



WYŻSZA SZKOŁA EKONOMII I INFORMATYKI

KATEDRA INFORMATYKI

Operation Deratization

Dokumentacja techniczna

Imię i nazwisko:

Dawid MUCHA
Filip KRAWIEC
Jakub MOLEK
Paweł TROJAŃSKI

22 lutego 2024

Spis treści

1 Wstęp	12
1.1 Cel	12
1.2 Dla kogo jest ten dokument?	12
1.3 Co znajdziesz w tym dokumencie?	13
2 Charakterystyka	14
2.1 Opis gry	14
2.2 Gatunek	14
2.3 Grupa docelowa	14
2.4 Czym się wyróżniamy?	14
2.5 Platforma	16
2.6 Silnik gry	16
3 Architektura i mechaniki gry	17
3.1 Struktura projektu	17
3.1.1 Organizacja katalogów	17
3.1.2 Struktura plików konfiguracyjnych	18
3.1.3 Narzędzia Zewnętrzne	19
3.2 Kluczowe mechaniki gry	20
3.2.1 Mechanika Sztucznej Inteligencji (AI)	20
3.2.2 Opisy Mechaniki Maszyny Stanów Sztucznej Inteligencji	23
3.2.3 Mechanika Wyposażenia	26
3.2.4 Mechanika Interaktywnych Obiektów	28

3.2.5	Mechaniki Środowiska	32
3.2.6	Mechaniki Gracza	32
3.2.7	Mechaniki Interakcji Środowiskowej	36
3.2.8	Interfejs Użytkownika i Mechaniki Interakcji	38
3.3	Interakcje między modułami	41
3.4	Wzorce projektowe	41
3.4.1	Wzorzec szablonu metody	42
3.4.2	Wzorzec stanu	42
3.4.3	Wzorzec puli obiektów	43
4	Interfejs użytkownika (UI) i grafika	46
4.1	Projektowanie interfejsu użytkownika	46
4.1.1	Tworzenie logo naszej gry	46
4.1.2	Tworzenie logo Menu głównego	48
4.1.3	Main Menu	49
4.1.4	Loading Screen	52
4.1.5	InGame Pause Menu	53
4.1.6	Grafiki wykorzystane do Cutscenek	55
4.2	Grafiki interfejsu	55
4.2.1	Przygotowanie listy grafik do stworzenia	56
4.2.2	Poszczególne kroki tworzenia ikon	56
4.2.3	Pokaz wszystkich granatów oraz broni zrobionych do gry	61
4.2.4	Pokaz wszystkich ikon UI zrobionych do gry	61
4.3	Integracja grafiki z interfejsem	62

5 Animacje i efekty wizualne	63
5.1 System animacji postaci i obiektów	63
5.1.1 Algorytmy wykorzystywane do animowania postaci	63
5.1.2 Animator przeciwników	67
5.1.3 Animator otwieranych/zamykanych obiektów	67
5.2 Specjalne efekty wizualne	68
5.2.1 Particle System	68
5.2.2 VFX	70
5.3 Oświetlenie	71
5.3.1 Dynamiczne oświetlenie	71
5.3.2 Wypieczone oświetlenie	71
5.4 Zastosowanie shaderów	71
6 Poziomy (Level design)	74
6.1 Planowanie i projektowanie poziomów	74
6.2 Implementacja interakcji w środowisku	75
6.2.1 Rodzaje przedmiotów do interakcji	75
6.2.2 Sposób implementacji	77
6.3 Rozkład przestrzeni	78
6.3.1 Struktura Ogólna Poziomu	78
6.3.2 Sekcje i Obszary Kluczowe	80
6.3.3 Połączenia Między Sekcjami	81
6.4 Estetyka i atmosfera	82
6.5 Przerywniki filmowe	83

6.5.1	Rodzaje i zastosowanie w grze cutscenek	83
6.5.2	Struktura sceny	84
6.5.3	Mechanizmy użyte w cutscenice	84
6.5.4	Cutscenka Intro - pozostałe w trakcie pracy	87
7	Audio	89
7.1	Kontrola ścieżek dźwiękowych	89
7.2	Ustawienia 3D dźwięków	90
7.3	Zarządzanie Zdarzeniami Dźwiękowymi	91
7.3.1	Opis Skryptu	92
7.3.2	Kluczowe Elementy	93
7.3.3	Wykorzystanie	93
7.3.4	Przykładowe Zastosowanie	93
8	Testowanie i debugowanie	94
8.1	Narzędzia debugujące	94
8.2	Raportowanie błędów	96
9	Optymalizacja i wydajność	99
9.1	Profilowanie kodu	99
9.2	Zarządzanie zasobami	101
9.2.1	Asynchroniczne Ładowanie	101
9.2.2	Object Pooling	102
9.3	Techniki optymalizacji wydajności	107
9.3.1	Wyszukiwanie Obiektów	107

9.3.2	Konstruktor Łańcucha Znaków	109
9.3.3	Pola Statyczne Obiektów	111
9.3.4	Pola GPU Instancing	112
9.3.5	Wypiekanie Światał	112
9.3.6	Optymalizacja Meshów	115

Spis rysunków

1	Komunikacja między modułami	41
2	Główne logo, w którym są nawiązania do tytułu tworzonej przez nas gry	47
3	Logo w głównym Menu, domyślnie jest z przezroczystym tłem (w tym przypadku tło dodane po ukończeniu logo)	48
4	Na powyższym obraz możemy zobaczyć bardzo uproszczony rysunek schematyczny z procesów tworzenia Menu	49
5	Nasze Main Menu utworzone już w grze	50
6	Plansza "Credits"	50
7	Plansza "Options"	51
8	Plansza "Quit"	52
9	Loading Screen po zaprojektowaniu i wykonaniu	53
10	InGame Pause Menu po zaprojektu i utworzeniu	54
11	InGame Options Menu po zaprojektu i utworzeniu	54
12	Grafiki użyte w cutscenach	55
13	Lista ikon do stworzenia	56
14	Screen prefabu	57
15	Usunięcie tła	57
16	Zmiana koloru na biały	58
17	Dodanie blasku zewnętrznego	58
18	Eksport grafiki	59
19	Zmiana kolory na szary	59
20	Zmiana przezroczystości	60

21	Eksport grafiki	60
22	Plansza przedstawiająca wszystkie ikony broni oraz granatów w grze	61
23	Plansza przedstawiająca wszystkie ikony broni oraz granatów w grze	61
24	Blend Tree dla płynnego przechodzenia między prędkościami poruszania dla AI	64
25	Warstwa broni Animatora przeciwników i zastosowanie Avatar Maska	67
26	Animator otwieralnego oraz zamkialnego obiektu na przykładzie skrzynki . . .	68
27	Przedstawienie konfiguracji gazu	69
28	Efekt podpalenia w grze	70
29	Efekt dymu w grze	70
30	Shader winiety odpowiedzialnej za wyświetlanie na ekranie gracza informacji o ilości utraconego życia	72
31	Shader wody, którą znajdziemy w fontannie znajdującej się na terenie parku . .	72
32	Prostsza wersja shadera zanikania, używana w przypadku podniesienia interaktywnych obiektów takich jak kamizelka kuloodporna, apteczka czy granaty	73
33	Zastosowanie gazu jako naturalnego ograniczenia wielkości poziomu	77
34	Hierarchia obiektów na mapie	78
35	Ustawienia drzew na mapie - z poziomu terenu	79
36	Użyte tekstury	79
37	Przykład hierarchii dla sekcji miasta	80
38	Ścieżki naniesione na teren za pomocą tekstury	82
39	Konfiguracja głosu dla Maxa - narratora pierwszej części intro	85
40	Konfiguracja głosu dla narratora drugiej części intro - Smart Chip	86
41	Konfiguracja głosu dla głównego bohatera	86
42	Ustawienia Audio Mixerów	89

43	Przykładowe ustawienie 3D dźwięku dla obiektu granatu	90
44	Okno Profilera dostępnego w edytorze Unity	95
45	Okno Frame Debuggera w edytorze Unity	96
46	Okno Console w edytorze Unity	96
47	Lista rozpoznanych błędów deweloperskich występujących w grze	97
48	Lista rozpoznanych błędów wizualnych występujących w grze	98
49	Okno Profilera dostępne w menu kontekstowym Window/Analysis/Profiler . .	99
50	Okno Frame Debug, które znajdziemy w menu kontekstowym Window/Analysis/Frame Debugger	100
51	Konfiguracja obiektu Particle System, który powinien wrócić do puli obiektów po zakończeniu emisji cząsteczek	105
52	Konfiguracja obiektu, który ma wrócić do puli obiektów po upływie czasu . .	106
53	Konfiguracja obiektu, który ma wrócić do puli obiektów po zakończeniu odtwarzania animacji	106
54	Optymalizacja pozyskiwania obiektów - porzucenie wywołań FindObjectOfType	109
55	Optymalizacja modyfikowania ciągów znaków - zastosowanie String Buildera .	110
56	Odpowiednie wykorzystanie pola Static na obiekcie	111
57	Renderowanie obiektów z użyciem GPU Instancing	112
58	Zakładka Scene ustawień oświetlenia, w której tworzymy plik dla sceny, a następnie konfigurujemy parametry i na koniec klikamy w Generate Lighting .	113
59	Katalog zawierający wszystkie pliki wypieczonego oświetlenia	114
60	Zakładka Baked Lightmaps okna oświetlenia, w której sprawdzimy wszystkie wygenerowane dane oświetlenia dla wybranego pliku oświetlenia sceny . . .	114
61	Konfiguracja obiektów światła - ustawiamy tryb Mixed lub Bake by móc korzystać z wypieczonych map	115

62	Ustawienia statycznego obiektu, który ma brać udział w wypiekaniu światła	115
63	Możliwe sposoby na optymalizację mesha obiektu	116

Spis fragmentów kodu

1	Klasa Interactable, z której dziedziczą wszystkie skrypty interaktywnych obiektów w grze	42
2	Implementacja wzorca stanu w klasie AiState	43
3	Klasa ObjectPoolManager odpowiedzialna za tworzenie pul obiektów	45
4	Fragment skryptu ObjectPoolManager.cs używany do zwracania obiektów do puli obiektów	45
5	Klasa WeaponIk służąca do realistycznego symulowania ruchów kości podczas korzystania z broni przez AI	65
6	Skrypt Ragdoll odpowiedzialny za symulowanie realistycznych reakcji postaci na siły zewnętrzne	66
7	Zmiany sposobu kontrolowania drzwi w skrypcie DoorMotionSensor.cs	88
8	Zarządzanie zdarzeniami dźwiękowymi w klasie AudioEventManager	92
9	Klasa SceneLoader obsługująca asynchroniczne przechodzenie między scenami	101
10	Zwrócenie obiektu do puli po zakończeniu trwania efektu systemu cząsteczkowego	103
11	Zwrócenie obiektu do puli po upływie określonego czasu	104
12	Klasa ReturnToPoolOnAnimationEnd wykorzystywana do zwrócenia obiektu do puli po zakończeniu trwania animacji przy użyciu zdarzeń animacji Unity .	104
13	Przykład podmiany funkcji Instantiate w skrypcie DISystem.cs na nowo utworzoną i zoptymalizowaną SpawnObject z klasy ObjectPoolManager	107
14	Przykład podmiany funkcji Destroy w skrypcie DamageIndicator.cs na nowo utworzoną i zoptymalizowaną ReturnObjectToPool z klasy ObjectPoolManager	107

1 Wstęp

Cześć! Witaj w naszej dokumentacji technicznej gry FPS pod nazwą *Operation Deratization*. To miejsce, gdzie znajdziesz masę informacji na temat tego, co się dzieje pod maską naszego projektu. Niezależnie od tego, czy grasz, czy programujesz, mamy nadzieję, że znajdziesz tu coś dla siebie.

1.1 Cel

Celem tego dokumentu jest przekazanie jasnych informacji na temat tego, co nasza gra potrafi. Dla programistów mamy trochę magicznych słów dotyczących interfejsów programistycznych, struktur danych i tego, jak współpracować z kodem. Dla reszty zainteresowanych – używajcie tego do odkrywania wszystkich fajnych rzeczy, jakie przygotowaliśmy!

1.2 Dla kogo jest ten dokument?

No cóż, myślimy, że każdy znajdzie tu coś dla siebie:

- **Programiści:** Jeśli kopiesz kod, to mamy dla ciebie szereg informacji na temat struktury projektu, funkcji i tego, jak wszystko działa.
- **Graficy:** Jeśli zajmujesz się tworzeniem tego, co widzi użytkownik, znajdziesz tu sporo o UI, animacjach i efektach wizualnych.
- **Projektanci Poziomów** Jeśli tworzysz światy, to mamy dla ciebie sekcję o planowaniu i projektowaniu poziomów, interakcjach i testowaniu środowiska gry.
- **Animatorzy i Muzycy:** Dla pasjonatów tworzenia ścieżek dźwiękowych i animacji, przygotowaliśmy specjalne rozdziały z informacjami na temat sterowania audio w grze, ustnień 3D dźwięków, dostosowywania głośności i wiele więcej.
- **Testerzy:** Jeśli spędzasz czas na szukaniu bugów, to znajdziesz tu informacje o strategiach testowania, narzędziach debugujących i tym, jak raportować błędy.

1.3 Co znajdziesz w tym dokumencie?

Dokumentacja jest podzielona na kilka rozdziałów, a każdy z nich skupia się na innym kawałku gry *Operation Deratization*. Poniżej krótkie wprowadzenie do każdego:

- **Rozdział 1:** Dowiecie się o strukturze gry i głównych ideach, które nami kierują.
- **Rozdział 2:** Ci którzy poszukują więcej informacji odnośnie samego gameplayu, na czym polega nasza gra, do kogo jest skierowana i czym się wyróżnia na tle innych FPSów nie będą zawiedzeni odwiedzając ten rozdział.
- **Rozdział 3:** Jeśli jesteś zainteresowany technicznymi detalami, to znajdziesz tu info o interfejsach programistycznych, strukturach danych i algorytmach.
- **Rozdział 4:** Graficy – tu rozdział dla Was! Projektowanie UI, grafika, animacje i to, jak to wszystko ze sobą współpracuje.
- **Rozdział 5:** Animacje postaci, efekty wizualne, oświetlenie – dla tych, którzy chcą, żeby gra wyglądała jak prawdziwe dzieło sztuki.
- **Rozdział 6:** Projektanci Poziomów – ten rozdział ma coś dla Was! Planowanie poziomów, interakcje, balansowanie i testowanie.
- **Rozdział 7:** Miłośnicy ścieżek dźwiękowych – specjalne sekcje dla Was! Sterowanie audio w grze, ustawienia 3D dźwięków, dostosowywanie głośności i wiele więcej.
- **Rozdział 8:** Testerzy, przygotowaliśmy coś specjalnie dla Was – strategie testowania, debugowanie, raportowanie błędów i testy jednostkowe/integracyjne.
- **Rozdział 9:** Na koniec, dla tych, którzy myślą o wydajności – profilowanie kodu, optymalizacja algorytmów i inne takie.

Zapraszamy do zgłębiania tajemnic *Operation Deratization*! Bawcie się dobrze!

2 Charakterystyka

2.1 Opis gry

Operation Deratization to gra first-person shooter osadzona w arenie Battle Royale z unikalnym zawirowaniem fabularnym. W tej grze gracze wcielają się w rolę agenta specjalnego, który ma za zadanie zapewnić, że żaden z więźniów biorących udział w grze nie przeżyje, aby zdobyć swoją wolność. Więźniowie są porównywani do szczurów walczących o swoje życie, a misją gracza jest "wyeliminowanie" ich.

Gra oferuje proste i intuicyjne sterowanie przy użyciu klawiszy WSAD oraz celowanie myszką. Gracze mogą zdobywać bronie rozrzucone na ogromnym terenie, aby dozbroić się do sprawdzenia sił w starciu z przeciwnikami sterowanymi przez sztuczną inteligencję. Gra osadzona jest w trójwymiarowym środowisku z jedną mapą zawierającą różne punkty spawnu i lokalizacje. W grze znajdują się również tracker oraz chip. Pierwszy pozwala na chwilowe wskazanie kierunku do najbliższego przeciwnika na mapie, drugi natomiast po zbliżeniu się do martwego oponenta pozwala na jego oznaczenie oraz co za tym idzie trwałą eliminację.

2.2 Gatunek

Operation Deratization to first-person shooter (FPS) z elementami gry Battle Royale. Gracze mają okazję doświadczyć intensywnych starć z przeciwnikami w dynamicznej arenie.

2.3 Grupa docelowa

Gra jest skierowana do miłośników gier first-person shooter, zwłaszcza tych, którzy cenią sobie unikalne koncepty i fabularne twisty. Gracze poszukujący wyzwań i dynamicznych starć w trybie Battle Royale znajdą w *Operation Deratization* emocjonującą rozgrywkę.

2.4 Czym się wyróżniamy?

Operation Deratization wyróżnia się na tle innych gier first-person shooter dzięki kilku unikalnym elementom:

- **Autorski system trackera** – W grze zaimplementowaliśmy autorski system trackera, który umożliwia graczom skuteczne śledzenie przeciwników. Każdy gracz może aktywować tracker, co pozwala im oznaczyć przeciwnika na pewien czas. To narzędzie strategiczne umożliwia lepsze planowanie działań, zwłaszcza w dynamicznych sytuacjach, gdzie precyzyjne lokalizowanie przeciwników może stanowić klucz do sukcesu.
- **Oznaczanie zneutralizowanych przeciwników** – Po zneutralizowaniu przeciwnika, gracz ma możliwość oznaczenia martwego ciała. To funkcjonalność, która może być kluczowa w dynamicznej rozgrywce Battle Royale, umożliwiając graczowi śledzenie przeciwników, którzy nadal stanowią dla niego zagrożenie. Jest to również niekiedy ryzykowny element rozgrywki z powodu wzrostu zagrożenia otrzymania obrażeń od pozostałych przeciwników, dla których możemy okazać się łatwym celem będąc na widoku. To unikalne podejście do różnych strategii, wprowadzają dodatkowy element taktyki poza samą walką.
- **Elementy fabularne** – *Operation Deratization* nie tylko oferuje intensywną rozgrywkę, ale również wplecone są elementy fabularne, które nadają grze głębię. Unikalne zawirowanie fabularne, w którym gracze pełnią rolę agenta specjalnego walczącego z więźniami, dodaje dodatkowy kontekst i motywację do działań postaci.
- **Prosta i intuicyjna mechanika sterowania** – Sterowanie w grze zostało zaprojektowane w sposób prosty i intuicyjny, dzięki czemu gracze mogą skupić się na samej rozgrywce. Kombinacja klawiszy WSAD i celowania myszką sprawia, że nawet nowi gracze szybko przyswajają mechanikę gry, co przyczynia się do przyjemnego doświadczenia.
- **Elastyczność mapy z różnymi lokacjami** – Jedna mapa gry oferuje różnorodne punkty spawnu i lokalizacje, co zapewnia elastyczność w planowaniu i podejmowaniu działań. Gracze muszą dostosować swoje strategie do zmieniającego się środowiska, co dodaje element nieprzewidywalności do każdej rozgrywki.
- **Zróżnicowane zestawy broni i wyposażenia** – Rozrzucone na mapie różnorodne bronie i wyposażenie pozwalają graczom dostosować swój styl gry. Bogaty wybór sprzętu dodaje głębi taktycznej, pozwalając graczom wybierać odpowiednie narzędzia do konkretnych sytuacji.

Operation Deratization to nie tylko kolejny FPS – to unikalne doświadczenie, które łączy w sobie intensywną rozgrywkę, elementy taktyczne i estetyczny design.

2.5 Platforma

Operation Deratization jest dostępne na platformy PC. Gracze mogą cieszyć się rozgrywką na komputerach osobistych, zapewniając płynność i pełną kontrolę podczas strzelania do wirtualnych przeciwników.

2.6 Silnik gry

W projekcie wykorzystano silnik gry **Unity** w wersji edytora **2021.3.16f1**. Unity to potężne narzędzie do tworzenia gier, oferujące szeroki zakres funkcji i możliwości.

3 Architektura i mechaniki gry

W tej sekcji przedstawimy kluczowe elementy dotyczące architektury oraz mechanik gry. Omówimy strukturę projektu, kluczowe mechaniki wpływające na rozgrywkę, interakcje między poszczególnymi modułami gry oraz wzorce projektowe zastosowane w projekcie.

3.1 Struktura projektu

W tym podrozdziale przeanalizujemy strukturę projektu, skupiając się na kluczowych aspektach organizacyjnych, które pomagają w zarządzaniu zasobami oraz utrzymaniu czytelności kodu. Szczególnie skoncentrujemy się na organizacji katalogów, strukturze plików konfiguracyjnych oraz narzędziach zewnętrznych.

3.1.1 Organizacja katalogów

Struktura projektu gry została starannie zaplanowana, aby ułatwić zarządzanie zasobami. Poniżej przedstawiono główne katalogi wraz z ich przeznaczeniem:

- **Assets/Animations/** – W tym katalogu znajdują się wszystkie pliki związane z animacjami w grze. Pliki te mogą obejmować animacje postaci, obiektów, interfejsu użytkownika i inne.
- **Assets/Art/** – Katalog z zasobami artystycznymi, takimi jak modele, tekstury, prefaby, materiały gotowych paczek plików.
- **Assets/Audio/** – Zawiera pliki dźwiękowe i muzykę używaną w grze.
- **Assets/Editor/** – Skrypty związane z Edytorem Unity, np. narzędzia pomocnicze do pracy w edytorze.
- **Assets/NavMeshComponents/** – Katalog z komponentami służącymi do dynamicznego generowania i obsługi nawigacji (*NavMesh*) w czasie rzeczywistym w środowisku gry.
- **Assets/Plugins/** – Zawiera zewnętrzne pluginy używane w projekcie.
- **Assets/Resources/** – Katalog przechowujący pliki audio, takie jak muzyka wykorzystywana w grze. Katalog ten jest często używany do przechowywania zasobów,

do których potrzebujemy dostępu w trakcie działania gry, a niekoniecznie podczas edycji w Unity.

- **Assets/Scenes/** – Zawiera pliki scen, reprezentujące poszczególne poziomy lub ekrany w grze.
- **Assets/ScriptableObjects/** – Zawiera pliki *Scriptable Objects*, które przechowują dane bez potrzeby tworzenia instancji.
- **Assets/Scripts/** – Tutaj przechowywane są skrypty odpowiedzialne za logikę gry.
- **Assets/Settings/** – Pliki ustawień m.in. oświetlenia, graficznych oraz związanych z Volume Profile.
- **Assets/Setup/** – Zawiera komponenty interfejsu gracza, takie jak ikony broni oraz pliki z prefabami używanymi w celach pomocniczych, np. wyświetlanie kamery uruchamianej podczas śmierci gracza.
- **Assets/Shaders/** – W tym katalogu znajdziemy wszystkie stworzone efekty graficzne, wykorzystujące Shader Graph w Unity. W tym katalogu zorganizowane są różne efekty, takie jak materiały post-processingu, efekty specjalne, czy niestandardowe materiały.
- **Assets/TextMeshPro/** – Pliki związane z używaniem narzędzia TextMesh Pro do obsługi tekstu w grze.
- **Docs/** – Katalog, w którym znajduje się dokumentacja projektu w formatach takich jak LaTeX.

3.1.2 Struktura plików konfiguracyjnych

Plików konfiguracyjnych odgrywają kluczową rolę w definiowaniu parametrów i ustawień w grze. Poniżej znajduje się struktura i opis głównych typów plików konfiguracyjnych używanych w projekcie:

- **Ustawienia Agentów AI: Assets/ScriptableObjects/Enemy/**
Plików konfiguracyjnych dotyczących agentów sztucznej inteligencji. Każdy plik w tym folderze to osobny Scriptable Object, definiujący parametry zachowania, umiejętności i strategii AI w grze.

- **Statystyki Broni: Assets/ScriptableObjects/Weapon**

W tym podfolderze umieszczone są pliki Scriptable Object, które przechowują statystyki poszczególnych rodzajów broni. Każdy plik konfiguracyjny definiuje parametry takie jak obrażenia, zasięg, magazynki i inne właściwości związane z uzbrojeniem.

- **Ustawienia Graficzne URP: Assets/Settings/GraphicsSettings/**

W tym podfolderze znajdują się pliki konfiguracyjne, które definiują ustawienia graficzne związane z Universal Render Pipeline (URP). Tutaj możemy dostosować parametry renderowania, oświetlenie, cienie i inne aspekty związane z wydajnością graficzną gry.

- **Ustawienia Oświetlenia: Assets/Settings/Lightning/**

Zawiera parametry dotyczące światła w grze. Może to obejmować kolor, intensywność, cień, odległość zasięgu, a także inne właściwości związane ze światłem punktowym, kierunkowym lub źródłem światła punktowego.

- **Volume Profiles: Assets/Settings/VolumeProfiles/**

W tym podfolderze przechowywane są pliki konfiguracyjne związane z Volume Profile. Volume Profile to narzędzie w Unity, które umożliwia skonfigurowanie efektów wizualnych i dźwiękowych na poziomie sceny. Pliki w tym folderze kontrolują parametry takie jak kolorystyka, efekty post-processingu czy ustawienia dźwięku.

3.1.3 Narzędzia Zewnętrzne

Narzędzia zewnętrzne wymienione poniżej są integralną częścią procesu tworzenia gry, wspierając różne aspekty projektu, od zarządzania zadaniami po rozwój kodu i dokumentację.

- **Adobe Photoshop** – Profesjonalne oprogramowanie do edycji grafiki, wykorzystywane do tworzenia i dostosowywania elementów interfejsu użytkownika oraz innych zasobów w grze.
- **Audacity** – Oprogramowanie do edycji dźwięku, używane w projekcie do obróbki i edycji plików dźwiękowych, takich jak efekty dźwiękowe czy muzyka.
- **GitHub** – Platforma do zarządzania kodem źródłowym, umożliwiająca kontrolę wersji, śledzenie zmian i współpracę w zespole.
- **GitHub Desktop / GitKraken** – Klient desktopowy ułatwiający interakcję z repozytorium GitHub poprzez intuicyjny interfejs graficzny.
- **Microsoft Visual Studio 2019** – Środowisko programistyczne używane do tworzenia, debugowania i rozwijania kodu źródłowego w języku C#.

- **Overleaf** – Platforma do współpracy nad dokumentacją w LaTeX, umożliwiająca tworzenie, edycję i udostępnianie dokumentów online.
- **Trello** – Platforma do zarządzania projektem, umożliwiająca tworzenie tablic, list i kart do organizacji zadań oraz śledzenia postępu.

3.2 Kluczowe mechaniki gry

Przeanalizujemy kluczowe mechaniki gry, które wpływają na doświadczenie gracza. Opiszemy, jak poszczególne mechaniki są zaimplementowane w kodzie oraz jakie mają znaczenie dla rozgrywki. Będziemy się skupiać na tych aspektach, które w największym stopniu determinują unikalność i atrakcyjność gry.

3.2.1 Mechanika Sztucznej Inteligencji (AI)

Klasa WeaponIK – Skrypt **WeaponIK** jest odpowiedzialny za obsługę kinematyki odwrotnej (IK) związanej z celowaniem broni. Kluczowe funkcje obejmują:

- **Metoda LateUpdate:** Dostosowuje rotacje kości na podstawie pozycji celu.
- **Metoda AimAtTarget:** Oblicza i stosuje rotacje do kości, aby celować w cel.
- **Metoda GetTargetPosition:** Oblicza pozycję celu, uwzględniając ograniczenia kątowe i odległościowe.
- **Metoda SetTargetTransform:** Ustawia transformację celu do celowania.
- **Metoda SetAimTransform:** Ustawia transformację celu (lufa) do celowania bronią.

Klasa EnemyShoot – Skrypt **EnemyShoot** zarządza zachowaniem strzelania przeciwników AI. Kluczowe funkcje obejmują:

- **Metoda Update:** Sprawdza obecną broń i ją konfiguruje.
- **Metoda StartFiring:** Inicjuje stan strzelania.
- **Metoda StopFiring:** Zatrzymuje stan strzelania.
- **Metoda Shoot:** Wykonuje logikę strzelania, w tym raycasting i efekty uderzenia.

- **Metoda ReloadCoroutine:** Zarządza korutyną do przeładowania broni.
- **Metoda SetCurrentWeapon:** Ustawia obecną broń na podstawie wyposażonej broni przeciwnika AI.

Klasa EnemyHealth – Skrypt **EnemyHealth** zarządza zdrowiem i zachowaniami związanymi ze śmiercią przeciwników AI. Kluczowe funkcje obejmują:

- **Metoda Update:** Obsługuje efekty związane ze śmiercią, jeśli przeciwnik nie żyje i jest blisko gracza.
- **Metoda TakeDamage:** Przetwarza obrażenia, uwzględnia pancerz i wywołuje śmierć, jeśli zdrowie spadnie poniżej zera.
- **Metoda IsLowHealth:** Sprawdza, czy przeciwnik ma niskie zdrowie.
- **Metoda Die:** Inicjuje stan śmierci dla przeciwnika.
- **Croutine HandleDeathEffects:** Zarządza efektami wizualnymi i czyszczeniem po śmierci przeciwnika.
- **Croutine FadeOutPromptText:** Stopniowo wygasza tekst zachęty po śmierci.
- **Metoda SetShaderParameters:** Dostosowuje parametry shadera dla efektów wizualnych.
- **Metoda RestoreHealth:** Przywraca zdrowie przeciwnika.
- **Metoda PickupArmor:** Podnosi pancerz, jeśli przeciwnik żyje i pancerz nie jest maksymalny.

Klasa AiWeapons – Skrypt **AiWeapons** zarządza funkcjonalnościami związanymi z bronią AI. Kluczowe cechy obejmują:

- **Metoda Update:** Obsługuje namierzanie i strzelanie na podstawie obecnego celu i broni.
- **Metoda SetFiring:** Rozpoczyna lub zatrzymuje strzelanie na podstawie wejścia.
- **Metoda StartShootingCoroutine:** Inicjuje korutynę do strzelania.
- **Metoda StopShootingCoroutine:** Zatrzymuje korutynę strzelania.
- **Metoda Shoot:** Realizuje logikę strzelania, w tym wywołuje metody strzelania dla konkretnej broni.

Klasa AiAudioSensor – Skrypt **AiSightSensor** odpowiada za wykrywanie dźwięków w otoczeniu agenta AI. Kluczowe funkcje obejmują:

- **Metoda Start:** Inicjuje komponenty i subskrybuje się do zdarzenia audio.
- **Metoda HandleAudioEvent:** Obsługuje zdarzenie dźwiękowe, sprawdzając, czy agent jest żywy, czy dźwięk jest słyszalny oraz czy należy zareagować na dźwięk wysokiego priorytetu.
- **Metoda IsAudioAudible:** Określa czy dźwięk jest słyszalny przez agenta, biorąc pod uwagę odległość oraz parametry dźwięku.
- **Metoda IsChildOfMyObject:** Sprawdza, czy dany obiekt jest dzieckiem obiektu kontrolującego sensor audio.
- **Metoda IsHighPrioritySound:** Określa, czy dany dźwięk jest dźwiękiem wysokiego priorytetu.
- **Metoda GetGameObjectPath:** Zwraca pełną ścieżkę do obiektu w hierarchii sceny.
- **Metoda OnDrawGizmosSelected:** Rysuje pomocnicze obiekty w edytorze Unity w celu wizualizacji zasięgu działania sensora.

Klasa AiSightSensor – Skrypt **AiSightSensor** reprezentuje komponent sensoryczny odpowiedzialny za wykrywanie celów. Kluczowe funkcje obejmują:

- **Metoda Start:** Inicjalizuje interwał skanowania sensora.
- **Metoda Update:** Wywołuje okresowe skany i aktualizuje listę wykrytych obiektów.
- **Metoda Scan:** Przeprowadza skan w celu zidentyfikowania ważnych celów w polu widzenia sensora.
- **Metoda IsValidTarget:** Sprawdza, czy wykryty obiekt jest ważnym celem na podstawie tagów i warstw.
- **Metoda IsInSight:** Określa, czy obiekt jest w polu widzenia sensora.
- **Metoda CreateWedgeMesh:** Generuje siatkę reprezentującą pole widzenia sensora.
- **Metoda OnValidate:** Aktualizuje siatkę i interwał skanowania podczas walidacji.
- **Metoda Filter:** Filtruje obiekty na podstawie kryteriów warstw i tagów.

