

P O L I T E C H N I K A Ś W I E T O K R Z Y S K A
Wydział Elektrotechniki, Automatyki i Informatyki

Anna Tutaj

Numer albumu: 070125

Opracowanie gry komputerowej FPS z wykorzystaniem silnika Unity 3D

Praca dyplomowa inżynierska
na kierunku Informatyka

Opiekun pracy dyplomowej:
dr inż. Katarzyna Rutczyńska-Wdowiak
Katedra Systemów Informatycznych

Kielce, 2016

POLITECHNIKA ŚWIĘTOKRZYSKA
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI

DZIEKAN
Zatwierdzam: Wydziału Elektrotechniki, Automatyki i Informatyki
Dziekan Wydziału dr hab. inż. Antoni Różowicz, prof. PŚk

Rok akademicki: 2015/16
Temat nr YD/1097/2015/16

Dnia 05.11.2015r.

ZADANIE NA PRACĘ DYPLOMOWĄ
INŻYNIERSKĄ

Wydano studentowi: Tutaj Anna

I. Temat pracy:

Opracowanie gry komputerowej FPS z wykorzystaniem silnika Unity 3D

II. Plan pracy:

1. Wprowadzenie.
2. Przegląd współczesnych, istniejących na rynku, gier komputerowych FPS.
3. Omówienie narzędzi programistycznych wykorzystanych w programie komputerowym.
4. Założenia i struktura gry.
5. Testy gry komputerowej.
6. Podsumowanie.

III. Cel pracy:

Celem pracy jest projekt i implementacja gry komputerowej FPS z wykorzystaniem silnika Unity 3D, oraz narzędzi, takich jak: Blender i Gimp.

IV. Uwagi dotyczące pracy:

V. Termin oddania pracy: 31 grudnia 2015

VI. Konsultant:

Kierownik Zakładu/Katedry

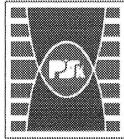
KIEROWNIK
Katedry Systemów Informatycznych
wydziału Elektrotechniki, Automatyki i Informatyki
prof. dr hab. Aleksander Jastrzębow

Opiekun pracy dyplomowej

Katarzyna Patyczko - Wolańska

Temat pracy dyplomowej celem jej wykonania otrzymałem(am):

Kielce, dnia 18.11.2015 r. Anna Tutaj
czytelny podpis studenta



Politechnika Świętokrzyska

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI

Kielce, dnia 24.01.2016 r.

Anna Tutej

Imię i nazwisko studenta

040125

nr albumu

ul. Małopolskiego 22/69 25-430 Kielce

Adres zamieszkania

Informatyka, Systemy informacyjne, II rok, studia stacjonarne

Kierunek, specjalność, rok studiów, rodzaj studiów (stacjonarne, niestacjonarne)

dr inż. Małgorzata Dutkiewicz-Adamska

Opiekun pracy dyplomowej inżynierskiej/magisterskiej*

OŚWIADCZENIE

Przedkładając w roku akademickim 2015/2016 opiekunowi pracy dyplomowej inżynierskiej/magisterskiej*, powołanemu przez Dziekana Wydziału Elektrotechniki Automatyki i Informatyki Politechniki Świętokrzyskiej, pracę dyplomową inżynierską/magisterską* pod tytułem:

Opracowanie nowej komputerowej FPS z wykorzystaniem silnika wentylatora 3D

oświadczam, że:

- 1) przedstawiona praca dyplomowa inżynierska/magisterska* została opracowana przeze mnie samodzielnie, stosownie do wskazówek merytorycznych opiekuna pracy,
- 2) przy wykonywaniu pracy dyplomowej inżynierskiej/magisterskiej* wykorzystano materiały źródłowe, w granicach dozwolonego użytku wymieniając autora, tytuł pozycji i miejsce jej publikacji,
- 3) praca dyplomowa inżynierska/magisterska* nie zawiera żadnych danych, informacji i materiałów, których publikacja nie jest prawnie dozwolona,
- 4) przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem stopnia zawodowego/naukowego w wyższej uczelni,
- 5) niniejsza wersja pracy jest identyczna z załączoną treścią elektroniczną (na CD i w systemie Archiwum Prac Dyplomowych).

