WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

PRACA DYPLOMOWA

STUDIA I°

Temat pracy: **PROJEKT I IMPLEMENTACJA GRY 3D Z WYKORZYSTANIEM TECHNOLOGII RZECZYWISTOŚCI WIRTUALNEJ**

INFORMATYKA

(kierunek studiów)

INŻYNIERIA SYSTEMÓW

(specjalność)

Dyplomant:

Mariusz GALJAN

Promotor:

Mgr inż. Paweł PIECZONKA

Warszawa 2022

OŚWIADCZENIE

Wyrażam zgodę / nie wyrażam[*] zgody
na udostępnianie mojej pracy w czytelni Archiwum WAT.

Dnia ....28.08.2022r..... ……
(podpis)

[*]*Niepotrzebne skreślić*

Spis treści

Wstęp

Rzeczywistość wirtualna stanowi obecnie jedną z najbardziej rozwojowych dziedzin technologicznych. Dzięki wsparciu takich międzynarodowych korporacji jak Meta czy Valve gogle rzeczywistości wirtualnej przestały być kosztownym narzędziem specjalistycznym i stały się powszechnie dostępnym produktem oferującym unikatowe, immersyjne doświadczenie. To spowodowało zwiększenie popytu na produkcje rozrywkowe wykorzystujące tę technologię, w szczególności gry.

Celem przedstawionej pracy jest stworzenie trójwymiarowej gry wykorzystującej technologię rzeczywistości wirtualnej, która stworzy dookoła użytkownika interaktywny świat przedstawiony, na który ten będzie mógł wpływać dzięki dedykowanym kontrolerom. Priorytetem przy tworzeniu tego projektu jest zaprojektowanie i zaimplementowanie aplikacji spójnej i stabilnej.

Pierwszy rozdział zawiera omówienie podstawowych pojęć związanych z tworzeniem gier. Przeprowadzona zostaje również analiza rynku i dostępnych technologii wykorzystywanych przy tworzeniu gier.

Rozdział drugi został poświęcony zagadnieniu rzeczywistości wirtualnej. Opisany został w nim sposób działania tej technologii, problemy związane z wytwarzaniem aplikacji na tę platformę i ograniczenia, z jakimi musi mierzyć się projektant takiej produkcji.

W rozdziale trzecim omówione zostało pojęcie silnika gry. Zawarte zostało zestawienie najpopularniejszych silników przedstawiające ich wady i zalety oraz uzasadnienie wyboru programu, w którym implementowana będzie tworzona gra.

Rozdział czwarty opisuje pozostałe wykorzystane narzędzia takie jak edytor kodu czy programy wspomagające zarządzanie projektem. Wymienione zostały również narzędzia niezbędne do pracy z zasobami audiowizualnymi stanowiącymi nieodłączną część każdej gry.

W rozdziale piątym przedstawiono kroki podjęte w fazie projektowania gry. Opisano sformułowane wymagania, a także stworzono diagramy i opisy pomocne przy skonkretyzowaniu wizji architektury projektu.

Rozdział szósty skupia się na implementacji wspomnianej aplikacji. Opisuje proces wytwórczy obrany podczas realizowania zaprojektowanych założeń oraz przedstawia implementację wybranych elementów aplikacji.

W rozdziale siódmym przedstawiono przeprowadzone testy wraz z ich wynikami. Zaprezentowano zdefiniowane scenariusze testowe, a także przeprowadzone badania zaczynając od testów najniższego poziomu (jednostkowych) i omawiając testy obejmujące coraz większe fragmenty końcowej aplikacji.

Wybór omawianej tematyki pozwoli na zgłębienie zagadnień technologii rzeczywistości wirtualnej oraz pozwoli na wykreowanie nowych doświadczeń dla użytkowników tej platformy.

Rozdział I. Analiza i omówienie zagadnień związanych z grami komputerowymi

I.1. Definicja gry komputerowej

Gry komputerowe można zdefiniować jako rodzaj oprogramowania umożliwiający przeprowadzenie rozgrywki. Celem każdej gry jest stworzenie spójnego środowiska o określonych zasadach funkcjonowania. [23]

W toku rozgrywki zadaniem gracza jest wykorzystywanie wspomnianych reguł, żeby przezwyciężać kolejne stawiane przed nim wyzwania. W zależności od rodzaju oferowanej rozgrywki, przeszkody mogą pochodzić z różnych źródeł np. z rywalizacji z innymi graczami, z odgórnie ustalonych przez twórców gry scenariuszy czy nawet z ograniczeń powstałych dzięki inwencji twórczej samego gracza.

I.2. Rozwój rynku gier komputerowych

Branża gier komputerowych stanowi jedną z najszybciej rozwijających się gałęzi informatycznych. Według danych Polskiej Agencji Inwestycji i Handlu dochody ze sprzedaży gier komputerowych w 2019 roku w samej Polsce przekroczyły 2,1 mld złotych, a średni roczny wzrost tej wartości względem poprzedniego roku na przestrzeni poprzednich 5 lat wynosił około 30%.

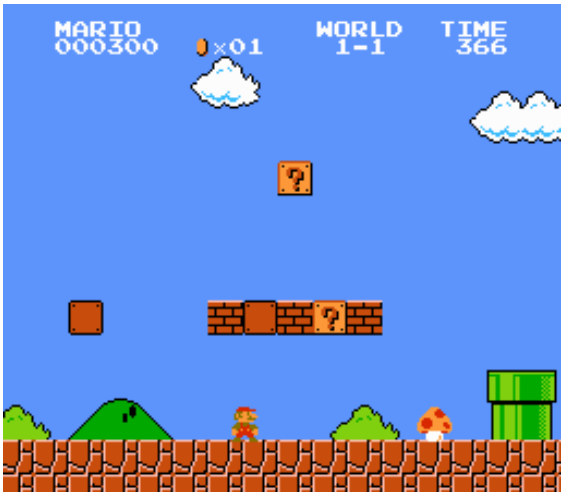
Początków gier komputerowych można doszukiwać się w latach 50. XX-ego wieku. Istnieją różne opinie odnośnie tego, jaka była pierwsza produkcja w historii, zależne najczęściej od kryteriów określania czegoś grą komputerową. [22]

W 1947 roku powstała gra „*Cathode-Ray Tube Amusement Device*", zbudowana w oparciu o technologię lamp elektronowych. Następnie, pod koniec lat 50., pojawiła się gra „*Spacewar*", która jako pierwsza oferowała funkcję generowania obrazu.

Przez wiele lat ograniczona moc obliczeniowa ówczesnych komputerów pozwalała jedynie na generowanie nieskomplikowanych obrazów dwuwymiarowych (w skrócie 2D). Obiekty w grze składały się wtedy z serii takich obrazków, które przełączane w odpowiednich momentach sprawiały wrażenie ruchu (niczym w klasycznej, filmowej animacji).

Rys. 1. Gra "Spacewar"

Źródło: https://www.flickr.com/photos/35034362831@N01/494431001



Rys. 2. Grafika dwuwymiarowa

Źródło: Super Mario Bros.. Nintendo

Kolejny przełom w dziedzinie tworzenia gier stanowiło wprowadzenie grafiki 3D. W tym przypadku poszczególne obiekty przechowywane są najczęściej w formie zbiorów wielokątów, w szczególności trójkątów. Każdy wielokąt ma określone współrzędne swoich wierzchołków, co pozwala tworzyć kształty zbliżone do przedmiotów ze świata rzeczywistego. Z biegiem czasu dane o wierzchołkach zostały wzbogacone o parametry takie jak kolor czy zdolność odbijania światła.

I.3. Gry z perspektywy pierwszoosobowej

Szczególnym rodzajem gier trójwymiarowych są gry z perspektywy pierwszoosobowej (ang. *FPP – First-Person Perspective*). Ich charakterystyczną cechą jest ustawienie kamery w pozycji oczu sterowanej przez gracza postaci. Jest to zabieg bardzo często stosowany w grach związanych ze strzelaniem i szeroko pojętym celowaniem, ponieważ stwarza wrażenie immersyjnego celowania z broni.

Ten sposób ustawienia kamery jest również najczęściej stosowany w grach wykorzystujących rzeczywistość wirtualną, ponieważ pozwala na naturalne i intuicyjne dla człowieka przedstawienie ruchów postaci odwzorowywanych w przestrzeni wirtualnej.

Rys. 3. Przykład perspektywy pierwszoosobowej

Źródło: The Elder Scrolls V: Skyrim, Bethesda Softworks

I.4. Gry w trybie Boga

Gatunek tak zwanych „Gier w trybie Boga” (ang. *God Game*) to odłam gier komputerowych, których akcja najczęściej toczy się samoistnie, bez ingerencji gracza. Użytkownik wciela się w rolę wszechobecnej istoty, której zadaniem jest uważna obserwacja rozwijającej się sytuacji i ingerowanie w rozgrywające się wydarzenia tak, żeby osiągnąć postawiony cel rozgrywki.

Gatunek ten powstał na początku XXI wieku. Za pierwszą grę z tego gatunku uznawana jest produkcja „*Populous*” autorstwa Petera Molyneux i kierowanego przez niego studia *Bullfrog Productions*. Popularność tych gier zmalała na przestrzeni lat,

jednakże wraz z rozpowszechnieniem się platform rzeczywistości wirtualnej znów pojawiają się nowe tytułu z tego gatunku.

Rys. 4. Przykład gry w trybie boga

Źródło: Townsmen VR, HandyGames™

I.5. Silniki do tworzenia gier

Silniki do tworzenia gier stanowią zbiór najczęściej powtarzających się w aplikacjach funkcji i komponentów takich jak symulacja fizyki czy poprawne wyświetlanie modeli i obrazów na ekranie. Twórcy mogą swobodnie wykorzystywać te elementy i dostosowywać je w celu wykorzystania w swoich projektach.

Do największych zalet korzystania z gotowych, ogólnodostępnych silników można zaliczyć:

- Szereg zaimplementowanych podstawowych funkcji związanych z tworzeniem gier
- Ustandaryzowana struktura projektów
- Wsparcie dla tworzenia gier na wiele platform jednocześnie
- Rozbudowaną dokumentację kodu i komponentów silnika
- Aktywną społeczność deweloperów pomagającą w rozwiązywaniu różnego rodzaju problemów i udostępniająca rozszerzenia silnika

Trzeba jednak zauważyć, że korzystanie z gotowych narzędzi wiąże się też z pewnymi ograniczeniami lub wręcz wadami:

- Twórcy silników nie mogą przewidzieć struktury gry, w związku z czym muszą uwzględniać pewne potencjalnie niepotrzebne zabezpieczenia spowalniające działanie programu
- W niektórych przypadkach gra może okazać się na tyle specyficzna, że domyślnie włączone systemy można zastąpić bardziej wydajnymi, uproszczonymi wersjami lub w ogóle je wyłączyć.
- Gotowe silniki są w wielu przypadkach płatne

Ze względu na powyższe aspekty większość studiów zajmujących się tworzeniem gier decyduje się jednak na korzystanie z gotowych silników. Jedynie największe korporacje, posiadające odpowiedni budżet, tworzą osobne zespoły zajmujące się niemal wyłącznie tworzeniem silników dedykowanych aktualnie tworzonym produkcjom.

Rozdział II. Technologia rzeczywistości wirtualnej

II.1. Opis pojęcia rzeczywistości wirtualnej

Rzeczywistość wirtualna to technika komputerowego modelowania i symulacji pozwalającej użytkownikowi na interakcję z wygenerowanym otoczeniem za pomocą dedykowanych kontrolerów, najczęściej gogli i rękawic. Użytkownik zakłada gogle, które reagują na ruchy jego głowy, co umożliwia zmianę pozycji i rotacji wyświetlanego obrazu. Rękawice natomiast pozwalają na interakcję z obiektami wewnątrz symulacji.

Przez wiele lat ta technologia wymagała specjalistycznego sprzętu niedostępnego komercyjnie. W związku z tym rozwijana była głównie w placówkach akademickich, wojskowych oraz medycznych.

Dzisiejsze komputery pozwalają jednak na generowanie obrazu rzeczywistości wirtualnej w czasie rzeczywistym, w związku z czym technologia ta staje się coraz bardziej powszechna w warunkach domowych.

II.2. Generowanie obrazu

Jedną z największych trudności w tworzeniu realistycznego obrazu jest stworzenie wrażenia głębi obrazu. Ludzki mózg określa odległość danego obiektu na podstawie tego, jak bardzo różny jest obraz z lewego oka od obrazu z prawego.

Gogle rzeczywistości wirtualnej rozwiązują ten problem poprzez wyświetlanie dwóch obrazów, po jednym dla każdego oka w następującym procesie:

Obrazy te generowane są na podstawie pojedynczego, klasycznie generowanego obrazu.

Widok ten jest następnie przetwarzany przez algorytm, który tworzy z niego dwa obrazy zniekształcone tak, żeby oczy mogły zinterpretować te obiekty jako mniej lub bardziej odległe.

Obrazy są oglądane przez specjalne soczewki, które nadają im wrażenia optycznej głębi.

Rys. 5. Podwójny obraz generowany przez VR

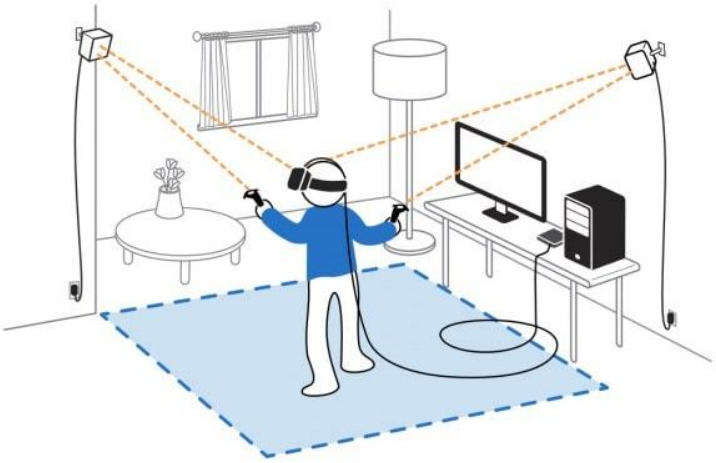
Źródło: hypergridbusiness.com

II.3. Odwzorowanie ruchu

W celu odwzorowania ruchów użytkownika wykorzystuje się czujniki ruchu. Najczęściej są one umieszczone w odległości 2-3 metrów od użytkownika i w odległości 1,5 – 2 metrów od siebie.

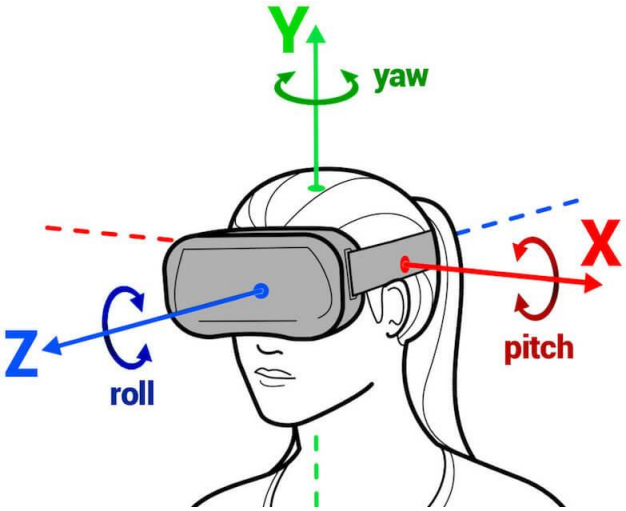
Czujniki skanują ruch użytkownika w przestrzeni trójwymiarowej pod kątem sześciu zmiennych tworzących tzw. Sześć Stopni Swobody (ang. *Six Degrees of Freedom*). Są to:

- Trzy współrzędne określające wektor przesunięcia na każdej z osi XYZ
- Trzy współrzędne określające zmianę rotacji wokół każdej z osi XYZ



Rys. 6. Szkic działania sensorów motorycznych

Źródło: https://4experience.co/vr-tracking-meet-degrees-of-freedom/



Rys. 7. : Sześć stopni swobody

Źródło: http://www.onlinecmag.com/virtual-reality-motion-tracking-technology/

II.4. Ograniczenia technologii

Mimo intensywnego rozwoju, technologia rzeczywistości wirtualnej nadal posiada wiele ograniczeń. Na wrażenie rzeczywistego świata składa się wiele czynników poza obrazem, dźwiękiem i ruchem.

Współczesne kontrolery pozwalają na poruszanie dłońmi w wygenerowanej przestrzeni i interakcje z przedmiotami za pomocą przycisków. Możliwe jest nawet wykorzystanie ruchu zaciskania dłoni przy podnoszeniu przedmiotów. Jednakże, nie istnieje obecnie technologia imitacji uczucia dotyku i odwzorowania tekstury dotykanego materiału. Dla przykładu, ludzka percepcja wymaga bodźca sygnalizującego, że dotykany wirtualny kubek wykonany został z porcelany.

Podobne problemy dotyczą innych aspektów takich jak ciepło przedmiotu, zapach, smak w przypadku przedmiotów jadalnych. Na chwilę obecną istnieją rozwiązania niektórych ze wspomnianych przeszkód, jednakże znajdują się w fazie wczesnych prototypów i jeszcze wiele lat minie, nim zostaną dopuszczone do sprzedaży komercyjnej.

Kolejnym bardzo istotnym problemem jest przemieszczanie się w świecie wirtualnym. Zmiana lokacji wirtualnej postaci przy jednoczesnym braku ruchu w świecie rzeczywistym może prowadzić do zawrotów głowy, a nawet objawów choroby lokomocyjnej.