Klasa AiTargetingSystem – Skrypt **AiTargetingSystem** zarządza systemem celowania i procesem podejmowania decyzji przez sztuczną inteligencję. Kluczowe funkcje obejmują:

- **Metoda Update:** Aktualizuje pamięć sensoryczną AI i ocenia wyniki celowania.
- **Metoda EvaluateScores:** Określa najlepszy cel na podstawie wyników obliczonych z pamięci sensorycznej.
- **Metoda Normalize:** Normalizuje wartość względem maksymalnej wartości.
- **Metoda CalculateScore:** Oblicza wynik celu na podstawie odległości, kąta i wieku.

Klasa AiSensoryMemory – Skrypt **AiSensoryMemory** zarządza pamięcią AI dotyczącą wykrytych celów. Kluczowe funkcje obejmują:

- **Metoda UpdateSenses:** Aktualizuje pamięć sensoryczną na podstawie wejściowego sensora.
- **Metoda RefreshMemory:** Odświeża lub tworzy wpis w pamięci dla wykrytego celu.
- **Metoda FetchMemory:** Pobiera wpis w pamięci dla określonego celu.
- **Metoda ForgetMemories:** Usuwa wspomnienia, które są starsze niż określony próg lub związane z nieistniejącymi celami.

3.2.2 Opisy Mechaniki Maszyny Stanów Sztucznej Inteligencji

Maszyna stanów została szerzej opisana w podrozdziale o Wzorcach Projektowych, możesz się tam przenieść klikając tutaj [Wzorzec stanu](#).

Klasa AiAttackTargetState – Klasa **AiAttackTargetState** reprezentuje stan, w którym wróg sterowany przez sztuczną inteligencję aktywnie zaangażowany jest w atakowanie celu. Główne cechy obejmują:

- **Zachowanie:** Sztuczna inteligencja skupia się na atakowaniu określonego celu za pomocą swojej wyposażonej broni.
- **Warunki Przejścia:**

- Przechodzi do innego stanu, jeśli cel nie znajduje się już w zasięgu detekcji sensora.
- Przechodzi do stanu **FindWeapon**, jeśli sztuczna inteligencja skończy amunicję podczas ataku.

Klasa AiDeathState – Stan **AiDeathState** reprezentuje zachowanie sztucznej inteligencji po śmierci. Główne cechy tego stanu obejmują:

- **Zachowanie:** AI przestaje poruszać się i aktywuje efekt ragdoll, aby zasymulować upadek ciała. Dodatkowo, AI odrzuca broń.
- **Warunki Przejścia:** Brak warunków przejścia, stan ten jest końcowym stanem AI po śmierci.

Klasa AiFindWeaponState – Klasa **AiFindWeaponState** oznacza stan, w którym sztuczna inteligencja poszukuje nowej broni. Główne cechy obejmują:

- **Zachowanie:** Sztuczna inteligencja bada otoczenie, aby zlokalizować i zdobyć nową broń.
- **Warunki Przejścia:**
 - Przechodzi do stanu **AttackTarget** po skutecznym zdobyciu broni.
 - Przechodzi do innych stanów w zależności od zmieniających się okoliczności, takich jak znalezienie amunicji lub apteczki.

Klasa AiFindAmmoState – Klasa **AiFindAmmoState** reprezentuje stan, w którym sztuczna inteligencja poszukuje amunicji do swojej obecnie wyposażonej broni. Główne cechy obejmują:

- **Zachowanie:** Sztuczna inteligencja bada otoczenie, aby zlokalizować i zebrać amunicję.
- **Warunki Przejścia:**
 - Powraca do stanu **AttackTarget** po zdobyciu wystarczającej ilości amunicji.
 - Przechodzi do innych stanów w zależności od zmieniających się okoliczności, takich jak znalezienie lepszej broni lub apteczki.

Klasa AiFindFirstAidKitState – Klasa **AiFindFirstAidKitState** oznacza stan, w którym sztuczna inteligencja poszukuje apteczki w celu przywrócenia zdrowia. Główne cechy obejmują:

- **Zachowanie:** Sztuczna inteligencja porusza się po otoczeniu, aby znaleźć i skorzystać z apteczki.
- **Warunki Przejścia:**
 - Powraca do stanu **AttackTarget** po udanym uleczeniu.
 - Przechodzi do innych stanów w zależności od zmieniających się okoliczności, takich jak znalezienie broni czy amunicji.

Klasa AiFindTargetState – Klasa **AiFindTargetState** reprezentuje stan, w którym sztuczna inteligencja aktywnie poszukuje potencjalnych celów do zaangażowania. Główne cechy obejmują:

- **Zachowanie:** Sztuczna inteligencja bada otoczenie, aby zidentyfikować i priorytetyzować potencjalne cele na podstawie predefiniowanych kryteriów.
- **Warunki Przejścia:**
 - Przechodzi do stanu **AttackTarget** po zidentyfikowaniu odpowiedniego celu.
 - Przechodzi do innych stanów na podstawie różnych wskazań otoczenia, takich jak znalezienie broni czy amunicji.

Klasa AiInvestigateSoundState – Klasa **AiInvestigateSoundState** reprezentuje zachowanie sztucznej inteligencji, gdy ta wykryje dźwięk i rozpoczyna dochodzenie w jego kierunku. Główne cechy tego stanu obejmują:

- **Zachowanie:** AI aktywnie porusza się w kierunku ostatniego wykrytego dźwięku, aby zbadać jego źródło.
- **Warunki Przejścia:**
 - AI powraca do poprzedniego stanu, jeśli nie ma już ścieżki do dźwięku lub dźwięk nie jest już słyszalny.
 - Jeśli AI wykryje cel w trakcie dochodzenia do dźwięku, przejdzie do stanu **AttackTarget**.

Klasa AiPatrolState – Stan **AiPatrolState** reprezentuje zachowanie sztucznej inteligencji podczas patrolowania okolicy. Główne cechy tego stanu obejmują:

- **Zachowanie:** AI porusza się po okolicy w poszukiwaniu potencjalnych celów.
- **Warunki Przejścia:**
 - Jeśli AI wykryje cel podczas patrolowania, przejdzie do stanu **AttackTarget**.
 - Jeśli AI wykryje źródło podejrzanego dźwięku, przejdzie do stanu **InvestigateSound**.

Te opisy stanów AI pokazują, jakie zachowania są realizowane przez AI w różnych sytuacjach. Każdy stan ma swoje określone cele i zachowania, które pozwalają AI efektywnie funkcjonować w środowisku gry.

3.2.3 Mechanika Wyposażenia

Klasa Destructible – Skrypt **Destructible** zarządza zachowaniem destrukcji obiektów. Główne funkcje obejmują:

- **Metoda Destroy:** Niszczy obiekt, tworząc zastępcę, jeśli dostępny, i zwraca obiekt do puli obiektów.

Klasa Flashbang – Skrypt **Flashbang** zarządza zachowaniem granatów ogłuszających. Główne funkcje obejmują:

- **Metoda Update:** Aktualizuje odliczanie do eksplozji granatu ogłuszającego.
- **Metoda DestroyObject:** Niszczy obiekt granatu ogłuszającego.
- **Metoda Flash:** Inicjuje efekt oślepiania na podstawie pozycji, kąta i odległości gracza.
- **Metoda FlashCoroutine:** Zarządza korutyną dla efektu oślepiania.
- **Metoda OnCollisionEnter:** Obsługuje kolizje, sprawdzając szkło i dostosowując prędkość.

Klasa Grenade – Skrypt **Grenade** zarządza zachowaniem wybuchowych granatów. Główne funkcje obejmują:

- **Metoda Update:** Aktualizuje odliczanie do eksplozji granatu.

- **Metoda Explode:** Inicjuje efekt eksplozji, uszkadzając pobliskie obiekty i stosując siły.
- **Metoda DestroyObject:** Niszczy obiekt granatu.
- **Metoda OnCollisionEnter:** Obsługuje kolizje, sprawdzając szkło i dostosowując prędkość.

Klasa GrenadeIndicator – Skrypt **GrenadeIndicator** zarządza wyświetaniem wskaźników odległości granatów. Główne funkcje obejmują:

- **Metoda Update:** Aktualizuje odległość wskaźnika na podstawie pozycji gracza.
- **Metoda FixedUpdate:** Aktualizuje obrotu wskaźnika na podstawie kamery gracza.

Klasa Molotov – Skrypt **Molotov** zarządza zachowaniem koktajli Mołotowa. Główne funkcje obejmują:

- **Metoda Explode:** Inicjuje efekt eksplozji i ognia koktajlu Mołotowa.
- **Metoda DestroyObject:** Niszczy obiekt koktajlu Mołotowa.
- **Metoda OnCollisionEnter:** Obsługuje kolizje, sprawdzając szkło i uruchamiając eksplozję koktajlu Mołotowa.

Klasa Smoke – Skrypt **Smoke** zarządza zachowaniem dymnych granatów. Główne funkcje obejmują:

- **Metoda Update:** Aktualizuje odliczanie do efektu dymu.
- **Metoda SmokeOn:** Inicjuje efekt dymu.
- **Metoda DestroyObject:** Niszczy obiekt dymnego granatu.
- **Metoda OnCollisionEnter:** Obsługuje kolizje, sprawdzając szkło i dostosowując prędkość.

Klasa WeaponRecoil – Skrypt **WeaponRecoil** zarządza zachowaniem odrzutu broni. Główne funkcje obejmują:

- **Metoda Update:** Aktualizuje efekt odrzutu na podstawie statusu celowania i właściwości broni.

- **Metoda RecoilFire:** Zastosowuje odrzut podczas strzału bronią.

Klasa WeaponSway – Skrypt **WeaponSway** zarządza zachowaniem kołysania broni.
Główne funkcje obejmują:

- **Metoda Update:** Aktualizuje efekt kołysania na podstawie wejścia gracza i ruchu.
- **Metoda GetInput:** Pobiera wejście dotyczące chodzenia i patrzenia.
- **Metoda CompositePositionRotation:** Łączy położenie i obroty kołysania.
- **Metoda SwayOffset:** Oblicza przesunięcie kołysania na podstawie ruchu i prędkości.
- **Metoda SwayRotation:** Oblicza obroty kołysania na podstawie ruchu i prędkości.

3.2.4 Mechanika Interaktywnych Obiektów

Klasa interaktywnych obiektów została szczegółowo opisana w podrozdziale o Wzorach Projektowych, możesz się tam przenieść klikając tutaj Wzorzec szablonu metody **Klasa AmmoBox** – Skrypt **AmmoBox** reprezentuje interaktywną skrzynię z amunicją. Główne funkcje obejmują:

- **Metoda Update:** Zarządza aktualizacjami związanymi z łupieniem i komunikatami.
- **Metoda Interact:** Inicjuje proces uzupełniania amunicji dla odpowiednich broni.
- **Metoda OnTriggerEnter:** Wykrywa interakcję wroga, uruchamiając uzupełnianie amunicji dla broni AI.

Klasa ArrowIndicator – Skrypt **ArrowIndicator** kontroluje wskaźniki strzałek.
Główne funkcje obejmują:

- **Metoda Start:** Inicjuje właściwości wskaźnika strzałki.
- **Metoda PlayArrowAnimation:** Rozpoczyna sekwencję animacji strzałki.
- **Metoda OnDestroy:** Zatrzymuje aktywne tweensy, gdy obiekt nadzędny jest niszczony.

Klasa BodyArmor – Skrypt **BodyArmor** reprezentuje interaktywny pickup pancerza.
Główne funkcje obejmują:

- **Metoda Interact:** Inicjuje proces podnoszenia pancerza dla gracza.
- **Croutine DestroyAfterSound:** Niszczy obiekt pancerza po odtworzeniu efektu dźwiękowego.
- **Metoda TryDifferentBonePrefixes:** Próbuje różnych prefiksów kości do przyczepienia pancerza.
- **Metoda SetShaderParameters:** Dostosowuje parametry shadera dla zniknięcia pancerza.
- **Metoda OnTriggerEnter:** Wykrywa interakcję wroga, uruchamiając podnoszenie pancerza dla przeciwników.

Klasa Container – Skrypt **Container** reprezentuje interaktywny kontener. Główne funkcje obejmują:

- **Metoda Update:** Zarządza aktualizacjami związanymi z otwarciem i komunikatami.
- **Metoda Interact:** Przełącza stan otwarcia/zamknięcia kontenera i odtwarza dźwięk.
- **Metoda DetectEnemyNearby:** Sprawdza, czy w pobliżu są wrogowie, i otwiera kontener, jeśli są wykryci.

Klasa Coffin – Skrypt **Coffin** działa podobnie do skryptu **Container**, ale obsługuje różne obiekty na scenie. Udostępnia te same funkcje co skrypt **Container**.

Należy zauważyć, że istnieje skrypt **Coffin**, który działa identycznie jak skrypt **Container**, ale jest przeznaczony dla różnych obiektów na scenie, a jego użycie jest wymienne z **Container**.

Klasa DoorMotionSensor – Skrypt **DoorMotionSensor** zarządza otwieraniem i zamykaniem drzwi na podstawie bliskości gracza, kamery i wrogów. Główne funkcje obejmują:

- **Metoda Update:** Monitoruje odległości do gracza, kamery i wrogów, aby określić, czy drzwi powinny być otwarte czy zamknięte.
- **Metoda OpenDoors:** Rozpoczyna sekwencję otwierania drzwi.
- **Metoda CloseDoors:** Rozpoczyna sekwencję zamykania drzwi.
- **Metoda SlideDoors:** Przesuwa drzwi poziomo na podstawie podanej ilości.

- **Metoda ScaleDoors:** Skaluje drzwi na podstawie podanej ilości i stanu otwarcia.
- **C coroutine LerpDoorPosition:** Lerpuje pozycję drzwi w czasie dla płynnego przesuwania.
- **C coroutine LerpDoorScale:** Lerpuje skalę drzwi w czasie dla płynnego skalowania.

Klasa FirstAidKit – Skrypt **FirstAidKit** reprezentuje interaktywny zestaw apteczny. Główne funkcje obejmują:

- **Metoda Interact:** Inicjuje proces przywracania zdrowia dla gracza.
- **C coroutine DestroyAfterSound:** Niszczy obiekt zestawu aptecznego po odtworzeniu efektu dźwiękowego.
- **Metoda SetShaderParameters:** Dostosowuje parametry shadera dla zniknięcia zestawu aptecznego.
- **Metoda OnTriggerEnter:** Wykrywa interakcję wroga, uruchamiając przywracanie zdrowia dla przeciwników.

Klasa Gate – Skrypt **Gate** reprezentuje interaktywną bramę. Główne funkcje obejmują:

- **Metoda Update:** Zarządza aktualizacjami związanymi z otwartością bramy i komunikatami.
- **Metoda Interact:** Przełącza stan otwarcia/zamknięcia bramy.
- **Metoda DetectEnemyNearby:** Sprawdza, czy w pobliżu są wrogowie, i otwiera bramę, jeśli są wykryci.

Klasa Glass – Skrypt **Glass** obsługuje rozbijanie szklanych obiektów. Główne funkcje obejmują:

- **Metoda Break:** Rozpoczyna rozbijanie szkła za pomocą pocisku.
- **Metoda BreakFromGrenade:** Rozpoczyna rozbijanie szkła za pomocą eksplozji granatu.

Klasa Interactable – Skrypt **Interactable** służy jako klasa podstawa dla wszystkich obiektów do oddziaływanego. Główne funkcje obejmują:

- **Metoda OnLook:** Udostępnia tekst komunikatu o interakcji.
- **Metoda BaseInteract:** Obsługuje podstawową logikę interakcji.
- **Metoda Interact:** Metoda abstrakcyjna do konkretnej logiki interakcji w klasach pochodnych.

Klasa LadderTrigger – Skrypt **LadderTrigger** reprezentuje interaktywną drabinę.

Główne funkcje obejmują:

- **Metoda Interact:** Przełącza stan wspinaczki gracza.
- **Metoda AttachToLadder:** Przyczepia gracza do drabiny.
- **Metoda DetachFromLadder:** Odczepia gracza od drabiny.
- **Metoda FixedUpdate:** Obsługuje logikę wspinaczki i ruchu gracza podczas korzystania z drabiny.

Klasa ShatteredGlass – Skrypt **ShatteredGlass** obsługuje rozbijanie szklanych obiektów na rozdrobnione kawałki. Główne funkcje obejmują:

- **Metoda ApplyForce:** Stosuje siłę na rozdrobnione kawałki szkła za pomocą pocisku.
- **Metoda ApplyForceFromGrenade:** Stosuje siłę na rozdrobnione kawałki szkła za pomocą eksplozji granatu.
- **Metoda DestroyObject:** Niszczy obiekt rozdrobnionego szkła po pewnym czasie.

Klasa Weapon – Skrypt **Weapon** reprezentuje interaktywną broń. Główne funkcje obejmują:

- **Metoda Update:** Monitoruje trafienia raycastów, aby określić odpowiedni komunikat o interakcji.
- **Metoda Interact:** Obsługuje podnoszenie i zarządzanie bronią w inwentarzu.
- **Coroutine DestroyAfterPickup:** Niszczy obiekt broni po jej podniesieniu.
- **Metoda SetShaderParameters:** Dostosowuje parametry shadera dla zniknięcia broni.
- **Metoda OnTriggerEnter:** Wykrywa interakcję wroga, uruchamiając podnoszenie broni dla przeciwników.

3.2.5 Mechaniki Środowiska

Klasa Skybox – Skrypt **Skybox** zarządza aspektem wizualnym środowiska. Główna funkcja skryptu to rotacja chmur Skybox w czasie.

Klasa SpawnPosition – Skrypt **SpawnPosition** organizuje losowe umieszczanie gracza i przeciwników w świecie gry. Główne funkcje obejmują:

- **Pozycja Startowa Gracza:** Losowo umieszcza gracza w dostępnych punktach teleportacji.
- **Spawn Przeciwnika:** Losowo pozycjonuje przeciwników w punktach teleportacji, zapewniając zróżnicowane lokalizacje startowe.

Skrypty **Skybox** i **SpawnPosition** wspólnie konfigurują wirtualne środowisko, kontrolując aspekty wizualne i zapewniając dynamiczne punkty startowe dla fascynującego doświadczenia gry.

3.2.6 Mechaniki Gracza

Klasa PlayerHealth – Skrypt **PlayerHealth** zarządza zdrowiem, pancerzem oraz efektami wizualnymi związanymi z graczem. Kluczowe funkcje to:

- **Aktualizacje Zdrowia i Pancerza:** Regularnie aktualizuje i ogranicza wartości zdrowia i pancerza gracza.
- **Obsługa Obrażeń:** Przetwarza różne rodzaje obrażeń (standardowe, upadkowe, od gazu, od ognia) i uruchamia odpowiednie reakcje.
- **Informacje Wizualne:** Modyfikuje efekty wizualne, takie jak intensywność vignette w zależności od procentu zdrowia.
- **Obsługa Śmierci:** Rozpoczyna sekwencję śmierci gracza z efektami dźwiękowymi i upuszczaniem przedmiotów.

Klasa PlayerInteract – Skrypt **PlayerInteract** umożliwia interakcję gracza z otoczeniem gry. Kluczowe funkcje to:

- **Promieniowanie:** Używa promieniowania do identyfikacji obiektów możliwych do zinterakcjonowania w otoczeniu.
- **Czas Odnowienia Interakcji:** Wprowadza czas odnowienia, aby zapobiec szybkim interakcjom.
- **Interakcja z Obiektami:** Uruchamia interakcje na podstawie wejścia gracza (np. naciśnięcie klawisza F).

Klasa PlayerInventory – Skrypt **PlayerInventory** zarządza inwentarzem gracza i zmianą broni. Kluczowe funkcje to:

- **Dodawanie i Usuwanie Przedmiotów:** Obsługuje dodawanie i usuwanie broni oraz przedmiotów.
- **Zmiana Broni:** Pozwala graczowi na zmianę broni przy użyciu różnych wejść.
- **Upuszczanie Przedmiotów:** Wprowadza upuszczanie broni z efektami fizycznymi.

Klasa PlayerLeaning – Skrypt **PlayerLeaning** odpowiada za obsługę mechaniki pochylenia gracza. Kluczowe funkcje to:

- **Wykrywanie Wejścia:**
 - Monitoruje wejście gracza, aby określić, czy gracz pochyla się w lewo czy w prawo (klawisze Q i E).
- **Obliczenia Pochylenia:**
 - Dostosowuje kąt pochylenia gracza na podstawie wejścia, stosując płynną interpolację dla bardziej naturalnego uczucia.
 - Aktualizuje lokalną rotację gracza, aby symulować pochylenie.

Klasa PlayerMotor – Skrypt **PlayerMotor** zarządza ruchem gracza oraz różnymi związanymi z nim mechanikami. Kluczowe funkcje to:

- **Ruch:**
 - Zbiera dane wejściowe dla ruchu gracza (klawisze W, A, S, D).
 - Rozróżnia pomiędzy chodzeniem, biegiem i kucaniem na podstawie działań gracza.

- **Kucanie:**

- Dostosowuje wysokość gracza oraz pozycję kamery podczas kucania.
- Obsługuje przełączanie pomiędzy kucaniem a staniem na podstawie wejścia.

- **Skakanie:**

- Wykrywa wejście skoku i wykonuje skoki, uwzględniając przeszkody oraz wytrzymałość gracza.

- **Grawitacja:**

- Symuluje grawitację dla gracza, uwzględniając czas spadania oraz potencjalne obrażenia z upadku.

- **Bieganie:**

- Pozwala graczowi na sprintowanie przytrzymując klawisz Shift i spełnieniu określonych warunków.

Klasa PlayerShoot – Skrypt PlayerShoot zarządza mechaniką strzelania gracza oraz interakcjami z bronią. Kluczowe funkcje to:

- **Strzelanie:**

- Obsługuje mechanikę strzelania, w tym oddawanie strzałów, zarządzanie amunicją i trajektorie pocisków.
- Zarządza trybami automatycznego i pojedynczego strzału.

- **Przeładowywanie:**

- Inicjuje i wykonuje sekwencję przeładowywania broni, uwzględniając różne typy broni.

- **Celowanie:**

- Dostosowuje widok gracza oraz pozycję broni na podstawie wejścia celowania.
- Wprowadza dynamiczne pole widzenia dla karabinów snajperskich.

- **Zmiana Broni:**

- Obsługuje zmianę broni, odtwarzając odpowiednie dźwięki.

- **Ataki Wręcz:**

- Wprowadza ataki wręcz z zużyciem wytrzymałości oraz blokowanie kolejnych ataków podczas przeładowywania.

Klasa PlayerStamina – Skrypt **PlayerStamina** zarządza wytrzymałością gracza, interfejsem użytkownika wytrzymałości oraz funkcjonalnościami związanymi. Kluczowe cechy to:

- **System Wytrzymałości:**

- Śledzi wytrzymałość gracza, w tym maksymalną wytrzymałość, koszty sprintu, skoku i ataku.
- Wprowadza regenerację wytrzymałości z upływem czasu.
- Blokuje regenerację wytrzymałości w określonych warunkach.

- **Elementy Interfejsu Użytkownika:**

- Zarządza wizualną reprezentacją paska wytrzymałości z płynnymi przejściami.
- Aktualizuje elementy interfejsu, takie jak tekst wytrzymałości i kolor w zależności od bieżącej wytrzymałości.
- Wyświetla komunikaty i odtwarza efekty dźwiękowe związane ze stanem wytrzymałości.

- **Zużycie Wytrzymałości:**

- Odbiera wytrzymałość podczas sprintu, skoku lub ataków.
- Dostosowuje elementy interfejsu zgodnie z tym oraz uruchamia blokowanie regeneracji wytrzymałości przy wyczerpaniu wytrzymałości.

Klasa PlayerStance – Skrypt **PlayerStance** definiuje różne postawy gracza i ich powiązane właściwości. Kluczowe cechy to:

- **Definicja Postawy:**

- Wymienia różne postawy gracza, w tym spoczynku, chodzenia, kucania i biegania.
- Przypisuje konkretne wartości wysokości kamery dla każdej postawy.

Klasa PlayerUI – Skrypt **PlayerUI** obsługuje elementy interfejsu użytkownika gracza, takie jak komunikaty, wyświetlacze amunicji i informacje zwrotne dotyczące interakcji. Kluczowe funkcje to:

- **Elementy Interfejsu Użytkownika:**

- Zarządza i aktualizuje elementy interfejsu, takie jak teksty komunikatów, teksty amunicji i suwak przeładowywania.
- Obsługuje dynamiczne wyświetlanie informacji o amunicji dla różnych typów broni.

- **Uzupełnianie Amunicji:**

- Inicjuje i wykonuje sekwencje uzupełniania amunicji, w tym informacje zwrotne wizualne i aktualizacje inwentarza.
- Wyświetla komunikaty dotyczące limitów granatów i ograniczeń uzupełniania amunicji.

- **Obsługa Coroutine:**

- Wprowadza korutyny dla płynnych przejść i efektów zanikania w komunikatach interfejsu.
- Zarządza czasem trwania i timingiem zanikania elementów interfejsu.

Te skrypty wspólnie przyczyniają się do złożonej mechaniki gracza, zarządzania zdrowiem, interakcji z otoczeniem, obsługi inwentarza, obejmując pochylenie, ruch, strzelanie, związane z nim interakcje, zarządzanie wytrzymałością, definicje postaw gracza oraz elementy interfejsu użytkownika w grze.

3.2.7 Mechaniki Interakcji Środowiskowej

Klasa FireParticleTrigger – Skrypt **FireParticleTrigger** obsługuje aktywację i efekty cząsteczek ognia. Kluczowe funkcje to:

- **Aktywacja Cząsteczek:**

- Aktywuje cząsteczki ognia po zderzeniu z wrogiem lub graczem.
- Zarządza słownikiem (**characterVfxMap**), śledząc aktywne postaci i ich efekty wizualne.

- **Obrażenia w Czasie:**

- Zadaje obrażenia ognistwe w czasie wrogom w zasięgu cząsteczek.

- Wykorzystuje korutyny do stosowania okresowych obrażeń i efektów wizualnych.

- **Obrażenia Gracza:**

- Inicjuje i obsługuje ciągłe obrażenia ogniowe dla gracza po kolizji.
- Wykorzystuje korutyny do okresowych obrażeń gracza.

- **Oczyszczanie przy Wyłączeniu:**

- Zatrzymuje korutyny i resetuje efekty wizualne po wyłączeniu skryptu.
- Zapewnia właściwe oczyszczanie zasobów.

Klasa GasParticleTrigger – Skrypt **GasParticleTrigger** zarządza efektami cząsteczek gazu i ich wpływem na gracza. Kluczowe funkcje to:

- **Aktywacja Gazu:**

- Aktywuje okresowe obrażenia gazowe, gdy gracz wchodzi w obszar cząsteczek gazu.
- Wykorzystuje **InvokeRepeating** do ciągłego zadawania obrażeń.

- **Obrażenia w Czasie:**

- Zadaje okresowe obrażenia gazowe graczowi w obszarze gazu.
- Stosuje losowe wartości obrażeń, aby symulować zmienną intensywność gazu.

Klasa HitBox – Skrypt **HitBox** reprezentuje hitboxy na postaciach i obsługuje obliczenia obrażeń. Kluczowe funkcje to:

- **Obsługa Obrażeń:**

- Oblicza obrażenia na podstawie lokalizacji hitboxu i właściwości broni.
- Uwzględnia mnożniki obrażeń dla różnych obszarów hitboxu.

- **Obrażenia od Eksplozji:**

- Pozwala na bezpośrednie zadawanie obrażeń od eksplozji postaciom.
- Wykorzystuje metody **OnExplosion** do obrażeń związanych z eksplozją.

- **Obrażenia Gracza:**

- Zadaje obrażenia graczowi, jeśli spełnione są warunki hitboxu i broni.
- Uwzględnia mnożniki obrażeń i specyficzne dla broni czynniki.

Klasa Ragdoll – Skrypt **Ragdoll** zarządza aktywacją i dezaktywacją fizyki ragdoll na postaci. Kluczowe funkcje to:

- **Aktywacja Ragdolla:**
 - Aktywuje fizykę ragdoll na postaci, umożliwiając realistyczne interakcje fizyczne.
 - Wykorzystuje komponenty **Rigidbody** i wyłącza Animator.
- **Dezaktywacja Ragdolla:**
 - Dezaktywuje fizykę ragdoll, umożliwiając kontrolę animacji przez Animatora.
 - Przywraca komponenty **Rigidbody** do kinematycznego stanu i włącza Animatora.
- **Zastosowanie Siły:**
 - Stosuje zewnętrzne siły do ragdolla, symulując wpływy lub eksplozje.
 - Używa **Rigidbody Hips** do zastosowania siły.

Te skrypty wspólnie przyczyniają się do efektów środowiskowych, interakcji postaci i obliczeń obrażeń w grze.

3.2.8 Interfejs Użytkownika i Mechaniki Interakcji

Klasa DamageIndicator – Skrypt **DamageIndicator** zarządza wskaźnikami obrażeń, które pojawiają się na ekranie, aby pokazać kierunek nadchodzących obrażeń. Kluczowe funkcje to:

- **Dynamiczna Rotacja:**
 - Dynamicznie obraca wskaźnik obrażeń w kierunku źródła obrażeń.
 - Wykorzystuje korutynę (**RotateToTheTarget()**) dla płynnych aktualizacji rotacji.
- **Mechanizm Odliczania:**
 - Inicjuje mechanizm odliczania dla czasu widoczności wskaźnika obrażeń.

- Płynnie zanika i pojawia się, używając wartości alfa w **CanvasGroup**.

- **Pula Obiektów:**

- Wykorzystuje pulę obiektów do efektywnego zarządzania i ponownego użycia instancji wskaźnika obrażeń.
- Zwraca obiekt do puli po zakończeniu odliczania.

Klasa DISystem – Skrypt **DISystem** zarządza Systemem Wskaźników Obrażeń, orkestrą tworzenia i ponownego użycia wskaźników obrażeń. Kluczowe funkcje to:

- **Dynamiczne Tworzenie Wskaźników:**

- Dynamicznie tworzy wskaźniki obrażeń na podstawie celu.
- Wykorzystuje pulę obiektów do efektywnego tworzenia i zarządzania.

- **Ponowne Użycie Wskaźników:**

- Ponownie używa istniejących wskaźników, gdy ten sam cel jest ponownie trafiony.
- Resetuje odliczanie wskaźnika dla ponownie używanych instancji.

- **Integracja z Systemem Akcji:**

- Integracja z systemem akcji w celu wywołania tworzenia wskaźników obrażeń.
- Nasłuchuje akcji **CreateIndicator**.

Klasa Tracker – Skrypt **Tracker** obsługuje mechanizm śledzenia, dostarczając funkcji śledzenia przeciwników, aktualizacji wskaźników i sprawdzania warunków zwycięstwa. Kluczowe funkcje to:

- **Inicjacja Śledzenia:**

- Inicjuje śledzenie po naciśnięciu określonego klawisza (np. Z).
- Rozpoczyna korutynę śledzenia i odtwarza efekt dźwiękowy.

- **Śledzenie Przeciwników:**

- Ciągle aktualizuje kierunek śledzenia na podstawie pozycji najbliższego przeciwnika.
- Obraca wskaźnik, aby pokazać kierunek najbliższego przeciwnika.

- **Czas Odprężania i Czas Trwania:**

- Wprowadza mechanizmy czasu odprężania i czasu trwania zdolności śledzenia.
- Zarządza inicjacją, zatrzymaniem i czasem odprężania śledzenia.

- **Skanowanie Sceny:**

- Inicjuje korutynę skanowania sceny dla efektów wizualnych podczas śledzenia.
- Wykonuje wiele skanów z różnymi poziomami przezroczystości i zasięgiem.