Przyjmuję do wiadomości, iż w przypadku ujawnienia naruszenia przepisów ustawy o prawie autorskim i prawach pokrewnych, praca dyplomowa inżynierska/magisterska* może być unieważniona przez Uczelnię, nawet po przeprowadzeniu obrony pracy.

Zostalem uprzedzony:

- 1) o odpowiedzialności karnej wynikającej z art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t. j. Dz. U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”
- 2) odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t. j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwany dalej sądem koleżeńskim”

Prawdziwość powyższego oświadczenia potwierdzam własnoręcznym podpisem.

Anna Tutej
czytelny podpis studenta

*) niepotrzebne skreślić

Opracowanie gry komputerowej FPS z wykorzystaniem silnika Unity 3D

Streszczenie

Niniejsza praca dyplomowa zawiera opracowanie procesu projektowania gry komputerowej z gatunku FPS. Wykorzystano silnik Unity 3D oraz narzędzia służące do modelowania i teksturowania: Blender, Gimp oraz Paint Tool SAI w celu zaprojektowania rozgrywki w świecie trójwymiarowym. Gracz posiada możliwości: używania różnych broni, zbierania przedmiotów, wykonywania misji oraz walki z przeciwnikami, którzy wykorzystują sztuczną inteligencję.

Słowa kluczowe: gra, komputerowa, FPS, Unity, 3D, sztuczna, inteligencja, C#

FPS game development using Unity3D engine

Summary

This diploma thesis contains a description of a process of designing a FPS video game. Unity 3D engine and tools for modeling and texturing: Blender, Gimp and Paint Tool SAI have been used in order to design three-dimensional game. Player can: use different weapons, pick items, do quests and fight with enemies, who using artificial intelligence.

Keywords: video, game, FPS, Unity, 3D, artificial, intelligence, C#

SPIS TREŚCI

1. WSTĘP.....	11
2. PRZEGŁĄD WSPÓŁCZESNYCH GIER WRAZ Z ICH SILNIKAMI.....	13
2.1 Przegląd istniejących gier FPS.....	13
2.2 Przegląd dostępnych silników.....	14
3. UŻYTE NARZĘDZIA SŁUŻĄCE DO ZAPROJEKTOWANIA GRY ORAZ MODELOWANIA I TEKSTUROWANIA.....	18
3.1 Unity 3D.....	16
3.1.1 Instalacja Unity 3D.....	16
3.1.2 Interfejs Unity 3D.....	17
3.2.3 Kolejność wykonywania zdarzeń.....	23
3.2 Blender.....	25
3.3 Programy grafiki rastrowej.....	26
3.3.1 Gimp.....	26
3.3.2 Paint Tool Sai.....	27
4. ZAŁOŻENIA, STRUKTURA I REALIZACJA GRY.....	28
4.1 Założenia projektowe.....	28
4.2. Architektura aplikacji.....	28
4.2.1 Model ECS.....	28
4.2.2 Pakiety.....	30
4.3 Obiekt gracza.....	32
4.3.1 Omówienie obiektu gracza.....	32
4.3.2 Skrypt statystyk gracza.....	33
4.3.3 System zadań.....	37
4.3.4 Pozostałe skrypty.....	43
4.4 System broni i zbierania przedmiotów.....	45
4.4.1 Używanie broni.....	45
4.4.2 Zbieranie przedmiotów.....	47
4.5 Przeciwnicy.....	49
4.5.1 Omówienie obiektu przeciwnika.....	49
4.5.2 Automat stanów skończonych.....	50

4.5.3 Inteligentne wyznaczanie ścieżek algorytmem A*	53
4.6 System kamer.....	55
4.6.1 Kamery podczas gry.....	55
4.6.2 Przerywnik filmowy.....	57
4.6.3 Ekran ładowania.....	59
4.7 System zapisu i odczytu gry.....	60
4.8 Grafika i dźwięk.....	62
4.8.1 Projektowanie map.....	62
4.8.2 Projektowanie modeli oraz grafiki 2D.....	66
5. TESTY GRY KOMPUTEROWEJ.....	69
5.1 Warunki techniczne przeprowadzonych testów.....	69
5.2 Wyniki testów wydajnościowych.....	69
6. ZAKOŃCZENIE.....	71
LITERATURA.....	72
ZAŁĄCZNIKI.....	74