Rys. 8. Prototyp urządzenia imitującego dotyk

Źródło: https://www.digitaltrends.com/news/wireality-carnegie-mellon-vr/

II.5. Wyzwania tworzenia aplikacji w rzeczywistości wirtualnej

II.5.1. Realistyczne odwzorowanie interakcji gracza z otoczeniem

Jednym z problemów stojących przed każdym twórcą aplikacji rzeczywistości wirtualnej jest obejście wspomnianych w poprzednim podrozdziale ograniczeń technologicznych w celu stworzenia jak najbardziej immersyjnego doświadczenia. Każde zdarzenie powinno uruchamiać szereg sygnałów dla zmysłów człowieka.

Przykładowo, jeśli użytkownik potrąci swoją wirtualną ręką puszkę, to powinny się wydarzyć następujące rzeczy:

- Puszka przewraca się i obraca zgodnie z prawami fizyki
- Użytkownik słyszy dźwięk przewracanej puszki
- Użytkownik odczuwa uderzenie np. za pomocą wibracji kontrolera

Takie informacje zwrotne można zapewnić przy obecnych ograniczeniach.

Kolejną sytuacją jest kontakt z przedmiotem nieporuszalnym jak np. ściana. Jeśli użytkownik spróbuje dotknąć ściany, to musi wiedzieć, kiedy rzeczywiście jej dotknął, a kiedy jego rzeczywista ręka zaczyna przenikać tę ścianę. Należy też ustalić, co dzieje się wtedy z ręką wirtualną. Jednym z rozwiązań jest ignorowanie ściany i pozwolenie na przeniknięcie wirtualnej ręki przez ścianę. Inny sposób to zatrzymanie wirtualnej kończyny i dopasowanie jej do pozycji ręki rzeczywistej, gdy powróci ona na miejsce.

To tylko pojedyncze przykłady wielu tego typu sytuacji, które należy rozstrzygnąć i zaprogramować.

II.5.2. Poruszanie się postaci w grze

Innym istotnym problemem jest poruszanie się. Płynne przesuwanie postaci jak w grach klasycznych może wywoływać u graczy zawroty głowy, a nawet objawy choroby lokomocyjnej.

W związku z tym powstało kilka różnych rozwiązań mających na celu ograniczenie tych objawów. Wielu twórców projektuje rozgrywkę tak, żeby nie wymagała ona zmiany pozycji postaci. Gracz kontroluje wtedy jedynie kończyny górne i głowę.

Innym rozwiązaniem jest teleportowanie postaci gracza we wskazane miejsce. Gracz po naciśnięciu przycisku musi wycelować w docelową lokalizację, w której następnie się po prostu pojawia.

II.6. Technologia wykorzystana w projekcie

Na potrzeby implementacji tego projektu konieczne jest wybranie technologii umożliwiającej korzystanie z rzeczywistości wirtualnej. Na rynku dostępne są zestawy od kilku różnych producentów. Wybór został zawężony do gogli działających na komputerach osobistych, co wyeliminowało np. gogle firmy Sony kompatybilne jedynie z konsolami *PlayStation*.

Na potrzeby implementacji tego projektu postanowiono wykorzystać zestaw *Oculus Rift* firmy *Facebook*. Do poprawnego działania wykorzystywane są gogle, dwa kontrolery oraz dwa sensory do śledzenia ruchu.

Rozdział III. Silnik gry

III.1. Przegląd dostępnych technologii

Wybór silnika do stworzenia tej gry sprowadza się do kilku silników wspierających tworzenie gier w wirtualnej rzeczywistości: *Unreal Engine*, *Unity* oraz *Godot*. Każdy z tych silników ma swoje wady i zalety. Poniżej przedstawione są parametry każdego z nich:

Tab. 1. Zestawienie parametrów silników do tworzenia gier

Parametr	Unreal Engine	Unity	Godot
Język programowania	C++	C#	GDScript, Visual Script, C#, C++
Data wydania	1998	2005	2014
Cena	5% zysków ze sprzedaży gry	Darmowe przy przychodzie brutto mniejszym niż 100,000$, potem płatna subskrypcja	Darmowy, open-source
Sklep z zasobami i modelami	Unreal Marketplace	Unity Asset Store	Godot Asset Library
Otwartość kodu	Publicznie dostępny do celów modyfikacji i rozszerzania silnika	Publicznie dostępna część silnika napisana w C#, część C++ niepubliczna	Publicznie dostępny, na licencji MIT

Źródła: https://www.unrealengine.com/en-US/, https://unity.com/, https://godotengine.org/

Jak widać, wszystkie silniki oferują na pierwszy rzut oka podobne rozwiązania. Ostatecznie jednak do implementacji wybrany został silnik *Unity*, ponieważ istnieje na rynku dłużej niż *Godot*, pozwala na pisanie kodu w języku C#, oferuje korzystny plan cenowy dla deweloperów mniejszych gier z mniejszym przychodem oraz posiada

dobrze skonstruowaną i rozbudowaną dokumentację. Istotnym argumentem było również wcześniejsze doświadczenie w pracy z tym silnikiem.

III.2. Podstawowe elementy silnika Unity

Rys. 9. Interfejs Unity

Źródło: https://docs.unity3d.com/520/Documentation/Manual/LearningtheInterface.html

III.2.1. Scena

Podstawowym elementem gry jest scena. Sceny stanowią zasoby (ang. *Asset)* przechowujące zbiór obiektów, z których zbudowany jest świat gry. Na grę może składać się wiele scen, między którymi można się swobodnie przełączać w trakcie rozgrywki. W ten sposób możemy np. stworzyć osobne sceny dla lokacji zamku i dla lokacji wiejskiej chaty.

III.2.2. Kamera

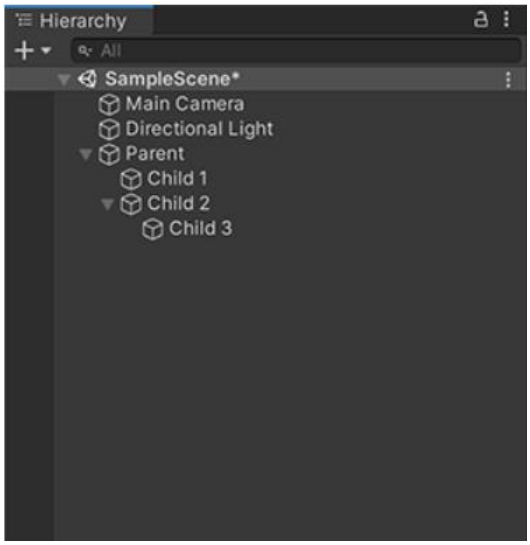
Żeby gra działała poprawnie, na scenie musi znajdować się kamera. Kamera odpowiada za poprawne wyświetlanie obiektów znajdujących się na scenie. Ustawienia

kamery są tak skonstruowane, żeby jak najlepiej odzwierciedlać ustawianie kamery np. przy kręceniu filmów.

III.2.3. Obiekt Gry (ang. *Game Object)*

Obiekt Gry stanowi najważniejszy element silnika Unity. Każdy obiekt w grze jest Obiektem Gry: drzewa, zwierzęta, telewizor czy nawet niebo.

Obiekty zorganizowane są w hierarchię. Każdy Obiekt może mieć przyporządkowane obiekty podległe. Przypisanie danego obiektu oznacza, że każde przesunięcie obiektu nadrzędnego w przestrzeni 3D poskutkuje przesunięciem obiektu podległego. Podobnie jeśli obiekt zostanie zdezaktywowany, to automatycznie zdezaktywowane zostaną też wszystkie obiekty podrzędne. Takie podejście pozwala na utworzenie przejrzystej hierarchii obiektów na scenie, dzięki której twórcy gier zyskują nowe funkcjonalności, a przy tym interfejs staje się bardziej czytelny.



Rys. 10. Przykładowa hierarchia obiektów na scenie

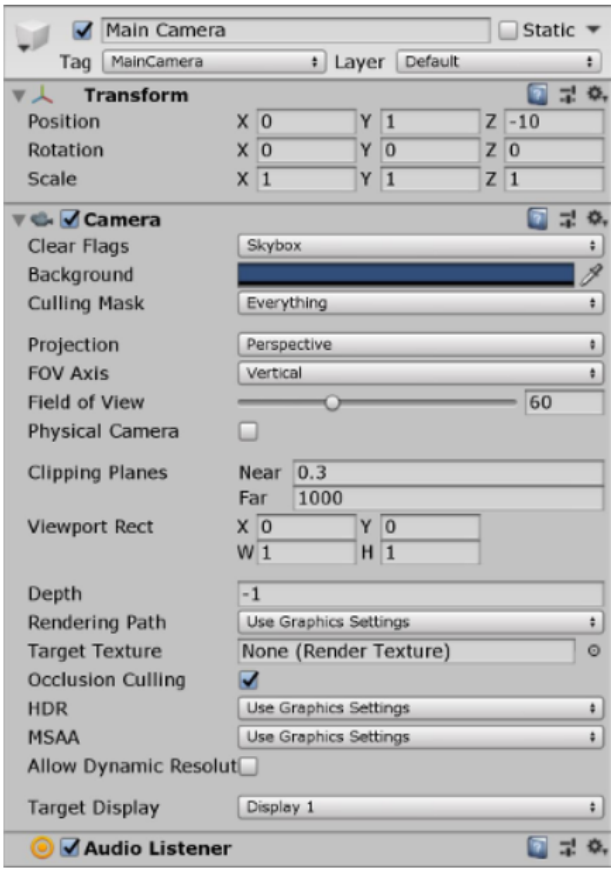
Źródło: https://docs.unity3d.com/Manual/Hierarchy.html

III.2.4. Komponent

Obiekt Gry stanowi jedynie reprezentację obiektu w świecie gry, jednakże sam obiekt nie posiada żadnych funkcji lub zachowań. Za to odpowiedzialne są komponenty, czyli instancje klas przechowywanych w skryptach, które są przypisane do obiektów. Przykładami komponentów mogą być klasy takie jak *Camera, Rigidbody* czy *Collider*.

Każda klasa komponentu dziedziczy z klasy *MonoBehaviour*, co daje jej dostęp do Obiektu Gry i pozwala na manipulację samym Obiektem oraz innymi komponentami nie tylko w obrębie tego obiektu, ale nawet całej sceny.

Jeden komponent może być przypisany do wielu różnych obiektów (każdy z nich ma wtedy przypisaną własną instancję klasy komponentu). Obiekty mogą składać się z dowolnych kombinacji różnych komponentów, co pozwala na elastyczne dopasowywanie zachowań poszczególnych obiektów do potrzeb projektu.



Rys. 11. Przykład widoku komponentów obiektu kamery

Źródło: https://docs.unity3d.com/Manual/Components.html

Rozdział IV. Pozostałe narzędzia

IV.1. Edytor kodu

Na potrzeby edytowania kodu źródłowego wybrany został program *Visual Studio Code*. Jest to narzędzie zajmujące mniej niż 500 MB pamięci dyskowej. Udostępnia wiele rozszerzeń ułatwiających pracę ze skryptami. *Visual Studio Code* umożliwia również obsługę systemu kontroli wersji *Git* oraz posiada wsparcie dla języka C# stworzone przez firmę *Microsoft*. Ponadto, Unity domyślnie udostępnia opcje współpracy z tym edytorem.

IV.2. Zarządzanie projektem

Z racji tego, że jest to projekt jednoosobowy, wykorzystana została metodyka *Kanban* przy pomocy platformy *Trello*. Ponadto, w ramach zarządzania projektem skorzystano z systemu kontroli wersji *Git* z repozytorium przechowywanym na platformie *GitLab*.

IV.3. Pozostałe narzędzia

Gra komputerowa stanowi doświadczenie multimedialne, w związku z czym oprócz samego silnika i edytora kodu do jej stworzenia niezbędne są między innymi:

- Modele 3D i elementy UI (2D)
- Animacje
- Pliki audio

W wielu sytuacjach stworzenie tych zasobów wymagałoby zatrudnienia szeregu specjalistów z różnych dziedzin. Jednakże, wielu twórców udostępnia swoje dzieła za darmo, dzięki czemu można je wykorzystać na potrzeby tego projektu. Warto wyróżnić tu takie serwisy jak *Unity Asset Store* czy *Mixamo*.

Pozostałe wykorzystane programy:

- *Blender* – program pozwalający na edycję modeli 3D i animacji
- *Audacity* – program do edycji dźwięku

Rozdział V. Projekt aplikacji

V.1. Ogólny opis projektu

Po uwzględnieniu informacji dotyczących dziedziny tworzenia gier komputerowych można przystąpić do fazy projektowania i konkretyzowania założeń tego projektu.

Tworzona aplikacja reprezentuje gatunek Gry w trybie boga. Centralną postać rozgrywki stanowi Bohater, który z upływem czasu automatycznie przemierza wyznaczoną, zapętloną ścieżkę złożoną z różnego rodzaju pokoi.

Do każdego pokoju przyporządkowany jest zbiór określonych efektów pokroju instancjonowania przeciwników czy leczenia bohatera. Do jednego pokoju może być przyporządkowanych wiele efektów jednocześnie. Każda z nich jest aktywowany, gdy przypisany do niego warunek jest spełniony. Warunkiem może być np. wejście Bohatera do tego pokoju lub upływ określonego czasu (odmierzanego w formie dni upływających w świecie gry). Gracz ma do dyspozycji wspomniane pokoje, które może ustawiać na odgórnie wyznaczonej siatce.

Zadaniem Gracza jest pomoc Bohaterowi w jak najdłuższym przetrwaniu na szlaku. Rozgrywka kończy się, gdy liczba Punktów Życia Bohatera spadnie do 0 lub Bohater pokona specjalnego przeciwnika pojawiającego się po ułożeniu przez Gracza odpowiedniej liczby pokoi.

Podczas rozgrywki odtwarzana jest również muzyka. Wybór aktualnie odtwarzanego podkładu dźwiękowego zależny jest od aktualnego stanu rozgrywki tj. podczas wyświetlania menu głównego słychać jeden, zawsze ten sam utwór, a podczas poruszania się podczas rozgrywki inny.

V.1.1. Opis przepływu rozgrywki

Po uruchomieniu gry ukazuje się menu główne, z poziomu którego gracz może:

- Rozpocząć nową rozgrywkę
- Kontynuować poprzednią rozgrywkę
- Zamknąć aplikację

W trakcie rozgrywki Gracz może w każdym momencie aktywować menu pauzy, które wstrzymuje rozgrywkę. Z poziomu tego menu Gracz może:

- Kontynuować rozgrywkę
- Wrócić do menu głównego

W grze istnieją również specjalne menu wyświetlane w przypadku zakończenia rozgrywki. Jeśli rozgrywka zakończy się porażką, to wyświetlane jest menu porażki. W

przeciwnym wypadku wyświetlane jest menu zwycięstwa. Oba widoki składają się z tych samych przycisków. Z ich poziomu gracz może:

- Rozpocząć nową rozgrywkę
- Wrócić do menu głównego

V.1.2. Bohater

Bohater w trakcie rozgrywki przemierza odgórnie ustaloną trasę złożoną z pokoi. Kiedy wkracza do pokoju i znajdują się w nim wrogowie, Bohater wdaje się z nimi w walkę. Jeśli starcie zakończy się sukcesem Bohatera, ten kontynuuje przemierzanie pokoi. W przeciwnym wypadku rozgrywka zostaje zakończona.

V.1.3. Pokoje

Pokoje ustawiane są na odgórnie wyznaczonej siatce. Pokój nie może zostać ustawiony w miejscu, w którym już znajduje się inny pokój. Istnieją pokoje dwóch typów: pokoje na ścieżce i pokoje na pozostałych polach siatki. Jak nazwa wskazuje, pokoje na ścieżce mogą być ustawione jedynie na drodze Bohatera, podczas gdy pozostałe pokoje na pozostałych polach siatki.

Funkcje pokoi mogą być aktywowane w jednej z trzech sytuacji:

- W momencie wkroczenia Bohatera do pokoju.
- W momencie rozpoczęcia kolejnego dnia.
- W momencie ustawienia pokoju na siatce.

Funkcje zaimplementowane w grze są następujące:

Tab. 2. Zestawienie efektów pokoi w grze

Lp.	Nazwa efektu	Efekt przy wkroczeniu Bohatera do pokoju	Efekt przy rozpoczęciu kolejnego dnia	Efekt przy ustawieniu pokoju na siatce
1.	Zwiększenie progresu	-	-	Zwiększa wartość progresu osiągniętego w grze.

2.	Leczenie	Leczy Bohatera o określoną liczbę Punktów Życia.	-	-
3.	Zradzanie przeciwników	-	Zrodzenie przeciwników zgodnie z ustawieniami danego pokoju.	-
3.	Dodanie przeciwnika do walk	W walce rozpoczętej w przylegającym pokoju uczestniczy dodatkowy, wskazany w ustawieniach przeciwnik.	-	-

Źródło: Badania własne

Kolejne pokoje są udostępniane graczowi za pomocą specjalnego generatora przedstawionego w formie szyny, na której pojawiają się nowe pokoje gdy Bohater zakończy walkę z sukcesem. Gracz może wtedy podnieść jeden z tych pokoi i umieścić go na siatce.

V.2. Specyfikacja wymagań

Priorytety poszczególnych wymagań zostały oznaczone zgodnie z następującą legendą:

Tab. 3. Objaśnienie priorytetów wymagań projektowych

Lp.	Priorytet	Opis
1.	Wysoki	Wymaganie musi zostać spełnione
2.	Średni	Wymaganie powinno zostać spełnione, ale nie musi
3.	Niski	Wymaganie może zostać spełnione, ale nie musi

Źródło: Badania własne

Na podstawie powyższego opisu sformułowane zostały wymagania dotyczące projektu. Wymagania zostały przedstawione w formie poniższych tabel w kolejnych punktach tego podrozdziału.