- **Warunki Zwycięstwa:**

- Sprawdza warunki zwycięstwa na podstawie eliminacji wszystkich przeciwników.
- Aktywuje ekran zwycięstwa, gdy wszyscy przeciwnicy zostaną wyeliminowani.

Klasa WeaponWheel – Skrypt **WeaponWheel** zarządza funkcjonalnością koła broni gracza, umożliwiając wybór broni i dostarczając informacje wizualne. Kluczowe funkcje to:

- **Aktywacja Koła:**

- Aktywuje koło broni po naciśnięciu określonego klawisza.
- Wyświetla ikony broni i związane z nimi informacje.

- **Interakcja z Kołem:**

- Umożliwia interakcję z kołem za pomocą pozycji myszy i kliknięć.
- Podświetla wybraną broń i wyświetla jej szczegóły.

- **Wybór Broni:**

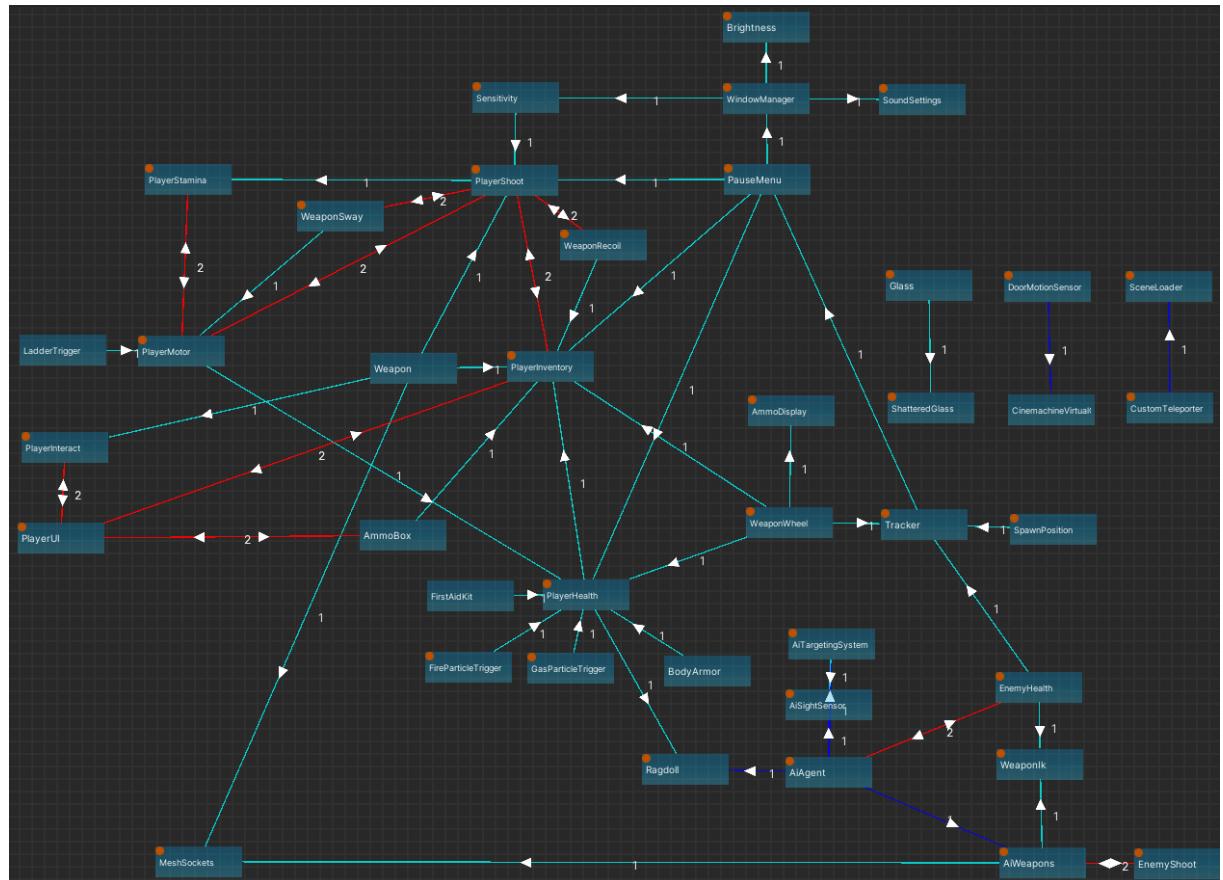
- Pozwala graczowi na wybór broni z koła.
- Wywołuje odpowiednią akcję na podstawie wybranego przedmiotu.

- **Skalowanie Czasu:**

- Dynamicznie dostosowuje skalę czasu w zależności od aktywacji koła.
- Modyfikuje dźwięk za pomocą tablicy mikserów audio.

3.3 Interakcje między modułami

W tym podrozdziale przyjrzymy się, jak poszczególne moduły gry ze sobą współpracują na podstawie załączonego diagramu. Zobaczysz, jakie interakcje zachodzą między różnymi częściami projektu, jak są przekazywane informacje oraz w jaki sposób poszczególne moduły komunikują się ze sobą, by zapewnić spójność i płynność działania gry.



Rysunek 1: Komunikacja między modułami

3.4 Wzorce projektowe

W tym podrozdziale opiszemy, w jaki sposób konkretnie wzorce projektowe są wykorzystywane w architekturze gry, jakie korzyści przynoszą, oraz jak wpływają na rozwój i utrzymanie kodu. To pozwoli Wam lepiej zrozumieć, dlaczego i w jaki sposób dany wzorzec jest stosowany w kontekście *Operation Deratization*.

3.4.1 Wzorzec szablonu metody

Wzorzec szablonu metody jest używany, gdy algorytm ma pewne kroki wspólne, ale pewne kroki są specyficzne dla konkretnych podklas. Poniżej znajduje się implementacja wzorca szablonu metody w klasie **Interactable**:

```
public abstract class Interactable : MonoBehaviour
{
    [Header("Interactables")]
    public bool useEvents;
    public string prompt;

    public virtual string OnLook()
    {
        return prompt;
    }
    public void BaseInteract()
    {
        if (useEvents)
            GetComponent<InteractionEvent>().OnInteract.Invoke();

        Interact();
    }
    protected virtual void Interact()
    {
    }
}
```

Fragment kodu 1: Klasa Interactable, z której dziedziczą wszystkie skrypty interaktywnych obiektów w grze

Wzorzec szablonu metody (*Template method pattern*): Metoda **BaseInteract** jest szablonem, a metoda **Interact** jest hookiem (hakiem), który może być nadpisywany przez klasy dziedziczące. Dzięki temu, wspólne kroki interakcji są zdefiniowane w klasie bazowej (**Interactable**), ale specyficzne zachowanie może być dostosowane przez podklasy.

3.4.2 Wzorzec stanu

Wzorzec stanu jest używany, gdy obiekt ma zmieniające się zachowanie w zależności od swojego stanu wewnętrznego. Poniżej znajduje się implementacja wzorca stanu w klasie **AiState**:

```

public enum AiStateId
{
    Death,
    Idle,
    FindWeapon,
    AttackTarget,
    FindTarget,
    FindFirstAidKit,
    FindAmmo,
    Patrol,
    InvestigateSound
}
public interface AiState
{
    AiStateId GetId();

    void Enter(AiAgent agent);
    void Update(AiAgent agent);
    void Exit(AiAgent agent);
}

```

Fragment kodu 2: Implementacja wzorca stanu w klasie AiState

Wzorzec stanu (*State pattern*): W tym przypadku, enum **AiStateId** reprezentuje różne stany, a interfejs **AiState** definiuje operacje związane z każdym stanem (**Enter**, **Update**, **Exit**). Każdy stan implementuje ten interfejs, co pozwala obiekowi AI zmieniać swoje zachowanie dynamicznie, w zależności od aktualnego stanu.

3.4.3 Wzorzec puli obiektów

Wzorzec puli obiektów został zaimplementowany w klasie **ObjectPoolManager**, która zarządza efektywnym ponownym użyciem obiektów, minimalizując koszty tworzenia i usuwania. Kluczowe funkcje obejmują dynamiczne tworzenie pul obiektów, efektywne tworzenie i zwracanie obiektów. Poniżej znajdziesz implementację wzorca pul obiektów:

```

public class ObjectPoolManager : MonoBehaviour
{
    // ... (Remaining part of the script)

    public static GameObject SpawnObject(GameObject objectToSpawn,
        Vector3 spawnPosition, Quaternion spawnRotation, PoolType
        poolType)
    {
        PooledObjectInfo pool = null;

        foreach (PooledObjectInfo p in ObjectPools)
        {
            if (p.LookupString == objectToSpawn.name)
            {
                pool = p;
                break;
            }
        }

        if (pool == null)
        {
            pool = new PooledObjectInfo() { LookupString =
                objectToSpawn.name };
            ObjectPools.Add(pool);
        }

        GameObject spawnableObj = pool.InactiveObjects.FirstOrDefault();

        if (spawnableObj == null)
        {
            GameObject parentObject = SetParentObject(poolType);
            spawnableObj = Instantiate(objectToSpawn, spawnPosition,
                spawnRotation);
            spawnableObj.transform.SetParent(parentObject.transform);
        }
        else
        {
            spawnableObj.transform.position = spawnPosition;
            spawnableObj.transform.rotation = spawnRotation;
            pool.InactiveObjects.Remove(spawnableObj);
            spawnableObj.SetActive(true);
        }

        return spawnableObj;
    }

    // ... (Remaining part of the script)
}

```

Fragment kodu 3: Klasa ObjectPoolManager odpowiedzialna za tworzenie pul obiektów

```
public class ObjectPoolManager : MonoBehaviour
{
    // ... (Remaining part of the script)

    public static void ReturnObjectToPool(GameObject obj)
    {
        string objName = obj.name.Substring(0, obj.name.Length - 7);
        PooledObjectInfo pool = null;

        foreach (PooledObjectInfo p in ObjectPools)
        {
            if (p.LookupString == objName)
            {
                pool = p;
                break;
            }
        }

        if (pool == null)
            Debug.LogWarning("Trying to release an object that is not
                            pooled: " + obj.name);
        else
        {
            obj.SetActive(false);
            pool.InactiveObjects.Add(obj);
        }
    }

    // ... (Remaining part of the script)
}
```

Fragment kodu 4: Fragment skryptu ObjectPoolManager.cs używany do zwracania obiektów do puli obiektów

Wzorzec puli obiektów (*State pattern*): W klasie **ObjectPoolManager** jest szeroko wykorzystywany do efektywnego zarządzania obiektami w grze, poprawiając wydajność i optymalizując zużycie pamięci. Jeżeli interesują Cię konkretne przykłady użycia tego wzorca zachęcamy do kliknięcia w odnośnik, który skieruje Cię do rozdziału o optymalizacji, gdzie szerzej opisaliśmy zalety stosowania tej techniki: Object Pooling

4 Interfejs użytkownika (UI) i grafika

W tej sekcji skoncentrujemy się na omówieniu kluczowych aspektów związanych z projektowaniem interfejsu użytkownika (UI) oraz doświadczenia użytkownika (UX), wraz z procesem tworzenia grafik dedykowanych do gry. Krok po kroku przewodniczyć będziemy przez projektowanie zarówno interaktywnego Menu Głównego, jak i ekranów w trakcie rozgrywki (Ingame).

Pierwszym etapem naszej analizy będzie zgłębienie szczegółów dotyczących UI i UX, identyfikując kluczowe elementy, które wpłyną pozytywnie na doświadczenie użytkownika. Starannie przyjrzymy się interaktywnym funkcjom, które mają być zawarte zarówno w Menu Głównym, jak i w trakcie samej rozgrywki, dążąc do stworzenia intuicyjnego i przyjaznego środowiska dla gracza.

Następnie skupimy się na procesie projektowania grafik, uwzględniając zarówno estetykę, jak i funkcjonalność. Przeanalizujemy style graficzne oraz czcionki, z myślą o tworzeniu spójnego i atrakcyjnego wizualnie interfejsu. Warto podkreślić, że każdy element graficzny będzie starannie dopasowany, aby nie tylko efektywnie przyciągnąć uwagę gracza, ale również ułatwić nawigację i zrozumienie dostępnych opcji.

Podsumowując, naszym celem jest stworzenie UI i UX, które nie tylko spełnią oczekiwania co do estetyki, ale przede wszystkim będą funkcjonalne i przyjazne dla gracza. Prześledzimy proces od projektowania do finalnej implementacji, dbając o każdy detal, aby zapewnić satysfakcję i wygodę użytkownikowi podczas eksploracji Menu Głównego oraz w trakcie fascynującej rozgrywki.

4.1 Projektowanie interfejsu użytkownika

W trakcie projektowania interfejsu użytkownika (UI) kluczowe jest osiągnięcie równowagi pomiędzy estetyką a funkcjonalnością. Oprócz estetycznego designu, priorytetem jest stworzenie interfejsu, który jest intuicyjny i efektywny dla użytkowników. Proces ten obejmuje systematyczne testowanie i uwzględnianie opinii, dążąc do idealnego połączenia atrakcyjnego wyglądu z praktycznym zastosowaniem, co sprawia, że interfejs nie tylko przyciąga uwagę, ale także doskonale spełnia potrzeby użytkowników.

4.1.1 Tworzenie logo naszej gry

Na wstępie skoncentrujmy się na kreowaniu głównego logo naszej gry, bowiem jest to nie tylko centralny element, ale także jedno z zadań, które mimo swojej doniosłości, należy do tych bardziej dostępnych w kwestii realizacji.

Rozpoczniemy od procesu projektowania, który skupi naszą uwagę na stworzeniu identyfikacyjnego znaku graficznego gry. Traktując to jako priorytetowe zadanie, pragniemy

nadać mu zarówno wyjątkowość, jak i łatwą rozpoznawalność. Celem jest opracowanie logo, które nie tylko odda charakter naszej gry, ale także wyróżni ją spośród innych.

W pierwszym etapie zwrócimy szczególną uwagę na koncepcję graficzną, starając się połączyć estetykę z funkcjonalnością. Zależy nam na tym, aby logo nie tylko zachwycało wizualnie, ale również komunikowało istotne elementy związane z tematyką gry. Podejdziemy do tego zadania z pełnym zaangażowaniem, starając się wykorzystać prostotę w projektowaniu, co z kolei sprawi, że logo będzie łatwo zapamiętywalne i zrozumiałe dla potencjalnych graczy.

W procesie tworzenia skoncentrujemy się na doskonałym dopasowaniu kolorów, kształtów i czcionek, aby osiągnąć optymalny efekt wizualny. Stworzymy coś nie tylko estetycznego, ale także trwałego i godnego reprezentowania gry na rynku.

W skrócie, celem jest opracowanie nie tylko logo, ale prawdziwej ikony charakteryzującej naszą grę, która będzie zarówno atrakcyjna wizualnie, jak i funkcjonalna w przekazywaniu istotnych informacji o produkcie.

Główne logo



Rysunek 2: Główne logo, w którym są nawiązania do tytułu tworzonej przez nas gry

W centralnym punkcie kompozycji wyróżnia się kontur szczura, symbolicznego odniesienia do graczy na mapie, określanych w potocznym żargonie jako "szczury". Wybór tego motywu wynika z naszej koncepcji, która nawiązuje do najbardziej niebezpiecznych postaci z różnych więzień i środowisk. W świecie gry, gracz zostaje postawiony przed zadaniem eliminacji tych "szczurów", reprezentujących najbardziej zróżnicowane i złożone przestępce postacie. Ostatecznym celem jest perfekcyjne zrealizowanie brutalnej rozgrywki, eliminując wszystkich przeciwników i upewniając się, że żaden z nich nie przeżyje starcia. Przyjęcie motywu szczura jako symbolu podkreśla nie tylko surowość środowiska przedstawionego w grze, ale także wprowadza unikalny aspekt, który stanowi istotny element narracyjny. Szczury, jako metafora

najbardziej zaciekłych przeciwników, dodają głębię fabularną oraz wywołują emocje związane z koniecznością pokonania skomplikowanego i nieprzewidywalnego przeciwnika.

W rezultacie, widoczna sylwetka szczura na głównym planie nie tylko stanowi element graficzny, ale także istotny komponent narracyjny, wzbogacający doświadczenie gracza poprzez ukazanie surowości oraz wyzwania, jakie stawia przed nim świat gry.

4.1.2 Tworzenie logo Menu głównego

Po stworzeniu głównego logo gry skoncentrujemy się na opracowaniu grafiki do menu głównego. W tym procesie zwrócimy szczególną uwagę na aspekty wizualne, aby logo doskonale wpasowało się w ogólny wygląd i tematykę gry. Naszym celem jest nie tylko estetyczna prezentacja w menu, ale także harmonijne zintegrowanie logotypu, tworząc spójną wizualną całość. Poprzez analizę detali kompozycji, kolorów i proporcji, dążymy do uzyskania efektu, który przyciągnie uwagę graczy i skutecznie przekazywać będzie charakter gry. Proces ten obejmie staranne dostosowanie grafiki do kontekstu menu, tworząc przyjemne wrażenie wizualne już na pierwszym spotkaniu gracza z grą.



Rysunek 3: Logo w głównym Menu, domyślnie jest z przezroczystym tłem (w tym przypadku tło dodane po ukończeniu logo)

Na przedstawionej grafice zauważamy, że logo zostało zręcznie wkomponowane jako litera w tytule gry. Ten subtelny zabieg nie tylko stanowi estetyczny atut dla oka, ale także wyróżnia się doskonałą czytelnością, nie dominując nad pozostałymi elementami. Stworzone w ten sposób logo idealnie komponuje się z ogólnym układem graficznym, prezentując równowagę między atrakcyjnym wyglądem a funkcjonalnością.

W trakcie tworzenia menu głównego gry planujemy skorzystać z tego logo, umieszczając je w sposób, który będzie sprzyjał spójności i łatwości odczytu dla graczy. Intencją jest stworzenie Main Menu, które nie tylko będzie wizualnie atrakcyjne, ale również praktyczne, umożliwiając płynną nawigację i dostarczając przyjemnego doświadczenia użytkownika od samego początku interakcji z grą.

4.1.3 Main Menu

Nadszedł moment, aby przystąpić do projektowania menu głównego naszej gry. W tym celu stworzyliśmy wstępny, aczkolwiek szczegółowy szkic, korzystając z popularnych narzędzi graficznych. Ten etap obejmuje opracowanie koncepcji, układu i funkcji, które będą dostępne dla gracza na etapie rozpoczęcia gry. Stworzyliśmy schematyczny rysunek, który stanowi swoiste mapowanie wizualne tego, jak zamierzamy zorganizować interakcję użytkownika z grą, uwzględniając zarówno estetyczne aspekty, jak i praktyczne potrzeby użytkowników.

W tym procesie zwracam uwagę na czytelność, intuicyjność nawigacji oraz estetyczne dopasowanie do ogólnej koncepcji gry. Wszystko to ma na celu stworzenie menu głównego, które nie tylko przyciągnie uwagę graczy swoim wyglądem, ale również zapewni im łatwą i satysfakcjonującą interakcję z grą od samego początku.

Rysunek schematyczny menu



Rysunek 4: Na powyższym obraz możemy zobaczyć bardzo uproszczony rysunek schematyczny z procesów tworzenia Menu

Po akceptacji wszystkich członków teamu możemy zacząć tworzenie naszego projektu w grze.

Gotowe Main Menu - w grze



Rysunek 5: Nasze Main Menu utworzone już w grze

Tworzenie zaczynamy od zrobienia sceny, na której ustawiliśmy dom, w którym znajduje się całość naszego Menu. Wokół postawiliśmy drzewa, które pełnią formę estetyczną, ponieważ wszystko widać przez okna a nie chcemy aby była widoczna pusta przestrzeń. Następnie w naszym budynku planujemy i tworzymy wnętrze, dodajemy kilka assetów aby nie świeciło ono pustkami. Jak możemy zobaczyć sugerowaliśmy się naszym wcześniej pokazywanym rysunkiem schematycznym. W górnej części widnieje zaprojektowane logo wraz z tytułem gry, natomiast poniżej funkcjonalne przyciski, które odpowiadają przypisanym im funkcjom. Dodatkowo pozwoliliśmy sobie dodać osobną planszę wraz z wyróżnieniem twórców gry znajdującą się pod przyciskiem "credits".

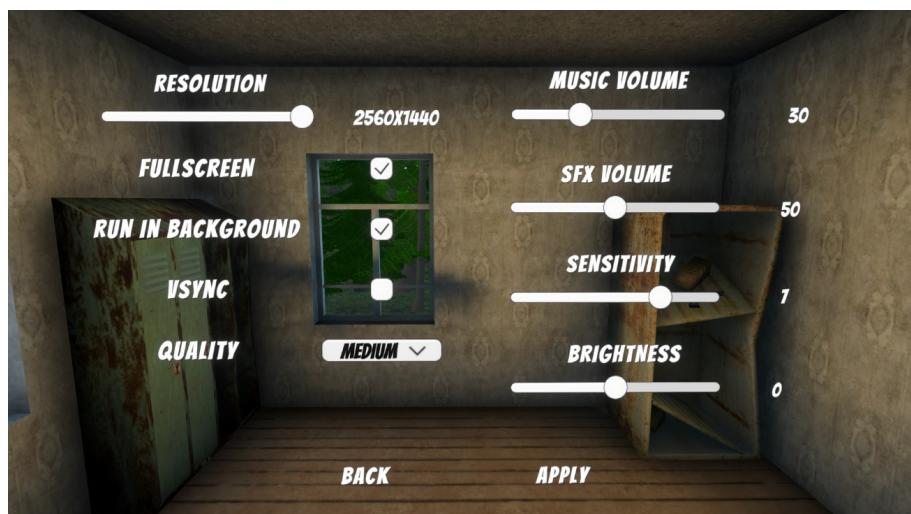
Credits - w grze



Rysunek 6: Plansza "Credits"

Jak możemy zobaczyć na powyższym obrazku plansza "Credits" zawiera informacje o autorach gry. W tle możemy zobaczyć wcześniej wspominane drzewa, które elegancko wypełniają na pustą przestrzeń na terenie przygotowanym pod Main Menu, w górnej części możemy zobaczyć również logo, które występuje w głównej planszy Menu. Poniżej znajdują się zdjęcia wraz z danymi autorów oraz w postaci dodatkowego smaczka mamy także przypisane do nich odpowiednie role przy tworzeniu gry. Przyciskiem "Back" znajdującym się na dole strony wracamy do głównej planszy naszego menu, gdzie możemy wejść w opcje naszej gry bądź z niej wyjść.

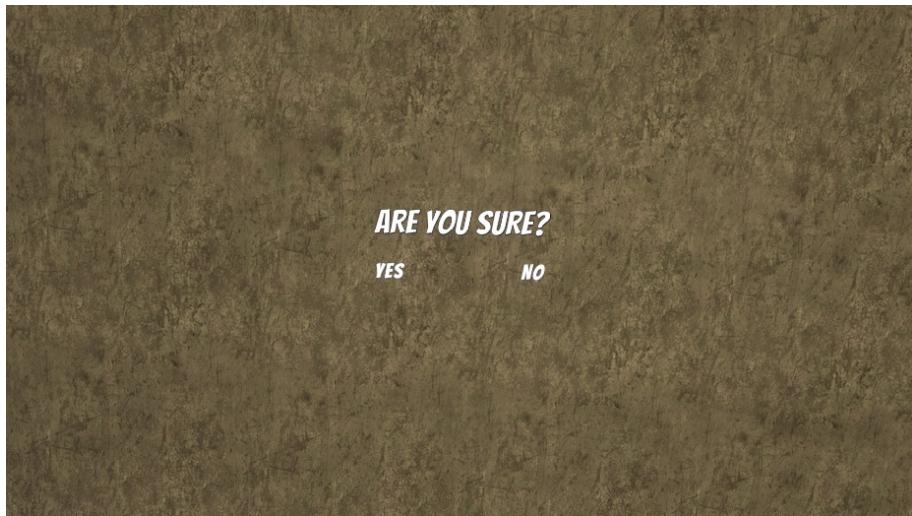
Options - w grze



Rysunek 7: Plansza "Options"

Kolejną pozycją na naszej liście przycisków są opcje gry. Tutaj możemy zmienić wiele aspektów naszej gry, począwszy od wersji graficznej, przechodząc do akustycznej i kończywszy na indywidualnych preferencjach ustawień takich jak czułość czy używanie pełnego ekranu. Poniżej znajdują się dwa przyciski, po naciśnięciu przycisku *Apply* wyskakuje nam okienko potwierdzenia (czy na pewno chcemy zapisać ustawienia gry), oraz po naciśnięciu przycisku *Back*, tak jak w poprzednim przypadku cofamy się do głównej planszy naszego Menu.

Quit - w grze



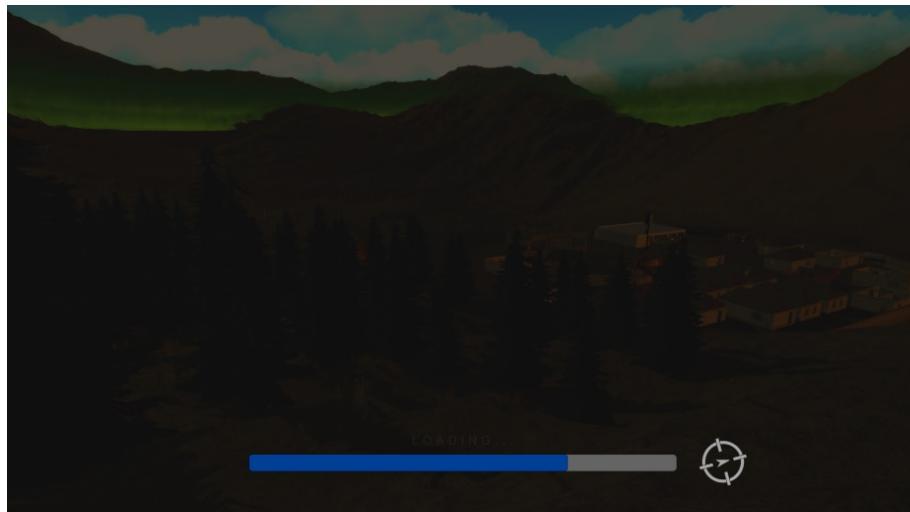
Rysunek 8: Plansza "Quit"

Jest to jedna z najprostszych i najbardziej podstawowych plansz w naszym Menu głównym. Jej zadaniem jest upewnić się czy gracz jest pewien swojej decyzji w stu procentach. Znajdują się na niej dwa przyciski *Yes* oraz *No*, jak możemy się domyślać po wciśnięciu przycisku po lewej stronie, nasza gra zostanie zamknięta i przejdziemy do swojego pulpitu. Ciekawostka ten ekran także jest wykorzystywany do zapisywania ustnień z planszy "Options", o której mowa była we wcześniejszym opisie.

4.1.4 Loading Screen

Jeżeli chodzi o sam "*Loading Screen*", pojawiającym się np. pomiędzy Menu a samą grą. Rozpoczeliśmy od przeglądnięcia ekranów wczytywania z rozmaitych gier i na tej podstawie stworzyliśmy nasz autorski projekt.

InGame Loading Screen



Rysunek 9: Loading Screen po zaprojektowaniu i wykonaniu

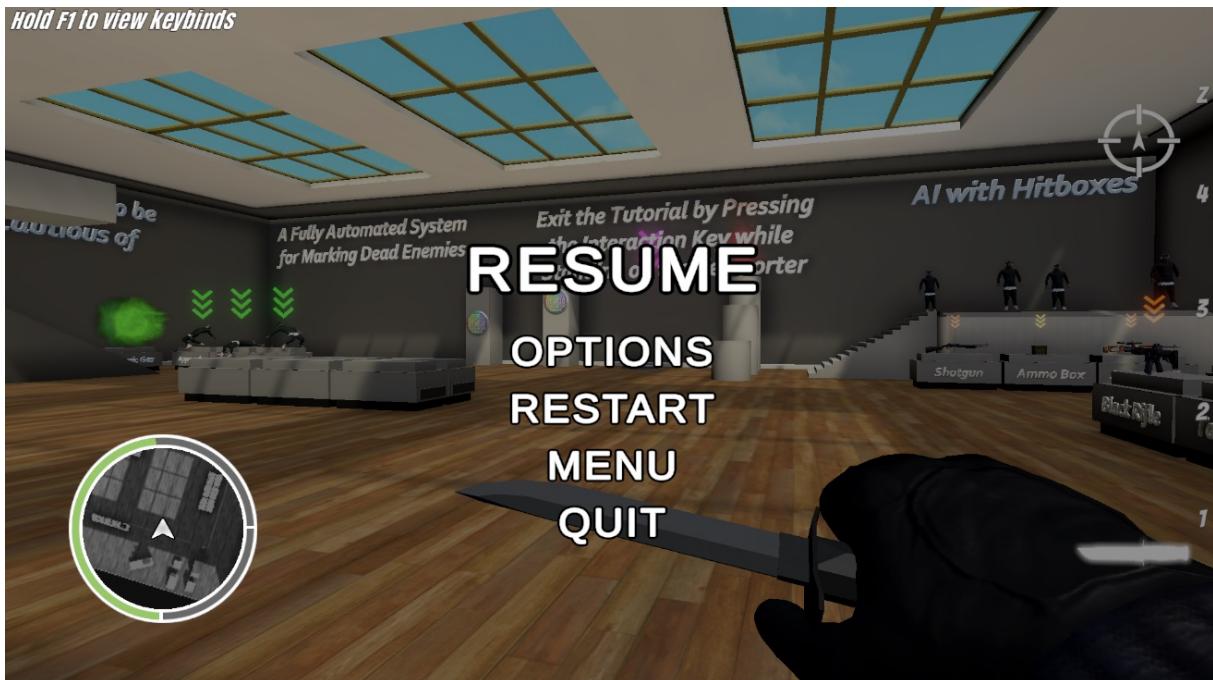
Jak możemy zobaczyć jest to jeden z podstawowych Loading Screenów. Znajduje się na nim pasek (w tym przypadku w trakcie ładowania, po prawej od paska oraz nad nim znajdują się animowany napis oraz animowana ikona naszego Trackera. W tle dodatkowo przedstawiona jest mapa, na której możemy zobaczyć przedsmak tego co się znajduje w głównej grze.

4.1.5 InGame Pause Menu

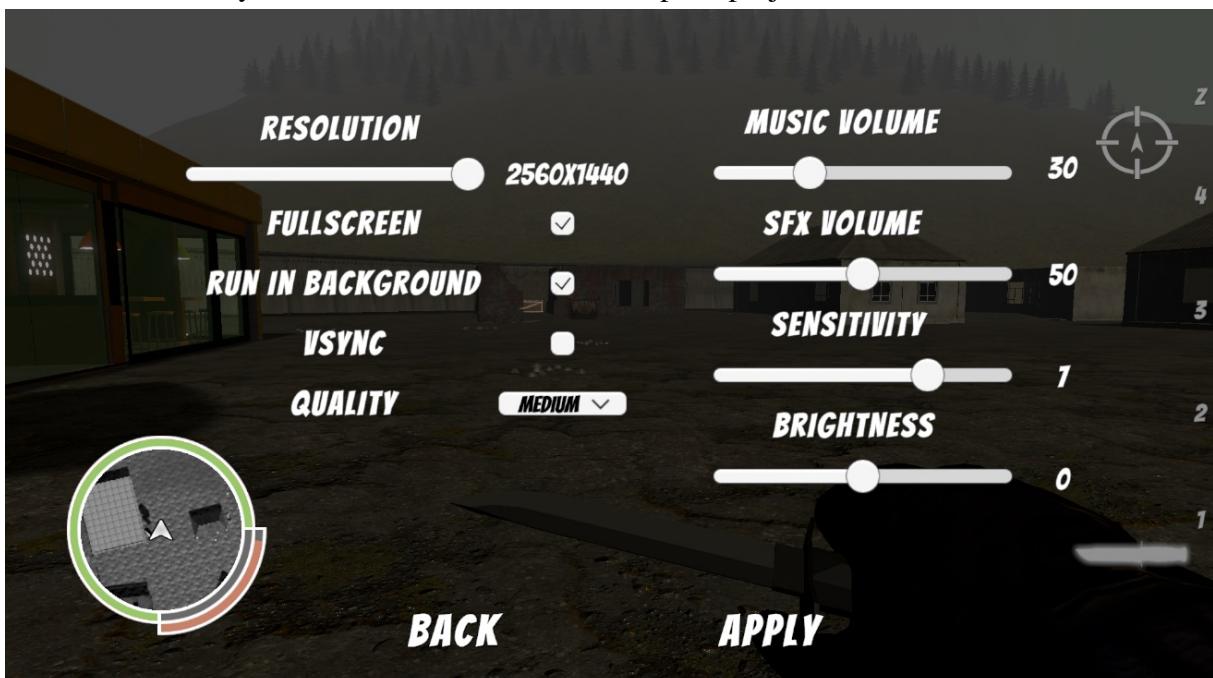
Istotnym elementem w doświadczeniu gracza jest "InGame Pause Menu", które można otworzyć za pomocą przycisku ESC. To istotne narzędzie zapewniające dostęp do szeregu opcji bezpośrednio w trakcie rozgrywki. W ramach tego menu, gracz ma możliwość wyjścia do menu głównego, zrestartowania rozgrywki oraz wejścia do "Ingame Options Menu". Ostatnie z wymienionych menu umożliwia dokonywanie zmian w ustawieniach gry, bez konieczności przerywania aktualnej rozgrywki.