1. WSTĘP

Temat pracy inżynierskiej został wybrany ze względu na wzrastające zainteresowanie zagadnieniem gier komputerowych oraz chęcią zaprojektowania własnej aplikacji z grafiką trójwymiarową. Ten rodzaj oprogramowania komputerowego może łączyć elementy edukacji oraz rozrywki, przez co szerokie grono odbiorców potrafi czerpać korzyści wynikające z użytkowania takiej aplikacji. Według Entertainment Software Association (stowarzyszenie zajmujące się przemysłem gier komputerowych) [1]:

- 155 milionów Amerykanów gra w gry komputerowe, z czego 56% stanowią mężczyźni, a 44% kobiety,
- procentowy udział graczy pod względem wieku:
 - 26% poniżej 26 lat,
 - 30%: 18-35 lat,
 - 17%: 36-49 lat,
 - 27% powyżej 50 lat.
- najpopularniejsze rodzaje gier: 31% towarzyskie, 30% akcji, 30% puzzle/ karciane/ planszowe/ teleturnieje.

Z analizy tych danych wynika, że niezależnie od płci oraz wieku ludzie grają w gry komputerowe i nie jest to domena tylko nastolatków. Ponadto ogromny udział mają gry, które bazują na zmaganiach w wirtualnym świecie i potrafią zapewnić człowiekowi towarzystwo. Wczuwanie się w skórę bohatera akcji jest jednym w kluczowych elementów przyjemnej rozgrywki dzięki zaangażowaniu jego umysłu, zmysłów oraz oddziaływaniu na emocje.

Rodzaj gry wybrany w temacie projektu to FPS (*ang. first-person shooter*), gdzie świat widać z widoku pierwszej osoby, a gracz ma zadania do wykonania i porusza się po świecie pełnym przeciwników. Ten temat pracy inżynierskiej oprócz spełnienia wcześniej omawianych potrzeb użytkownika pozwolił na: sprawdzenie i rozwijanie umiejętności autora z zakresu programowania, projektowania oraz modelowania.

Celem pracy był projekt i implementacja gry komputerowej FPS z wykorzystaniem silnika Unity 3D oraz narzędzi, takich jak: Blender i Gimp. Skrypty były pisane w języku C# w edytorze MonoDevelop dostarczanym wraz ze środowiskiem Unity 3D. Ponadto do wykonywania własnych grafik został wykorzystany program Paint Tool Sai.

Aplikacja powinna wykorzystywać jak najlepiej możliwości oferowane przez

środowisko Unity 3D, które doskonale nadaje się do zaprojektowania wybranego typu gry. Praca inżynierska powinna przybliżyć początkującym programistom proces tworzenia gier skupiając się na wybranym środowisku.

Rozdział 1 stanowi wprowadzenie w tematykę pracy, sformułowano w nim cel i zakres pracy.

Rozdział 2 to przegląd istniejących na rynku gier FPS z uwzględnieniem, na jakich silnikach powstawały. Zawarte jest też uzasadnienie, dlaczego w pracy został wybrany silnik Unity 3D oraz porównanie go do innych dostępnych silników, ze szczególnym uwzględnieniem jego zalet.

Rozdział 3 stanowi opis narzędzi programistycznych wykorzystanych do wykonania projektu kładąc nacisk na instalację środowiska Unity 3D oraz dokładny opis interfejsu.

Rozdział 4 stanowi najważniejszą część pracy, ponieważ zawiera założenia, struktury i realizację gry, a także dużo zrzutów ekranu prezentujących gotową grę.

Rozdział 5 opisuje proces testowania gry podczas tworzenia oraz testy gotowego już produktu.

Rozdział 6 jest zakończeniem i podsumowaniem pracy z wyszczególnieniem zalet oraz wad. Zawiera także szersze spojrzenie na projekt i przemyślenia dotyczące dalszego rozwoju projektu.

2. PRZEGŁĄD WSPÓŁCZESNYCH GIER WRAZ Z ICH SILNIKAMI

2.1 Przegląd istniejących gier FPS

Do stycznia 2016 r. na rynku pojawiło się wiele ciekawych tytułów, ale głównie są to kontynuacje znanych serii. W tabeli 2.1 przedstawiono zestawienie tych gier wraz z użyтыm silnikiem.