V.2.1. Wymagania funkcjonalne

W pierwszej kolejności należy określić wymagania funkcjonalne dotyczące mechanik zawartych w tworzonej grze:

Tab. 4. Wymagania funkcjonalne dotyczące mechanizmów rozgrywki

ID	Treść	Priorytet
MR1	Gracz musi mieć możliwość ustawiania pokoi na specjalnie wyznaczonych do tego miejscach.	Wysoki
MR2	Pokoje ustawione na siatce muszą aktywować odpowiednie efekty, gdy rozpocznie się kolejny dzień	Wysoki
MR3	Pokoje ustawione na siatce muszą aktywować odpowiednie efekty, gdy Bohater do nich wkroczy	Wysoki
MR4	Pokoje ustawione na siatce powinny aktywować odpowiednie efekty w momencie ustawienia na siatce.	Średni
MR5	Gracz powinien mieć możliwość włączania i wyłączania pauzy w podczas rozgrywki	Średni
MR6	Gracz może mieć możliwość manipulowania prędkością upływu czasu podczas rozgrywki.	Niski

Źródło: Badania własne

Należy również wyróżnić wymagania dotyczące wybranych efektów:

Tab. 5. Wymagania funkcjonalne dotyczące efektów przypisanych do pokoi

ID	Treść	Priorytet
EF1	Pokój z przypisanym efektem leczenia Bohatera musi zwiększać liczbę Punktów Życia Bohatera gdy ten wkroczy do tego pokoju.	Wysoki

VR2	Pokój z przypisanym efektem zradzania przeciwników musi zradzać nowych przeciwników i umieszczać ich w środku tego pokoju w momencie upłynięcia wskazanego w ustawieniach pokoju czasu.	Wysoki

Źródło: Badania własne

Następnie można wyodrębnić wymagania dotyczące korzystania z technologii rzeczywistości wirtualnej:

Tab. 6. Wymagania funkcjonalne dotyczące mechanizmów rzeczywistości wirtualnej

ID	Treść	Priorytet
VR1	Kiedy gracz przesunie prawy kontroler o wektor [x, y, z] metrów w przestrzeni rzeczywistego świata, gra musi przesunąć wirtualną rękę przyporządkowaną do tego kontrolera o taki sam wektor w przestrzeni wirtualnej.	Wysoki
VR2	Kiedy wirtualna ręka znajdzie się w odległości mniejszej niż 0,2m od obiektu interaktywnego i gracz naciśnie przycisk interakcji, musi zostać uruchomiona interakcja przypisana do tego obiektu.	Wysoki
VR3	Gracz powinien mieć możliwość podnoszenia niektórych przedmiotów interaktywnych	Średni
VR4	Gracz może mieć możliwość aktywowania trzymanych obiektów interaktywnych (jeśli taki obiekt ma przypisaną dodatkową aktywację)	Niski

Źródło: Badania własne

Warto również wymienić wymagania dotyczące widoków menu:

Tab. 7. Wymagania funkcjonalne dotyczące interfejsów użytkownika

ID	Treść	Priorytet
MR1	Gracz musi mieć możliwość uruchomienia nowej rozgrywki z poziomu głównego menu.	Wysoki

MR2	Gracz musi mieć możliwość zamknięcia aplikacji z poziomu menu głównego.	Wysoki
MR3	Po uruchomieniu aplikacja powinna pokazywać menu główne.	Średni
MR4	Gracz powinien mieć uruchomienia menu pauzy w trakcie rozgrywki poprzez naciśnięcie odpowiedniego przycisku na kontrolerze.	Średni
MR5	Gracz powinien mieć możliwość przejścia do głównego menu z poziomu menu pauzy	Średni
MR6	Gracz może mieć możliwość otworzenia widoku ustawień gry z poziomu menu głównego.	Niski

Źródło: Badania własne

V.2.2. Wymagania pozafunkcjonalne

W poniższej tabeli zawarto wymagania pozafunkcjonalne dotyczące tworzonej aplikacji:

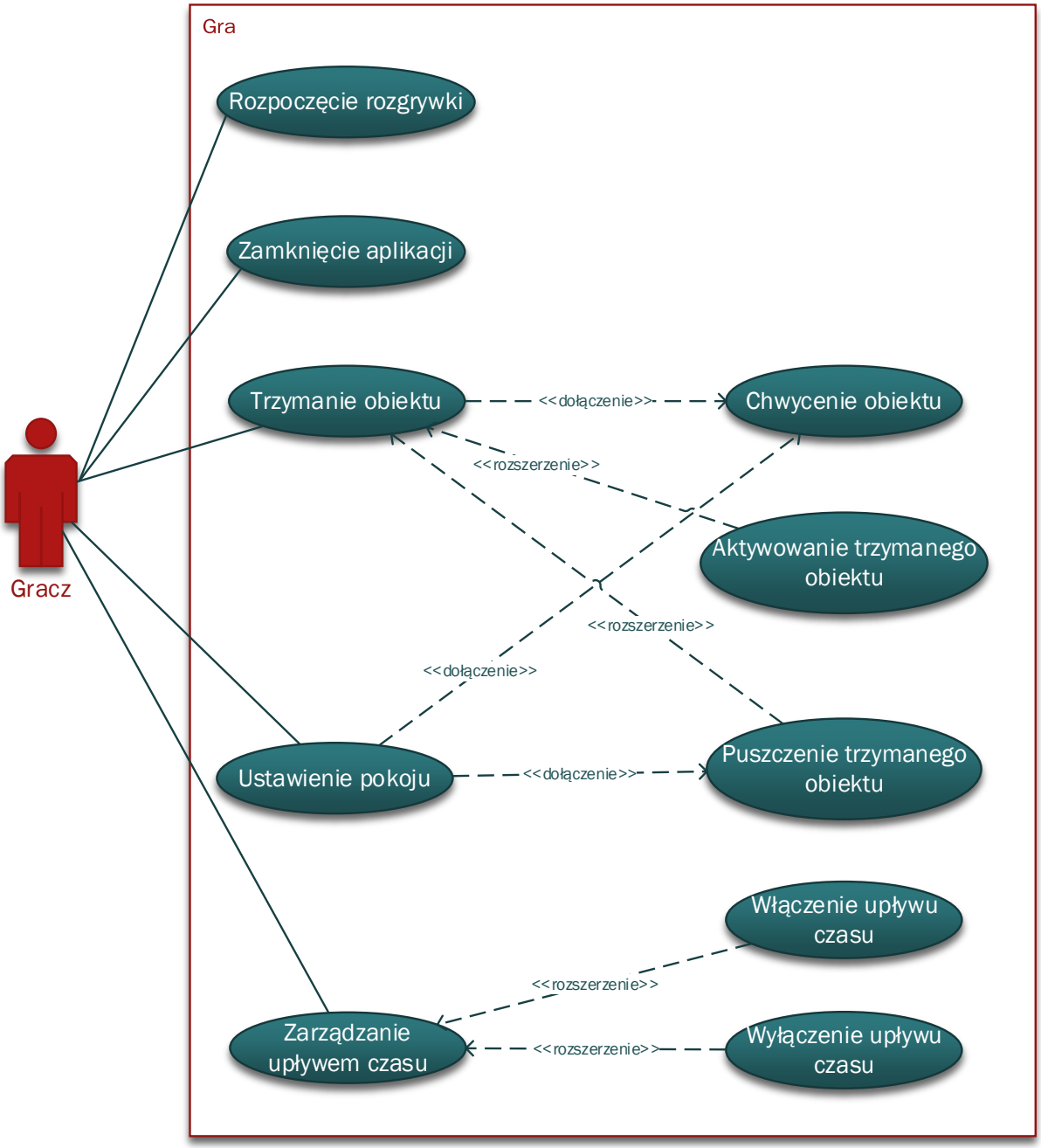
Tab. 8. Wymagania pozafunkcjonalne

ID	Treść	Priorytet
PF1	Gra musi działać poprawnie na urządzeniach Oculus Rift podłączonych do komputerów z systemem Windows w wersji 10 lub nowszej.	Wysoki
PF2	Czas przetwarzania pojedynczej klatki nie powinien przekraczać 33ms na komputerze z zadaną specyfikacją techniczną: • Procesor: Intel Core™ i7-7700HQ (2.8GHz) • Karta graficzna: NVIDIA GeForce GTX 1060 6GB • Pamięć RAM: 16GB	Średni
PF3	Gra musi być zlokalizowana w języku angielskim	Wysoki
PF4	Gra może być zlokalizowana w języku polskim	Niski

Źródło: Badania własne

V.3. Diagramy przypadków użycia

W projekcie można zdefiniować jednego aktora, którym jest gracz. W związku z tym i z powyższymi wymaganiami, można sformułować następujący diagram przypadków użycia:



Rys. 12. Diagram przypadków użycia

Źródło: Badania własne

V.4. Wybrane scenariusze przypadków użycia

V.4.1. Rozpoczęcie nowej rozgrywki

Aktor: Gracz

Opis: Po uruchomieniu aplikacji Graczowi ukazane jest menu główne. Żeby rozpocząć rozgrywkę, Gracz musi wybrać przycisk nowej gry. Po naciśnięciu przycisku menu główne zostaje zastąpione widokiem danych dotyczących stanu rozgrywki i zostaje załadowana nowa sesja tejże rozgrywki.

Warunki wstępne:

1. Aplikacja została uruchomiona.

Przebieg wydarzeń:

1. Aktywowany jest widok głównego menu.
2. Wirtualne ręce Gracza przełączają się w tryb interakcji za pomocą wskaźnika pod postacią promienia.
3. Gracz wskazuje przycisk rozpoczęcia nowej gry.
4. Gracz naciska przycisk interakcji na kontrolerze.
5. Menu główne zostaje ukryte.
6. Wirtualne ręce Gracza zostają przełączone w tryb interakcji bezpośredniej. Wskaźnik znika.
7. Aktywowany jest widok parametrów aktualnej sesji rozgrywki.
8. Zostaje załadowana nowa sesja rozgrywki.

Warunki końcowe:

1. Menu główne jest nieaktywne.
2. Widok parametrów aktualnej sesji jest aktywny.
3. Rozgrywka została rozpoczęta.
4. Gracz może używać jedynie interakcji bezpośredniej.

V.4.2. Ustawienie pokoju

Aktor: Gracz

Opis: Aby Gracz ustawił pokój na Planszy, musi za pomocą wirtualnej ręki przypisanej do kontrolera podnieść obiekt Pokoju, a następnie ustawić go na Planszy. Ustawiony pokój zostaje aktywowany i zaczyna wywoływać swoje funkcje w określonych momentach (różnych dla każdej funkcji lub pokoju).

Warunki wstępne:

1. Rozgrywka została rozpoczęta.
2. Istnieje przynajmniej jeden Pokój na „szynie” do generowania pokoi, który Gracz może podnieść
3. Istnieje przynajmniej jedno dostępne pole na Planszy, na którym Gracz może ustawić Pokój.

Przebieg wydarzeń:

1. Gracz przesuwa wirtualną rękę za pomocą kontrolera rzeczywistości wirtualnej do obiektu Pokoju tak, żeby znalazła się w obrębie Zderzacza (ang. *Collider*) przypisanego do tego obiektu.
2. Obiekt Pokoju zostaje wybrany do możliwej interakcji.
3. Gracz naciska i przytrzymuje przycisk na kontrolerze przypisany do akcji podniesienia obiektu. Obiekt Pokoju zostaje podniesiony.
4. Gracz przesuwa wirtualną rękę tak, żeby znalazła się w obrębie Zderzacza przypisanego do obiektu miejsca na Planszy, w którym chce postawić Pokój.
5. Obiekt miejsca na Planszy zostaje wybrany do możliwej interakcji.
6. Gracz puszcza przycisk przypisany do akcji podniesienia przedmiotu.
7. Pokój zostaje ustawiony we wskazanym miejscu na Planszy, a jego funkcje zostają aktywowane.

Warunki końcowe:

1. Pokój jest ustawiony w odpowiednim miejscu na Planszy.
2. Pokój został aktywowany.
3. Wykorzystane miejsce na Planszy jest zajęte, co blokuje możliwość ustawienia następnych Pokoi w tym miejscu.

V.4.3. Zatrzymanie upływu czasu

Aktor: Gracz

Opis: W celu przemyślenia sytuacji i zyskania czasu na podjęcie decyzji, Gracz ma możliwość zatrzymania upływu czasu. Odbywa się to poprzez naciśnięcie przypisanego do tej akcji przycisku na kontrolerze. Zatrzymanie czasu wpływa jedynie na wydarzenia na Planszy, nie na Gracza.

Warunki wstępne:

1. Rozgrywka została rozpoczęta

Przebieg wydarzeń:

1. Gracz naciska przypisany przycisk.
2. Upływ czasu w świecie gry zostaje zatrzymany tj.:
    a. postacie nie zmieniają swojej pozycji.
    b. Przebieg animowania postaci zostaje zatrzymany.
    c. licznik cyklu dnia i nocy zostaje zatrzymany.

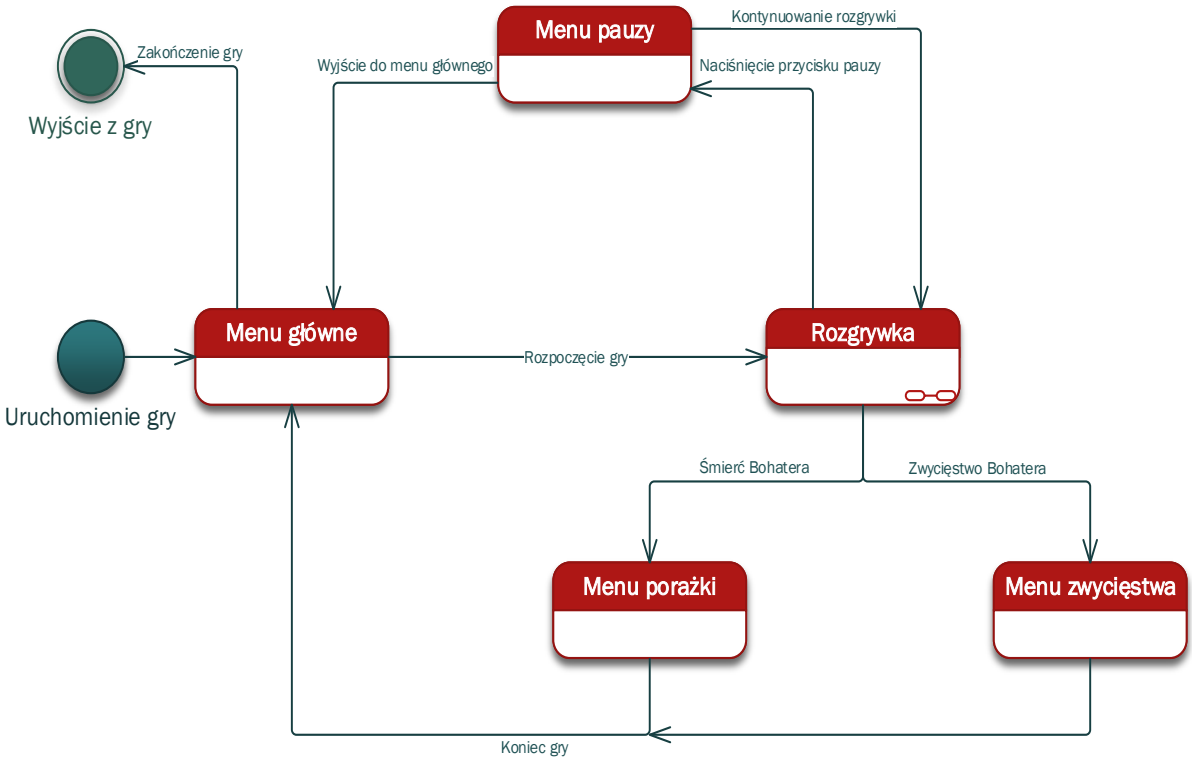
Warunki końcowe:

1. Upływ czasu w świecie wirtualnym zostaje zatrzymany.
2. Gracz może się dalej poruszać i wykonywać domyślne akcje.

V.5. Diagramy stanów gry

W celu zapewnienia przejrzystej i czytelnej struktury gry i jej działania, można na podstawie opisu projektu podzielić ją na poszczególne stany, w których może się znajdować. Dzięki temu można określić, co dzieje się w trakcie każdego stanu, a także w chwilach zmiany stanu gry.