Warto jednak zauważyć, że po wejściu do "InGame Pause Menu" gra może ulec chwilowemu zatrzymaniu, co zapewnia bezpieczne i spokojne środowisko do dokonywania zmian w ustawieniach bez pośpiechu. Ten krótkotrwały "freeze" gry jest integralną częścią procesu i zapewnia stabilność operacji dokonywanych w trakcie korzystania z "InGame Pause Menu". Dzięki tej funkcji, gracz ma pewność, że wszelkie zmiany w ustawieniach odbywają się w kontrolowany sposób, nie zakłócając płynności rozgrywki.

InGame Pause Menu



Rysunek 10: InGame Pause Menu po zaprojektu i utworzeniu



Rysunek 11: InGame Options Menu po zaprojektu i utworzeniu

4.1.6 Grafiki wykorzystane do Cutscenek

Wspomniane CutSceny, czyli sekwencje filmowe w grach, stanowią niezwykle istotny element w kreowaniu bogatego narracyjnego doświadczenia. Na specjalne życzenie jednego z naszych animatorów, podjęliśmy się stworzenia kilku "memowych grafik", które stanowią kreatywną odmianę tego konwencjonalnego aspektu. Poniżej znajdują się zdjęcia przedstawiające te unikatowe grafiki, które zostały zrealizowane z myślą o dodaniu elementu humorystycznego i nieformalnego charakteru do świata gry.

Rozwinięta kreatywność w obszarze CutScen sprawia, że gracze mogą doświadczać nie tylko bogatej narracji, ale także elementów zabawy i humoru. Memowe grafiki stanowią oryginalne podejście do tego, co często bywa traktowane poważnie, dodając unikalny, lekki akcent do animacji. To przykład, jak z dbałością o detale podejście do tworzenia rozrywki może wprowadzić nową, niekonwencjonalną jakość do gier.

Cutscenes graphic



Rysunek 12: Grafiki użyte w cutscenach

4.2 Grafiki interfejsu

Przechodzimy teraz do etapu przygotowywania grafiki interfejsu gracza. W tej fazie skupiamy się na opracowywaniu elementów wizualnych, które będą bezpośrednio wpływać na doświadczenie użytkownika podczas interakcji z grą. Wykorzystujemy różnorodne narzędzia graficzne, aby stworzyć estetyczne, funkcjonalne elementy, obejmujące wszystko, od ikon po przyciski i panele.

Nasz cel to nie tylko stworzenie atrakcyjnego wizualnie interfejsu, ale także dostosowanie go do ergonomicznych potrzeb graczy, zapewniając czytelność, intuicyjność i efektywność w obszarze interakcji. Grafiki interfejsu gracza będą nie tylko ozdobą, ale przede wszystkim praktycznym narzędziem, które ułatwi użytkownikom korzystanie z różnych funkcji gry, podnosząc tym samym jakość ich doświadczenia.

4.2.1 Przygotowanie listy grafik do stworzenia

Przygotowywanie listy grafik do stworzenia to kluczowy krok w procesie projektowania. W tym etapie identyfikujemy i szczegółowo opisujemy wszelkie grafiki, które będą niezbędne do kompleksowego stworzenia wizualnej struktury projektu. Lista ta obejmuje różnorodne elementy, takie jak logo, ikony interfejsu, tła, animacje, czy też wszelkie inne grafiki, które są istotne dla spójnego wizualnego doświadczenia użytkownika.

W trakcie tworzenia tej listy uwzględniamy zarówno elementy dekoracyjne, jak i funkcjonalne, dbając o to, aby każdy element pełnił określona rolę w kontekście użytkowego aspektu projektu. Sprecyzowane opisy pomagają zrozumieć, jakie oczekiwania mamy co do każdej grafiki, a także ułatwiają koordynację pracy zespołu, dając klarowny plan dla artystów grafików i projektantów.

Lista grafik do stworzenia

1. Ikona broni głównej - M4 - ON/OFF
2. Ikona broni głównej - Shotgun - ON/OFF
3. Ikona broni głównej - Snajperka SVD - ON/OFF
4. Ikona broni bocznej - Pistolet Beretta - ON/OFF
5. Ikona broni bocznej - Revolver - ON/OFF
6. Ikona granatu wybuchowego - HE - ON/OFF
7. Ikona granatu błyskowego - Flashbang - ON/OFF
8. Ikona granatu dymnego - Smoke - ON/OFF
9. Ikona granatu zapalającego - Molotov - ON/OFF
10. Ikona Trackera
11. Strzałka Trackera
12. Strzałka Minimapy
13. Ikona noża - ON/OFF
14. Ikona indyktora obrażeń
15. Ikona koła do maskowania minimapy
16. Ikona Indykatora granatu

Łącznie 26 ikon do stworzenia

Rysunek 13: Lista ikon do stworzenia

4.2.2 Poszczególne kroki tworzenia ikon

Tutaj utworzymy ikonę, która będzie się wyświetlać podczas trzymania broni/granatu w ręku.

1. Robimy screena prefabu wybranej broni. W tym przypadku Molotov.



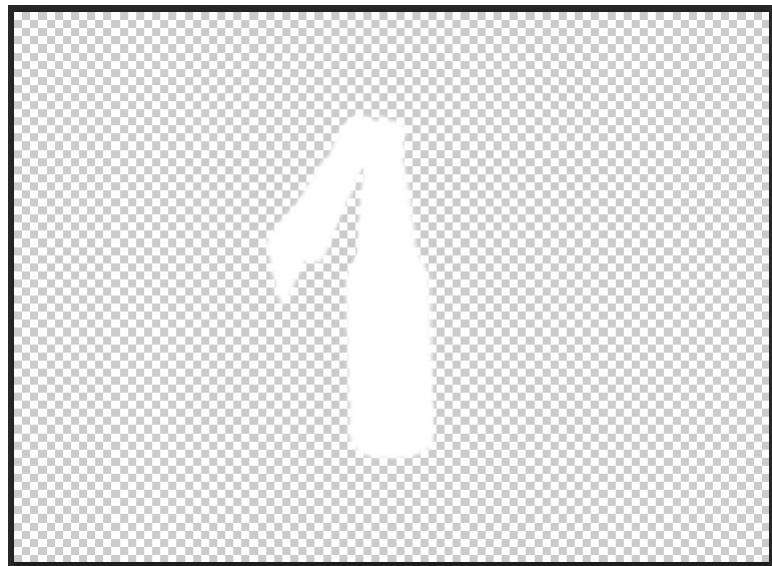
Rysunek 14: Screen prefabu

2. Wycinamy tło molotova, tak aby było przezroczyste.



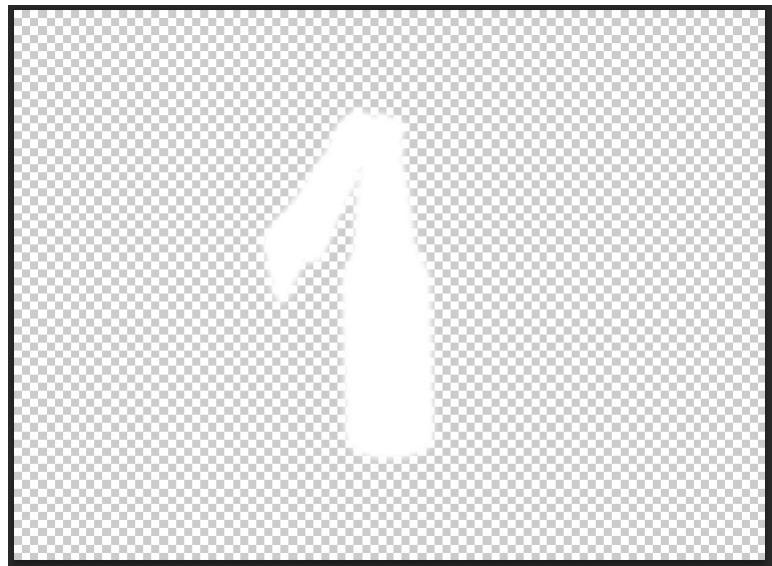
Rysunek 15: Usunięcie tła

3. Nakładamy na naszego Molotova biały kolor.



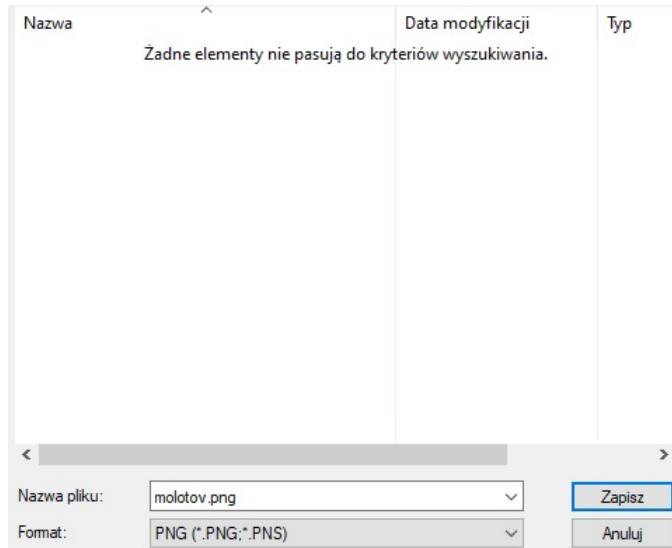
Rysunek 16: Zmiana koloru na biały

4. W tym momencie nadajemy blask zewnętrzny grafice.



Rysunek 17: Dodanie blasku zewnętrznego

5. Eksportujemy grafikę jako PNG.



Rysunek 18: Eksport grafiki

Teraz utworzymy ikonę broni/granatu, która będzie się wyświetlała gdy posiadamy dany sprzęt bojowy w ekwipunku, ale nie trzymamy go w rękach. W tym przypadku powtarzamy **krok 1** oraz **krok 2**. Jeśli wykonamy poprzednie kroki możemy przystąpić do kolejnych.

1. Nakładamy na naszego Molotova szary kolor.



Rysunek 19: Zmiana kolory na szary

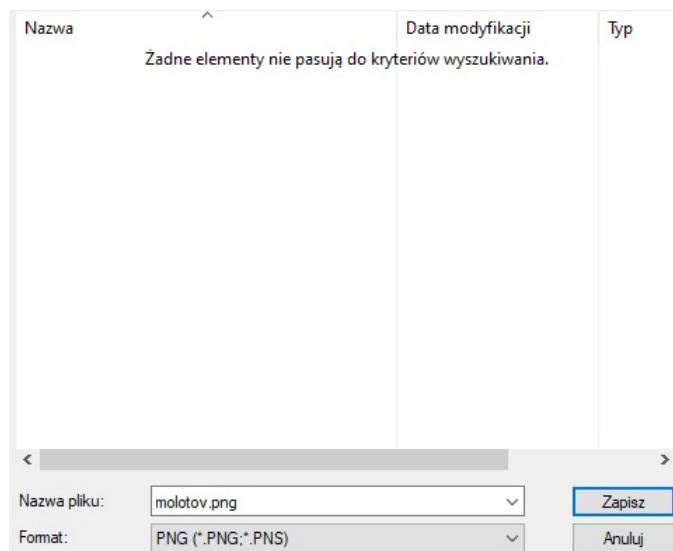
2. Zmieniamy jego przezroczystość wedle upodobań.



Rysunek 20: Zmiana przezroczystości

Jak możemy zauważyc nasze "przezroczyste" tło widać przez naszą ikonę, co oznacza, że nasza praca została wykonana pomyślnie.

3. Eksportujemy grafikę jako PNG.



Rysunek 21: Eksport grafiki

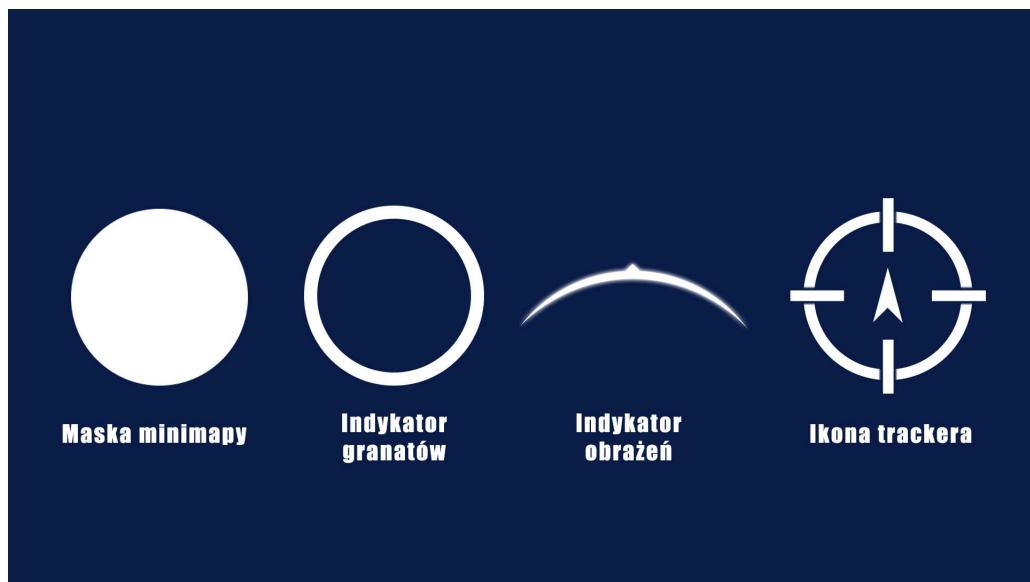
Kolejnym krokiem jest powtórzenie tego samego procesu dla pozostałych ikon broni oraz granatów. W analogiczny sposób planujemy i tworzymy grafiki reprezentujące różne rodzaje broni oraz elementy wyposażenia, dbając o spójność wizualną z wcześniej opracowanymi elementami. Długość procesu oraz staranność w przygotowywaniu każdej ikony są kluczowe, aby zagwarantować zróżnicowanie i czytelność, jednocześnie zachowując estetyczny charakter całego zestawu. Ostatecznym celem jest stworzenie kompleksowego zestawu ikon, które będą jednoznacznie identyfikować różne elementy związane z uzbrojeniem w grze.

4.2.3 Pokaz wszystkich granatów oraz broni zrobionych do gry



Rysunek 22: Plansza przedstawiająca wszystkie ikony broni oraz granatów w grze

4.2.4 Pokaz wszystkich ikon UI zrobionych do gry



Rysunek 23: Plansza przedstawiająca wszystkie ikony broni oraz granatów w grze

4.3 Integracja grafiki z interfejsem

Integracja grafiki z interfejsem w Unity 3D obejmuje proces łączenia elementów wizualnych, takich jak obrazy, tekstury, czy modele 3D, z interfejsem użytkownika (UI) gry. Poniżej przedstawiam ogólny opis tego procesu:

1. **Utworzenie Interfejsu Użytkownika (UI):** Zaczni od stworzenia elementów interfejsu użytkownika, takich jak przyciski, teksty, obrazy, paski postępu, itp. W Unity, możesz użyć narzędzi dostępnych w systemie UI, takich jak Canvas, Image, Text itp.
2. **Import Grafiki:** Przygotuj grafiki, które chcesz zintegrować z interfejsem. To mogą być pliki PNG, JPEG, czy inne formaty obsługiwane przez Unity. Importuj je do projektu Unity i umieść w odpowiednich folderach.
3. **Umieszczanie Grafiki na Canvase:** Przeciągnij i upuść grafiki na odpowiednie elementy UI, takie jak Image dla obrazków, czy Text dla tekstów. Upewnij się, że elementy graficzne są odpowiednio umieszczone na Canvase, aby skonfigurować układ interfejsu.
4. **Konfiguracja Grafiki w Edytorze:** W Edytorze Unity możesz dostosowywać właściwości grafik, takie jak rozmiar, kolor, przezroczystość, czy inne parametry. Edytor dostarcza intuicyjne narzędzia do manipulowania grafikami bez konieczności korzystania z zewnętrznych programów graficznych.
5. **Animacje i Efekty Specjalne:** Jeśli chcesz dodać animacje lub efekty specjalne do grafik w interfejsie, możesz korzystać z komponentu Animator lub użyć skryptów w języku programowania obsługiwany przez Unity, takim jak C#.
6. **Dostosowanie Interakcji:** Jeśli interakcje z elementami graficznymi mają być obsługiwane przez skrypty, napisz odpowiednie skrypty, aby reagować na interakcje gracza, takie jak kliknięcia myszą czy dotknięcia na ekranie dotykowym.
7. **Testowanie na Różnych Rozdzielczościach:** Przetestuj interfejs na różnych rozdzielczościach, aby upewnić się, że elementy graficzne skalują się poprawnie i są czytelne na różnych urządzeniach.
8. **Optymalizacja:** Dostosuj grafiki pod kątem optymalizacji, aby zoptymalizować wydajność gry, zwłaszcza jeśli planujesz publikację na różnych platformach.

Integracja grafiki z interfejsem w Unity wymaga uwzględnienia aspektów wizualnych, funkcjonalności oraz dostosowania do potrzeb użytkownika, co pozwoli uzyskać atrakcyjny i skuteczny interfejs gry.

5 Animacje i efekty wizualne

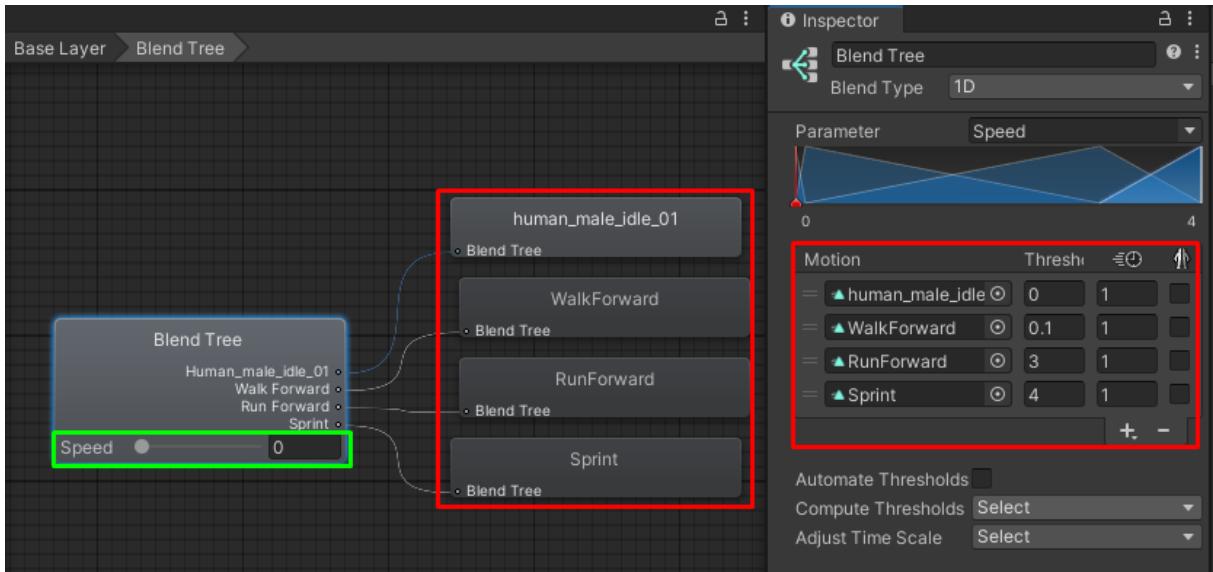
W tym rozdziale zostaną omówione kluczowe aspekty związane z animacją postaci i obiektów w grze, specjalnymi efektami wizualnymi, oświetleniem oraz zastosowaniem shaderów z przykładami prostych shader graphów.

5.1 System animacji postaci i obiektów

W świecie gier komputerowych, animacje postaci oraz obiektów odgrywają kluczową rolę w kształtowaniu wrażeń gracza. Optymalne i realistyczne animacje są nieodzowne dla zapewnienia płynnej rozgrywki oraz ułatwienia zanurzenia się w wirtualnym środowisku. W tej sekcji omówione zostaną zaawansowane algorytmy stosowane do animowania postaci, takie jak Blend Trees, Inverse Kinematics (IK) oraz Ragdoll physics oraz animacje interaktywnych obiektów występujące w grze.

5.1.1 Algorytmy wykorzystywane do animowania postaci

Blend Trees: Technika Blend Trees jest wykorzystywana w animacjach postaci, umożliwiając płynne przejścia między różnymi animacjami w zależności od czynników takich jak ruch postaci, skok czy interakcje z otoczeniem. W grze jest stosowana dla osiągnięcia bardziej naturalnych i realistycznych animacji postaci. Modyfikując w tym przypadku parametr *Speed* oznaczony zielonym prostokątem możemy zobaczyć jak poszczególne animacje są przełączane w zależności od jego wartości. Prostokątem oznaczonym kolorem czerwonym po prawej zaznaczyliśmy miejsce gdzie możemy wybrać nasze animacje, jego poziom progowy do przejścia oraz prędkość odtwarzania animacji i ewentualne jej odbicie lustrzane.



Rysunek 24: Blend Tree dla płynnego przechodzenia między prędkościami poruszania dla AI

Inverse Kinematics (IK): Algorytm IK to technika matematyczna wykorzystywana w animacji komputerowej do określania pozycji stawów kości w celu osiągnięcia pożądanej pozycji końcowej. W grze IK jest używane do uzyskania bardziej realistycznych ruchów postaci poprzez dynamiczne dostosowywanie rotacji kości. Poniżej znajduje się fragment kodu (*code snippet*), który wykorzystuje IK do dynamicznego dostosowywania rotacji określonych kości postaci przeciwnika, w celu precyzyjnego celowania w kierunku gracza:

```
public class WeaponIk : MonoBehaviour
{
    // ... (Remaining part of the script)

    private void LateUpdate()
    {
        if (aimTransform == null)
            return;
        if (targetTransform == null)
            return;

        Vector3 targetPosition = GetTargetPosition();

        for(int i = 0; i < iterations; i++)
            for(int j = 0; j < boneTransforms.Length; j++)
            {
                Transform bone = boneTransforms[j];
                float boneWeight = humanBones[j].weight * weight;
                AimAtTarget(bone, targetPosition, boneWeight);
            }
    }

    // ... (Remaining part of the script)
}
```

Fragment kodu 5: Klasa WeaponIk służąca do realistycznego symulowania ruchów kości podczas korzystania z broni przez AI

Ragdoll physics: Algorytm Ragdoll physics symuluje realistyczne reakcje postaci na siły zewnętrzne. W grze jest używany do nadawania postaci dynamicznych właściwości fizycznych po trafieniu lub upadku, co prowadzi do naturalnych animacji. W poniższym fragmencie skryptu odpowiadającym za fizykę ragdoll, zaimplementowano funkcje umożliwiające aktywację i dezaktywację ragdolla, co pozwala na dynamiczną zmianę zachowania postaci w zależności od sytuacji w grze. Dodatkowo, istnieje możliwość zastosowania sił zewnętrznych:

```
public class Ragdoll : MonoBehaviour
{
    // ... (Remaining part of the script)

    public void ActivateRagdoll()
    {
        foreach (var rigidBody in rigidBodies)
            rigidBody.isKinematic = false;

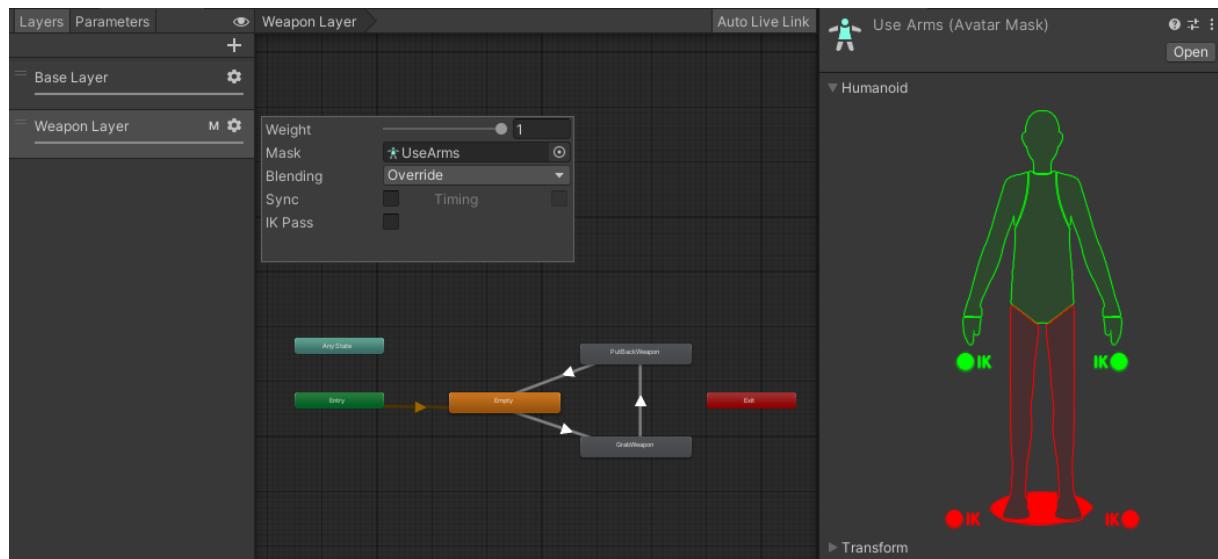
        if (animator != null)
            animator.enabled = false;
    }

    public void ApplyForce(Vector3 force)
    {
        var rigidBody = animator.
            GetBoneTransform(HumanBodyBones.Hips).
            GetComponent<Rigidbody>();
        rigidBody.AddForce(force, ForceMode.VelocityChange);
    }
}
```

Fragment kodu 6: Skrypt Ragdoll odpowiedzialny za symulowanie realistycznych reakcji postaci na siły zewnętrzne

5.1.2 Animator przeciwników

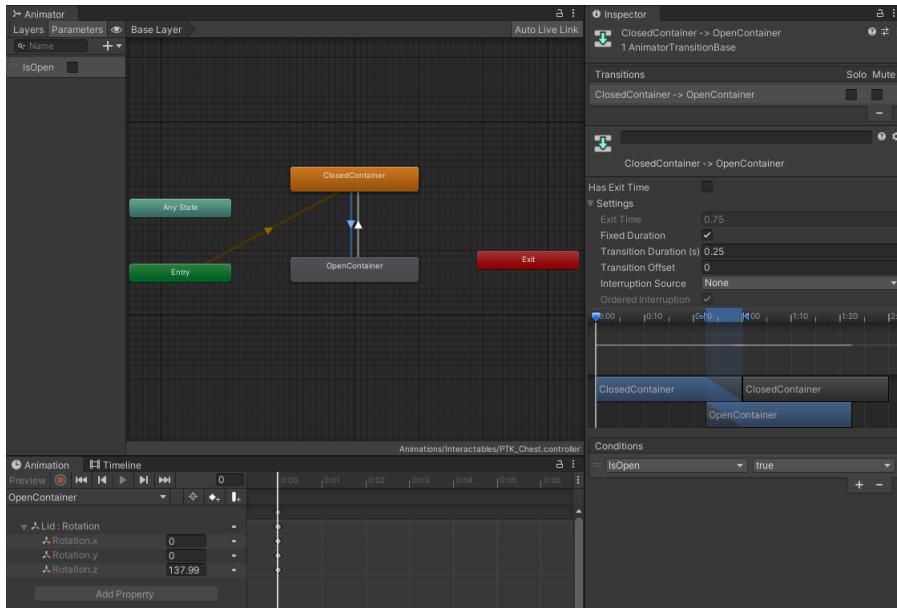
W tej sekcji przedstawiono użycie animatora przeciwników, gdzie wykorzystano warstwę broni wraz z zastosowaniem Avatar Mask. Avatar Mask został użyty do nadpisania animacji dla konkretnych części ciała przeciwnika. W tym przypadku zmienione zostało zachowanie głowy, tułowia oraz rąk podczas odtwarzania animacji na warstwie broni. Celem było zachowanie ruchu nóg z warstwy domyślnej do poruszania się, ale dostosowanie reszty kości do animacji wyciągania/trzymania/chowania broni.



Rysunek 25: Warstwa broni Animatora przeciwników i zastosowanie Avatar Maska

5.1.3 Animator otwieralnych/zamykalnych obiektów

W tej sekcji znajduje się animator otwieralnych i zamykanych obiektów, z ilustracją przykładowej skrzynki. W animatorze umieszczono dwie proste animacje: jedną z otwartą skrzynką, a drugą z zamkniętą. Zastosowano również parametr bool, który jest sterowany z poziomu kodu. Parametr ten działa na zasadzie toggla, co oznacza, że przy zmianie stanu otwiera lub zamyka skrzynkę, co skutkuje odtworzeniem odpowiedniej animacji.