Tabl. 2.1. Popularne gry FPS

Tytuł gry	Silnik	Autor Silnika
Counter-Strike: Global offensive	Source	Valve Corporation [2]
Fallout 4	Creation Engine	Bethesda Game Studios [3]
Call of Duty: Black Ops III	Black Ops III engine	Infinity Ward [4]
Battlefield , Medal of Honor: Warfighter	Frostbite 2	Digital Illusions CE [5]
Medal of Honor (tryb jednoosobowy)	Unreal Engine 3.0	Epic Games
Crysis 3	CryEngine	Crytek Studios

Jak widać różne produkcje korzystają z odmiennych silników i żaden autor nie ma monopolu w tej dziedzinie, więc przy tworzeniu własnej gry wybór silnika jest praktycznie dowolny. Motywacja autora odnośnie wyboru Unity 3D została omówiona w kolejnym podrozdziale 2.2.

Zaznaczyć należy, że wszystkie te gry, niezależnie od silnika, a ze względu na rodzaj jakim jest FPS, są do siebie podobne. Ich wspólnymi cechami są:

- posiadanie fabuły,
- możliwość zbierania i wykorzystywania przedmiotów,
- możliwość wyboru broni,
- zróżnicowane misje,
- oprawa graficzna i muzyczna stojąca na wysokim poziomie.

Z takimi produkcjami stoi kilkanaście zespołów ludzi (przeważnie około 100 osób), np. Call of Duty: Black Ops III tworzyło około 300 osób.

Tabela 2.1 zawiera zestawienie gier z dostępnymi trybami jedynie singleplayer (np. Fallout 4) , jak i multiplayer (np. Counter-Strike: Global Offensive). Założenie projektowe niniejszej pracy dotyczyło pierwszego typu, a więcej szczegółów na ten temat zawiera podrozdział 4.1.

2.2 Przegląd dostępnych silników

Zdobyta na studiach wiedza z zakresu OpenGL'a, która pokazała jak można tworzyć proste obiekty 3D oraz operować na nich za pomocą kodu była inspiracją do poszukania narzędzia bardziej rozbudowanego, które umożliwiły wykonanie bardziej złożonego projektu przy większym komforcie tworzenia.

Postęp w dziedzinie informatyki następuje bardzo szybko. Częste problemy i najpowszechniejsze rozwiązania są zbierane i analizowane, żeby zaoferować programistom jak najlepsze narzędzie do pracy. Było tak w np. przypadku powstawania specyfikacji HTML 5, dzięki czemu wprowadzono takie elementy blokowe jak: footer, header, article. Wcześniej programiści nagminnie sami tworzyli takie bloki poprzez znacznik div, ponieważ element nagłówka, czy stopki był po prostu konieczny przy tworzeniu strony internetowej. Zamiast używać konstrukcji `<div id="header">...</div>` HTML 5 wprowadzić bardziej przystępna możliwość używania `<header> ...</header>`, który w działaniu jest taki sam, ale używanie go jest o wiele prostsze i zwięzlejsze.

Powyższy przykład prowadzi do następującego wniosku - nie ma potrzeby korzystania ze starych, topornych rozwiązań, skoro postęp w technologii dostarcza czegoś nowego, wartego uwagi, a do tego sprawnego. Zatem napisanie własnego silnika do gry FPS w czystym i znanym OpenGL'u nie był konieczny. W celu realizacji projektu wybrano silnik Unity 3D Personal Edition ze względu na kilka następujących czynników.

Pierwszym z nich była licencja umożliwiająca darmowe pobranie oraz zarabianie na wykonanym produkcie. Jeśli jednak zysk przekroczy \$100 000 jest obowiązek zakupienia wersji Professional, która posiada wiele rozszerzeń. Kosztuje ona \$1 000 lub \$85 miesięcznie [6]. Drugim ważnym czynnikiem był interfejs - na pierwszy rzut oka wydający się być intuicyjny i przystępny. Kolejnym powodem była możliwość pisania skryptów w języku C# albo JavaScript, do których jest dostępna na stronie domowej Unity rzetelnie napisana dokumentacja oraz duża liczba poradników. Ponadto są dostępne książki, które wprowadzają do pracy z Unity i tworzenia gier. Przy tworzeniu pracy korzystano z Unity 3.x Game Development Essentials [7] oraz Unity Game Development with C# [8]. Następną przyczyną było umożliwienie tworzenie aplikacji 2D albo 3D, które można uruchomić w przeglądarce internetowej dzięki wtyczce Unity Web Player albo działałyby na różnych platformach:

- Komputery: Windows, Linux, Mac,
- Mobilne np. iOS, Windos Phone, Android,
- Konsole np. PS4, Xbox360.