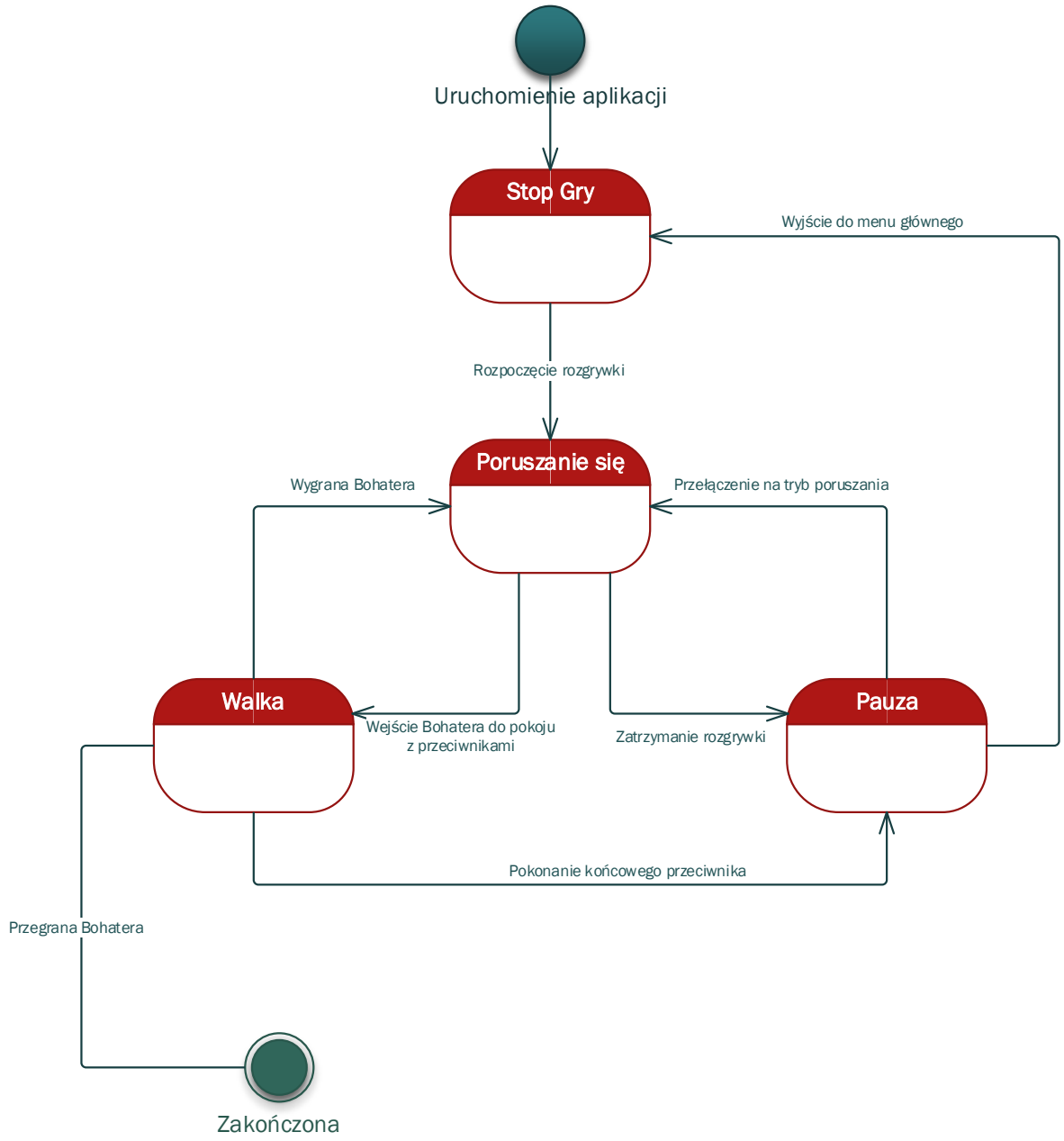
Stany gry i warunki przejść między nimi zostały zaprezentowane na poniższym diagramie:



Rys. 13. Diagram stanów gry

Źródło: Badania własne

Oprócz stanów całej gry, możemy również wyznaczyć stany samej rozgrywki na siatce:



Rys. 14. Diagram stanów rozgrywki na siatce

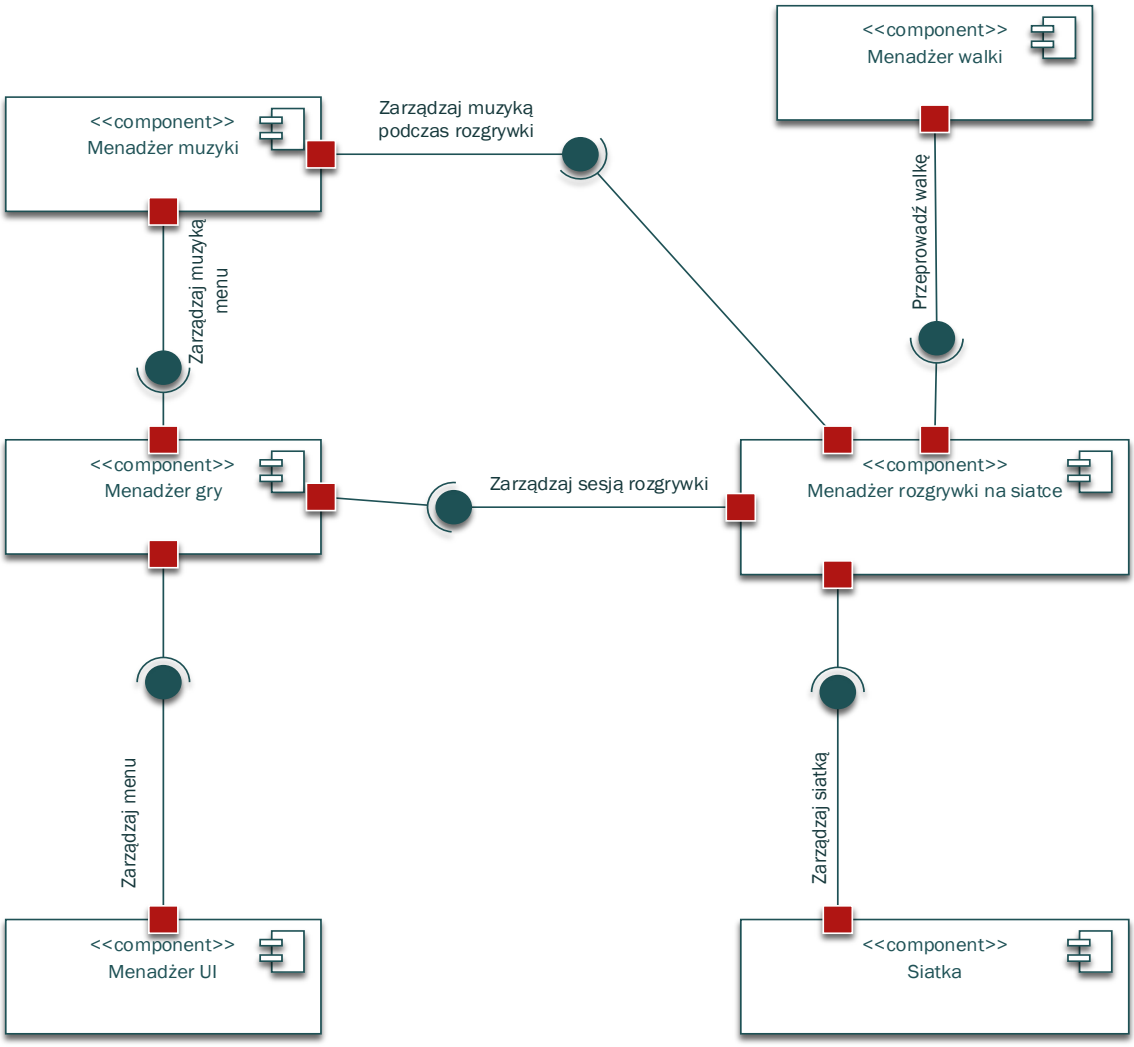
Źródło: Badania własne

V.6. Architektura systemów i diagram komponentów

W celu spełnienia powyższych wymagań należy wyszczególnić kilka fundamentalnych elementów aplikacji:

- Menadżer gry – element odpowiedzialny za zarządzanie przepływem całej rozgrywki
- Menadżer rozgrywki w pętli – element odpowiedzialny za zarządzanie pojedynczą sesją rozgrywki na siatce
- Menadżer UI – element odpowiedzialny za przełączanie widoczności wymienionych wcześniej menu i widoków

Oprócz tego możemy również wyróżnić bardziej specyficzne elementy takie jak kontroler siatki czy menadżer walki istot na siatce. Wszystkie te elementy i łączące je interfejsy zostały przedstawione na poniższym diagramie komponentów:



Rys. 15. Diagram komponentów gry

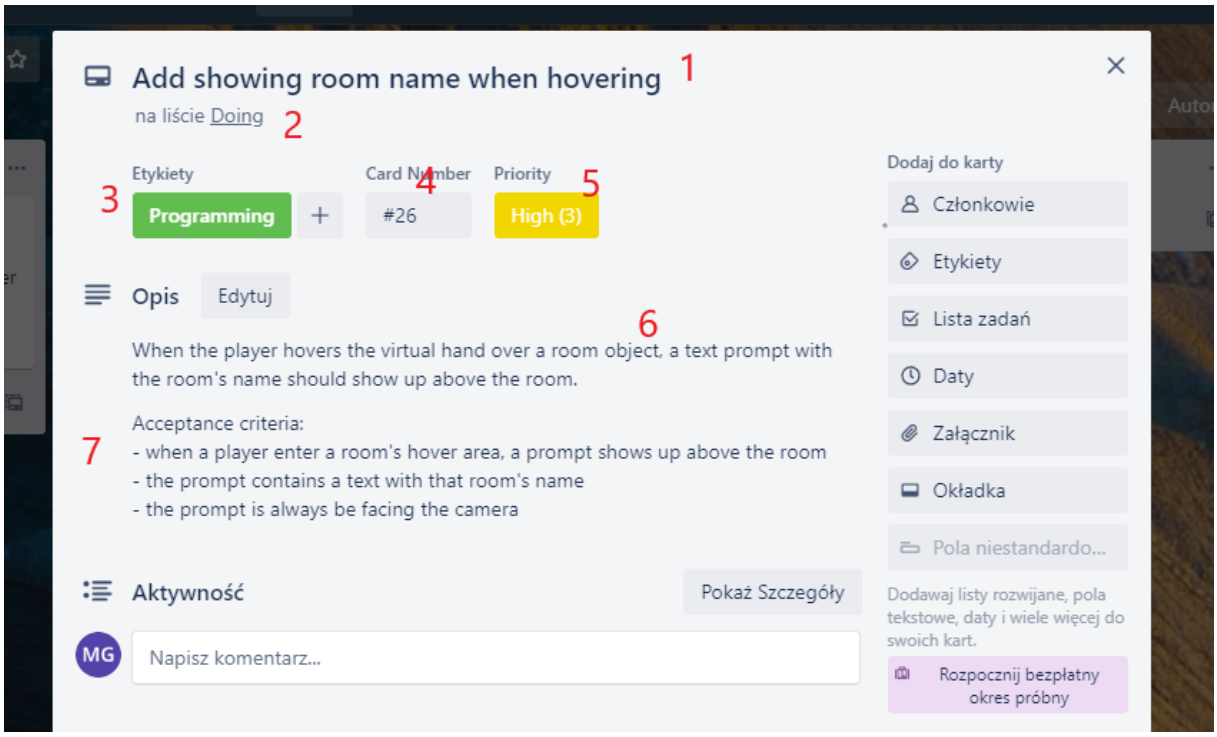
Źródło: Badania własne

Rozdział VI. Implementacja

VI.1. Proces implementowania aplikacji

VI.1.1. Definiowanie i zarządzanie zadaniami do wykonania

Po utworzeniu zdefiniowaniu aplikacji należało rozpisać zadania do zrealizowania. Zadania zostały spisane w aplikacji internetowej *Trello*, która umożliwia tworzenie wielu tablic *Kanban* w ramach jednej przestrzeni pracy (ang. *workspace*).



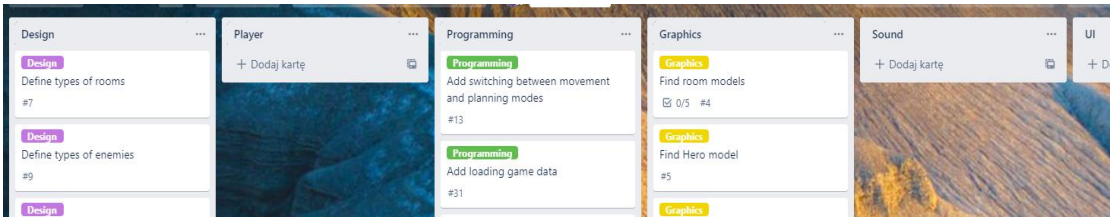
Rys. 16. Definicja zadania

Źródło: Badania własne

Zgodnie z numerami pokazanymi na Rys. 16 definicja zadania składa się z:

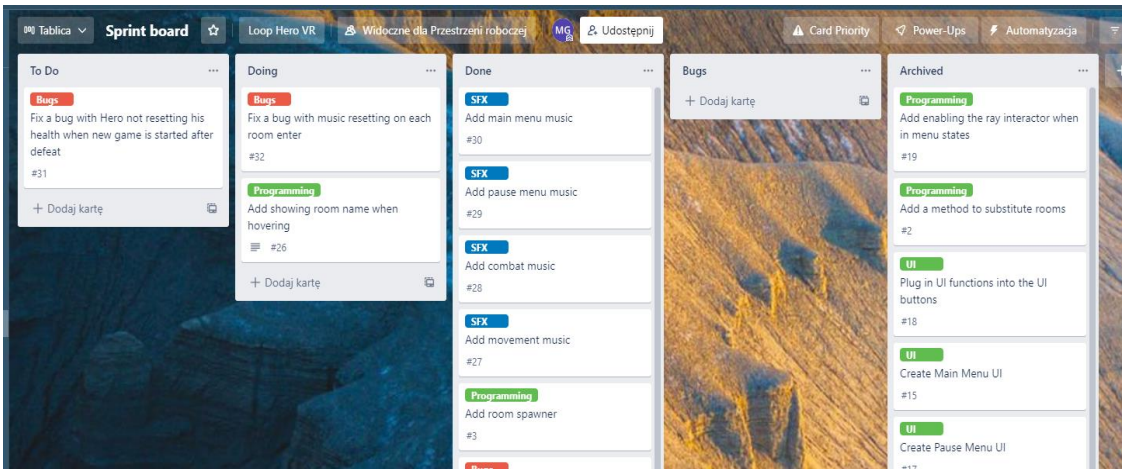
1. Nazwy zadania
2. Listy, do której zadanie jest przypisane
3. Etykiety zadania, określającej zakres projektu, do którego to zadanie należy (np. programowanie lub tworzenie interfejsu użytkownika)
4. Unikatowy identyfikator zadania
5. Priorytet zadania
6. Krótki opis treści zadania
7. Kryteria akceptacji

Zadania zostały podzielone na listy z poszczególnymi etykietami w tablicy *Backlog*. Następnie poszczególne zadania były przesuwane do tablicy *Sprint Board*, gdzie znajdowały się w jednej z trzech list: *Do zrobienia, W trakcie* oraz *Zrobione*. Oprócz tych list tablica *Sprint Board* zawiera również tablicę *Bugi*, do której na bieżąco były dodawane znalezione problemy aplikacji oraz *Archiwum*, do której były z czasem przenoszone zadanie z listy *Zrobione*. Aktualnie wykonywane zadania były umieszczane w liście *W trakcie.*



Rys. 17. Tablica *Backlog*

Źródło: Badania własne



Rys. 18. Tablica *Sprint Board*

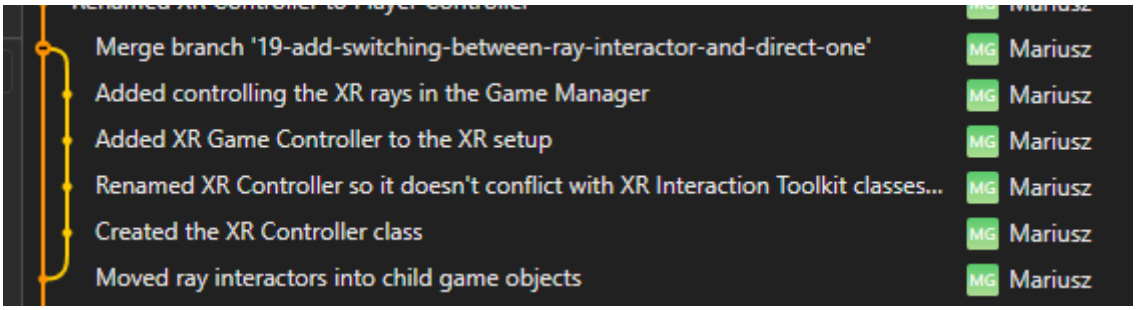
Źródło: Badania własne

VI.1.2. System kontroli wersji Git

Projekt aplikacji tworzony wewnątrz silnika Unity był utrzymywany w systemie kontroli wersji Git. Po wybraniu zadania do implementacji tworzona była nowa gałąź (ang. *branch*). Gałęzie były opisywane przez tytuł zadania oraz jego identyfikator. W związku z tym nazwą gałęzi dla zadania z Rys. 16 byłoby *26-add-showing-room-name-when-hovering*.

Po ukończeniu każdego definiowalnego kroku prowadzącego do wykonania zadania wykonywane było zatwierdzenie (ang. *commit*). Po wykonaniu zadania wykonywana była instrukcja złączenia (ang. *merge*) gałęzi zadania z gałęzi główną, w dzięki czemu wprowadzone zmiany były aplikowane do końcowej aplikacji. Po wykonaniu tej operacji gałąź związana z zadaniem była usuwana.

Graf gałęzi repozytorium przy przyjętym podejściu wyglądał następująco:



Rys. 19. Graf repozytorium projektu

Źródło: Badania własne

VI.2. Język programowania

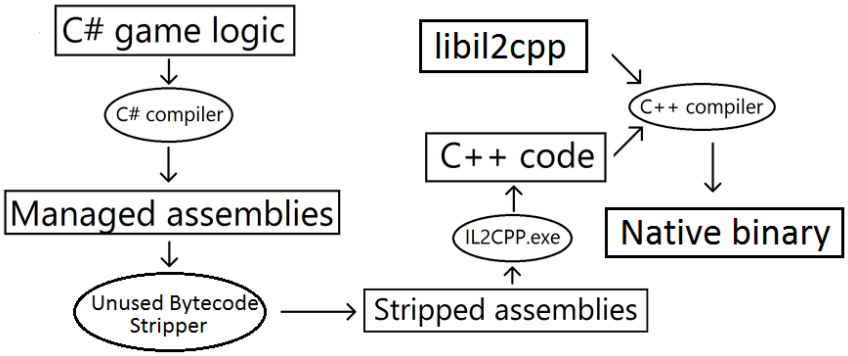
Na początku swojego istnienia silnik Unity umożliwiał pisanie skryptów w trzech językach: *C#, Boo i JavaScript*. Z biegiem czasu jednak twórcy silnika postanowili porzucić dwa ostatnie ze wspomnianych języków i skupić się na rozwoju własnego kompilatora języka *C#*.