Rysunek 26: Animator otwieralnego oraz zamkialnego obiektu na przykładzie skrzynki

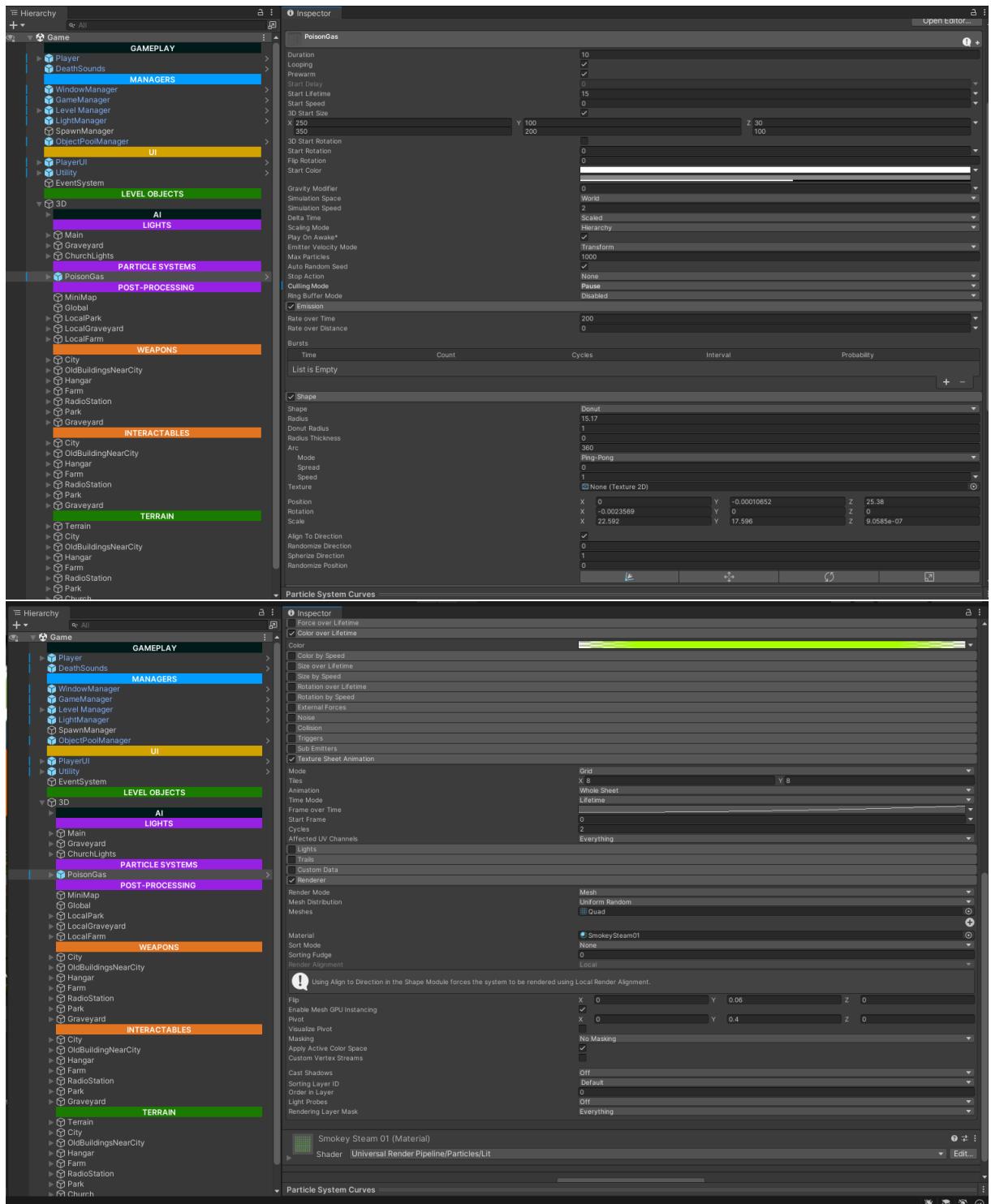
5.2 Specjalne efekty wizualne

W celu zwiększenia atrakcyjności wizualnej naszej rozgrywki zdecydowaliśmy się na wplecenie wielu elementów VFX czy systemów cząsteczkowych. Ich wykorzystanie w znacznej części powiązane jest z mechanikami, takimi jak granaty:

- Dymny - po wyrzuceniu go przez gracza następuje odliczanie, które po osiągnięciu wartości 0 wywołuje funkcję **ObjectPoolManager.SpawnObject**, która tworzy w miejscu granatu VFX dymu
- Podpalający (koktajl Mołotova) - po rozbiciu obiektu w jego miejscu pojawia się system cząsteczkowy, który wraca do puli obiektów po upływie czasu trwania oraz za pomocą VFX grafu nakłada na trafionych oponentów efekt podpalenia

5.2.1 Particle System

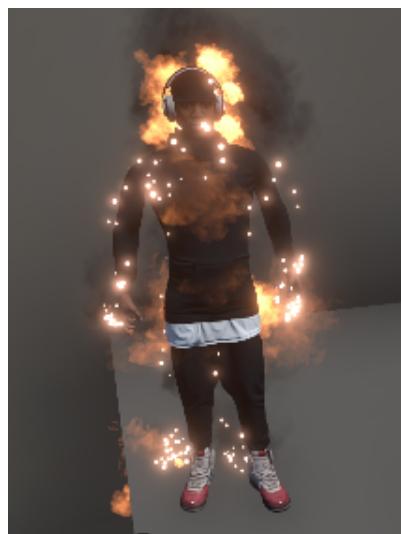
Największym zauważalnym jego zastosowaniem jest otaczający obszar rozgrywki gaz. Do jego stworzenia użyty została tekstura przykładowego gazu zaprezentowanego w paczce "**Particle Pack**" dostarczona przez Unity Technologies. Sam Particle System posiada ograniczoną ilość generowanych cząsteczek (max 1000), został zapętlony by utrzymywał się podczas całej rozgrywki. Ze względu na złożoność i sporą ilość rozmaitych ustawień w obrębie samego Particle Systemu, zdecydowanie łatwiej jest przedstawić konfigurację za pomocą zdjęć:



Rysunek 27: Przedstawienie konfiguracji gazu

5.2.2 VFX

- **Efekt podpalenia postaci:** Ten efekt pojawia się, gdy przeciwnik wejdzie w obszar pokryty ogniem z koktajlu Mołotowa. Postać staje się ognistą, co nie tylko wprowadza elementy wizualne, ale także może wpływać na mechanikę rozgrywki, dodając elementy takie jak obrażenia od ognia lub zmienione zachowanie postaci w płomieniach.



Rysunek 28: Efekt podpalenia w grze

- **Efekt dymu:** Ten efekt aktywuje się w momencie użycia granatu dymnego. Tworzy gęsty dym, który może pełnić różne funkcje w grze, takie jak ukrywanie ruchów gracza przed przeciwnikami, zmiana taktyki lub tworzenie warunków do uniknięcia wykrycia. Jest to ważny element taktyczny wpływający zarówno na rozgrywkę, jak i na aspekty wizualne.



Rysunek 29: Efekt dymu w grze

5.3 Oświetlenie

Oświetlenie odgrywa kluczową rolę w tworzeniu atmosfery i nastroju w grze. W naszym projekcie skupiamy się na zróżnicowanym wykorzystaniu oświetlenia, dostosowanym do różnych lokacji i sytuacji. Duża ilość oświetlenia znajduje się np. na cmentarzu, gdzie atmosfera musi być odpowiednio tajemnicza i klimatyczna.

5.3.1 Dynamiczne oświetlenie

Dynamiczne oświetlenie zostało zastosowane w głównej scenie gry w celu symulacji zmiany cyklu dnia i nocy. Szczególnie wykorzystane do tworzenia efektu realistycznego oświetlenia w różnych momentach rozgrywki. Więcej informacji na temat oświetlenia dynamicznego oraz jego wpływu na atmosferę gry można znaleźć w sekcji Mechaniki Środowiska.

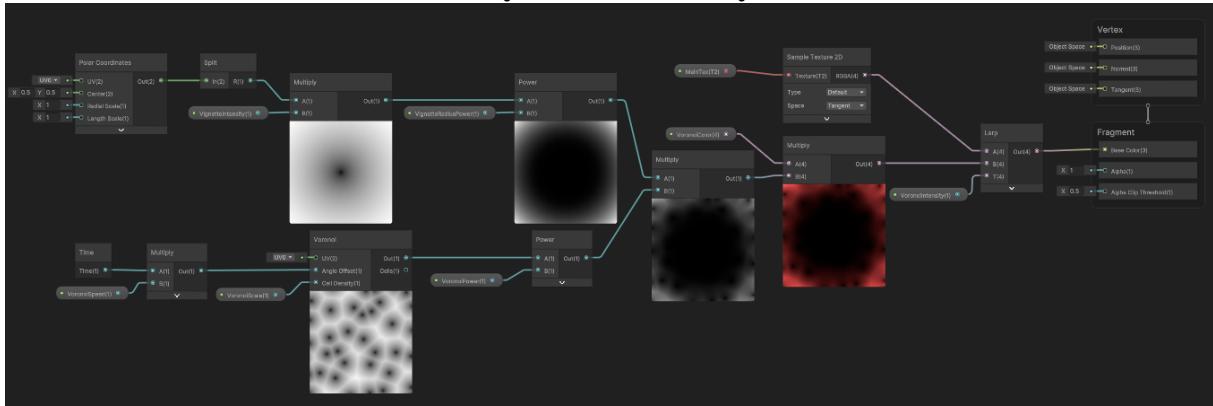
5.3.2 Wypieckone oświetlenie

Wypieckone oświetlenie jest używane dla światel, które nie wymagają dynamicznej zmiany w czasie rzeczywistym, co zdecydowanie poprawia optymalizację gry. Proces ten, znany jako wypiekanie światel, pozwala na wcześniejsze przygotowanie oświetlenia dla statycznych scen, co przyczynia się do wydajniejszego renderowania. Aby dowiedzieć się więcej o wypiekanych oświetleniu, zobacz sekcję Wypiekanie Światel.

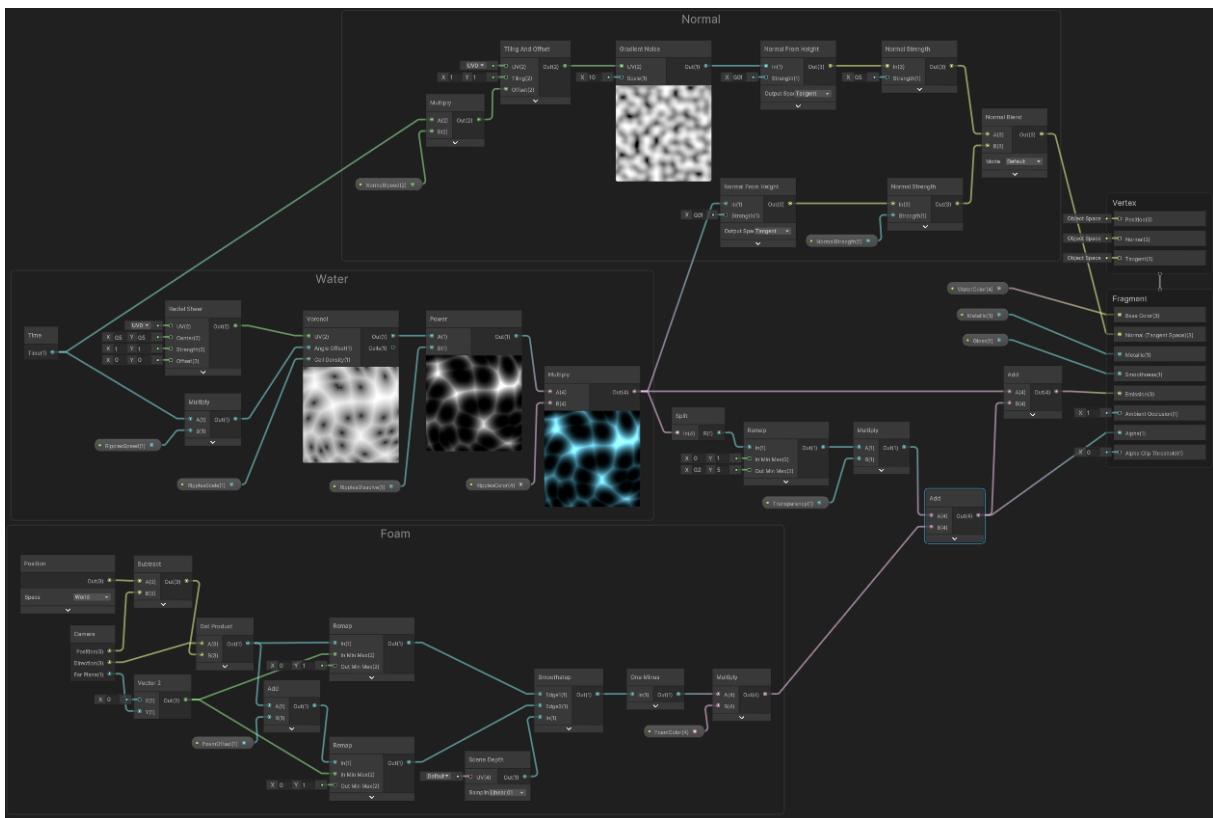
5.4 Zastosowanie shaderów

W projekcie wykorzystano szereg zaawansowanych shaderów, które mają istotny wpływ na wizualną i atmosferyczną stronę gry. Poniżej znajdują się przykłady kilku zastosowanych shaderów. Warto zauważyć, że są to jedynie podstawowe i najprostsze przykłady użycia shaderów w naszym projekcie, ale używamy również bardziej skomplikowanych (np. do stworzenia okręgu z możliwością wyboru jego koloru czy ilości i odstępów jego segmentów lub do stworzenia fali, która porusza się po otaczającym nas terenie i ma za zadanie prezentować wizualnie wykorzystanie w grze trackera).

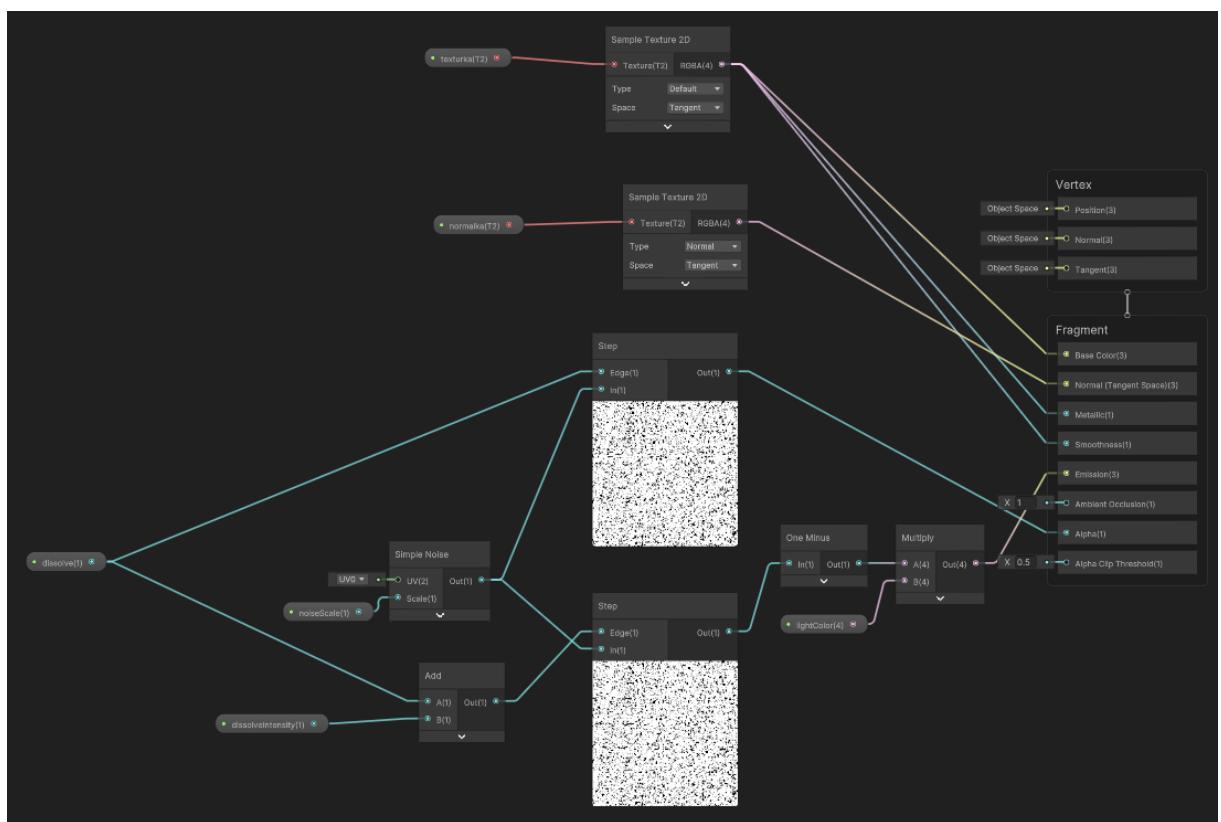
Przykładowe Shadery:



Rysunek 30: Shader winiety odpowiedzialnej za wyświetlanie na ekranie gracza informacji o ilości utraconego życia



Rysunek 31: Shader wody, którą znajdziemy w fontannie znajdującej się na terenie parku



Rysunek 32: Prostsza wersja shadera zanikania, używana w przypadku podniesienia interaktywnych obiektów takich jak kamizelka kuloodporna, apteczka czy granaty

6 Poziomy (Level design)

Rozdział poświęcony poziomom stanowi kluczową część procesu tworzenia gier komputerowych, wpływając na doświadczenie gracza, narrację gry oraz ogólny poziom zabawy. W tym rozdziale omówione zostaną fundamentalne aspekty związane z projektowaniem poziomów.

6.1 Planowanie i projektowanie poziomów

Przed rozpoczęciem prac nad poziomem w celu uniknięcia konieczności poprawek oraz gruntownych przebudów poziomu został sporządzony plan poziomu, który po dyskusjach w obrębie zespołu został zaakceptowany oraz wdrożony.

W trakcie planowania poziomu zostały przedyskutowane oraz uwzględnione następujące zagadnienia

- **Tematyka poziomu w powiązaniu z założeniami gry** - jaki poziom pasuje do założeń naszej gry? Gdzie się znajduje? Jak fabuła tłumaczy wybór owego miejsca i w drugą stronę - w jaki sposób dobór lokacji i historia opowiadana przez miejsce wiąże się z naszą fabułą i ubogaca ją? Wybór padł na specjalnie odizolowane za pomocą gazu opuszczone miasto oraz okoliczną farmę i cmentarz. Głębsza analiza wyboru znajduje się tutaj: Estetyka i atmosfera
- **Cele poziomu** - jakich doświadczeń ma on dostarczyć graczowi? Jakie są warunki zwycięstwa? Czy celem jest stworzenie otwartego świata i pozwolenie graczowi na eksploracje - czy raczej należy stworzyć poziom liniowy, w którym gracz ma za zadanie dostać się z punktu A do punktu B - i jest on prowadzony w łatwy sposób?
- **Historia opowiadana przez poziom** - czy chcemy by nasz poziom opowiadał sobą historię - czy raczej chcemy ją graczowi przedstawić w inny sposób, a sam poziom ma być tylko planszą do rozgrywki.
- **Inspiracje i odniesienia** - czy już istnieją poziomy o podobnej tematyce lub założeniach? Jeżeli tak - to czy były dobre/złe? Jakie były ich mocne i słabe strony - oraz jak możemy uniknąć popełniania błędów innych ?

Po przeanalizowaniu powyższych pytań powstał pierwszy zamysł poziomu. W pierwotnych założeniach miał on ograniczać się do zamkniętego miasta, które po spełnieniu celu - pokonania wszystkich przeciwników - przekierowałoby gracza do kolejnego poziomu - cmentarza.

Następnie z cmentarza gracz po pokonaniu oponentów miał być kierowany do kolejnego, finałowego poziomu - lotniska. Poziomy te w założeniach różniły się ilością przeciwników oraz dostępnym broniami.

Jednakże dalsza analiza potencjalnych plusów oraz minusów tego rozwiązania oraz porównanie z innymi grami podobnego gatunku doprowadziły do następujących wniosków:

- Taki ciąg poziomów nie jest w żaden sposób uzasadniony fabularnie
- Liniowość tego rozwiązania negatywnie wpływałyby na satysfakcje graczy
- Gry z gatunku battle-royal uniwersalnie stosują jeden, rozległy poziom, podzielony na pomniejsze lokacje.

W związku z powyższymi wnioskami końcowy koncept uległ zmianie i zamiast trzech odrębnych poziomów, użyty został koncept jednego, większego poziomu, który zawiera w sobie powyższe lokacje. Dodatkowo, lotnisko będące planowanym ostatecznym poziomem, zostało zamienione na farmę. Decyzja ta uzasadniona była wielkością lokacji oraz prędkością poruszania się gracza - w skali świata lokacja ta zaburzałaby proporcje pomniejszych lokacji jak również nadmiernie spowalniałaby tempo rozgrywki.

6.2 Implementacja interakcji w środowisku

6.2.1 Rodzaje przedmiotów do interakcji

Ze względu na charakterystykę gatunku gry, główną częścią interaktywnych obiektów stanowią bronie. Zaimplementowane w grze zostało 5 broni (2 z nich występują w różnych wariantach kolorystycznych):

- Pistolet (warianty czarny, biały, brązowy)
- Rewolwer
- Strzelba
- Karabin (warianty jak pistolet)
- Karabin Sniperski

Wprowadzone zostały również obiekty powiązane z mechanikami w grze:

- Apteczka - w celu uzupełniania punktów zdrowia - występują obecnie dwa warianty - bazujące na takim samym meshu, lecz używające innej tekstury
 1. Czarna - odnawia 15 % maksymalnych punktów zdrowia
 2. Czerwona - odnawia 35 % maksymalnych punktów zdrowia

- Skrzynka z amunicją - po wejściu w interakcję sprawdza ilość amunicji gracza, jeżeli jest ona różna od maksymalnej (przez maksymalną rozumie się trzykrotność pojemności magazynka) dla dowolnej z broni noszonej przez gracza - pozwala na uzupełnienie amunicji do ilości maksymalnej
- Pancerz - po wejściu w interakcję sprawdza obecną wartość pancerza gracza - jeżeli jest różna od 100%, uzupełnia ją do maximum
- Gaz - interakcja odbywa się nie jak w wszystkich powyższych - przez odpowiedni klawisz - lecz przez wejście gracza w zakres collidera.
Wejście w zakres collidera odpala funkcje zadającą bohaterowi obrażenia w czasie.
- Szkło - została zaimplementowana możliwość rozbijania obiektów.// Żeby tego dokonać, za pomocą kodu obiekt szkła sprawdza za pomocą collidera kolizje z pociskami wystrzelonymi przez gracza/bota.
W momencie wykrycia takowej kolizji następuje eksplozja niszcząca szkło oraz podmiana prefabu na odłamki szkła.
Otwiera to szerokie spektrum możliwości w kontekście tworzenia poziomu - bowiem zniszczenie takowego szkła posiada również towarzyszący temu dźwięk - co może informować gracza o obecności przeciwników. Dodatkowo, otwiera to możliwość strzelania przez przestrzeń które nie są blokowane szkłem po zniszczeniu go.
- Skrzynie z granatami - zasada działania jest podobna do skrzyni z amunicją (tutaj maksimum dla danego typu granatu wynosi 3) - jednak sprawdza ona wszystkie z 3 możliwych typów granatów
 1. Wybuchowy - warto wspomnieć że on również jest w stanie wchodzić w interakcję z szkłem tj. niszczyć
 2. Dymny - po rzuceniu triggeruje się Particle Effect, tworzący dym na małym obszarze
 3. Hukowo-błyskowy - po rzuceniu, wybucha, w momencie kiedy gracz patrzy w kierunku granatu w momencie, na ekranie zostaje wyświetlony biały canvas zasłaniający cały ekran, który wraz z czasem zanika (poprzez obniżanie jego wartości alpha), dodatkowo wybuchowi towarzyszy głośny dźwięk
- Skrzynie
trumny - wejście w interakcję triggeruje animacje otwarcia skrzyni (lub trumny), wewnątrz której można znaleźć broń (nie każda skrzynia zawiera broń, obiekty broni są dodawane ręcznie do owych skrzynek)

Oprócz wyżej wymienionych elementów związanych z mechanikami, gracz może wejść również w interakcje z:

- drzwiami przesuwnymi - otwierają się w momencie wejścia w kolizje z colliderem gracza
- drabiną - podejście pozwala na wspinanie się gracza w celu uzyskania dostępu do wybranych lokacji
- bramą - wejście w interakcje otwiera ją, umożliwiając graczowi wejście na cmentarz

6.2.2 Sposób implementacji

Wszystkie z powyższych interaktywnych obiektów zostały przygotowane jako prefaby, z podpiętymi wszystkimi niezbędnymi komponentami oraz skryptami

Następnie owe prefaby zostały wkomponowane w poziom, z uwzględnieniem balansu poprzez rozmieszczenie słabszych broni w łatwiej dostępnych lub bardziej oczywistych lokacjach, natomiast późniejsze wyposażenie zostało ukryte w skrzyniach w cięzszych do osiągnięcia lokacjach (przez trudność dostania się do lokacji rozumiane jest albo wyzwanie pod kątem zręcznościowym - wymaga skoków i odpowiedniego poruszania, lub spora ilość przeciwników w danej lokacji) Wyjątkiem od wyżej wymienionej zasady jest gaz - który został użyty jako immersyjny sposób do ograniczenia wielkości poziomu.



Rysunek 33: Zastosowanie gazu jako naturalnego ograniczenia wielkości poziomu

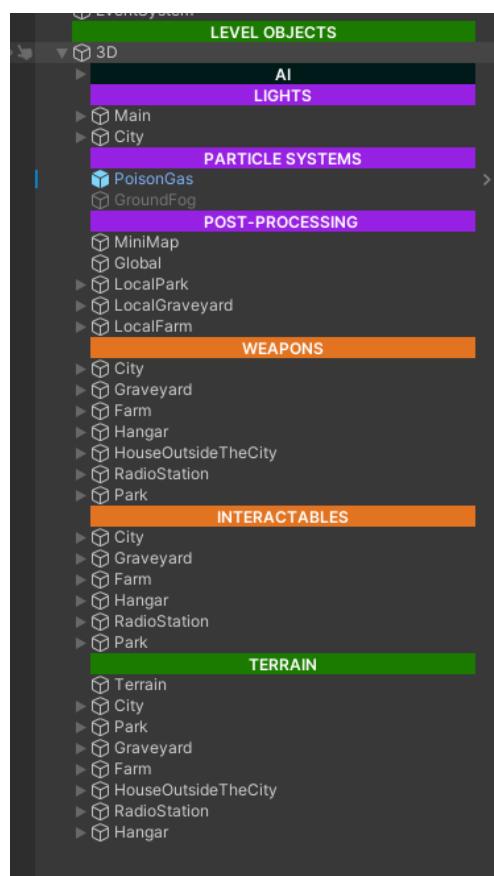
6.3 Rozkład przestrzeni

6.3.1 Struktura Ogólna Poziomu

To tutaj kształtuje się świat gry, definiując główne obszary, trasy poruszania się gracza oraz punkty kluczowe. Zrozumienie tej struktury jest kluczowe dla zapewnienia płynności i logiczności rozgrywki. W tym kontekście, zajmiemy się analizą, jak zorganizować przestrzeń, aby stworzyć poziom, który nie tylko przyciąga wzrok, ale również dynamicznie reaguje na decyzje gracza.

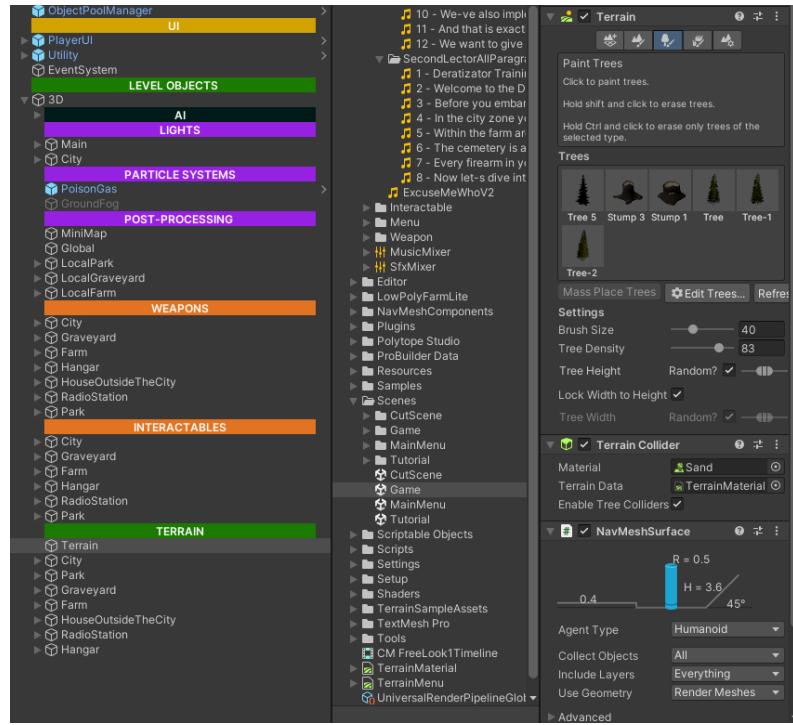
W celu zapewnienia dynamiki naszemu poziomowi, postawiliśmy na rozwiążanie pozwalające uniknąć nadmiernego wczytywania - czyli umieszczenie wszystkich segmentów poziomu - w obrębie jednej sceny Unity. Dodatkowo zastosowanie asynchronicznego ładowania sceny pozwoliło na stworzenie dużego poziomu bez większych problemów pod kątem wydajności.

Rozwiązanie to ma jednak swoje ograniczenia - utrudnia wprowadzanie zmian w scenie więcej niż jednej osobie ze względu na sposób przechowywania danych na temat sceny w jednym pliku - więc potencjalne jednoczesne wprowadzania zmian może prowadzić do niemożliwości otwarcia sceny/problemów z wprowadzaniem zmian. Wszelkie obiekty umieszczone na scenie znajdują się w hierarchii sceny tutaj:

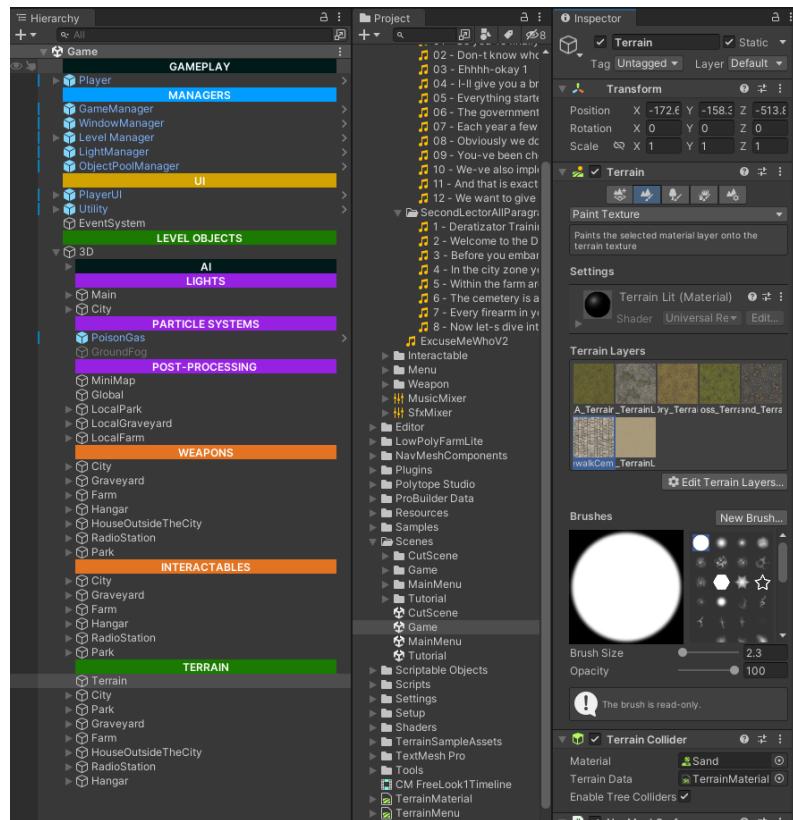


Rysunek 34: Hierarchia obiektów na mapie

Obiekty jak trawa, drzewa oraz kamienie dostępne są do edycji oraz rozmieszczania z poziomu ustawień terenu - to samo dotyczy wszelkich tekstur użytych do malowania podłoża:



Rysunek 35: Ustawienia drzew na mapie - z poziomu terenu



Rysunek 36: Użyte tekstury

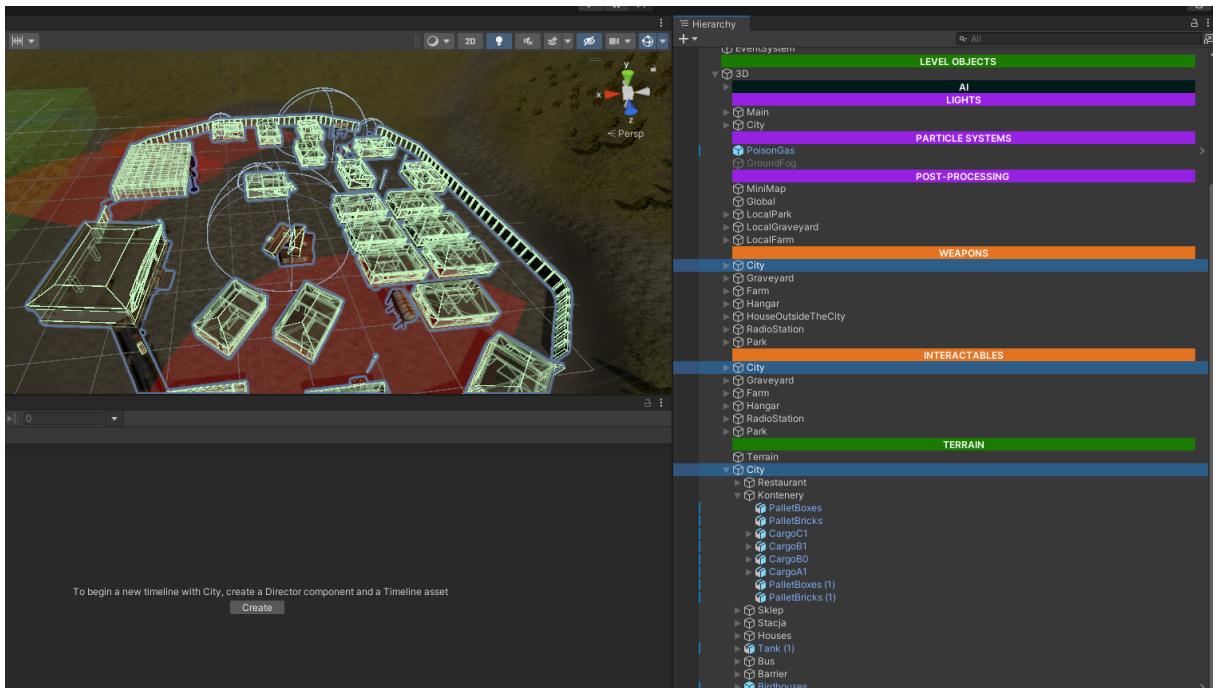
6.3.2 Sekcje i Obszary Kluczowe

Poziom został podzielony na 4 główne sekcje, które wraz z postępami prac zostały rozdzielone na mniejsze obszary kluczowe. Dodatkowo powstała również lokacja będąca łączniem między sekcjami - w celu bardziej angażującej podróży pomiędzy owymi strefami - oraz makietą potencjalnej lokacji, która jednakże nie została wykorzystana - dostęp do niej jest zablokowany przez gaz - jednakże ze względu na aspekt wizualny (daje ona wrażenie że poza gazem również są tereny) została ona zachowana w obecnym scenariuszu Strefy które możemy wyróżnić w grze to:

- Cmentarz
- Miasto
- Farma
- Radiostacja

Wszystkie lokacje połączone są z sobą - dostęp do nich jest nieograniczony - co daje graczy dowolność w kolejności odwiedzania.