Konkurencją dla Unity 3D jest niewątpliwie Unreal Engine, na którym powstały bardzo popularne tytuły, takie jak Unreal, Assassin's Creed albo Postal, a także znane Batman Arkham Knight, czy Mortal Kombat X. W chwili, gdy autor interesował się silnikami było możliwe korzystanie z Unreal Engine przy opłacaniu comiesięcznego abonamentu, a do tego dochodził procent odtrącanego za każdą sprzedaną kopię. Na chwilę obecną jest możliwe darmowe pobranie tego silnika, ale licencja zawiera zapis o tym, że przy zysku powyżej \$3000 twórcom silnika należy się 5% za każdy sprzedany egzemplarz. Ta zmiana w licencji była spowodowana właśnie wzrastającym zainteresowaniem silnikiem Unity 3D, na które decydowało się coraz więcej producentów (m. in powstały wtedy Dead Trigger 2, Wasteland 2, Tactics, Deus Ex: The Fall) [9].

Na wspomnienie zasługuje również GameMaker: Studio, istniejące na rynku od 1999 r. Jest jednak zbyt ograniczone pod względem możliwości, przystosowane raczej do tworzenia prostych gier, głównie 2D. Do tego licencja jest niezbyt atrakcyjna – darmowa pozwala na publikowanie tylko dla Windowsa, należy zapłacić \$100, aby móc publikować na inne platformy.

3. UŻYTE NARZĘDZIA SŁUŻĄCE DO ZAPROJEKTOWANIA GRY ORAZ MODELOWANIA I TEKSTUROWANIA

3.1 Unity 3D

Unity 3D jest elastycznym i potężnym narzędziem programistycznym do tworzenia wieloplatformowych, interaktywnych gier zarówno trójwymiarowych, jak i dwuwymiarowych [10]. Zalety Unity 3D zostały omówione w porozdziale 2.2.

3.1.1 Instalacja Unity 3D

Aktualne wymagania systemowe zarówno dla developera jak i użytkownika są dostępne na stronie [11]. Na chwilę obecną w pierwszym wypadku należy mieć system operacyjny Windows 7/8/10 albo Mac OS X 10.8 +. Windows XP oraz Vista nie są wspierane. Wymagana jest także karta graficzna kompatybilna z DirectX 9.0.

Aby pobrać instalator Unity 3D należy wykonać kilka prostych kroków. Pierwszym krokiem jest wejście na stronę internetową z plikami instalacyjnymi. Link do nich został zamieszczony w odnośniku [12]. Następnie należy kliknąć na przycisk „Choose your Unity + Download”. Po zapoznaniu się z różnicami dotyczącymi płatnej i darmowej wersji (wybrane elementy przedstawione w tabeli 3.1) należy wybrać odpowiednią. Na początku polecana jest wersja niepłatna, zatem należy kliknąć na przycisk „Free Download”. Finalnie należy kliknąć na przycisk „Download Installer”.