C# to język programowania rozwijany przez firmę *Microsoft* od 2002 roku. Należy do grupy języków zorientowanych na programowanie obiektowe ze statyczną kontrolą typów. *C#* zaliczany jest do rodziny języków *C*, w związku można w składni tego języka można zauważyć wiele podobieństw do *C i C++*, a także języków *Java* czy *JavaScript*. [12]

Tworzenie aplikacji w języku *C#* nie wymaga ręcznego zarządzania pamięcią, ponieważ programy uruchamiane są na platformie *.NET* wyposażonej w tak zwany kolektor śmieci (ang. *Garbage collector*), którego zadaniem jest zwalnianie pamięci zajmowanej przez obiekty, do których nie ma żadnych aktywnych referencji w programie. [13]

Interesującym zagadnieniem jest to, że o ile skrypty silnika Unity pisane są w języku *C#*, podczas kompilacji silnik dokonuje konwersji napisanego kodu na język *C++*. Tradycyjne aplikacje *C#* są kompilowane do Powszechnego Pośredniego Języka (ang. *Common Intermediate Language)*, który przy uruchomieniu aplikacji jest kompilowany na kod maszynowy natywny dla danej platformy. Umożliwia to pisanie aplikacji na wiele platform jednocześnie, ale wiąże się z bardzo poważną wadą: wydajnością.

Kompilacja w momencie uruchomienia aplikacji jest procesem zbyt wolnym przy tak intensywnie wykorzystujących zasoby aplikacjach, jakimi są gry komputerowe. W związku z tym silnik Unity w momencie budowania gry najpierw kompiluje kod *C#*, a następnie na podstawie plików wyjściowych generuje kod *C++* i dokonuje jego kompilacji do plików wykonywalnych (proces przedstawiony na Rys. 20).



Rys. 20. Proces kompilacji aplikacji przez silnik Unity

Źródło: https://docs.unity3d.com/2019.3/Documentation/Manual/IL2CPP-HowItWorks.html

VI.3. Wykorzystane biblioteki zewnętrzne

VI.3.1. XR Interaction Toolkit

XR Interaction Toolkit to biblioteka udostępniająca system interakcji za pomocą sprzętu rzeczywistości wirtualnej z obiektami w grze. Pakiet ten bazuje na mechanizmie komponentów, dzięki czemu pozwala na modularne i elastyczne podejście do tworzenia nowych zachowań. [18]

Podstawowymi elementami tego systemu są:

- Interaktory (ang. *Interactor)*, czyli obiekty, które mogą inicjować interakcje
- Obiekty interaktywne (ang. *Interactable*), czyli obiekty, z którymi można wchodzić w interakcje
- Menadżer interakcji, który spaja funkcjonalności dwóch powyższych elementów

Każdy z obiektów interaktywnych może znajdować się w jednym z trzech stanów:

- Naniesiony (ang. *Hover*), kiedy interaktor znajduje się w zasięgu interakcji z obiektem interaktywnym.
- Wybrany (ang. *Select*) – kiedy interactor wciśnie przycisk interakcji z obiektem w stanie naniesienia, czyli np. chwyci za ten przedmiot.
- Aktywowany (ang. *Activate*) – dodatkowa akcja uruchamiana gdy gracz wciśnie specjalny przycisk trzymając obiekt w stanie wybrania. To może oznaczać np. akcję wystrzelenia z trzymanego pistoletu.

Omawiany pakiet obsługuje zarządzanie tymi stanami dla każdego przedmiotu w grze. Zadaniem twórcy gry jest zaprogramowanie obsługi poszczególnych stanów przedmiotów interaktywnych.

VI.3.2. Zenject

Zenject to biblioteka wzbogacająca Unity o funkcjonalność wstrzykiwania zależności (ang. *Dependency injection*) do obiektów. Pozwala to na zwiększenie modularności poszczególnych elementów projektu i na bezproblemowe pozyskiwanie referencji do pożądanych obiektów. [19]

Zenject działa w oparciu o tzw. konteksty. Kontekst stanowi osobny obiekt, który przechowuje wskazane przez programistę referencje. Wewnętrzny menadżer tych referencji udostępnia obiekty ze wspomnianych kontekstów, gdy jakiś skrypt o nie poprosi. Każdy kontekst ma również określony zasięg: projektu, sceny czy pojedynczego obiektu.

VI.3.3. Odin Inspector and Serializer

Unity jest silnikiem wyposażonym w zaawansowane narzędzia graficzne, które można modyfikować na potrzeby projektu. W tym celu należy utworzyć osobny skrypt odpowiedzialny za generowanie takiego edytora. *Odin Inspector* to biblioteka rozszerzająca wbudowane narzędzia tworzenia edytorów o dodatkowe rodzaje okien, opcje wyświetlania kolekcji. Dużą przewagą tego pakietu nad wbudowanymi mechanizmami jest możliwość modyfikacji edytorów bez potrzeby tworzenia osobnego skryptu.

VI.4. Wykorzystane wzorce projektowe

VI.4.1. Wzorzec komponentu

Wzorzec komponentu stanowi podstawę silnika Unity. Jak zostało wspomniane we wcześniejszym rozdziale, każdy Obiekt Gry stanowi kontener na zbiór komponentów, w związku z czym jest to nieustannie wykorzystywany wzorzec w tym projekcie. Ten wzorzec jednak rzadko pojawia się we fragmentach kodu, ponieważ Unity umożliwia organizację komponentów (które dziedziczą z klasy *MonoBehaviour*) za pomocą interfejsu graficznego.

Celem wzorca komponentu jest umożliwienie pojedynczemu bytowi realizacji zachowań z wielu dziedzin przy jak największej izolacji poszczególnych zachowań. [14]

Jako przykład można wymienić obiekt samochodu. Samochód powinien reagować na ruchy kierownicą, móc się przemieszczać, a także reagować na kolizje czy prawa fizyki. Implementacja wszystkich powyższych elementów wiązałaby się ze stworzeniem bardzo długiego i ciężkiego do zrozumienia skryptu. W związku z tym wspomniany kod można podzielić na „komponenty" sterowania, poruszania, kolizji czy fizyki. Każdy z tych elementów realizuje własne zachowanie komunikując się jedynie z niezbędnymi komponentami.

VI.4.2. Wzorzec skończonej maszyny stanów

Kolejnym wykorzystanym schematem jest wzorzec maszyny stanów. Celem tego wzorca jest jasne rozdzielenie poszczególnych stanów, w których może znajdować się obiekt danej klasy. To pozwala na większą kontrolę nad przejściami pomiędzy poszczególnymi stanami, ułatwia ich inicjalizację i zwalnianie zasobów. Przykładowe maszyny stanów można zobaczyć w rozdziale o projektowaniu.

Wzorzec opiera się na następujących założeniach [14]:

- Skończona liczba stanów, w której maszyna może się znajdować.
- Maszyna nie może być w wielu stanach jednocześnie..
- Dane wejściowe są przesyłane do maszyny.

VI.4.3. Wzorzec pyłku (ang. *Flyweight)*

Ostatnim wzorcem jest wzorzec Pyłku (ang. *Flyweight*). Zaliczany jest on do grupy strukturalnych wzorców projektowych, którego celem jest zmniejszenie wykorzystywanej pamięci przez współdzielenie przez obiekty powtarzających się elementów danych.[1]

Silnik *Unity* pozwala na komfortowe wykorzystanie tego wzorca poprzez tak zwane Obiekty Skryptowalne (ang. *Scriptable Object)*. Są to kontenery danych, które można definiować za pomocą skryptów, następnie tworzyć ich instancje i przypisać referencje do nich w poszczególnych obiektach korzystających z tych danych (zamiast tworzenia osobnej kopii tych danych dla każdego obiektu). [17]

[1] (Wikipedia, 2022)

VI.5. Implementacje wybranych elementów aplikacji

W następnych punktach przedstawiono implementacje wybranych elementów aplikacji wraz z objaśnieniami przedstawionych fragmentów kodu.

Na początek warto omówić interfejs i klasę bazową, po których dziedziczą kluczowe elementy architektury aplikacji, czyli menadżer gry, menadżer rozgrywki oraz kontroler Bohatera. Tymi elementami są interfejs maszyny stanów *IStateMachine* i implementująca go klasa *MonoStateMachine*.

VI.5.1. Interfejs maszyny stanów

Interfejs maszyny stanów zdefiniowany został w sposób następujący:

Kod. 1. Interfejs maszyny stanów

```
1   namespace Utilities
2   {
3       public interface IStateMachine<TStateType>
4       {
5           TStateType ActState { get; }
6
7
8           void OnStateEnter(TStateType state);
9
10          /// <returns>Next state</returns>
11          TStateType OnStateUpdate();
12
13          void OnStateExit(TStateType state);
14      }
15  }
```

Źródło: Badania własne

Jest to interfejs generyczny z typem *TStateType*. Oznacza to, że można go zaimplementować wykorzystując różne, niepowiązane ze sobą typy. W praktyce pozwala to na zdefiniowanie stanów zarówno jako osobnych klas dziedziczących po jakiejś klasie bazowej stanu, jak i określenie ich za pomocą typu wyliczeniowego (ang. *Enum values*).

Interfejs złożony jest z właściwości (ang. *property*) *ActState*, która pozwala na uzyskanie dostępu do aktualnego stanu implementującej maszyny (ale tylko do odczytania jej wartości za pomocą metody pozyskującej (ang. getter)) i trzech metod:

- *OnStateEnter* – jest to metoda wykonywana przy wejściu do nowego, określonego w argumencie *state* stanu.
- *OnstateUpdate* – metoda wykonywana okresowo, np. raz na każdą wyświetloną klatkę. Pozwala np. na zaimplementowanie logiki określającej,

kiedy stan maszyny ulega zmianie, dlatego też zwraca wartość następnego stanu maszyny.

- *OnStateExit* – metoda wywoływana w momencie wyjścia z danego stanu. Może być np. wykorzystana do zwolnienia zasobów i odwrócenia zmian wprowadzonych w *OnStateEnter*.

VI.5.2. Klasa bazowa maszyny stanów

Implementacja klasy *MonoStateMachine* wygląda nastepujaco:

Kod. 2. Klasa *MonoStateMachine*

```
1   public abstract class MonoStateMachine<TStateType>
2       : MonoBehaviour, IStateMachine<TStateType>
3   {
4       [Title("State Machine")]
5
6       [ShowInInspector, ReadOnly]
7       protected TStateType _actState;
8
9       public TStateType ActState => _actState;
10
11
12      protected virtual void Update()
13      {
14          var nextState = OnStateUpdate();
15
16          if (!nextState.Equals(_actState))
17              SetState(nextState);
18      }
19
20
21      public abstract void OnStateEnter(TStateType state);
22
23      public abstract TStateType OnStateUpdate();
24
25      public abstract void OnStateExit(TStateType state);
26
27
28      protected void SetState(TStateType state)
29      {
30          OnStateExit(_actState);
31          OnStateEnter(state);
32
33          _actState = state;
34      }
35  }
```

Źródło: Badania własne

Z deklaracji klasy w linijkach 1-2 można wyczytać, że jest to generyczna klasa abstrakcyjna implementująca interfejs *IStateMachine*. Te zabiegi umożliwiają

dynamiczne definiowanie stanów wedle potrzeb, czy to za pomocą typów wyliczeniowych (ang. *Enum values*) czy jako osobne klasy.

Abstrakcyjność tej klasy wynika z faktu, że metody zadeklarowane w *IStateMachine* tu są zadeklarowane w liniach 21-25 jako abstrakcyjne. Wynika to z faktu, każda klasa dziedzicząca po klasie maszyny stanów powinna zdefiniować własną obsługę stanów.

Właściwość *ActState* jest tu zaimplementowana poprzez odnośnik do strzeżonego (ang. p*rotected*) pola *_actState* klasy *MonoStateMachine.* Wykorzystanie modyfikatora dostępu *protected* oznacza, że bezpośredni dostęp do tej wartości ma jedynie ta klasa i klasy po niej dziedziczące. [12]

MonoStateMachine stanowi implementację wzorca maszyny stanów opartą o wspomnianą wcześniej klasę komponentu w Unity, czyli *MonoBehaviour*. Oznacza to, że wewnątrz klas dziedziczących można zdefiniować określone metody, które Unity będzie wykonywać w ściśle określonych momentach wykonywania programu. W tym przypadku w wierszach 9-19 wykorzystano metodę *Update*, która jest wykonywana w każdej klatce gry. Metoda ta zadeklarowana jest jako metoda wirtualna, co oznacza, że istnieje możliwość zmiany jej działania w klasie dziedziczącej.

Wewnątrz metody *Update* w każdej klatce gry jest wykonywana metoda *OnStateUpdate*. Jeśli stan zwrócony przez tę metodę jest inny niż aktualny stan maszyny, wykonywana jest metoda *SetState*, której zadaniem jest poprawne przełączenie maszyny z jednego stanu na drugi poprzez:

- wyjście z dotychczasowego stanu (*OnStateExit* w linijce 30),
- wejście do nowego stanu (*OnStateEnter* w linijce 31),
- zmianę wartości aktualnego stanu na nowy stan (linijka 33),

VI.5.3. Uruchomienie gry

W języku *C#* pierwszą wykonywaną metodą dla każdej aplikacji jest metoda *main*. Silnik *Unity* jednak wykorzystuje tę metodę do wewnętrznej inicjalizacji i nie pozwala na jej modyfikację. W związku z tym dla zewnętrznego programisty metodą najbliższą tej metodzie jest procedura *Start*. Jest to metoda klasy *MonoBehaviour* wykonywana w momencie instancjonowania i aktywowania nowego Obiektu Gry, który zawiera dany komponent. [17]

Przy uruchomieniu aplikacji podstawową funkcją jest właśnie metoda *Start* menadżera gry:

Kod. 3. Metoda *Start* menadżera gry

```
1   private void Start()
2   {
3       // load environment scene
4       if (!SceneManager.GetSceneByName(_environmentSceneName).IsValid())
5           SceneManager.LoadScene(
6               _environmentSceneName, LoadSceneMode.Additive
7           );
8
9       bool showMainMenu = _gameConfig.Game.ShowMainMenuOnStart;
10
11      GameState initialState = showMainMenu ?
12          GameState.MainMenu : GameState.LoopGameplay;
13
14      SetState(initialState);
15  }
```

Źródło: Badania własne

Cała gra złożona jest z dwóch scen (III.2.1):

- sceny z elementami rozgrywki – Zawiera wszelkie obiekty interaktywne oraz obiekty realizujące pewną logikę
- sceny z elementami środowiska – Zawiera obiekty statyczne i nie wykonujące żadnej logiki, czyli np. modele otoczenia pokroju podłoża, beczek itp.

Obie sceny zawierają wzajemnie uzupełniające się elementy, przez co powinny być załadowane jednocześnie. Silnik Unity w momencie uruchomienia aplikacji pozwala na załadowanie tylko jednej sceny (w tym przypadku sceny z rozgrywką), w związku z czym w linijkach 3-7 menadżer gry sprawdza, czy scena z otoczeniem została załadowana. Jeśli nie, to wymusza jej załadowanie.

W linijce 9 zostaje pobrana wartość flagi logicznej z konfiguracji gry przechowywanej w Obiekcie Skryptowalnym (VI.4.3). Flaga ta określa, czy na początku gry powinno być wyświetlone główne menu, czy gra powinna przejść od razu do rozgrywki.

Klasa *GameManager* jest klasą dziedziczącą po klasie *MonoStateMachine* (VI.5.2) Zależnie od wartości *showMainMenu*, stanem początkowym menadżera gry staje się albo stan wyświetlania menu głównego, albo stan rozgrywki (linijki 11-14). Flaga *showMainMenu* została stworzona głównie na potrzeby testowania. W skompilowanej aplikacji jej wartość zawsze wynosi *true*, więc docelowym stanem początkowym gry jest pokazywanie menu głównego.

Stany menadżera gry zostały zdefiniowane zgodnie z diagramem stanów (Rys. 13) za pomocą typu wyliczeniowego *GameState*:

Kod. 4. Definicja stanów menadżera gry

```
1   public enum GameState
2   {
3       MainMenu = 0,
4       LoopGameplay = 1,
5       PauseMenu = 2,
6       DefeatMenu = 3,
7       WinMenu = 4,
8   }
```

Źródło: Badania własne

Zgodnie z implementacją metody *SetState*, w następnej kolejności wykonana zostanie metoda *OnStateEnter* dla stanu wyświetlania menu głównego. W przypadku menadżera gry stanu gry metody *OnStateEnter, OnStateUpdate i OnSstateExit* wykorzystują instrukcję *switch*, żeby określić zachowania dla różnych stanów. Przykładowo, metoda *OnStateExit* w menadżerze gry wygląda następująco:

Kod. 5. Metoda *OnStateExit* menadżera gry

```
1   public override void OnStateExit(GameState state)
2   {
3       switch (state)
4       {
5           case GameState.LoopGameplay:
6               _playerController.SwitchToInteractor(InteractorType.Ray);
7
8               break;
9
10          case GameState.PauseMenu:
11              _pauseSound.Play();
12
13              break;
14
15          default:
16              break;
17      }
18  }
```

Źródło: Badania własne

W tym przypadku wyjście ze stanu musi zostać obsłużone tylko przy wyjściu ze stanów gry w pętli (linie 5-8) i stanu menu pauzy (10-13). Jeśli jakiś stan nie wymaga obsługi danego stanu, to instrukcja *switch* przechodzi do bloku *default* (15-16), który jedynie wychodzi z instrukcji *switch*.

W ten sposób zaimplementowane są metody *OnStateEnter*, *OnStateUpdate* i *OnStateExit* zarówno w menadżerze gry, jak i menadżerze rozgrywki w pętli i kontrolerze Bohatera. W związku z tym, dla zwiększenia czytelności, fragmenty kodu dla poszczególnych stanów będą pomijały deklarację metody i zawierały jedynie kod dotyczący obsługi danego stanu.