Wszystkie obiekty związane z danymi lokacjami są odpowiednio pogrupowane w celu ułatwienia nawigacji przy potencjalnym wprowadzaniu zmian: Każda z sekcji jest również



Rysunek 37: Przykład hierarchii dla sekcji miasta

rozbita na obszary kluczowe dla niej - główne punkty uwagi gracza, które będą najczęściej odwiedzane Są to:

- Dla miasta są to:
 - Restauracja
 - Stos kontenerów na środku
 - Wrak autobusu
 - Sklep
 - Sekcja domków szeregowych - 7 identycznych domów w rzędach
 - Stacja benzynowa
- Sekcja cmentarza jest podzielona na:
 - Sekcja wejściowa - brama przed nagrobki
 - Sekcja środkowa - ołtarz oraz kolumny
 - Sekcja tylna - kurhany, mniejszy wydzielony fragment z nagrobkami
- Farma została podzielona na
 - Sekcja domów
 - Pola uprawne
- Radiostacja posiada następujące punkty zainteresowania:
 - Budynek Radiostacji
 - Kontenery z skrzyniami

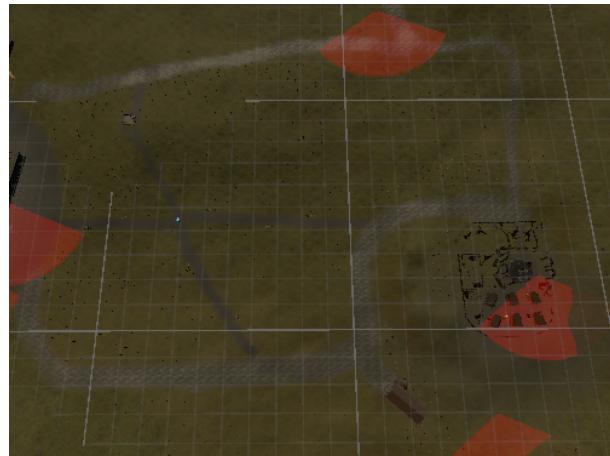
Taki rozkład znacznie ułatwiał pracę nad poszczególnymi lokacjami - każdy z punktów kluczowych został osobno zaprojektowany i wdrożony, następnie połączenie ich dawało praktycznie gotową lokację.

6.3.3 Połączenia Między Sekcjami

Połączenia między sekcjami muszą być zarówno interesujące pod kątem posiadanej zawartości - jak również wizualnie. Jednocześnie w dalszym ciągu powinny być łatwo widoczne oraz intuicyjne dla gracza. Ze względu na rozmiar mapy jak i zastosowanie minimapy intuicyjność wyboru ścieżki i miejsca dokąd prowadzi nie stanowi żadnego wyzwania pod kątem tworzenia poziomu - gdyż lokacje są widoczne.

W ramach zachęty do stosowania przez gracza wytyczonych ścieżek - pozostałe trasy posiadają dużą ilość drzew. Powoduje to znaczne spowolnienie w przemierzaniu terenu - jednakże gwarantuje większą ochronę - gdyż drzewa posiadają collidery, co sprawia że pocisk wystrzelony w stronę gracza może zatrzymać się na obiekcie drzewa.

Zastosowane w grze ścieżki zostały wykonane przez naniesienie odpowiedniej, różnej od pozostałej powierzchni, tekstury - by za pomocą skojarzeń z dnia codziennego przekazać informacje "to jest ścieżka" graczowi.



Rysunek 38: Ścieżki naniesione na teren za pomocą tekstuury

W celu urozmaicenia podróży gracza, na przecięciu najważniejszych ścieżek prowadzących do głównych sekcji wymienionych wyżej, została wprowadzona nowa mniejsza lokacja - Park. Umieszczenie kilku ławek, koszy oraz latarni z dodaną emisją sprawiło że ścieżka nie sprawia wrażenia pustej, świat wydaje się żywlszy i realniejszy. Dodatkowo gracz może znaleźć po drodze na kilka broni - co okazało się konieczne w sytuacjach gdzie AI w agresywny sposób wykorzystało dostępne na mapie zasoby i zmusiło gracza do ucieczki - pozwoliło to uniknąć sytuacji gdzie gracz ginie bez możliwości sensownej obrony.

Warto również wspomnieć że wszelkie lokalizacje mijane po drodze posiadają również teleportery, będące punktami startowymi dla gracza oraz botów. Poszczególne lokacje posiadają różne warianty kolorystyczne owych teleporterów, co może być swego rodzaju informacją dla gracza dzięki dobraniu kolorów powodującym skojarzenia:

- Cmentarz & kościół - pomarańczowy
- Radiostacja & okolice - niebieski
- Farma - żółty
- Park& okolice lasu między lokacjami - zielony
- Miasto - różowy

6.4 Estetyka i atmosfera

Podział poziomu na mniejsze sekcje doprowadził do decyzji zastosowania różnych estetyk dla poszczególnych lokacji. Każda z nich posiada inną muzykę, zastosowane palety barw oraz

assetty czy oświetlenie. Dzięki takiemu rozwiązaniu przemierzając poszczególne lokacje gracz w mniejszym stopniu odczuwa fakt że mapa jest zamkniętym wycinkiem - gdyż każda lokacja ma swoją historię którą graficznie przekazuje, i mogłaby stanowić osobny poziom.

Główne założenia odnośnie estetyki dla poszczególnych sekcji to:

- Cmentarz - mroczna, ciężka, gotycka stylistyka, chłodne, ciemne barwy, odcienie czerni, szarości, miejscami rozświetlone przez pomarańczowe oświetlenie latarni oraz świec. Assety używające teksturow kamienia, grafitu, by jeszcze bardziej podkreślić mrok tego miejsca. Brak/minimalna roślinność, jeżeli występuje - to uschnięte krzewy, drzewa bez liści. Podłoże korzystające z teksturow odzwierciedlających suchą, uschniętą ziemię. Inspirowane scenami z filmów jak "Harry Potter i Czara Ognia" (sekwencja końcowa na cmentarzu), "Smętarz dla zwierzaków"
- Miasto - klimat znacznie lżejszy niż na cmentarzu, mimo to surowe miasto, wzorowane na miastach Europy wschodniej (wielka płyta). Miasto ma w założeniach sprawiać wrażenie opustoszałego - porozbijane szyby, zniszczone budynki, zardzewiałe wraki samochodów. Odcienie szarości i brązu, zastosowane materiały - beton, ciemne drewno, metale. Wyjątek - restauracja - rozświetlona, jasna, nowoczesna, wydaje się nie pasować do pozostałych budynków. Brak roślinności. Podłoże - tekstura betonu.
- Farma - kontrast dla dwóch poprzednich lokacji, jasna, rozświetlona, sielankowo cukerkowa, ciepłe, żywe barwy, odcienie żółci i pomarańcza, kolory mocno nasycione, jaskrawe. Sporo roślinności, podłoże - jasna trawa. Budynki drewniane, proste, wiejskie.
- Park - klimat letniego popołudnia, jako strefa środkowa mapy ma za zadanie wprowadzić pewien balans pomiędzy cmentarzem, a miastem tak aby gracz doświadczył w miarę płynnego przejścia klimatu pomiędzy lokacjami.
- Radiostacja - położona na wzgórzu w okolicach parku z widokiem na całą mapę. Pełni po części funkcję punktu widokowego. Prosta drewniana konstrukcja z drobną piwniczką na sprzęt.

6.5 Przerywniki filmowe

6.5.1 Rodzaje i zastosowanie w grze cutscenek

Po zebraniu feedbacku po playtestach część graczy zauważyła brak powiązania fabularnego między lokacją, do której trafia gracz a tytułem gry oraz stylistyką głównego menu. Po przedyskutowaniu w obrębie zespołu postanowiliśmy stworzyć cutscenek wprowadzającą(intro), której zadaniem będzie stworzyć pomost między sceną na której znajduje się gracz, a tytułem.

Dodatkowo, postanowiliśmy również stworzyć odrębne cutscenki w ramach zakończenia rozgrywki (outro) - w dwóch wariantach, w zależności czy gracz wygrał czy przegrał. Z uwagi na decyzję o wdrożeniu również tutorialu w ramach cutscenki - polegającego na przeniesieniu gracza na osobną scenę, która wcześniej była używana do testowania nowych mechanik - a następnie możliwość przejścia z poziomu samouczka do gry głównej - implementacja intro odbyła się na osobnej scenie z specjalnie przygotowanym środowiskiem którego przedstawienie znajduje się poniżej. Takie rozwiązanie pozwoliło nam na zwiększenie ilości klatek na sekundę (FPS) podczas cutscenki - ze względu na brak konieczności renderowania oraz symulacji zachowania botów, pozbyciu się assetów broni, elementów interfejsu gracza, oraz samych mechanik powiązanych z graczem - takich jak poruszanie się, system ekwipunku, dynamiczna kamera (gracza jak i minimapy)

6.5.2 Struktura sceny

Do stworzenia scen outro - w obu wariantach - została wykorzystana scena **Game** - czyli scena główna na której odbywa się rozgrywka - ze względu na fakt że po zakończeniu rozgrywki, niezależnie od wygranej lub przegranej gracze - nic nie powstrzymuje nas przed usunięciem obiektów powiązanych z botami oraz graczem - oraz aktywowania obiektów stricte niezbędnych do scenki - gdyż rozgrywka sterowana przez gracza de facto dobiera końca.

Natomiast ze względu na wyżej opisane powody scena intro odbywa się na specjalnie przygotowanej scenie **Cutscene**. Jej struktura pod kątem obecnych obiektów rozbita została na 3 główne sekcje:

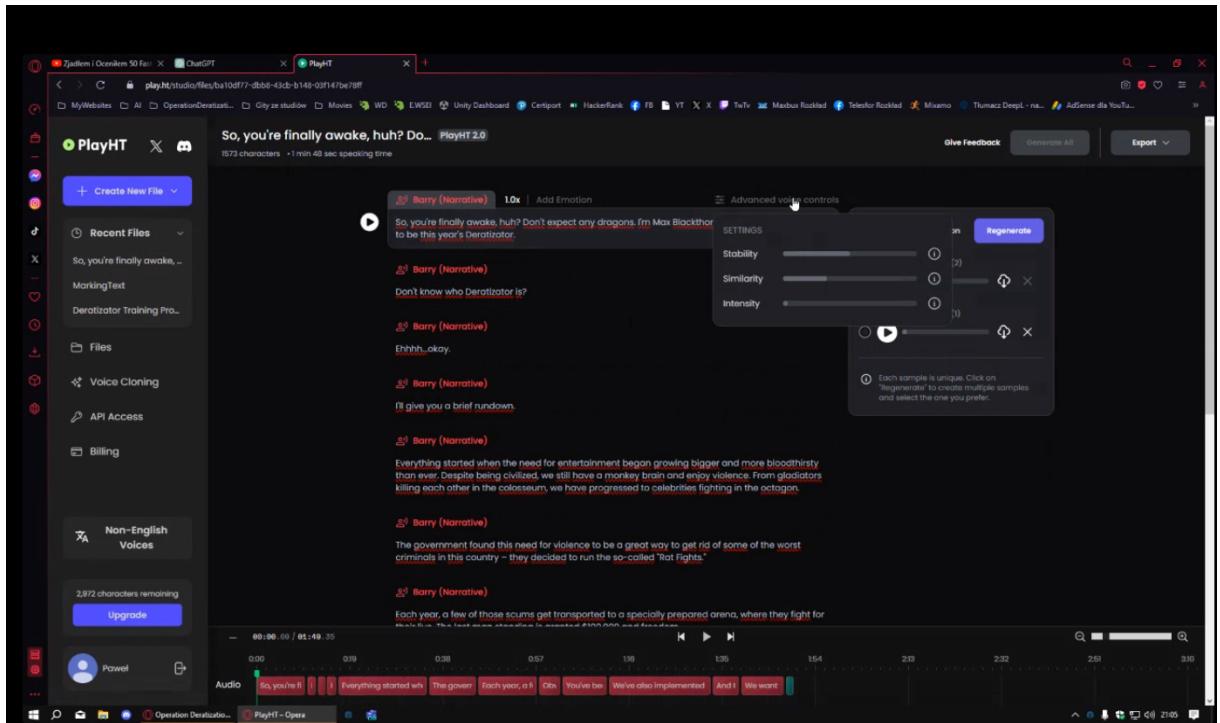
- Lokacja 1 - dom - wykorzystane tutaj zostało identyczne ułożenie obiektów jak do menu głównego.
- Lokacja 2 - teren - jest to kopia terenu głównego rozgrywki - zmniejszona o nieużywane elementy powiązane z rozgrywką
- Obiekty związane z samą scenką - najszerzsza kategoria. znajdują się tutaj wszystkie kamery wraz z dollyCartami, warstwa UI odpowiedzialna za wyświetlanie napisów przy dialogach, wszelkie efekty wizualne oraz obiekty animowane na osi czasu. Dodatkowo w tej kategorii znajdziemy wszelkie Managery wykorzystane do zarządzania przepływem pomiędzy scenami rozgrywki.

6.5.3 Mechanizmy użyte w cutscenice

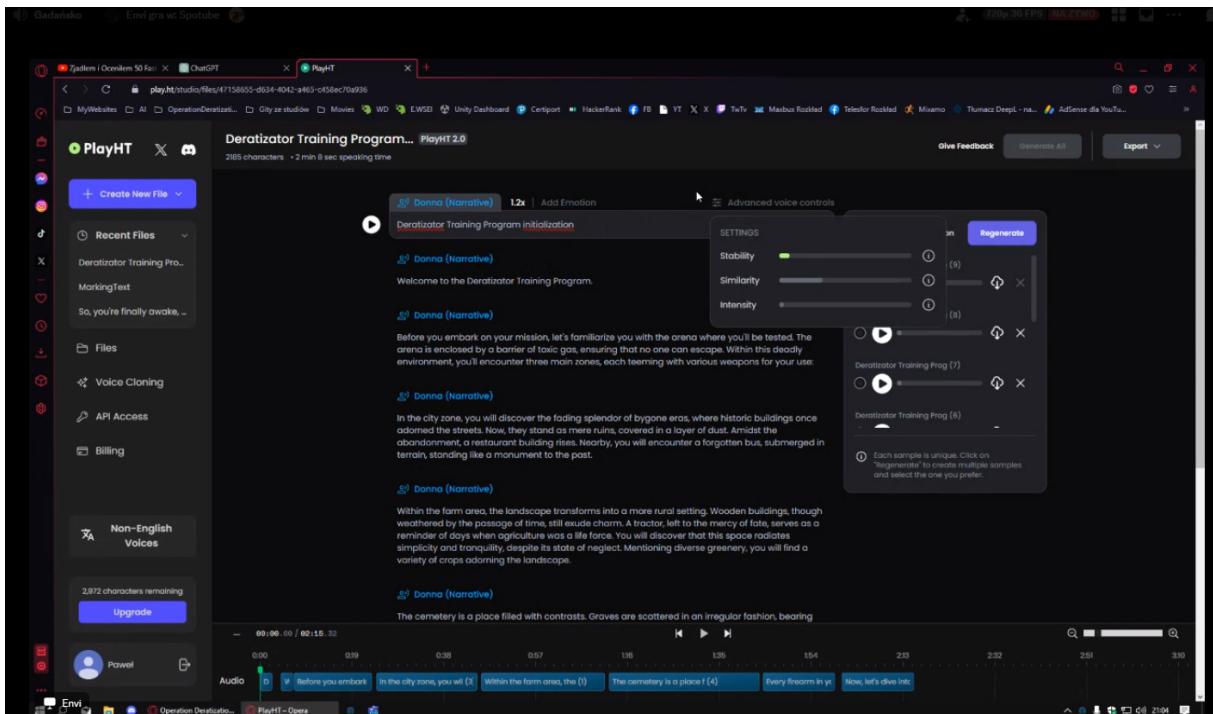
Rdzeniem naszych przerywników filmowych są dwie dostępne w samym Unity paczki - Cinemachine oraz Timeline. Dodatkowo, bardzo ważną rolę odgrywa również paczka

pozwalającą na sterowanie tekstami z TextMeshPro z poziomu timeline, stworzona przez Krisa Kreja.

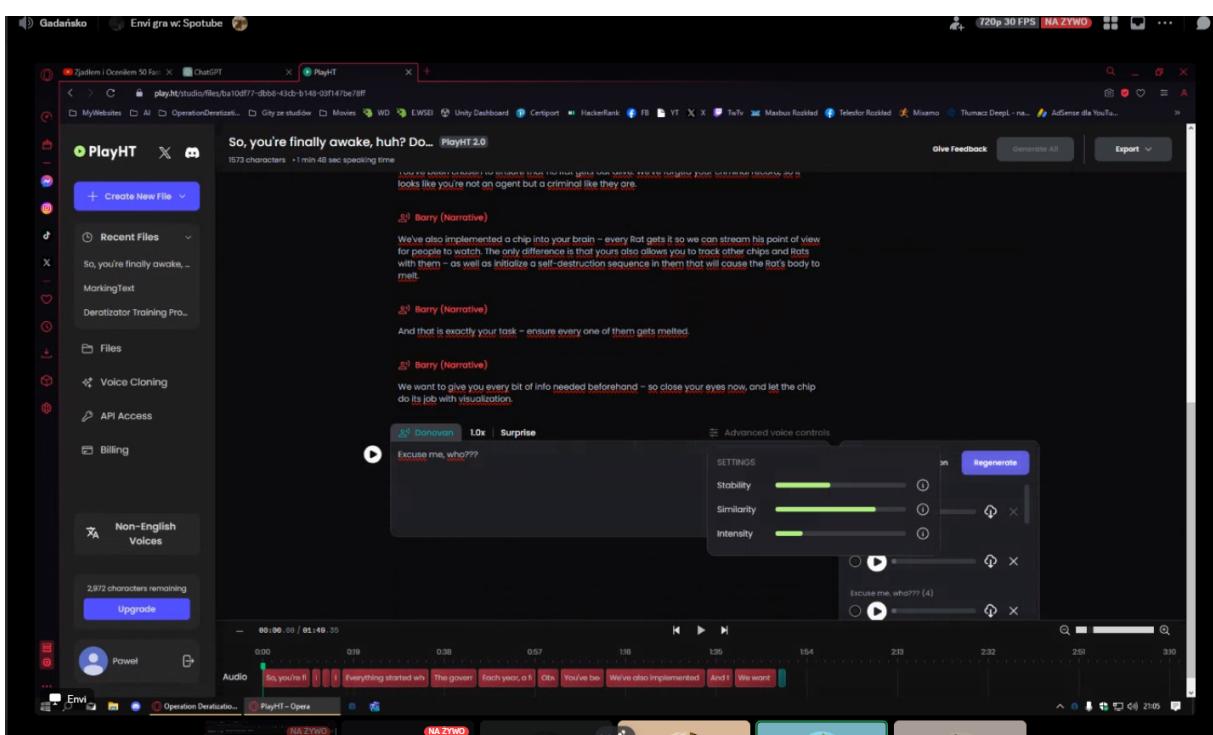
Warto również wspomnieć o dubbingu, który został wygenerowany przy użyciu narzędzia AI text to speech. Poniżej screenshotsy przedstawiające konkretne konfiguracje dla poszczególnych głosów:



Rysunek 39: Konfiguracja głosu dla Maxa - narratora pierwszej części intro



Rysunek 40: Konfiguracja głosu dla narratora drugiej części intro - Smart Chip



Rysunek 41: Konfiguracja głosu dla głównego bohatera

Dialogi zostały rozdzielone na oddzielne pliki - po jednym zdaniu per linijka. Po umieszczeniu na timeline wszystkich dialogów, dostosowane zostało poruszanie się postaci.

6.5.4 Cutscenka Intro - pozostałe w trakcie pracy

Zamysł cutscenki stanowi rozwinięcie fabuły przedstawionej tutaj Opis gry

Cutscenka Intro ze względu na swoją strukturę oraz historię przedstawioną może zostać na dwie osobne sekcje - pierwszą, w domku, gdzie narratorem jest postać Maxa Blackthorna - którego osoba pozostaje dla gracza pewnego rodzaju zagadką, na podstawie dialogów można snuć domyśl że jest to agent rządowy, który wdraża nas - graczy - w role którą pełnimy w rozgrywce.

Początek scenki informuje nas nie bezpośrednio że wszystkie zdarzenia oglądane są z perspektywy postaci gracza - poprzez efekt zamykania i otwierania oka.

Efekt ten został osiągnięty poprzez Sugeruje nam to również że gracz mógł zostać porwany. Następnie narrator odchodzi, a na ścianie za nim zostaje za pomocą animatora naniesiona grafika w Sprite2D - oraz aktywowane i dezaktywowane jest światło (Spot Light), co emituje efekt odtwarzania slajdów. Pozostały fragment pierwszej części cutscenki sprowadza się do kolejnych linii dialogu oraz zmiany slajdów (wyłączanie i analogiczna animacja dla innych obiektów Sprite2D)

Końcowy dialog jest punktem przełomowym - gracz jest informowany o posiadaniu przez postać "chipa" w głowie - co stanowi uzasadnienie mechanik takich jak oznaczenie ciał przeciwników czy nawet użyty interfejs - dodatkowo wprowadzenie owego "chipu" jako postaci pozwala zmienić narratora i w naturalny sposób zaprezentować graczowi poziom - co odbywa się w drugiej części intro

Druga część intro to pokazanie wszystkich lokacji oraz punktów zainteresowania - wraz z narracją wprowadzającą historię owych miejsc.

Głównym elementem zastosowanym w tej części cutscenki jest użycie wielu wirtualnych kamer oraz przełączanie się między nimi - oraz zmiana ich rotacji wraz z czasem z poziomu Animatora.

W celu poprawy wrażeń wizualnych niektóre z kamer posiadają zablokowany punkt na który patrzą (Look At) - dzięki czemu rozwiązanie polegające na pokazaniu całego poziomu stało się znacznie łatwiejsze - kamera porusza się dookoła mapy, patrząc na pusty obiekt znajdujący się idealnie w centrum.

Na potrzeby tej części sceny zmodyfikowany również został skrypt odpowiedzialny za kontrolę drzwi:

```

public class DoorMotionSensor : MonoBehaviour
{
    // ... (Remaining part of the script)

    if (isUsingCameraCheck)
    {
        float distanceToCamera = Vector3.Distance(transform.position,
            vCamera.transform.position);
        distance.Add(distanceToCamera);
        CinemachineTrackedDolly dollyCart =
            vCamera.GetCinemachineComponent<CinemachineTrackedDolly>();

        if (dollyCart != null)
        {
            float pathPosition = dollyCart.m_PathPosition;

            if (pathPosition >= 3f && pathPosition <= 4f || pathPosition
                >= 4.1f && pathPosition <= 5f)
                count++;
        }
    }

    // ... (Remaining part of the script)
}

```

Fragment kodu 7: Zmiany sposobu kontrolowania drzwi w skrypcie DoorMotionSensor.cs dzięki czemu obiekt DollyCart był w stanie aktywować animację otwierania drzwi podczas "wjazdu" kamery do restauracji, gdzie takowe drzwi zostały użyte. Następnie zakończenie scenki prowadzi do przekierowania i wczytania sceny **Tutorial** - na której gracz może zapoznać się z mechanikami już z poziomu rozgrywki.

7 Audio

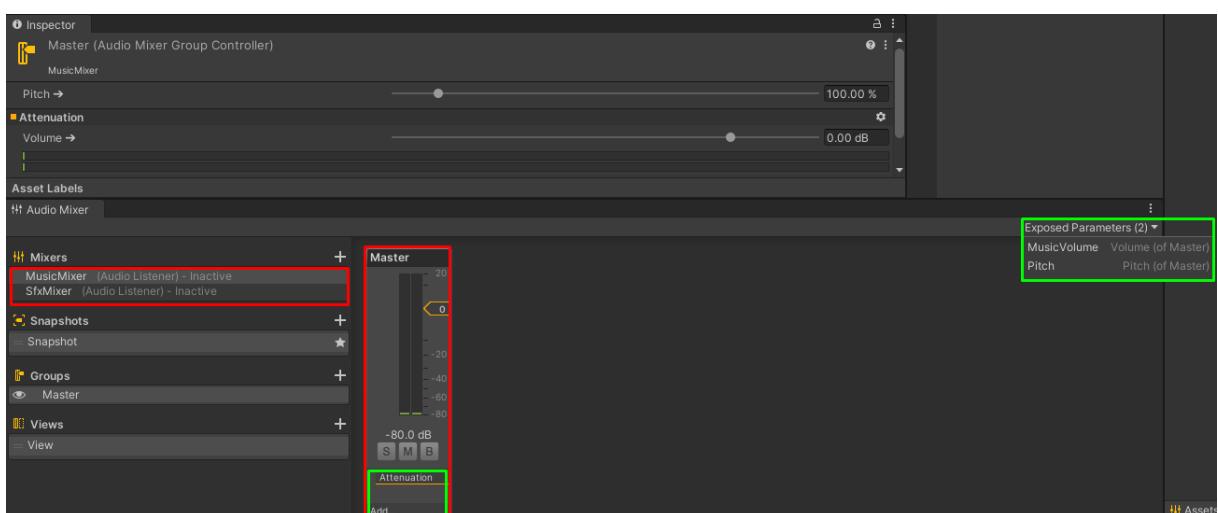
Rozdział poświęcony dźwiękowi i audio w grach komputerowych zajmuje się kluczowymi aspektami związanymi z projektowaniem, implementacją i optymalizacją dźwięku w grach.

7.1 Kontrola ścieżek dźwiękowych

W tej sekcji znajdziesz informacje na temat zarządzania różnymi ścieżkami dźwiękowymi w grze. Możesz kontrolować poziom głośności bezpośrednio z gry lub menu głównego, przechodząc do ustawień i przesuwając suwak od kontroli głośności muzyki lub osobnego suwaka dla reszty dźwięków. Ustawienia te są zapisywane w **PlayerPrefs**, co umożliwia utrzymanie preferencji gracza nawet po wyłączeniu gry.

Kontrola głośności jest również dostępna za pośrednictwem *Audio Mixerów*. Modyfikacja zmiennych dostępnych w **PlayerPrefs** umożliwia kontrolę głośności z poziomu gry. Warto zauważyć, że te zmienne są udostępniane (zielone zaznaczenie po prawej na rysunku poniżej) do kontrolowania z *Audio Mixerów* poszczególnych elementów dźwiękowych z poziomu skryptu, który obsługuje slidery głośności dostępne dla użytkownika w ustawieniach.

Oprócz tego, masz możliwość dostosowania głośności i efektów dźwiękowych z poziomu sceny. Modyfikowanie suwaków *Audio Mixerów* od głośności (czerwone zaznaczenie na obrazku poniżej) lub suwaków komponentów *Audio Source* na poszczególnych obiektach pozwala na precyzyjną kontrolę dźwięków w trakcie rozgrywki.



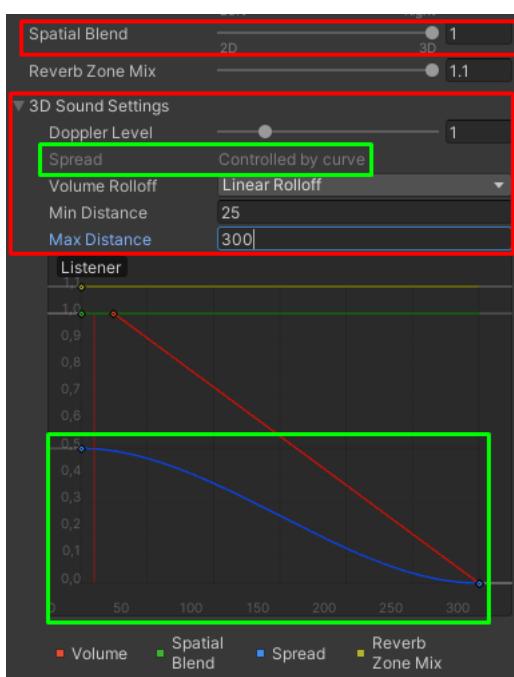
Rysunek 42: Ustawienia Audio Mixerów

7.2 Ustawienia 3D dźwięków

W sekcji dotyczącej ustawień 3D dźwięków, skupiamy się na implementacji efektów przestrzennych, które pomagają uzyskać bardziej realistyczne doznanie dźwiękowe w grze. W przypadku naszej gry, kontrola nad efektami przestrzennymi staje się istotna dla zwiększenia immersji gracza.

Przykłady sytuacji, w których kontrola 3D dźwięków może być kluczowa:

- Dźwięk broni:** Podczas strzelania z broni, efekty przestrzenne pozwalają na dokładne odwzorowanie źródła dźwięku w zależności od kierunku, w którym strzelasz. To może poprawić orientację gracza w otoczeniu oraz dostarczyć dodatkowych wrażeń związanych z użytkowaniem broni.
- Dźwięk granatów:** Efekty przestrzenne umożliwiają precyzyjne określenie pozycji, w której eksplodował granat oraz odległości od naszego miejsca. Gracz będzie w stanie lepiej zrozumieć, czy zagrożenie pochodzi z przodu, tyłu czy z boku, co może wpływać na jego taktykę.



Rysunek 43: Przykładowe ustawienie 3D dźwięku dla obiektu granatu

W ustawieniach dźwięku 3D dla granatu, warto zwrócić uwagę na kilka kluczowych parametrów:

- Spatial Blend:** Określa, jak bardzo dźwięk jest przestrzenny w stosunku do źródła. Wartość 1 oznacza, że dźwięk jest w pełni przestrzenny, natomiast wartość 0 oznacza, że dźwięk jest zupełnie dwuwymiarowy.