Obsługa wejścia do stanu wyświetlania menu głównego wygląda następująco:

Kod. 6. *OnStateEnter* dla stanu wyświetlania menu głównego menadżera gry

```
1   case GameState.MainMenu:
2       _loopGameplayManager.StopGame();
3
4       _uiManager.ShowMainMenu();
5       _musicManager.PlayMainMenuMusic();
6       break;
```

Źródło: Badania własne

W linii 2 menadżer rozgrywki zostaje wprowadzony w stan zatrzymanej gry. Ten zabieg przywraca wszelkie elementy rozgrywki do stanu początkowego. Dzięki temu do stanu wyświetlania menu głównego można przejść nie tylko przy rozpoczęciu gry, ale również w jej trakcie, np. z poziomu menu pauzy.

W linii 4 menadżer interfejsów użytkownika zmienia aktualnie wyświetlony interfejs na widok głównego menu, a w wierszu 5 menadżer muzyki zaczyna odtwarzać motyw menu głównego.

Metoda *StopGame* menadżera rozgrywki wygląda następująco:

Kod. 7. Metoda *StopGame* menadżera rozgrywki

```
1   public void StopGame()
2   {
3       SetState(LoopGameplayState.GameStopped);
4   }
```

Źródło: Badania własne

Menadżer rozgrywki implementuje maszynę stanów w sposób analogiczny do menadżera gry, w związku z czym metoda *StopGame* stanowi jedynie publicznie dostępną funkcję pozwalającą na zmianę jego stanu na stan zatrzymanej gry. Obsługa wejścia w ten stan natomiast wygląda następująco:

Kod. 8. *OnStateEnter* dla stanu zatrzymanej gry menadżera rozgrywki

```
1   case LoopGameplayState.GameStopped:
2       StopGameInternal();
3       break;
```

Źródło: Badania własne

Kod. 9. Metoda *StopGameInternal* menadżera rozgrywki

```
1   private void StopGameInternal()
2   {
3       _hero.gameObject.SetActive(false);
4       _roomSpawner.DespawnAll();
5
6       _loopGrid.ResetToInitialState();
7   }
```

Źródło: Badania własne

W momencie zatrzymania gry menadżer rozgrywki dezaktywuje obiekt Bohatera, co powoduje jego zniknięcie ze świata gry oraz zatrzymanie wykonywania metod *Update* komponentów przypisanych do jego Obiektu Gry.

_roomSpawner reprezentuje wspomniany w opisie gry generator pokoi, czyli obiekt odpowiadający za instancjonowanie kolejnych, wybieranych losowo pomieszczeń, które Gracz może podnieść i umieścić na siatce. Metoda *DespawnAll* ma za zadanie zniszczyć wszystkie aktywne w danym momencie pokoje znajdujące się na tej „szynie”.

Kod. 10. Metoda *DespawnAll* klasy *RoomSpawner*

```
1   public void DespawnAll()
2   {
3       foreach (var room in _spawnedRooms)
4           Destroy(room.gameObject);
5
6       _spawnedRooms.Clear();
7   }
```

Źródło: Badania własne

DepawnAll iteruje po prywatnej liście istniejących pokoi i niszczy każdy znajdujący się w niej obiekt. Przez niszczenie rozumiane jest usunięcie obiektu z hierarchii na scenie i zwolnienie zajmowanej przez niego pamięci (linijki 3-4). Następnie w linii 6 lista zostaje wyczyszczona.

_loopgrid to obiekt klasy *LoopGrid*, którego zadaniem jest zarządzanie siatką pokoi. *ResetToInitialState* to metoda niszcząca wszystkie pokoje znajdujące się na siatce.

Kod. 11. Metoda *ResetToInitialState* klasy *LoopGrid*

```
1   public void ResetToInitialState()
2   {
3       // clear all rooms
4       foreach (var socket in _gridElements)
5       {
6           if (socket != null)
7               socket.SetRoom(null);
8       }
9
10      _roomsPlaced = 0;
11  }
```

Źródło: Badania własne

Metoda *ResetToInitialState* iteruje po wszystkich gniazdach (ang. *socket*) znajdujących się na siatce w dwuwymiarowej, strzeżonej tablicy *_gridElements* (linia 4). Jeśli dany element tej tablicy nie przyjmuje wartości *null*, to pokój znajdujący się wewnątrz tego gniazda jest ustawiany na *null* (linijki 6-7). Następnie wartość prywatnego pola przechowującego liczbę pokoi ustawionych na siatce zostaje ustawiona na 0. (linia 10).

Metoda *ShowMainMenu* menadżera interfejsów użytkownika wygląda następująco:

Kod. 12. Metoda *ShowMainMenu* menadżera interfejsów użytkownika

```
1   public void ShowMainMenu() => ShowMenu(_mainMenu);
2
3
4   private void ShowMenu(UIMenu menu)
5   {
6       foreach (var actMenu in _menus)
7       {
8           if (!Utils.AssertNotNull(actMenu, this))
9               continue;
10
11          if (actMenu == menu)
12              actMenu.Show();
13          else
14              actMenu.Hide();
15      }
16  }
```

Źródło: Badania własne

Metoda *ShowMainMenu* odwołuje się do prywatnej metody *ShowMenu*, która przyjmuje jako argument obiekt klasy *UIMenu*. Metoda ta iteruje po wszystkich menu, do których posiada referencję w prywatnej liście *_menus*. Jeśli referencja do danego menu ma wartość *null*, to zostaje pominięta (linijki 8-9). Jeśli *actMenu* wskazuje na ten sam obiekt co argument *menu*, to interfejs zostaje pokazany, w przeciwnym wypadku zostaje ukryty (linie 11-14).

VI.5.4. Generowanie pokoi

Żeby ustawić pokój na siatce, w pierwszej kolejności pokój ten musi zostać stworzony przez generator pokoi. W tym punkcie zostanie opisany proces generowania takiego pomieszczenia, które Gracz może potem chwycić w świecie gry.

Za cały proces odpowiada klasa *RoomSpawner*, a punktem wyjściowym całego procesu jest metoda *EvaluateSpawner*.

Kod. 13. Metoda *EvaluateSpawner* klasy *RoomSpawner*

```
1   public void EvaluateSpawner()
2   {
3       if (_config.Rooms.Count == 0)
4           return;
5
6       if (!EvaluateRoomSpawnChance())
7           return;
8
9       if (!TryToChooseRoomToSpawn(out var room))
10      {
11          Debug.Log("Failed to choose a room to spawn.", this);
12          return;
13      }
14
15      SpawnRoom(room.Room);
16      UpdateWeights(room);
17
18      MoveSpawnedRoomsToRightPositions();
19  }
```

Źródło: Badania własne

Metoda ta najpierw sprawdza, czy pula dostępnych pokoi w pliku konfiguracji gry nie jest pusta (linijki 3-4). Jeśli tak, to działanie funkcji zostaje zakończone. Następnie ewaluowana jest szansa na wygenerowanie pokoju (linie 6-7). Jeśli wynik okaże się negatywny, to metoda kończy działanie. W linii 9 zostaje podjęta próba wybrania pokoju, który ma zostać stworzony. Jeśli to zakończy się niepowodzeniem, metoda wypisuje komunikat o błędzie na konsolę i kończy działanie (linie 11-12). Następnie pokój zostaje wygenerowany (linia 15), a pozycje pokoi w świecie gry zostają zaktualizowane (linia 18). W linii 16 następuje kalkulacja wag poszczególnych pokoi, co zostanie zaraz bardziej szczegółowo wyjaśnione.

Kod. 14. Metoda *EvaluateRoomSpawnChance* klasy *RoomSpawner*

```
1   private bool EvaluateRoomSpawnChance()
2   {
3       return _config.RoomSpawnChance >= UnityEngine.Random.Range(0f, 1f);
4   }
```

Źródło: Badania własne

Metoda *EvaluateRoomSpawnchance* zwraca wynik testu na prawdopodobieństwo. Jeśli wylosowana liczba z przedziału [0, 1] jest mniejsza lub równa wartości *RoomSpawnChance* z pliku konfiguracji gry, to funkcja zwraca wartość *true*. W przeciwnym wypadku zwraca wartość *false*.

Kod. 15. Metoda *TrytoChooseRoomToSpawn* klasy *RoomSpawner*

```
1   private bool TryToChooseRoomToSpawn(out RoomSpawnSettings chosenRoom)
2   {
3       float totalWeight = CalculateTotalWeight();
4       float randomValue = UnityEngine.Random.Range(0f, totalWeight);
5
6       foreach (var room in _actRoomWeights)
7       {
8           randomValue -= room.Weight;
9           if (randomValue <= 0f)
10          {
11              chosenRoom = room;
12              return true;
13          }
14      }
15
16      chosenRoom = default;
17      return false;
18  }
```

Źródło: Badania własne

Metoda *TryToChooseRoomToSpawn* wykorzystuje prawdopodobieństwo ważone, w którym do każdego obiektu przypisywana jest pewna waga. Żeby wybrać losowy obiekt z tego przedziału, należy najpierw zsumować wagi wszystkich obiektów (linia 3), a następnie wylosować wartość z przedziału [0, suma wag] (linia 4). Kolejnym krokiem jest iteracja po kolejnych elementach i odejmowanie od wylosowanej wartości wagi aktualnie rozpatrywanego elementu (linie 6-8). Jeśli po odjęciu wartość wylosowanej liczby jest mniejsza lub równa 0, aktualnie rozpatrywany element jest zwracany jako ten wylosowany (linie 9-13). Jeśli po rozpatrzeniu wszystkich pokoi wylosowana liczba nadal jest większa od 0, to żaden pokój nie jest zwracany (linie 16-17).

Kod. 16. Metoda *SpawnRoom* klasy *RoomSpawner*

```
1   private void SpawnRoom(Room room)
2   {
3       var spawnedRoom = _container
4           .InstantiatePrefabForComponent<Room>(room,
5               _newRoomPositionT.position, _newRoomPositionT.rotation,
6               _spawnedRoomsParentT
7           );
8
9       _spawnedRooms.AddFirst(spawnedRoom);
10
11      if (_spawnedRooms.Count > _config.MaxRoomsOnTheSpawner)
```

```
12      {
13          var lastRoom = _spawnedRooms.Last.Value;
14          _spawnedRooms.RemoveLast();
15          Destroy(lastRoom.gameObject);
16      }
17  }
```

Źródło: Badania własne

Metoda *SpawnRoom* w pierwszej kolejności tworzy nową instancję przekazanego w argumencie pokoju (linie 3-6). Wykorzystywany jest do tego obiekt *_container*, który jest obiektem klasy *DiContainer* z biblioteki Zenject. Używanie tego obiektu zamiast domyślnej funkcji *Instantiate* stosowanej w Unity zapewnia, że wszelkie zależności komponentów przypisanych do utworzonego pokoju zostaną wstrzyknięte (ang. *injected*). Nowo utworzony obiekt jest ustawiony w pozycji i rotacji określonej przez obiekt *_newRoomPositionT*, a rodzic tego obiektu w hierarchii sceny zostaje ustawiony na obiekt *_spawnedRoomParentT*.

W linii 9 obiekt zostaje dodany do prywatnej listy *_spawnedRooms*. Jeśli liczba pokoi udostępnianych Graczowi przez generator przekracza maksymalny limit, ostatni pokój z listy zostaje zniszczony (linie 11-16).

Kod. 17. Metoda *MoveSpawnedRoomstoRightPositions* klasy *RoomSpawner*

```
1   private void MoveSpawnedRoomsToRightPositions()
2   {
3       Vector3 shiftDirection =
4           _lastRoomPositionT.position - _firstRoomPositionT.position;
5
6       var actNode = _spawnedRooms.First;
7       int i = 0;
8
9       while (actNode != null)
10      {
11          var room = actNode.Value;
12
13          room.transform.position =
14              _firstRoomPositionT.position + i * shiftDirection
15              / _config.MaxRoomsOnTheSpawner;
16
17          actNode = actNode.Next;
18          i++;
19      }
20  }
```

Źródło: Badania własne

Na początku metody *MoveSpawnedRoomstoRightPositions* wyznaczany jest wektor przesunięcia między pozycją pierwszego pokoju na „szynie”, a ostatniego (linie 3-4). Następnie inicjalizowane są wartości wykorzystywane do iterowania po liście *_spawnedRooms* (linie 6-7), po czym metoda iteruje po elementach listy i przesuwa

każdy na pozycję między pierwszym a ostatnim elementem, ustawione w równych odstępach od siebie (linie 13-15).

Po wykonaniu powyższych metod nowo utworzony pokój powinien znaleźć się w odpowiedniej pozycji w świecie wirtualnym, a pozostałe dostępne dla Gracza pokoje powinny zostać odpowiednio przesunięte.

VI.5.5. Podniesienie i ustawienie pokoju na siatce

Za śledzenie ruchów kontrolera i zarządzanie przebiegiem interakcji odpowiada przede wszystkim biblioteka *XR Interaction Toolkit*. (VI.3.1) Zadaniem programisty jest zaprogramowanie zachowań wykonywanych w różnych momentach interakcji oraz ustawienie parametrów interakcji takich jak warstwy interakcji określające jakie obiekty mogą ze sobą wchodzić w te interakcje.

Ustawienie komponentów kontrolera gracza wykonywane jest głównie za pomocą interfejsu graficznego Unity (Rys. 21).

Rys. 21. Obiekt Gry gracza w hierarchii sceny

Źródło: Badania własne

Obiekt Gracza ma również przypisany osobny komponent *PlayerController*, który pozwala na zarządzanie interaktorami oraz zawiera metody łączące *Interaction Toolkit* z resztą gry.

W momencie najechania wirtualną ręką na jakiś obiekt interaktywny wywoływana jest metoda *OnHoverEnter:*

Kod. 18. Metoda *OnHoverEnter* klasy *PlayerController*

```
1   public void OnHoverEnter(HoverEnterEventArgs eventArgs)
2   {
3       var selectedObjectT = eventArgs.interactableObject.transform;
4
5       if (selectedObjectT.TryGetComponent<Room>(out var room))
6           room.OnPlayerHoverEnter();
7   }
```

Źródło: Badania własne

W metodzie *OnHoverEnter* następuje sprawdzenie, czy obiektem interaktywnym jest obiekt klasy Pokój. Jeśli tak, to wywoływana jest jego metoda *OnPlayerHoverEnter*.

Kod. 19. Metoda *OnPlayerHoverEnter* klasy *Room*

```
1   public void OnPlayerHoverEnter()
2   {
3       if (_roomInfo == null)
4           return;
5
6       _roomInfo.Show();
7   }
```

Źródło: Badania własne

W metodzie *OnPlayerHoverEnter* pokazywany jest panel zawierający informacje o tym pokoju. Panel ten jest zawieszony nad pokojem w świecie gry i pozwala na zdobycie informacji na temat tego pokoju.

Gdy Gracz wejdzie w interakcje z pokojem w zasięgu jego ręki i go podniesie, wywoływana jest metoda *OnSelectEnter:*

Kod. 20. Metoda *OnSelectEnter* klasy *PlayerController*

```
1   private void OnSelectEnter(SelectEnterEventArgs eventArgs)
2   {
3       var selectedObjectT = eventArgs.interactableObject.transform;
4
5       if (selectedObjectT.TryGetComponent<Room>(out var room))
6       {
7           room.OnPlayerHoverExit();
8           OnRoomPickedUp?.Invoke(room);
9       }
10  }
```

Źródło: Badania własne

Podobnie jak w *OnHoverEnter*, w tej metodzie sprawdzane jest, czy obiektem interaktywnym jest obiekt klasy *Room*. Jeśli tak, to wywoływana jest jego metoda *OnHoverExit*, a następnie wywoływane jest zdarzenie (ang. *event*) *OnRoomPickedUp.*

Kod. 21. Metoda *OnPlayerHoverExit* klasy *Room*

```
1   public void OnPlayerHoverExit()
2   {
3       if (_roomInfo == null)
4           return;
5
6       _roomInfo.Hide();
7   }
```

Źródło: Badania własne

Zadaniem metody *OnPlayerHoverExit* jest odwrócenie zmian wprowadzonych w *OnPlayerHoverEnter*, w związku z czym metoda ta ukrywa panel informacyjny przypisany do tego pokoju.

Kiedy Gracz puści pokój w pobliżu gniazda na siatce, które nie zawiera wcześniej ułożonego pokoju, to wywołana zostanie metoda *OnSelectEnter* klasy *GridSocket* będącej komponentem przypisanym do każdego gniazda na siatce.

Kod. 22. Metoda *OnSelectEnter* klasy *GridSocket*

```
1   public void OnSelectEnter(SelectEnterEventArgs eventArgs)
2   {
3       var interactableRigidbody =
4       eventArgs.interactableObject.colliders[0].attachedRigidbody;
5
6       if (interactableRigidbody == null
7       || !interactableRigidbody.TryGetComponent<Room>(out var room))
8           return;
9
10      SetRoom(room);
11  }
12
13
14  [Button]
15  public void SetRoom(Room newRoom)
16  {
17      if (_room != null)
18          Destroy(_room.gameObject);
19
20      if (newRoom == null)
21      {
22          _socketInteractor.enabled = true;
23          return;
24      }
25
26      newRoom.transform.SetParent(this.transform);
27      newRoom.transform.ResetLocalCoordinates();
28
29      newRoom.OnRoomPlacedInSocket(this);
```

```
30      _room = newRoom;
31
32      _loopGrid.OnRoomPlaced(this);
33
34      _socketInteractor.enabled = false;
35  }
```

Źródło: Badania własne

Metoda *OnSelectEnter* w liniach 3-7 sprawdza, czy obiekt, z którym to gniazdo weszło w interakcję posiada komponent typu *Room* tj. czy jest pokojem. Jeśli nie, to w linii 8 kończone jest działanie tej metody. W przeciwnym wypadku wywoływana jest metoda *SetRoom* (linijka 10).