- **3D Sounds Settings:** Pozwala na dostosowanie wielu parametrów dźwięku przestrzennego, takich jak odległość, w jakiej dźwięk słychać, szybkość spadku głośności w zależności od odległości czy także szybkość przemieszczania się dźwięku.
 - **Spread:** Określa rozproszenie dźwięku, czyli jak szeroko dźwięk jest rozprzestrzeniony w przestrzeni. Wartość 0 oznacza brak rozproszenia, a wartość 360 oznacza pełne rozproszenie wokół źródła dźwięku.
3. **Dźwięk otwieranych skrzynek:** Kontrola 3D dźwięków może pomóc w tworzeniu bardziej realistycznego doświadczenia podczas otwierania skrzynek czy innych kontenerów. Dźwięk tych elementów będzie można precyzyjnie zlokalizować w przestrzeni, co dodatkowo podniesie realizm gry.
 4. **Dźwięk drzwi:** Efekty przestrzenne pozwalają na dokładne odwzorowanie dźwięku otwieranych drzwi. Gracz będzie w stanie odróżnić, czy drzwi otwierają się po lewej czy prawej stronie, co może mieć znaczenie taktyczne w przypadku szybkiego przemieszczania się po lokacji.

7.3 Zarządzanie Zdarzeniami Dźwiękowymi

Aby umożliwić agentom AI reakcję na zdarzenia dźwiękowe w środowisku gry, wykorzystujemy skrypt **AudioEventManager**. Ten skrypt odpowiada za zarządzanie przekazywaniem informacji o zdarzeniach dźwiękowych agentom AI.

```

public class AudioEventManager : MonoBehaviour
{
    private static AudioEventManager instance;
    public static AudioEventManager Instance
    {
        get
        {
            if (instance == null)
            {
                instance =
                    GameObject.FindGameObjectWithTag("AudioEventManager")
                    .GetComponent<AudioEventManager>();

                if (instance == null)
                {
                    GameObject obj = new GameObject();
                    obj.name = typeof(AudioEventManager).Name;
                    instance = obj.AddComponent<AudioEventManager>();
                }
            }

            return instance;
        }
    }

    public event Action< AudioSource > OnAudioEvent;

    public void NotifyAudioEvent( AudioSource audioSource )
    {
        if (OnAudioEvent != null)
            OnAudioEvent( audioSource );
    }
}

```

Fragment kodu 8: Zarządzanie zdarzeniami dźwiękowymi w klasie AudioEventManager

7.3.1 Opis Skryptu

Skrypt **AudioEventManager** jest odpowiedzialny za zarządzanie zdarzeniami dźwiękowymi w grze. Jest to skrypt jednostki w Unity, który można przypisać do dowolnego obiektu w scenie.

7.3.2 Kluczowe Elementy

- **Singleton Instance:** Skrypt zawiera mechanizm singletona, który zapewnia dostęp do jednej instancji obiektu **AudioEventManager** w całej grze. Jest to osiągane poprzez statyczną właściwość **Instance**, która zwraca referencję do tej instancji.
- **Zdarzenie Dźwiękowe (OnAudioEvent):** Jest to zdarzenie publiczne typu **Action**, które może być subskrybowane przez inne skrypty. Gdy zostanie wywołane, wysyła informacje o źródle dźwięku (**AudioSource**) do wszystkich subskrybentów.
- **Metoda NotifyAudioEvent:** Jest to publiczna metoda, która służy do powiadamiania o zdarzeniach dźwiękowych. Przyjmuje jako parametr obiekt **AudioSource**, który reprezentuje źródło dźwięku, które wywołało zdarzenie. Po otrzymaniu zdarzenia, metoda wysyła informacje o tym zdarzeniu do wszystkich subskrybentów zdarzenia dźwiękowego.

7.3.3 Wykorzystanie

Aby skorzystać ze skryptu **AudioEventManager.cs** w grze, należy przypisać go do dowolnego obiektu w scenie. Następnie, inne skrypty mogą subskrybować zdarzenie dźwiękowe **OnAudioEvent** i reagować na wykryte dźwięki poprzez implementację odpowiednich metod obsługi zdarzeń.

7.3.4 Przykładowe Zastosowanie

Skrypt **AiAudioSensor.cs** korzysta z **AudioEventManager** do reakcji na zdarzenia dźwiękowe w otoczeniu agenta AI. Gdy **AudioEventManager** wykryje dźwięk, informacja o tym zdarzeniu jest przekazywana do **AiAudioSensor**, który podejmuje odpowiednie działania na podstawie otrzymanych danych.

8 Testowanie i debugowanie

Rozdział o testowaniu stanowi kluczową część procesu tworzenia oprogramowania, obejmującą różnorodne strategie, metody i narzędzia służące zapewnieniu jakości produktu. Testowanie jest nieodłącznym elementem każdego projektu, bez względu na jego skalę czy rodzaj. Ten rozdział skupia się na zasadach i praktykach związanych z testowaniem w kontekście tworzenia gier komputerowych. Omówimy różne strategie testowania, które pomogą zapewnić wysoką jakość gry oraz zidentyfikować potencjalne problemy jeszcze przed wykonaniem ostatecznego buildu.

W rozdziale omówione zostaną różne aspekty testowania, począwszy od planowania strategii testowej, przez wykorzystane narzędzia testerskie, aż po analizę wyników i raportowanie błędów. Podkreślona zostanie rola testowania w zapewnianiu jakości oprogramowania, identyfikowaniu i usuwaniu błędów oraz poprawianiu doświadczenia użytkownika.

8.1 Narzędzia debugujące

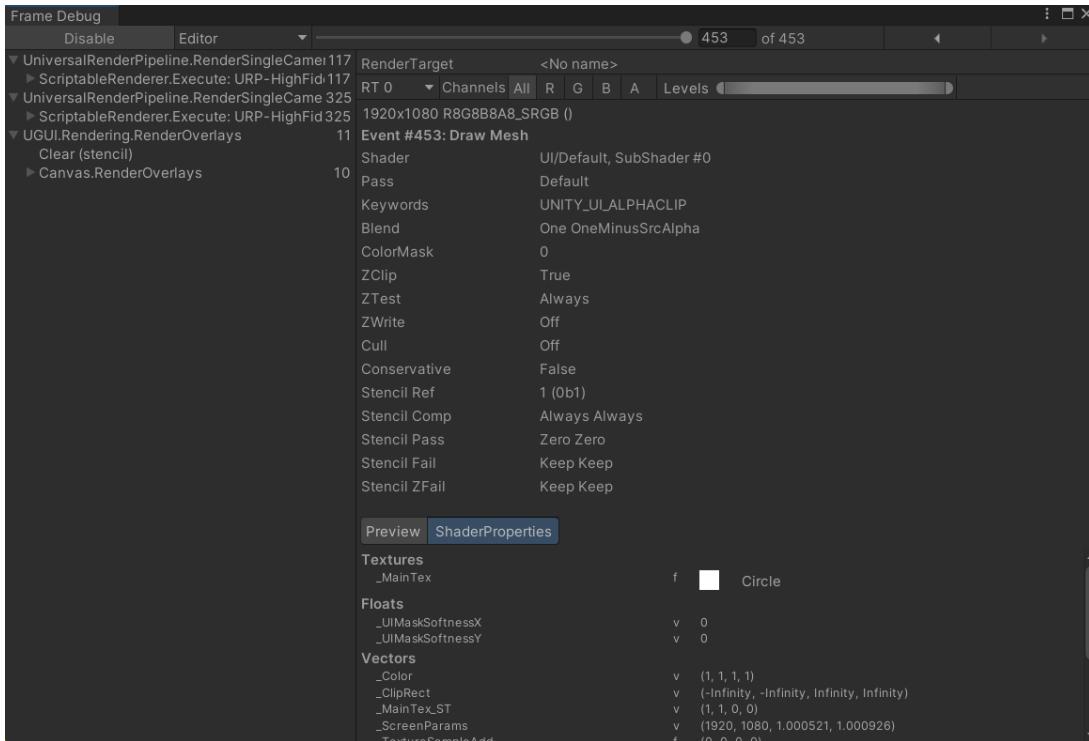
W procesie tworzenia gry FPS na silniku Unity kluczowym elementem jest skuteczne debugowanie, czyli identyfikowanie, analizowanie i rozwiązywanie błędów w kodzie i mechanice gry. Poniżej przedstawione zostaną niektóre z najważniejszych narzędzi debugujących, które mogą być wykorzystane w tym kontekście.

Unity Profiler – to narzędzie, które pozwala na analizę wydajności gry. Pomaga identyfikować miejsca, gdzie gra zużywa najwięcej zasobów komputera (GPU, CPU i pamięć RAM). Dla naszej gry Profiler jest niezastąpiony przy optymalizacji wydajności renderowania i skryptów odpowiedzialnych za obsługę mechanik.



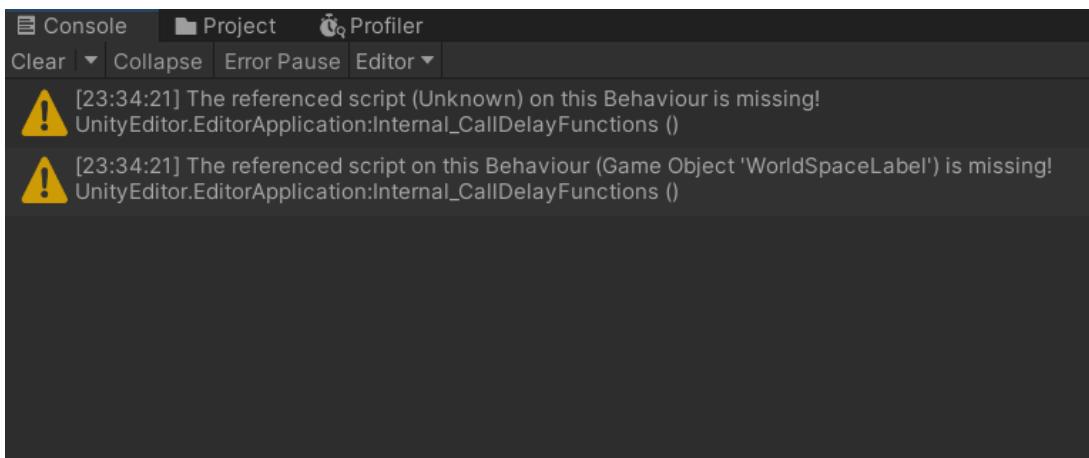
Rysunek 44: Okno Profilera dostępnego w edytorze Unity

Unity Frame Debugger – to narzędzie dostępne w środowisku Unity, które umożliwia programistom analizę renderowania klatki gry. Jest to szczególnie ważne w grach FPS, gdzie płynność i wydajność renderowania mają kluczowe znaczenie dla doświadczenia gracza.



Rysunek 45: Okno Frame Debuggera w edytorze Unity

Unity Console – Bardzo prosta ale przydatna konsola dostępnna bezpośrednio w edytorze Unity. Wyświetla komunikaty, ostrzeżenia oraz błędy co ułatwia śledzenie zmian i problemów w grze. Konsola bardzo pomaga w identyfikacji problemów związanych z interakcjami z otoczeniem czy działaniem broni.



Rysunek 46: Okno Console w edytorze Unity

8.2 Raportowanie błędów

Rozwój gier wideo to złożony proces, który wymaga nieustannej uwagi i zaangażowania zespołu programistycznego. Jednym z kluczowych elementów tego procesu jest skuteczne

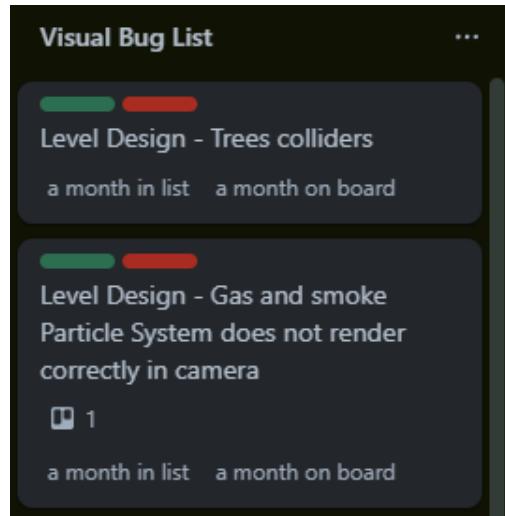
zarządzanie i monitorowanie błędów, które mogą pojawić się podczas różnych faz produkcji. Trello, jako narzędzie do zarządzania projektami, może stanowić doskonałe wsparcie w raportowaniu i śledzeniu błędów w grze. Poniżej omówiony zostanie sposób zastosowania Trello w naszym projekcie.

Dev Bug List – Na tej liście dodajemy karty ze zgłoszeniami błędów deweloperskich, które zostały odkryte i należy je naprawić. W karcie wybieramy etykietę *Bug* oraz ewentualnie *Important* jeżeli zgłoszenie ma wysoki priorytet i wybieramy kategorię, której dotyczy błąd (np. *Mechaniki* lub *Audio*). Następnie dodajemy opis, z którym musi zapoznać się deweloper jeżeli błąd jest złożony. Podpinamy się do zadania jako osoba zgłaszająca, a programista podpisuje się do taska samodzielnie. Wewnątrz karty z zadaniem możemy śledzić co w tym momencie dzieje się z tematem, który zgłosiliśmy poprzez zapoznanie się z komentarzami dewelopera.



Rysunek 47: Lista rozpoznanych błędów deweloperskich występujących w grze

Visual Bug List – Na tą listę trafiają zgłoszenia rozpoznanych błędów wizualnych, które należy naprawić. Podobnie jak w przypadku karty z błędami deweloperskimi w karcie wybieramy odpowiednie etykiety, dodajemy opis dla osoby, która będzie odpowiedzialna za naprawę błędu. Następnie podpinamy się do karty jako zgłaszający i czekamy aż osoba odpowiedzialna za naprawienia buga się do niej przypnie sama. Wewnątrz karty sprawdzimy postęp naprawy.



Rysunek 48: Lista rozpoznanych błędów wizualnych występujących w grze

Osoba odpowiedzialna za naprawę błędu po zaznajomieniu się z zadaniem przenosi kartę do listy **Work In Progress**.

Po naprawie błędu osoba odpowiedzialna za niego przerzuca go do listy **Internal Test**, gdzie tester lub osoba zgłaszająca problem sprawdza czy błąd rzeczywiście udało się wyeliminować i czy przy okazji nie powstał z tego powodu inny bug:

- W przypadku, gdzie problem został naprawiony i pomyślnie przeszedł testy karta z zadaniem zostaje przeniesiona do **Done** i cykl życia taska się zamyka.
- W przypadku, gdzie problem nie został naprawiony/powoduje innego rodzaju problemy karta z zadaniem zostaje przeniesiona do listy **Work In Progress** ze stosownym komentarzem dla osoby odpowiedzialnej za ten task (co nie zadziałało/jak odtworzyć błąd/jaka inna mechanika przestała działać po zmianach).

9 Optymalizacja i wydajność

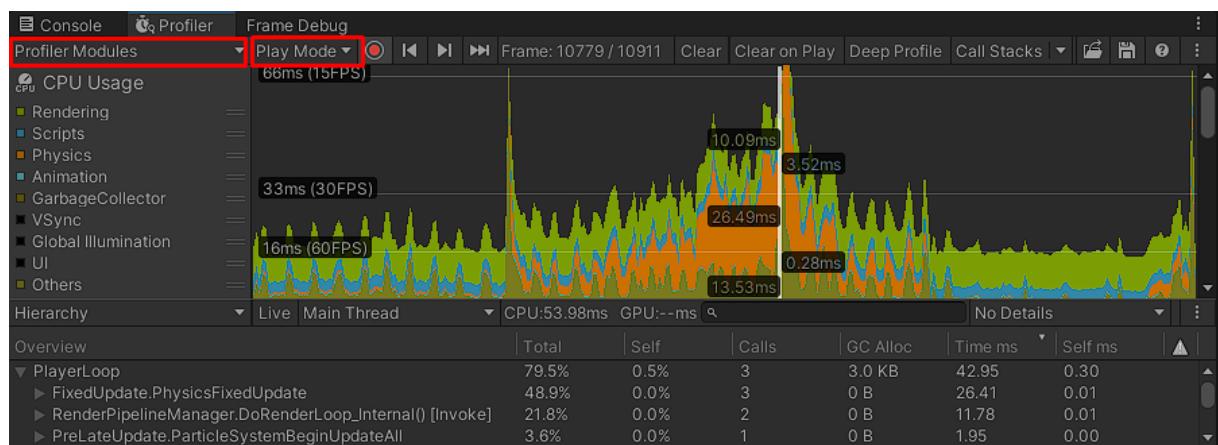
Optymalizacja stanowi kluczowy element procesu tworzenia gry w środowisku Unity, skupiając się na poprawie efektywności działania i zapewnieniu płynności doświadczenia gracza. W obrębie tego rozdziału, eksplorujemy różnorodne aspekty optymalizacji, koncentrując się na trzech głównych sekcjach, które znajdziesz poniżej.

9.1 Profilowanie kodu

Profilowanie kodu jest kluczowym etapem w procesie optymalizacji. W tej sekcji przedstawimy i omówimy narzędzia dostępne w środowisku Unity do analizy wydajności kodu, a także skonfigurujemy profiler w celu efektywnego monitorowania wydajności.

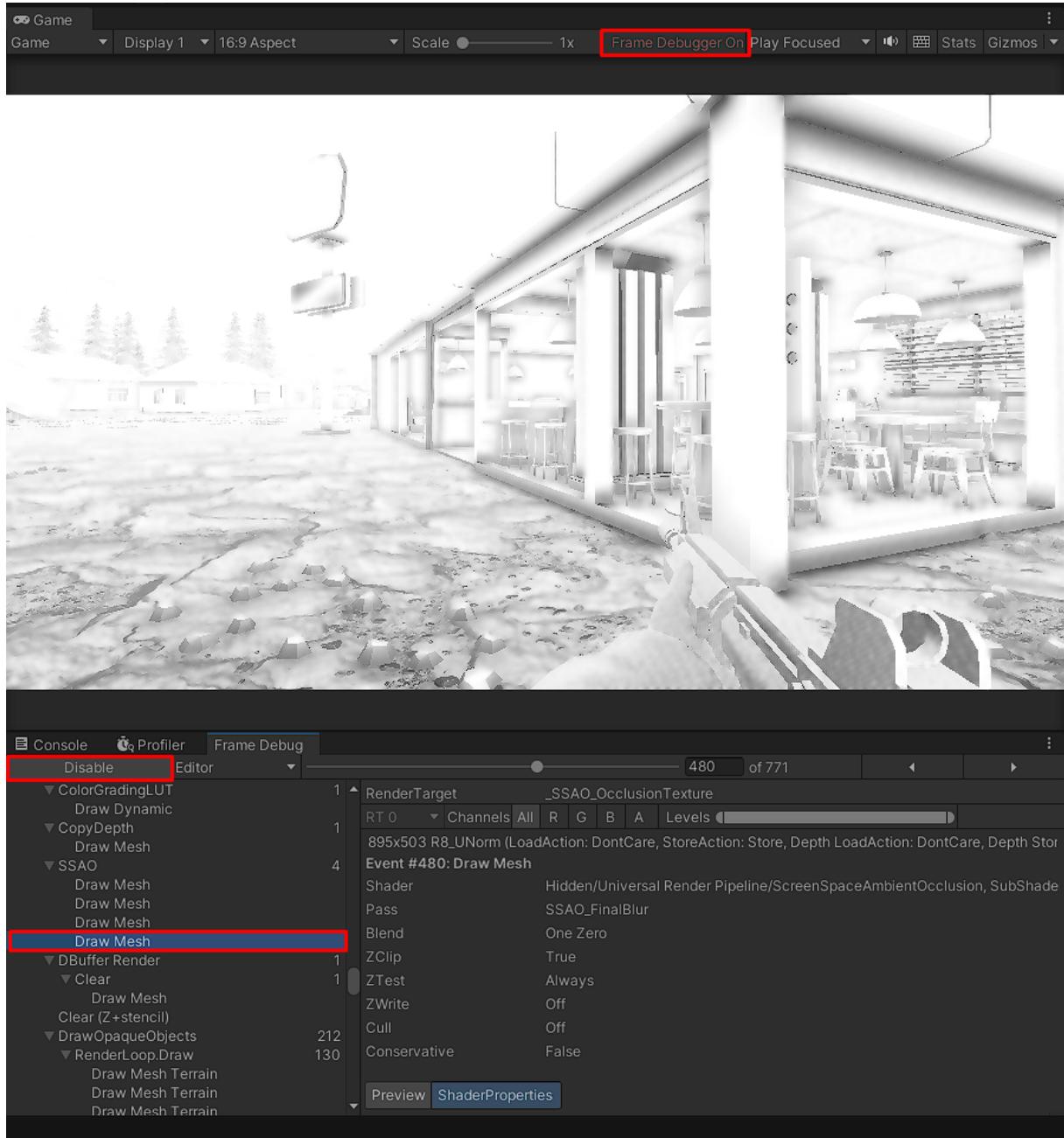
Unity udostępnia zaawansowane narzędzia do profilowania kodu, pozwalające na analizę wydajności w różnych aspektach. Poniżej przedstawiamy kluczowe narzędzia wbudowane w Unity:

- **Profiler Unity** - Profiler to narzędzie, umożliwiające śledzenie i analizę wydajności podczas działania gry lub z okna edytora po przełączeniu trybu na *Edit Mode*. Pozwala ono na monitorowanie zużycia zasobów, czasu renderowania, czasu CPU, alokacji pamięci i innych parametrów, których widoczność możemy przełączać klikając w *Profiler Modules* w oknie Profilera.



Rysunek 49: Okno Profilera dostępne w menu kontekstowym Window/Analysis/Profiler

- **Unity Frame Debugger** - Frame Debugger pozwala na dokładną analizę, jak Unity renderuje każdą klatkę. Wystarczy kliknąć przycisk *Enable* w oknie Frame Debug i wybrać interesującą nas klatkę. To narzędzie jest szczególnie przydatne do identyfikacji problemów związanych z grafiką i renderowaniem.



Rysunek 50: Okno Frame Debug, które znajdziemy w menu kontekstowym Window/Analysis/Frame Debugger

9.2 Zarządzanie zasobami

Wydajne zarządzanie zasobami jest kluczowym aspektem zapewnienia płynności działania gry. W tym podrozdziale omówimy strategie ładowania i zwalniania zasobów w zależności od potrzeb sceny. Przyjrzymy się tematom takim asynchroniczne ładowanie czy korzyści płynące z użycia wzorca *puli obiektów*.

9.2.1 Asynchroniczne Ładowanie

Asynchroniczne ładowanie sceny jest kluczowym elementem optymalizacji gier, umożliwiając płynne przejścia między różnymi fragmentami rozgrywki. W tym kontekście przedstawiamy skrypt **SceneLoader.cs**, który implementuje asynchroniczne ładowanie sceny w grze.

```
public class SceneLoader : MonoBehaviour
{
    // ... (Remaining part of the script)

    public IEnumerator LoadScene_Coroutine(int index)
    {
        AsyncOperation asyncOperation =
            SceneManager.LoadSceneAsync(index);
        asyncOperation.allowSceneActivation = false;
        float progress = 0;

        while (!asyncOperation.isDone)
        {
            progress = Mathf.MoveTowards(progress,
                asyncOperation.progress, Time.deltaTime);
            progressSlider.value = progress;

            if (progress >= 0.9f)
            {
                progressSlider.value = 1;
                asyncOperation.allowSceneActivation = true;
            }

            yield return null;
        }
    }
}
```

Fragment kodu 9: Klasa SceneLoader obsługująca asynchroniczne przechodzenie między scenami

Skrypt ten umożliwia dynamiczne ładowanie sceny, co jest szczególnie użyteczne przy dużych i rozbudowanych projektach. Kluczowym elementem jest użycie klasy **AsyncOperation**, która pozwala na asynchroniczne ładowanie sceny w tle, bez blokowania głównego wątku gry.

Elementy kluczowe skryptu:

- **LoaderUI:** Obiekt reprezentujący interfejs użytkownika (UI) wykorzystywany podczas ładowania sceny.
- **progressSlider:** Pasek postępu ładowania sceny, umożliwiający informowanie gracza o aktualnym stanie procesu.
- **gOToDeactivate:** Obiekt do dezaktywacji podczas ładowania sceny, co może być przydatne, aby ukryć niepotrzebne elementy w trakcie przejścia między scenami.
- **LoadScene:** Metoda rozpoczynająca proces ładowania sceny. Wywołuje **LoadScene_Coroutine** w ramach coroutine.
- **LoadScene_Coroutine:** Coroutine odpowiedzialna za asynchroniczne ładowanie sceny. W trakcie procesu aktualizuje pasek postępu, a po osiągnięciu 90% pozwala na aktywację sceny.

Użycie asynchronicznego ładowania sceny poprzez ten skrypt pozwala na zachowanie płynności rozgrywki nawet w przypadku dużych i złożonych scen, jednocześnie umożliwiając informowanie gracza o postępie za pomocą paska ładowania. Skrypt osiąga stan 90% ładowania, a następnie pozwala na aktywację sceny. Ograniczenie ładowania do 90% zanim scena zostanie aktywowana ma na celu zminimalizowanie zakłóceń związanych z ostatecznym przejściem do nowej sceny, umożliwiając jednocześnie aktualizację interfejsu użytkownika i przygotowanie do gry. Jest to istotny element optymalizacji, który przyczynia się do lepszych doświadczeń graczy.

9.2.2 Object Pooling

Object Pooling umożliwia efektywne ponowne wykorzystanie obiektów w grze, zamiast dynamicznego tworzenia i niszczenia ich. Podejście to pomaga zminimalizować obciążenie systemu, zwłaszcza w przypadku obiektów, które są często tworzone i usuwane, takich jak efekty cząsteczkowe, wskaźniki obrażeń, popupy z obrażeniami czy rany po pociskach.

Skrypt **ObjectPoolManager.cs** jest centralnym elementem zarządzającym pulą obiektów. Jeżeli interesuje Cię sama implementacja funkcji do tworzenia puli obiektów oraz tej

odpowiedzialnej za ich powrót do puli to sugerujemy powrót do podsekcji Wzorzec puli obiektów gdzie możesz znaleźć code snippet, którego szukasz! Poniżej przedstawiono główne funkcje skryptu:

- **SpawnObject:** Funkcja ta służy do tworzenia obiektów z puli. Sprawdza, czy dany obiekt już istnieje w puli. Jeśli nie, tworzy nowy obiekt; jeśli tak, ponownie go aktywuje i umieszcza na odpowiedniej pozycji.
- **ReturnObjectToPool:** Funkcja odpowiada za zwrot obiektu do puli po zakończeniu jego używania. Obiekt jest dezaktywowany i dodawany z powrotem do puli.
- **SetParentObject:** Funkcja ta ustawia rodzica dla danego obiektu w zależności od jego typu. Pomaga to w utrzymaniu porządku w hierarchii obiektów w scenie.

Dodatkowo, skrypty które znajdziesz poniżej prezentują konkretne przypadki użycia *puli obiektów* w akcji:

- Skrypt **ReturnParticlesToPool.cs:** Skrypt ten reprezentuje konkretny przypadek użycia *puli obiektów* w kontekście efektów cząsteczkowych. Po zakończeniu emisji cząsteczek wywołuje funkcję zwracającą obiekt do puli.

```
public class ReturnParticlesToPool : MonoBehaviour
{
    private void OnParticleSystemStopped()
    {
        ObjectPoolManager.ReturnObjectToPool(gameObject);
    }
}
```

Fragment kodu 10: Zwrócenie obiektu do puli po zakończeniu trwania efektu systemu cząsteczkowego

- Skrypt **ReturnToPoolAfterTimer.cs** Ten skrypt ilustruje użycie *puli obiektów* w kontekście czasowego zwalniania zasobów. Po aktywowaniu obiektu, rozpoczyna odliczanie czasu i po upływie określonego czasu, zwraca obiekt do puli.

```

public class ReturnToPoolAfterTimer : MonoBehaviour
{
    public float timeToDespawn = 1f;
    private Coroutine _timerCoroutine;

    private void OnEnable()
    {
        _timerCoroutine =
            StartCoroutine(ReturnToPoolAfterTime());
    }

    private IEnumerator ReturnToPoolAfterTime()
    {
        float elapsedTime = 0f;

        while(elapsedTime < timeToDespawn)
        {
            elapsedTime += Time.deltaTime;
            yield return null;
        }

        ObjectPoolManager.ReturnObjectToPool(gameObject);
    }
}

```

Fragment kodu 11: Zwrócenie obiektu do puli po upływie określonego czasu

- Skrypt **ReturnToPoolOnAnimationEnd.cs**: Skrypt ten prezentuje wykorzystanie *puli obiektów* w kontekście zwalniania zasobów po zakończeniu animacji. Po zakończeniu animacji obiektu, rodzic tego obiektu zostaje zwrócony do puli.

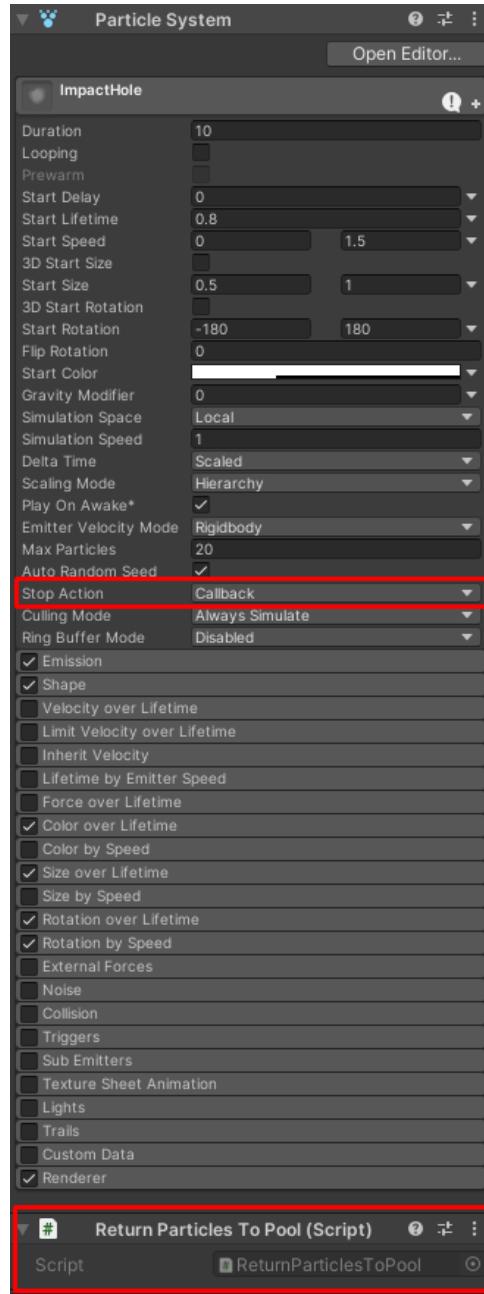
```

public class ReturnToPoolOnAnimationEnd : MonoBehaviour
{
    public void DestroyParent()
    {
        GameObject parentObject =
            gameObject.transform.parent.gameObject;
        ObjectPoolManager.ReturnObjectToPool(parentObject);
    }
}

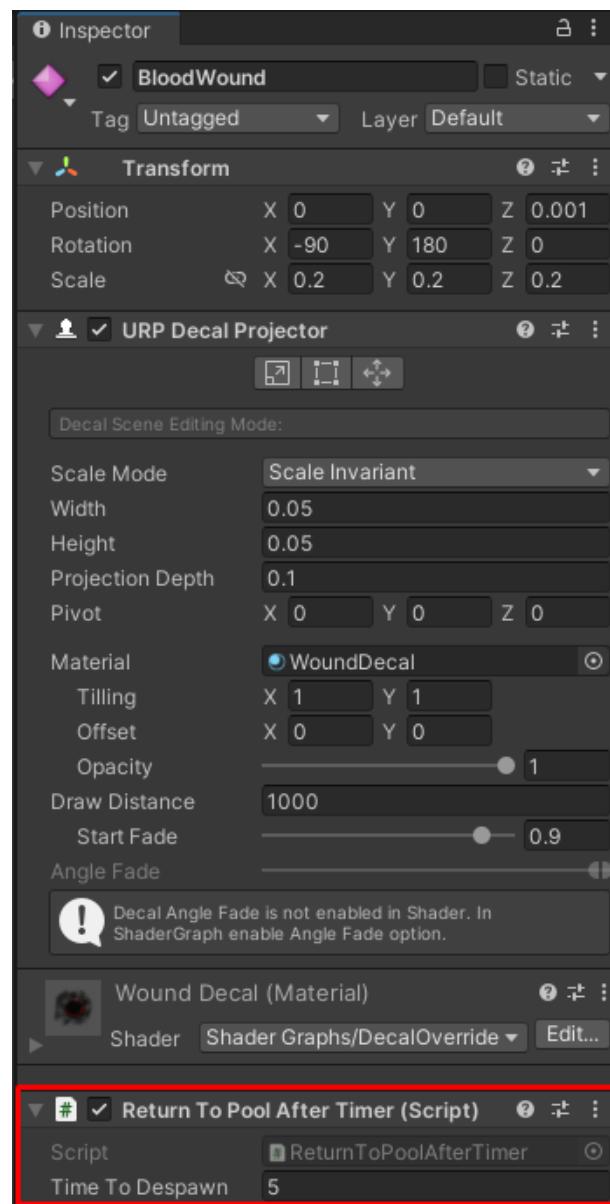
```

Fragment kodu 12: Klasa ReturnToPoolOnAnimationEnd wykorzystywana do zwrócenia obiektu do puli po zakończeniu trwania animacji przy użyciu zdarzeń animacji Unity

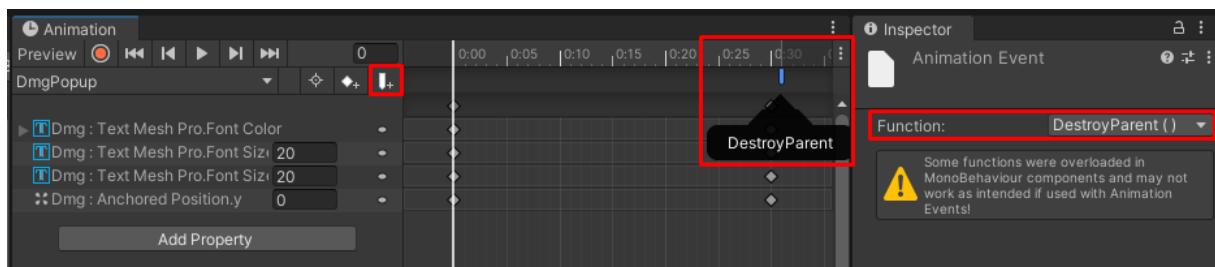
Poniżej znajdziesz konfiguracje obiektów w zależności od sposobu w jaki możesz je dodać do systemu *puli obiektów* oraz przykład podmiany wbudowanych i kosztownych funkcji silnika Unity, czyli *Instantiate* oraz *Destroy*.



Rysunek 51: Konfiguracja obiektu Particle System, który powinien wrócić do puli obiektów po zakończeniu emisji cząsteczek



Rysunek 52: Konfiguracja obiektu, który ma wrócić do puli obiektów po upływie czasu



Rysunek 53: Konfiguracja obiektu, który ma wrócić do puli obiektów po zakończeniu odtwarzania animacji

```

public class DISystem : MonoBehaviour
{
    [SerializeField] private GameObject indicatorPrefab;
    [SerializeField] private RectTransform holder;

    // ... (Remaining part of the script)

    private void Create(Transform target)
    {
        // ... (Remaining part of the function)

        GameObject indicator =
            ObjectPoolManager.SpawnObject(indicatorPrefab,
                Vector3.zero, Quaternion.identity, holder);

        // ... (Remaining part of the function)
    }
}

```

Fragment kodu 13: Przykład podmiany funkcji Instantiate w skrypcie DISystem.cs na nowo utworzoną i zoptymalizowaną SpawnObject z klasy ObjectPoolManager

```

public class DamageIndicator : MonoBehaviour
{
    // ... (Remaining part of the script)

    private IEnumerator Countdown()
    {
        // ... (Remaining part of the function)

        unRegister();
        ObjectPoolManager.ReturnObjectToPool(gameObject);
    }
}

```

Fragment kodu 14: Przykład podmiany funkcji Destroy w skrypcie DamageIndicator.cs na nowo utworzoną i zoptymalizowaną ReturnObjectToPool z klasy ObjectPoolManager

9.3 Techniki optymalizacji wydajności

9.3.1 Wyszukiwanie Obiektów

Podczas programowania w Unity, efektywne wyszukiwanie obiektów w scenie jest kluczowe dla wydajności gry. Unikamy zasobozernych funkcji, takich jak **FindObjectOfType**, szczególnie w każdej klatce, aby uniknąć zbędnego obciążenia. Ostatecznie wyeliminowaliśmy wszystkie użycia tej funkcji z kilku prostych powodów, ale m.in. dlatego, że funkcja

ta przeszukuje całą scenę (mówimy tu o wszystkich obiektach gry i o wszystkich ich komponentach!) oraz nie jest ograniczona do konkretnego obszaru lub hierarchii.

Zamiast tego, zalecamy korzystanie z bardziej efektywnych metod, takich jak:

- **Bezpośrednie Referencje w Edytorze Unity:** Przypisywanie referencji bezpośrednio w inspektorze Unity może być bardzo efektywne. Jeśli dany obiekt jest dostępny bezpośrednio poprzez referencję, nie trzeba korzystać z funkcji wyszukujących.
- **GetChild:** Funkcja pozwalająca na odnalezienie konkretnego dziecka obiektu, co jest przydatne, gdy hierarchia obiektów jest złożona.
- **GetComponent:** Pozwala na bezpośrednie odnalezienie komponentu przypisanego do obiektu. Jest to efektywna metoda, zwłaszcza gdy wiemy, że dany obiekt posiada określony komponent.
- **GetComponentInParent:** Metoda umożliwiająca znalezienie komponentu w hierarchii rodzica obiektu.
- **GetComponentInChildren:** Funkcja wyszukująca komponenty wśród dzieci danego obiektu.
- **FindGameObjectWithTag:** Efektywne wyszukiwanie obiektów po tagu. Jest to skuteczna metoda, jeśli obiekty są odpowiednio oznaczone tagami.
- **Find:** Możemy używać ogólnego wyszukiwania obiektów, takiego jak Find, w celu znalezienia obiektu po jego nazwie. Należy jednak używać go z umiarem, aby uniknąć nadmiernego obciążenia.

Optymalnym rozwiązańiem jest również projektowanie systemu, w którym unikamy częstego wyszukiwania obiektów w czasie rzeczywistym. Zamiast tego, stosujmy strategie takie jak cachowanie referencji, używanie menedżerów lub struktur organizacyjnych w scenie, aby minimalizować liczbę operacji wyszukiwania. To podejście przyczyni się do płynności działania gry, zwłaszcza w przypadku dużych projektów.

Refactor code for better performance

- Improved performance by completely removing FindObjectOfType calls
- Improved performance by caching repeated GetComponent calls

Paweł Trojański - 26edbea8 +116 -106

17 changed files	Assets\Scripts\Player\PlayerInventory.cs
Assets\Scenes\Game.unity	13 13 public Image grenadeWeaponImage;
Assets\Scenes\TutorialUnity	14 14 public Image flashbangWeaponImage;
Assets\Scripts\Equipment\WeaponRecoil.cs	15 15 public Image smokeWeaponImage;
Assets\Scripts\Equipment\WeaponSway.cs	16 17 private PlayerUI playerUI;
Assets\Scripts\Interactable\AmmoBox.cs	17 18 [Header("Weapon")]
Assets\Scripts\Interactable\BodyArmor.cs	18 19 [SerializeField] private Gun melee;
Assets\Scripts\Interactable\Coffin.cs	39 40 @@ -39,6 +40,7 @@ namespace RatGamesStudios.OperationDeratization.Player
Assets\Scripts\Interactable\Container.cs	40 41 MeshRenderer meshRenderer = mesh.GetComponent<MeshRenderer>();
Assets\Scripts\Interactable\Door.cs	41 42 meshRenderer.shadowCastingMode = ShadowCastingMode.Off;
Assets\Scripts\Interactable\DoorMotionSensor.cs	42 43 meshRenderer.receiveShadows = false;
Assets\Scripts\Interactable\FirstAidKit.cs	43 44 playerUI = GetComponent<PlayerUI>();
Assets\Scripts\Interactable\LadderTrigger.cs	44 45 weapons = new Gun[6];
Assets\Scripts\Interactable\Weapon.cs	45 46 weapons[0] = melee;
Assets\Scripts\Player\PlayerInventory.cs	112 114 weapons[1] = null;
Assets\Scripts\Player\PlayerMotor.cs	113 115 @@ -112,7 +114,7 @@ namespace RatGamesStudios.OperationDeratization.Player
Assets\Scripts\Player\PlayerShoot.cs	114 116 if (newItem.currentAmmoCount < newItem.editorAmmoValue)
	115 117 newItem.currentAmmoCount = newItem.editorAmmoValue;
	116 118 else
	117 119 FindObjectOfType<PlayerUI>().ShowGrenadePrompt(newItem.gunName);
	playerUI.ShowGrenadePrompt(newItem.gunName);

Rysunek 54: Optymalizacja pozyskiwania obiektów - porzucenie wywołań FindObjectOfType

9.3.2 Konstruktor Łańcucha Znaków

String Builder to klasa w języku C#, która umożliwia dynamiczne tworzenie, manipulowanie i modyfikowanie łańcuchów znaków. Główną różnicą między String Builder a standardowymi łańcuchami znaków (typ string) jest to, że String Builder jest mutowalny, co oznacza, że można go modyfikować bez tworzenia nowych instancji obiektów za każdym razem, gdy dokonywane są zmiany.

1. Zalety String Buildera

- **Efektywność pamięciowa:** W przypadku operacji na łańcuchach znaków, szczególnie gdy wymagane są powtarzane modyfikacje, String Builder oferuje znaczną efektywność pamięciową. Ponieważ obiekty typu string są niemodyfikowalne, każda operacja modyfikacji tworzy nowy obiekt. String Builder minimalizuje ten problem, pozwalając na modyfikację istniejącego obiektu bez konieczności tworzenia nowego za każdym razem.
- **Szybkość operacji modyfikacji:** Operacje modyfikacji, takie jak dodawanie, usuwanie lub zamiana znaków, są szybsze w przypadku String Builder niż dla typu string. W przypadku typu string, każda operacja modyfikacji tworzy nowy obiekt, co wpływa na wydajność, szczególnie w przypadku dużej liczby operacji.
- **Efektywność czasowa:** Dla operacji, które wymagają wielokrotnych modyfikacji, String Builder jest bardziej efektywny czasowo niż konkatenacja łańcuchów znaków

(string + string). Konkatenacja za każdym razem tworzy nowy obiekt, co może prowadzić do kosztownych operacji kopiowania danych.

2. Zastosowanie String Buildera

- **Budowanie długichłańcuchów znaków:** Jeśli musisz dynamicznie budować długiełańcuchy znaków, String Builder jest bardziej efektywny niż konkatenacja standardowychłańcuchów.
- **Częstymodyfikacje tekstu:** W przypadku, gdy tekst podlega częstym zmianom, a konieczność tworzenia nowych obiektów jest nieefektywna, String Builder pozwala na płynne modyfikacje bez nadmiernego obciążenia pamięciowego.
- **Wydajneskładanie wiadomości lub komunikatów:** W przypadku tworzenia komunikatów, logów lub innych dynamicznych komunikatów tekstowych, String Builder umożliwia efektywne tworzenie i modyfikowanie treści.

3. Tworzenie Obiektu String Buildera

String Builder może być inicjowany różnymi sposobami. Poniżej przedstawiono jedną z metod tworzenia obiektu String Builder:

```
StringBuilder stringBuilder = new StringBuilder();
```

Ten konstruktor tworzy pusty obiekt String Builder, który może być następnie modyfikowany przez różne operacje, takie jak dodawanie, usuwanie czy zamiana znaków.

The screenshot shows the Unity Editor interface with the title "String Builder partially added". It displays a list of changed files: Assets\Scenes\Game.unity, Assets\Scripts\Enemy\EnemyHealth.cs, Assets\Scripts\Player\PlayerUI.cs (highlighted with a red box), and Assets\Setup\Utility\PlayerUI.prefab. The code editor shows the PlayerUI.cs script with the following content:

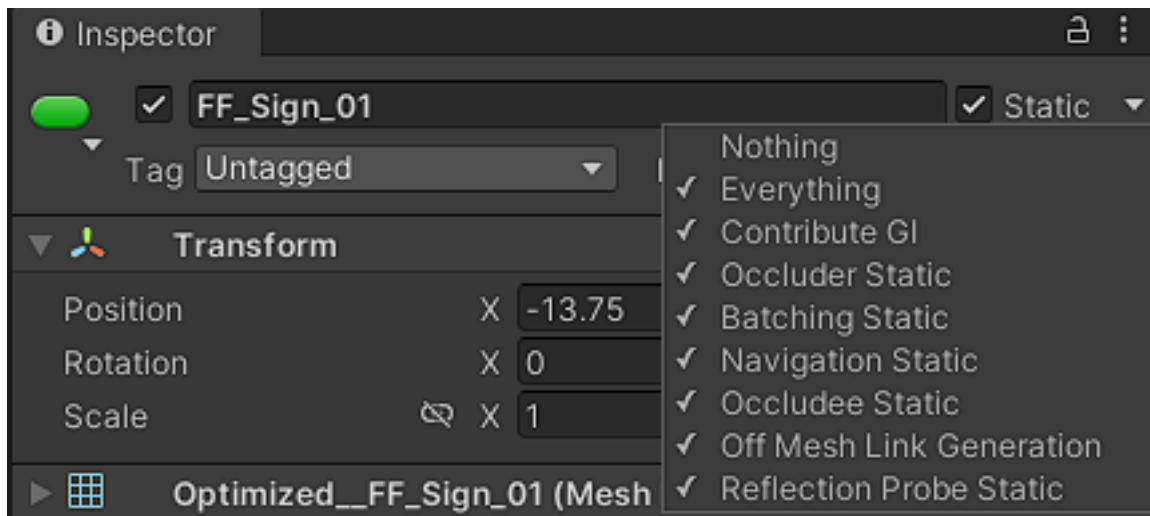
```
private void UpdateAmmoText()
{
    ammoTextBuilder.Clear();
    if (currentWeapon.gunStyle != GunStyle.Melee)
    {
        if (currentWeapon.gunStyle == GunStyle.Grenade || currentWeapon.gunStyle == GunStyle.Flashbang
            || currentWeapon.gunStyle == GunStyle.Smoke)
        {
            ammoText.text = currentWeapon.currentAmmoCount.ToString();
            ammoTextBuilder.Append(currentWeapon.currentAmmoCount.ToString());
        }
        else
        {
            ammoText.text = currentWeapon.currentAmmoCount + " / " + currentWeapon.maxAmmoCount;
            ammoTextBuilder.Append($"{currentWeapon.currentAmmoCount} / {currentWeapon.maxAmmoCount}");
        }
    }
    else
    {
        ammoText.text = "";
        ammoTextBuilder.Append("");
    }
    ammoText.text = ammoTextBuilder.ToString();
}
```

Red arrows highlight the use of the Append method to build the string step-by-step, illustrating the performance benefits over traditional string concatenation.

Rysunek 55: Optymalizacja modyfikowania ciągów znaków - zastosowanie String Buildera

9.3.3 Pola Statyczne Obiektów

Pola statyczne mogą być używane w celu przechowywania danych, które są wspólne dla wszystkich instancji danej klasy. Omówimy, kiedy i jak używać pól statycznych w celu zoptymalizowania zarządzania danymi w grze.



Rysunek 56: Odpowiednie wykorzystanie pola Static na obiekcie

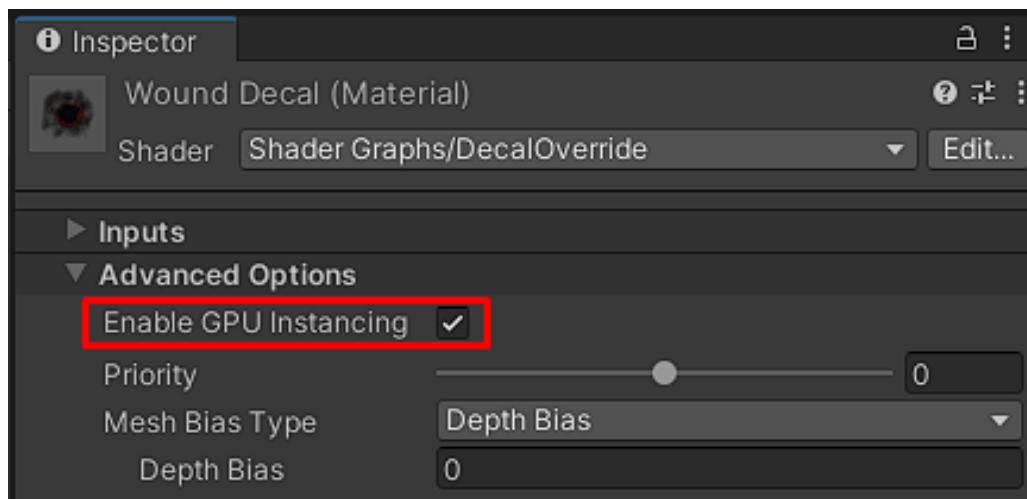
Opcje Statycznych Pól Obiektów w Unity – Statyczne pola obiektów w Unity oferują kilka opcji do kontrolowania zachowania obiektu w scenie. Oto opcje i ich znaczenia:

- **Nothing:** Obiekt nie przyczynia się do statycznego zbiorczego renderowania (batching), eliminowania elementów niewidocznych (occlusion culling) ani globalnego oświetlenia (GI).
- **Everything:** Obiekt uczestniczy w batchingu, occlusion cullingu oraz GI.
- **Contribute GI:** Obiekt przyczynia się do obliczeń GI, ale nie uczestniczy w batchingu ani occlusion cullingu.
- **Occluder Static:** Obiekt działa jako zakrywacz dla statycznych obliczeń occlusion culling.
- **Batching Static:** Obiekt uczestniczy w batchingu, ale nie w GI ani occlusion cullingu.
- **Navigation Static:** Obiekt jest uwzględniany w generowaniu statycznej siatki nawigacyjnej.
- **Occludee Static:** Obiekt jest brany pod uwagę w occlusion cullingu, ale nie działa jako zakrywacz.

- **Off Mesh Link Generation:** Obiekt jest uwzględniany w generowaniu połączeń międzymeshowych dla nawigacji.
- **Reflection Probe Static:** Obiekt przyczynia się do statycznych obliczeń sond odbicia.

9.3.4 Pola GPU Instancing

Aktywowanie opcji GPU Instancing dla powtarzających się obiektów umożliwia renderowanie ich jednym wywołaniem, co przyśpiesza proces renderowania. Poniżej przedstawiamy konfigurację materiału w celu zastosowania tej techniki.



Rysunek 57: Renderowanie obiektów z użyciem GPU Instancing

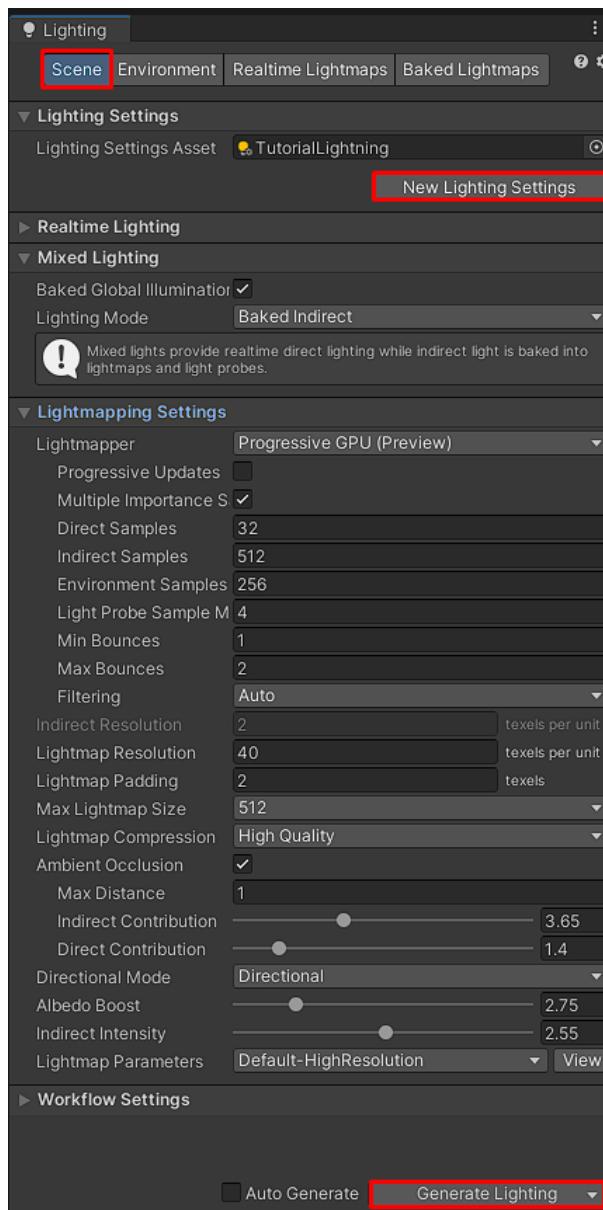
9.3.5 Wypiekanie Światel

Wypiekanie światel to proces wstępnych obliczeń oświetlenia, co może znacznie poprawić wydajność renderowania w czasie rzeczywistym. Przedstawimy korzyści i kroki do przeprowadzenia wypiekania światel w projekcie.

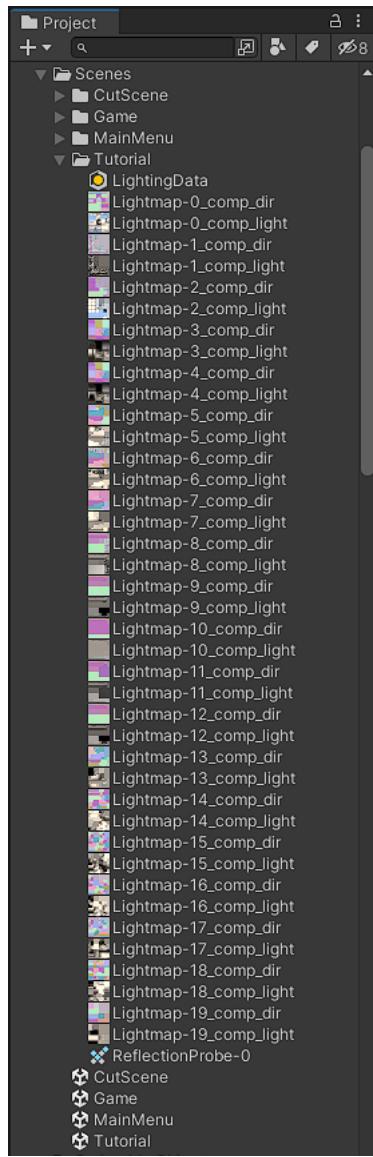
- **Poprawiona Wydajność Renderowania w Czasie Rzeczywistym:** Poprzez wstępne obliczenia informacji oświetleniowej, zmniejsza się konieczność obliczeń w czasie rzeczywistym podczas rozgrywki, co przekłada się na płynniejszą wydajność renderowania.
- **Spójność Oświetlenia:** Wypiekanie światel zapewnia spójność oświetlenia pomiędzy scenami, poprawiając jednolity wygląd i utrzymując bardziej wyszukany charakter wizualny.
- **Zwiększone Detale i Cienie:** Wypiekanie oświetlenia pozwala uzyskać skomplikowane detale i realistyczne cienie, przyczyniając się do bardziej immersyjnego i atrakcyjnego otoczenia.

- **Optymalizacja dla Urządzeń Nisko-Wydajnych:** Wypiekanie światel może być szczególnie korzystne dla optymalizacji projektów skierowanych na urządzenia nisko-wydajne, zapewniając lepsze doświadczenie użytkownika na różnych rodzajach sprzętu.
- **Przewidywalne Wyniki Oświetleniowe:** Ponieważ oświetlenie jest wstępnie obliczane, można osiągnąć przewidywalne i możliwe do odtworzenia wyniki oświetleniowe, co ułatwia precyzyjne dostrojenie wizualnej estetyki projektu.

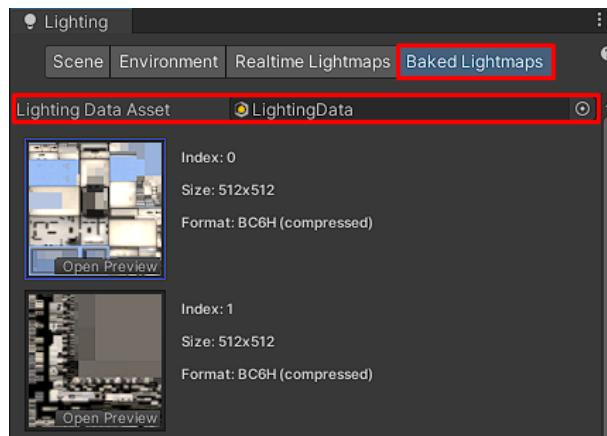
Dzięki wykorzystaniu wypiekania światel można znaleźć równowagę między jakością wizualną a wydajnością renderowania, tworząc bardziej przyjemne doświadczenie dla graczy.



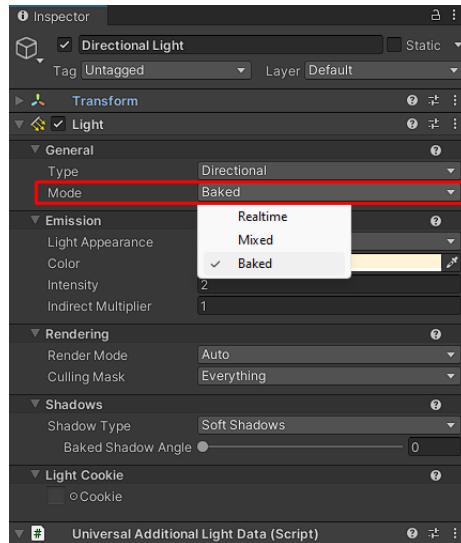
Rysunek 58: Zakładka Scene ustawień oświetlenia, w której tworzymy plik dla sceny, a następnie konfigurujemy parametry i na koniec klikamy w Generate Lighting



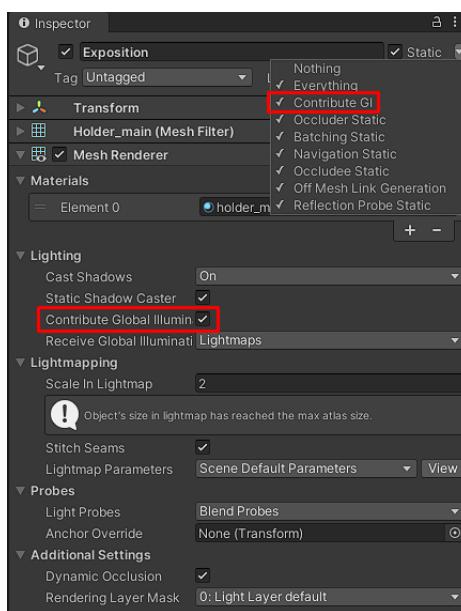
Rysunek 59: Katalog zawierający wszystkie pliki wypieczonego oświetlenia



Rysunek 60: Zakładka Baked Lightmaps okna oświetlenia, w której sprawdzimy wszystkie wygenerowane dane oświetlenia dla wybranego pliku oświetlenia sceny



Rysunek 61: Konfiguracja obiektów światła - ustawiamy tryb Mixed lub Bake by móc korzystać z wypiekanych map

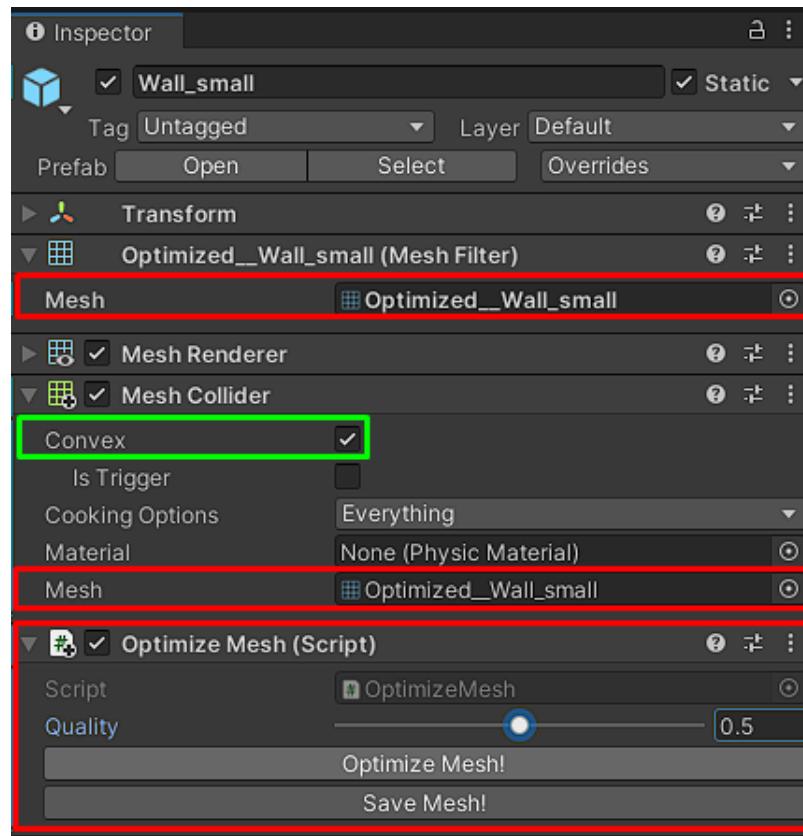


Rysunek 62: Ustawienia statycznego obiektu, który ma brać udział w wypiekaniu światła

9.3.6 Optymalizacja Meshów

Wprowadzenie odpowiednich optymalizacji meshów może znacznie poprawić wydajność renderowania. Poniżej przedstawiamy przykładowy obiekt, który wykorzystuje opcjonalnie zoptymalizowany mesh i Mesh Collider z low poly wersją oryginału. Mesh Collider dodatkowo może być skonfigurowany poprzez zaznaczenie pola "Convex". Zaleca się używanie tej opcji w przypadku, gdy mesh jest całkowicie zamknięty, bez otworów czy dziur. Dodatkowo używamy ze skryptu `OptimizeMesh.cs` z paczki dostępnej na Asset Store: **Mesh Optimizer**. Ten skrypt

umożliwia dynamiczną optymalizację mesha przy użyciu suwaka "Quality". Zmniejszając wartość suwaka, można skutecznie obniżyć jakość mesha, co może być szczególnie użyteczne dla optymalizacji renderowania lub samego collidera. Po dostosowaniu jakości, istnieje opcja zapisu zoptymalizowanego mesha do zasobów projektu.



Rysunek 63: Możliwe sposoby na optymalizację mesha obiektu

Literatura

- [1] Unity3d quicktip 31 – optymalizacja gry: Skrypty. <https://mwin.pl/unity3d-quicktip-31-optymalizacja-gry-part-i-skrypty/>, August 2015.
- [2] Difference between system.random and unityengine.random. <https://discussions.unity.com/t/what-is-the-difference-between-system-random-and-unityengine-random/160442>, March 2016.
- [3] High-level navmesh building components. <https://github.com/Unity-Technologies/NavMeshComponents/tree/master/Documentation>, April 2017.
- [4] Unity ui best practices. <https://medium.com/@dariarodionovano/unity-ui-best-practices-40964a7a9aba>, June 2018.
- [5] 10 unity audio tips (that you won't find in the tutorials). <https://gamedevbeginner.com/10-unity-audio-tips-that-you-wont-find-in-the-tutorials-2/?fbclid=IwAR0X451xD8c1iP9wiQjQ6fY723mUs75ed9pesUX0b4iEzdDWvrqq0-t0gns>, August 2022.