Metoda *SetRoom* niszczy poprzednio przypisany pokój, jeśli takowy istnieje (linie 17-18). Jeśli pokój przekazany w argumencie przyjmuje wartość *null*, to oznacza to, że to gniazdo zostało zwolnione, w związku z czym komponent *_socketInteractor*, który jest częścią *XR Interaction Toolkit* i odpowiada za zarządzanie interakcjami z tym gniazdem zostaje włączony (linie 20-23), a działanie metody zostaje zakończone.

Jeśli argument *newRoom* nie przyjmuje wartości *null*, to jego rodzic w hierarchii sceny jest ustawiany na dane gniazdo, a pozycja i rotacja przypisanego pokoju przyjmują takie same wartości, jak pozycja i rotacja tego gniazda (linie 26-27). Następnie wywoływane są metody *OnRoomPlacedInSocket* ustawionego pokoju oraz *OnRoomPlaced obiektu _loopGrid*, który jest kontrolerem siatki. Na koniec interakcje z tym gniazdem zostają wyłączone (linia 34).

Kod. 23. Metoda *OnRoomPlacedInSocket* klasy *Room*

```
1   public void OnRoomPlacedInSocket(GridSocket socket)
2   {
3       _xrInteractable.enabled = false;
4       _rigidbody.isKinematic = true;
5       Init();
6   }
```

Źródło: Badania własne

W metodzie *OnRoomPlacedInSocket* wyłączane są interakcje z tym pokojem (linia 3), wyłączany jest wpływ fizyki na ten pokój (linia 4), a także wywoływana jest funkcja *Init*, która aktywuje efekty tego w momencie ustawienia go na siatce.

Kod. 24. Metoda *Init* klasy *Room*

```
1   private void Init()
2   {
3       ApplyToAllEffects(effect => effect.OnInit(this));
4
5       _loopManager.OnNextDayEvent += OnNextDay;
6   }
7
```

```
8
9   private void ApplyToAllEffects(Action<EffectBase> Method)
10  {
11      foreach (var effect in _effects)
12      {
13          if (!Utils.AssertNotNull(effect, this))
14              continue;
15
16          Method(effect);
17      }
18  }
```

Źródło: Badania własne

Wewnątrz metody *Init* w pierwszej kolejności wywoływana jest metoda *ApplyToAllEffects*, która przyjmuje jako argument inną metodę, która jest aplikowana do każdego efektu przypisanego do tego pokoju (linie 9-18).

Rozdział VII. Testowanie

VII.1. Scenariusze i przypadki testowe

Poniżej przedstawiono scenariusze testowe dla wybranych elementów aplikacji.

VII.1.1. Ustawianie pokoju na siatce

Opis: Sprawdzenie poprawności procesu ustawiania pokoju na siatce.
Czynności przygotowawcze:
- Poprawna inicjalizacja rozgrywki.
- Utworzenie nowej instancji pokoju przez generator pokoi.

Czynności końcowe:
- Zakończenie rozgrywki

Powyższy scenariusz testowy wiąże się z następującymi przypadkami testowymi:

Tab. 9. Przypadki testowe ustawiania pokoju na siatce

ID	Nazwa	Kroki wykonania	Oczekiwany rezultat
1	Poprawne wstawienie pokoju	1. Chwycenie Pokoju przez Gracza. 2. Wstawienie Pokoju do gniazda.	Pokój został przypisany do gniazda. Znacznik postępu w grze zostaje zaktualizowany.
2	Wstawienie pokoju do już zajętego gniazda	1. Chwycenie pokoju przez Gracza. 2. Wstawienie pokoju do gniazda. 3. Powtórzenie czynności dla następnego pokoju.	Pierwszy pokój zostaje ustawiony poprawnie. W przypadku następnego pokoju interakcja nie powinna być możliwa.
3	Próba aktywowania	1. Gracz próbuje aktywować gniazdo nie trzymając pokoju w wirtualnej ręce.	Próba interakcji powinna zostać zignorowana.

	gniazda bez trzymanego pokoju.		
4	Zwolnienie gniazda przy nowej rozgrywce	1. Chwycenie Pokoju przez Gracza. 2. Wstawienie Pokoju do gniazda. 3. Wyjście do menu głównego. 4. Rozpoczęcie nowej rozgrywki.	Pokój znajdujący się w gnieździe powinien zostać zniszczony.
5	Wstawienie pokoju do gniazda przy nowej rozgrywce	1. Chwycenie Pokoju przez Gracza. 2. Wstawienie Pokoju do gniazda. 3. Wyjście do menu głównego. 4. Rozpoczęcie nowej rozgrywki. 5. Powtórzenie punktów 1 i 2 dla nowego pokoju.	Pierwszy ustawiony pokój został zniszczony, a nowy pokój został przypisany do gniazda.

Źródło: Badania własne

VII.1.2. Uruchomienie rozgrywki

Opis: Sprawdzenie poprawności procesu uruchomienia rozgrywki
Czynności przygotowawcze:
- Uruchomienie gry.

Czynności końcowe:
- Zakończenie rozgrywki i wyjście z aplikacji

Powyższy scenariusz testowy wiąże się z następującymi przypadkami testowymi:

Tab. 10. Przypadki testowe rozpoczęcia nowej rozgrywki

ID	Nazwa	Kroki wykonania	Oczekiwany rezultat
1	Rozpoczęcie nowej rozgrywki z poziomu menu głównego	1. Gracz wybiera przycisk rozpoczęcia nowej rozgrywki za pomocą interakcji wskaźnikowej.	Nowa rozgrywka została rozpoczęta.

2	Rozpoczęcie z poziomu widoku porażki	1. Gracz rozpoczyna nową rozgrywkę. 2. Gracz ustawia pokoje z przeciwnikami tak, żeby poziom punktów życia Gracza spadł do 0. 3. Gracz wybiera w widoku porażki rozpoczęcie nowej gry.	Nowa rozgrywka została rozpoczęta.

Źródło: Badania własne

W następnych podrozdziałach zostaną przedstawione kolejne testy zaczynając od testów najniższego poziomu, czyli jednostkowych, a na testach funkcjonalnych kończąc.

VII.2. Testy jednostkowe

VII.2.1. Maszyna stanów *MonoStateMachine*

Na potrzeby testów jednostkowych abstrakcyjnej klasy *MonoStateMachine* została zdefiniowana klasa *StateMachineMock*:

Kod. 25. Kod klasy *StateMachineMock*

```
1   public enum MockMachineStates { A, B }
2
3
4   public class StateMachineMock : MonoStateMachine<MockMachineStates>
5   {
6       public override void OnStateEnter(MockMachineStates state)
7       {
8           Debug.Log($"Entering state {state}");
9       }
10
11      public override void OnStateExit(MockMachineStates state)
12      {
13          Debug.Log($"Exitting state {state}");
14      }
15
16      public override MockMachineStates OnStateUpdate()
17      {
18          return ActState;
19      }
20
21      public void ChangeStateTo(MockMachineStates newState)
22      {
23          SetState(newState);
24      }
25  }
```

Źródło: Badania własne

Klasa ta implementuje 2 stany: A i B. Przy wejściu i wyjściu ze stanu metody wypisują komunikaty o tych przejściach informujące. Udostępnia również publiczną metodę *ChangeStateTo*, za pomocą której można przełączać aktualny stan z zewnątrz.

Pierwszym testem dla tej klasy jest zmiany wartości aktualnego stanu:

Kod. 26. Testy przełączania stanu klasy *MonoStateMachine*

```
1   [UnityTest]
2   public IEnumerator ChangingState()
3   {
4       var stateMachine = new GameObject()
5           .AddComponent<StateMachineMock>();
6
7       stateMachine.ChangeStateTo(MockMachineStates.A);
8
9       // Use the Assert class to test conditions.
10      // Use yield to skip a frame.
11      yield return null;
12
13      Assert.AreEqual(stateMachine.ActState, MockMachineStates.A);
14  }
```

Źródło: Badania własne

Test sprawdza, czy wywołanie metody *SetState* klasy *MonoStateMachine* zmienia wartość aktualnego stanu. W związku z tym tworzona jest instancja nowej maszyny stanów, stan jest zmieniany, a następnie test odczekuje jedną klatkę wyświetlanej aplikacji i sprawdza, czy aktualny stan zgadza się ze stanem oczekiwanym (w tym przypadku stanem A).

Następny test sprawdza, czy zmiana stanu poprawnie wywołuje metody *OnStateEnter i OnStateExit:*

Kod. 27. Test wywołania funkcji klasy *MonoStateMachine*

```
1   [UnityTest]
2   public IEnumerator TransitionCheck()
3   {
4       var stateMachine = new
GameObject().AddComponent<StateMachineMock>();
5
6       stateMachine.ChangeStateTo(MockMachineStates.A);
7
8       yield return null;
9
10      LogAssert.Expect(
11          LogType.Log, $"Exitting state {MockMachineStates.A}");
12      LogAssert.Expect(
13          LogType.Log, $"Entering state {MockMachineStates.B}");
14
15      stateMachine.ChangeStateTo(MockMachineStates.B);
16
17      Assert.AreEqual(stateMachine.ActState, MockMachineStates.B);
18  }
```

Źródło: Badania własne

Test tworzy nową instancję maszyny stanów. Następnie wywołuje zmianę stanu, odczekuje jedną klatkę i sprawdza, czy w konsoli zostały wypisane odpowiednie wiadomości.

VII.2.2. Kontroler punktów życia *HealthController*

Następną testowaną klasą jest kontroler życia *HealthController*, którego zadaniem jest zarządzanie punktami życia każdej istoty i informowanie pozostałych o śmierci tej istoty. Pierwszy test sprawdza poprawne działanie procesu otrzymywania obrażeń:

Kod. 28. Test otrzymywania obrażeń klasy *HealthController*

```
1   [Test]
2   public void ReceivingDamage()
3   {
4       var testEntitySettings =
5           ScriptableObject.CreateInstance<EntitySettingsSO>();
6       testEntitySettings.MaxHealth = 3;
7
8       var healthController =
9           new GameObject().AddComponent<HealthController>();
10      healthController.Initialize(testEntitySettings);
11
12      int damage = 1;
13      int expectedHP = 2;
14
15      healthController.DealDamage(damage);
16
17      Assert.AreEqual(expectedHP, healthController.ActHealth);
18  }
```

Źródło: Badania własne

Powyższy test tworzy kontroler życia, w którym maksymalna wartość punktów życia wynosi 3. Następnie wywoływana jest metoda otrzymywania obrażeń z parametrem 1 punktu obrażeń. Na końcu sprawdzane jest, czy aktualny stan punktów życia zgadza się ze stanem oczekiwanym.

Zadaniem kolejnego testu jest sprawdzenie, czy kontroler życia prawidłowo ustawia wartość flagi określającej, czy istota ta posiada jeszcze punkty życia:

Kod. 29. Test flagi stanu śmierci w *HealthController*

```
1   [Test]
2   public void Dying()
3   {
4       var testEntitySettings =
5           ScriptableObject.CreateInstance<EntitySettingsSO>();
6       testEntitySettings.MaxHealth = 3;
```

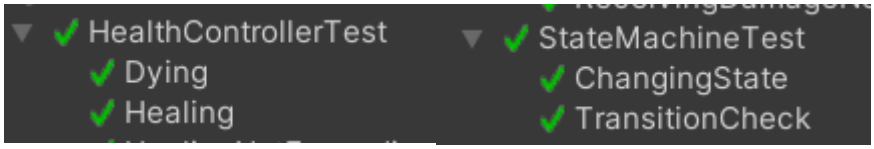
```
7
8       var healthController =
9           new GameObject().AddComponent<HealthController>();
10      healthController.Initialize(testEntitySettings);
11
12      int damage = 100;
13
14      healthController.DealDamage(damage);
15
16      Assert.True(healthController.IsDead);
17  }
```

Źródło: Badania własne

Powyższy test, podobnie jak wcześniejszy tworzy kontroler życia, w którym maksymalna wartość punktów życia wynosi 3. Następnie wywoływana jest metoda zadawania obrażeń z parametrem 100 punktów obrażeń. Na końcu sprawdzane jest, czy wartość flagi *IsDead*, która to określa, czy postać jest martwa, jest ustawiona na *True*.

VII.2.3. Wyniki testów jednostkowych

Powyższe testy po kilku iteracjach testowania i wprowadzania poprawek zwróciły wyniki pozytywne:



Rys. 22. Wyniki testów jednostkowych

Źródło: Badania własne

VII.3. Testy integracyjne

VII.3.1. Wstawianie pokoju do gniazda - testy *GridSocket* i *Room*

Pierwszy test sprawdza, czy przypisanie pokoju do pustego gniazda przebiega poprawnie:

Kod. 30. Test wstawiania pokoju do pustego gniazda klasy *GridSocket*

```
1   [UnityTest]
2   public IEnumerator InsertingRoomIntoSocket()
3   {
4       PreInstall();
5
6       // Call Container.Bind methods
7       Container.Bind<LoopGameplayManager>().FromInstance(null);
8       Container.Bind<LoopGrid>().FromInstance(null);
9
10      LogAssert.ignoreFailingMessages = true;
11
12      var testRoom = Container
13          .InstantiateComponentOnNewGameObject<Room>();
14      var testSocket = Container
15          .InstantiateComponentOnNewGameObject<GridSocket>();
16
17      PostInstall();
18
19      testSocket.SetRoom(testRoom);
20
21      yield return null;
22
23      Assert.AreEqual(testSocket.Room, testRoom);
24
25      yield break;
26  }
```

Źródło: Badania własne

Test na początek ustawia wstrzykiwane referencje do klas *LoopGameplayManager* i *LoopGrid* na wartości *null*. Następnie tworzy nowe instacje pokoju i gniazda jako odpowiednio *testRoom* i *testSocket. testRoom* jest następnie wstawiany do gniazda za pomocą metody *SetRoom* klasy *GridSocket*. Na koniec sprawdzane jest, czy pokój przypisany do testowego gniazda jest sprawdzany pokojem testowym.

Następny test sprawdza poprawne działanie zwolnienia gniazda i zniszczenia pokoju, który aktualnie się w nim znajdował:

Kod. 31. Test zwalniania gniazda klasy *GridSocket*

```
1   [UnityTest]
2   public IEnumerator InsertingNullInPlaceOfRoom()
3   {
4       PreInstall();
5
6       // Call Container.Bind methods
7       Container.Bind<LoopGameplayManager>().FromInstance(null);
8       Container.Bind<LoopGrid>().FromInstance(null);
9
10      LogAssert.ignoreFailingMessages = true;
11
12      var testSocket = Container
13          .InstantiateComponentOnNewGameObject<GridSocket>();
14      var testRoom = Container
15          .InstantiateComponentOnNewGameObject<Room>();
16
17      PostInstall();
18
19      testSocket.SetRoom(testRoom);
20
21      yield return null;
22
23      Assert.AreEqual(testSocket.Room, testRoom);
24
25      testSocket.SetRoom(null);
26
27      yield return null;
28
29      Assert.True(testRoom == null);
30      Assert.True(testSocket.Room == null);
31
32      yield break;
33  }
```

Źródło: Badania własne

Test, podobnie jak poprzedni, inicjuje elementy i wstrzykiwane zależności. Następnie tworzona jest testowa instancja pokoju oraz gniazda. Testowa instancja pokoju jest wstawiana do gniazda. Następnie program wykonuje sprawdzenie, czy pokój znajdujący się w gnieździe to pokój testowy, po czym ustawia pokój na wartość *null*. To w założeniu ma zniszczyć obecnie przypisany pokój i zwolnić gniazdo tj. przywrócić je do stanu, w którym można włożyć do niego kolejny pokój.

VII.3.2. Inicjalizacja elementów sceny rozgrywki

Ostatnie z omawianych testów integracyjnych sprawdzają, czy gra jest poprawnie inicjalizowana tj. czy podczas ładowania poszczególnych scen nie występują żadne błędy:

Kod. 32. Test inicjalizacji sceny rozgrywki

```
1   [UnityTest]
2   public IEnumerator GameplaySceneStartup()
3   {
4       yield return LoadScene(SceneNames.Gameplay);
5
6       yield return new WaitForSeconds(3.0f);
7   }
```

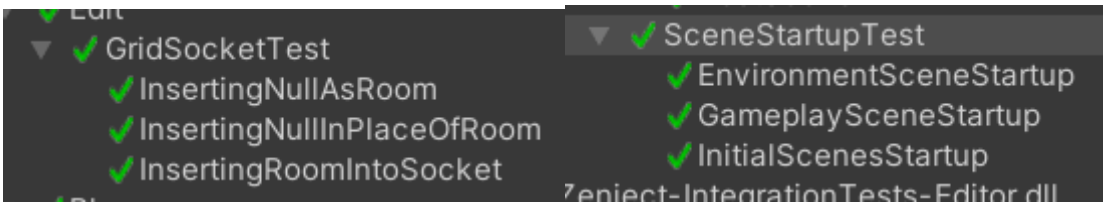
Źródło: Badania własne

Powyższy test ładuje scenę rozgrywki, a następnie odczekuje 3 sekundy. Każdy test domyślnie kończy się niepowodzeniem, gdy w trakcie jego przeprowadzania zostanie zgłoszony jakikolwiek błąd. Ładowanie sceny automatycznie inicjalizuje obiekty znajdujące się na tej scenie, w związku z czym cała gra jest wtedy automatycznie inicjalizowana. Test musi jedynie poczekać na zakończenie procesu inicjalizacji i sprawdzać, czy pojawiły się jakieś błędy.

Analogiczne testy należy wykonać dla sceny z otoczeniem, jak i załadowania obu scen.

VII.3.3. Wyniki testów integracyjnych

Powyższe testy również po kilku iteracjach testowania i wprowadzania niezbędnych poprawek zwróciły wyniki pozytywne:



Rys. 23. Wyniki testów integracyjnych

Źródło: Badania własne

VII.4. Testy czarnej skrzynki na podstawie scenariuszy testowych

VII.4.1. Ustawianie pokoju na siatce

Po rozpoczęciu rozgrywki Gracz może podnieść pokoje z generatora pokoi:

Rys. 24. Szyna generująca pokoje z początkowymi pokojami

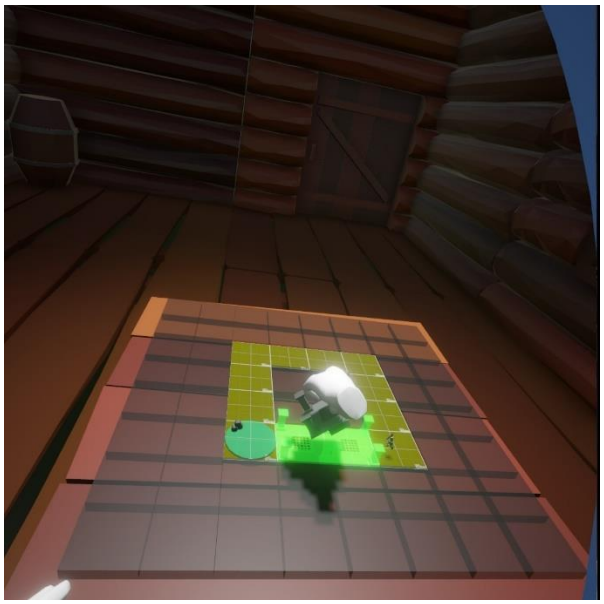
Źródło: Badania własne

Po zbliżeniu wirtualnej ręki do pokoju wyświetlana zostaje informacja o rodzaju pokoju:

Rys. 25. Widok informacji o pokoju

Źródło: Badania własne

Następnie Gracz chwyta pokój i przystawia rękę trzymającą pokój do wolnego gniazda. Gniazda, w które można wstawić pokój danego rodzaju podświetlają się:



Rys. 26. Przystawienie pokoju do wolnego gniazda

Źródło: Badania własne

Warto zaznaczyć, że jest to pokój, który można ustawić jedynie na ścieżce Bohatera (zielone platformy). Tego pokoju nie można ustawić poza ścieżką, w związku z czym pozostałe, czarne platformy nie reagują na zbliżenie tego pokoju:

Rys. 27. Przystawienie pokoju nie niekompatybilnego gniazda

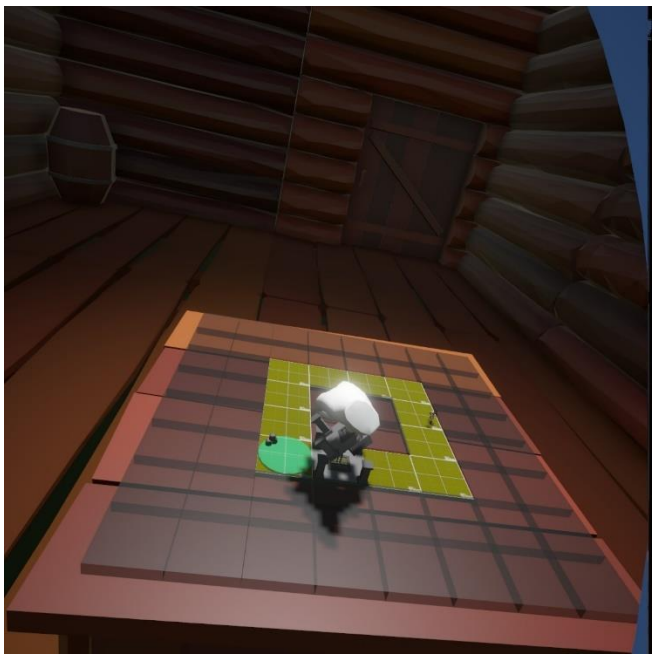
Źródło: Badania własne

Następnie, gdy przycisk aktywacji pokoju zostaje zwolniony, trzymany pokój zostaje ustawiony w wolnym gnieździe:

Rys. 28. Wstawienie pokoju do gniazda

Źródło: Badania własne

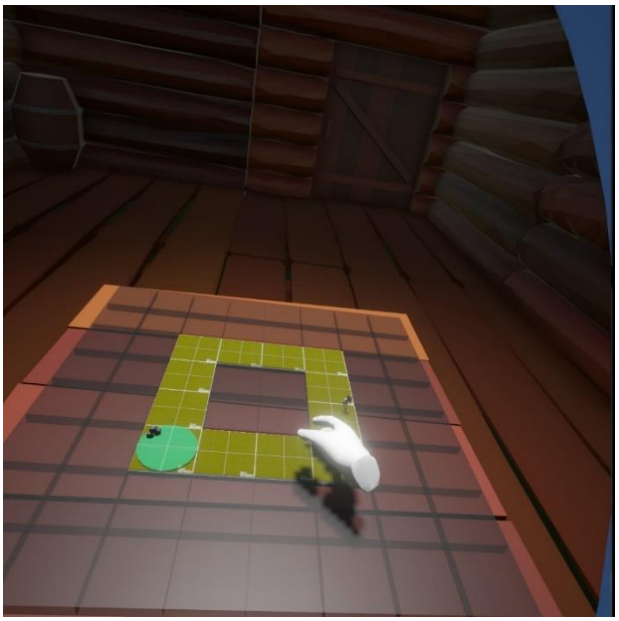
W sytuacji, w której gracz próbuje wejść w interakcję z tym gniazdem trzymając kolejny pokój, interakcja zostaje zignorowana:



Rys. 29. Próba ustawienia pokoju w zajętym gnieździe

Źródło: Badania własne

W przypadku pustego gniazda, jeśli Gracz próbuje dokonać interakcji, to ta również zostaje zignorowana:



Rys. 30. Próba aktywowania gniazda bez trzymanego pokoju

Źródło: Badania własne

VII.4.2. Rozpoczęcie rozgrywki

Po uruchomieniu gry wyświetlony zostaje widok menu głównego:

Rys. 31. Widok menu głównego

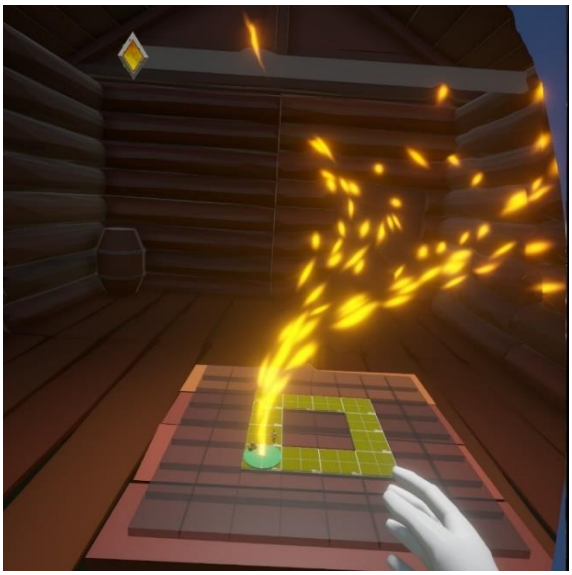
Źródło: Badania własne

Gdy jedna z wirtualnych rąk Gracza wyceluje w przycisk nowej gry za pomocą wskaźnika (czerwone promienie przytwierdzone do rąk), to przycisk zostanie podświetlony:

Rys. 32. Wybór przycisku w menu głównym

Źródło: Badania własne

Po wciśnięciu przycisku selekcji na kontrolerze wirtualnej rzeczywistości, widok menu głównego znika, a w jego miejsce pojawia się widok danych określających status nowej rozgrywki. Rozgrywka zostaje rozpoczęta, a na siatce pojawia się Bohater oraz pokój początkowy:



Rys. 33. Widok rozpoczętej rozgrywki

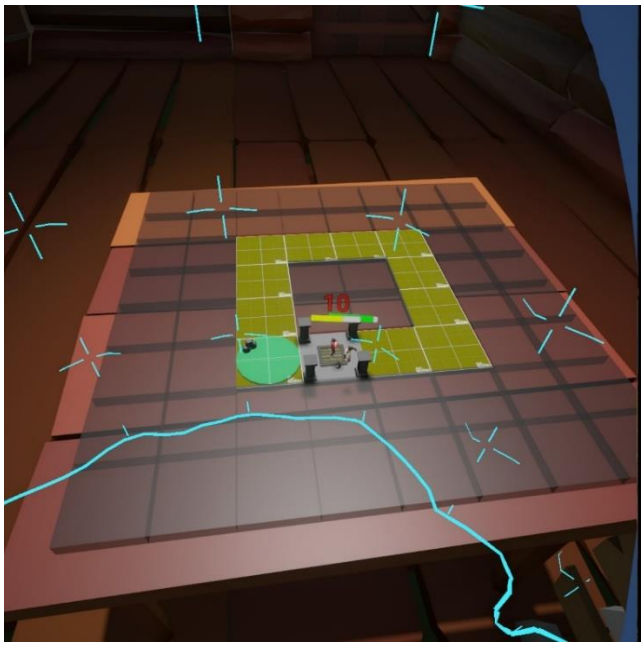
Źródło: Badania własne

Oprócz tego, na generatorze pokoi pojawiają się pokoje startowe:

Rys. 34. Pokoje początkowe

Źródło: Badania własne

W celu przetestowania rozpoczęcia rozgrywki z poziomu widoku porażki, należy ustawić na siatce pokoje z przeciwnikami, w których Bohater będzie odbywał walki (widok błękitnego systemu zabezpieczeń wynika z ograniczonej przestrzeni testowej):



Rys. 35. Widok walki

Źródło: Badania własne

Takie pokoje należy ustawić w taki sposób, żeby doprowadzić do spadku poziomu punktów życia Bohatera do 0. Wtedy wyświetlony zostaje widok porażki (widok błękitnego systemu zabezpieczeń wynika z ograniczonej przestrzeni testowej):

Rys. 36. Widok porażki

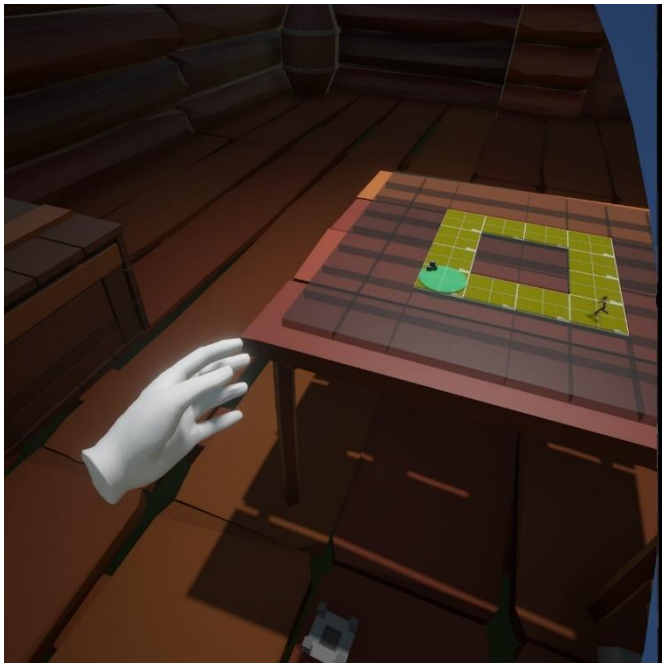
Źródło: Badania własne

Gracz celuje wtedy wskaźnikiem na przycisk spróbowania ponownie, który zostaje podświetlony:

Rys. 37. Wycelowanie wskaźnika w menu porażki

Źródło: Badania własne

Wtedy uruchomiona zostaje nowa rozgrywka:



Rys. 38. Nowa rozgrywka po porażce

Źródło: Badania własne

VII.4.3. Wyniki testów czarnej skrzynki

Na podstawie powyższych zrzutów ekranu z aplikacji można stwierdzić, że omawiane scenariusze i przypadki testowe zostały przetestowane z wynikiem pozytywnym.

Podsumowanie

Celem tej pracy było zaprojektowanie i zaimplementowanie trójwymiarowej gry wykorzystującej technologię rzeczywistości wirtualnej. Po przeprowadzeniu testów i przeanalizowaniu wyników końcowych można stwierdzić, że postawiony cel został zrealizowany mimo niezaimplementowania niektórych wymagań oznaczonych najniższym priorytetem takich jak implementacja niektórych rodzajów efektów.

Do najważniejszych osiągnięć przy tworzeniu tej aplikacji należy zaliczyć stworzenie systemu kontrolującego siatkę pokoi, na której można ustawiać nowe elementy, implementację systemu zapewniającego poprawne przechodzenie między stanami gry oraz implementację zestawu immersyjnych interakcji ze światem wirtualnym.

W celu zachowania obiektywnej oceny wyników należy również wspomnieć o rzeczach, które można było zrealizować lepiej. Można wskazać na przykład zwiększenie liczby przeprowadzonych testów i wskazanych scenariuszy i przypadków testowych, żeby zwiększyć niezawodność aplikacji.

Architektura stworzonego projektu pozwala na dalszy rozwój projektu i dodawanie nowych funkcjonalności. Możliwe jest przykładowo rozszerzenie gry o nowe efekty i co za tym idzie nowe rodzaje pokoi. Interesującym kierunkiem rozwoju byłoby również wzbogacenie systemu walki o różnego rodzaju modyfikatory wpływające na przebieg pojedynku. Pod kątem wykorzystywania technologii rzeczywistości wirtualnej interesującym kierunkiem byłoby wzbogacenie aplikacji o przedmioty lub umiejętności wymagające wykonywania gestów za pomocą wirtualnych rąk.

Oprócz dodawania nowych funkcjonalności można też wzbogacić już istniejące przez zaimplementowanie sygnałów zwrotnych pokroju dodatkowych efektów dźwiękowych i wizualnych pojawiających się w określonych sytuacjach. Ponadto, warto byłoby rozważyć dodanie różnych klas bohaterów, z których każda posiada inne parametry pokroju zadawanych obrażeń czy maksymalnych punktów życia.

Przedstawiony projekt reprezentuje aplikację zgodną z najnowszymi trendami i standardami rynkowymi oraz prezentuje wiele fascynujących perspektyw dalszego rozwoju. Gra spełnia postawione założenia i stwarza nowe możliwości przeżywania immersyjnych doświadczeń w rzeczywistości wirtualnej.

Bibliografia

[1]

[2] **Anderson David J. 2010.** *Kanban: Successful Evolutionary Change for Your Technology Business.* brak miejsca : Blue Hole Press, 2010. ISBN.

[3] **Borromeo Nicolas Alejandro. 2021.** *Hands-On Unity 2021 Game Development.* Birmingham : Packt Publishing, 2021. ISBN.

[4] **Brackeys. 2020.** *Unity YouTube Tutorials.* [Online] 2020. https://www.youtube.com/c/Brackeys/featured.

[5] **Chacon Scott i Straub Ben. 2014.** *Pro Git.* brak miejsca : APress, 2014. ASIN.

[6] **Davis Bradley Austin. 2015.** *Oculus Rift in Action.* Nowy Jork : Manning, 2015. ISBN.

[7] **Fowler Martin. 2003.** *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* brak miejsca : Addison-Wesley Professional, 2003. ISBN.

[8] **International Organisation for Standardization. 2021.** Project, Programme and Portfolio Management - Context and concepts (ISO/DIS Standard Nr 21500). 2021.

[9] **La Counte Scott. 202.** The Ridiculously Simple Guide to Trello: A Beginners Guide to Project Management with Trello. Berkeley : Niezależne wydawnictwo, 202. ISBN.

[10] **Lee Graham. 2019.** Modern Programming: Object Oriented Programming and Best Practices. Birmingham : Packt Publishing, 2019. ISBN.

[11] **Lowood Henry E. 2021.** Virtual Reality. [Online] Maj 2021. https://www.britannica.com/technology/virtual-reality.

[12] **Microsoft Inc. 2022.** *A tour of the C# language.* [Online] 07 2022. https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

[13] **Microsoft Inc. 2022.** *Garbage Collection.* [Online] 07 2022. https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/.

[14] **Nystrom Robert. 2014.** *Game Programming Patterns.* brak miejsca : Genever Benning, 2014.

[15] **Parisi Tony. 2015.** Learning Virtual Reality: Developing Immersive Experiences and Applications for Desktop, Web, and Mobile. Sebastopol : O'Reilly Media, 2015. ISBN.

[16] **Schell Jesse. 2019.** *The Art of Game Design: A Book of Lenses.* Boston : A K Peters/CRC Press, 2019. ISBN.

[17] **Unity Technologies. 2022.** *Unity Documentation.* [Online] 2022. https://docs.unity3d.com/Manual/index.html.

[18] **Unity technologies. 2022.** XR Interaction Toolkit - Manual. [Online] 07 2022. https://docs.unity3d.com/2019.3/Documentation/Manual/IL2CPP-HowItWorks.html.

[19] **Vermeulen Steve. 2020.** Zenject Documentation. [Online] May 2020. https://github.com/modesttree/Zenject.

[20] **Weisfeld Matt. 2019.** *The Object-Oriented Thought Process.* brak miejsca : Addison-Wesley Professional, 2019. ISBN.

[21] **Wikipedia. 2022.** Pyłek (wzorzec projektowy). *Wikipedia.* [Online] 13 Maj 2022. https://pl.wikipedia.org/wiki/Pyłek_(wzorzec_projektowy).

[22] **Wolf, Mark J. P. 2012.** Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming. s.l. : Greenwood, 2012. ISBN: 978-0313379369.

[23] **Wydawnictwo Naukowe PWN . 2021.** Gry komputerowe. *Encyklopedia PWN.* [Online] 06 2021. https://encyklopedia.pwn.pl/haslo/gry-komputerowe;3908365.html..

Spis rysunków

Spis tabel

Załączniki

1.Płyta CD/DVD zawierająca:

a. Prezentację wyników pracy dyplomowej
b. Kody źródłowe oprogramowania
c. Biblioteki programowe niezbędne do zbudowania i uruchomienia oprogramowania