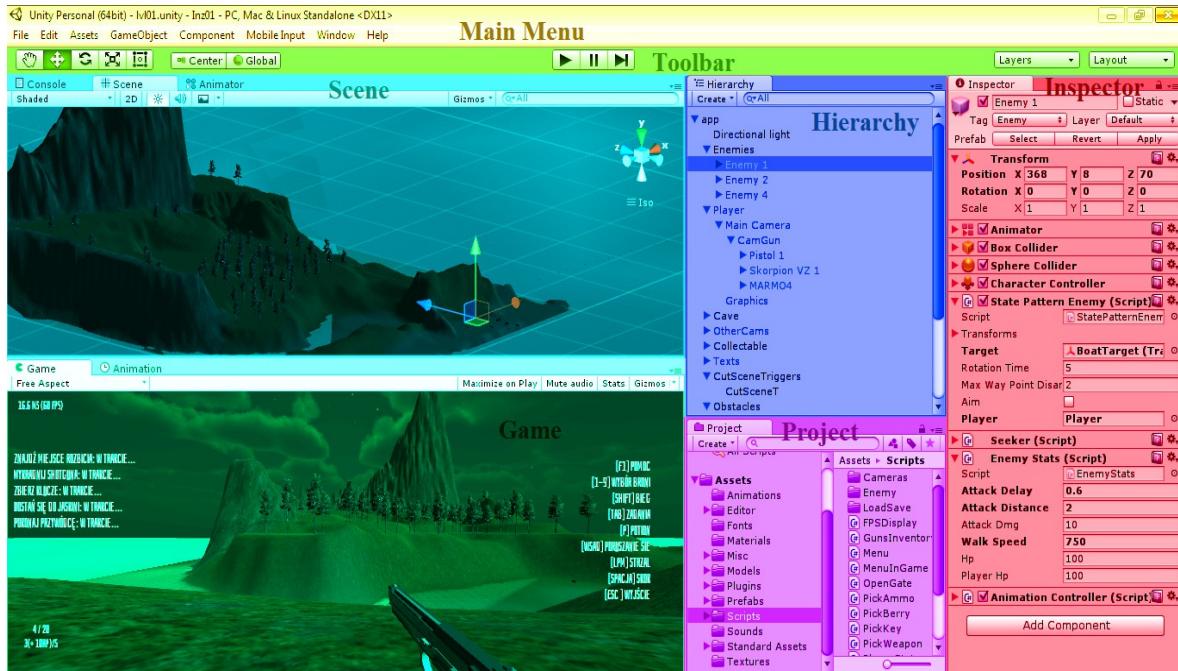
W celu instalacji środowiska należy otworzyć plik .exe znajdujący się we wcześniej wybranym folderze. Instalacja przebiega standardowo i w każdym kroku postępowania wystarczy wybierać przycisk „Next”. Dopiero przy wyborze elementów warto jest upewnić się, że środowisko MonoDevelop jest zaznaczone. Posiada darmową licencję i doskonale nadaje się do pisania w językach takich jak: C#, F#, C/C++, Visual Basic [13].

Tabl. 3.1.Porównanie dwóch wersji silnika Unity

Wersja silnika/ Cecha	Personal Edition	Professional Edition
12 miesięczny dostęp do Unity Cloud Build pro	nie	tak
Silnik ze wszystkimi funkcjami	tak	tak
Wspiera wszystkie platformy	tak	tak

3.1.2 Interfejs Unity 3D

Duży wpływ na komfort pracy ma przyjazny interfejs użytkownika. Unity 3D sprostało temu wymaganiu, co przedstawia rys. 3.1.



Rys. 3.1. Interfejs Unity 3D

Poniżej omówiono widoki interfejsu Unity 3D:

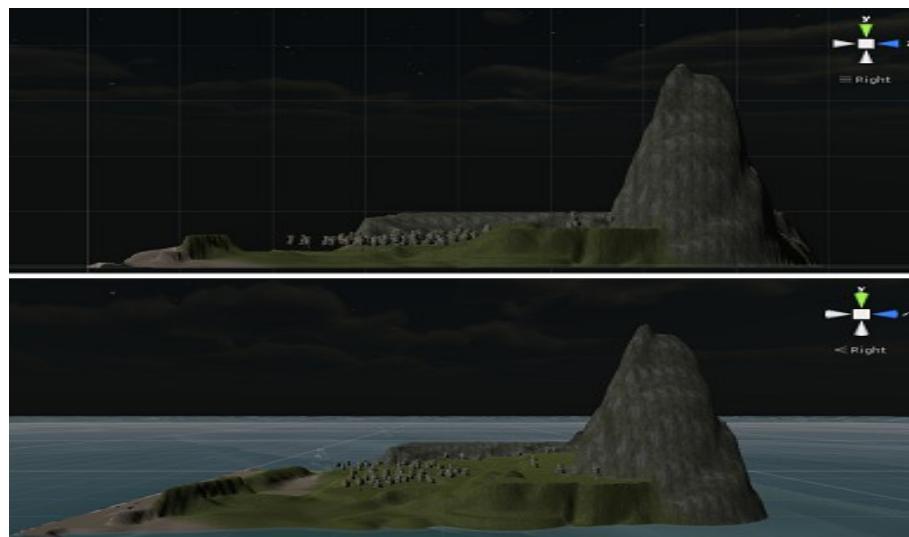
- **Main Menu** – dzięki niemu można dodawać nowe sceny, zapisywać je, tworzyć i budować projekty, ustawiać preferencje, dodawać proste obiekty (w tym kamery i światła), zarządzać oknami,
- **Toolbar** – czyli pasek narzędzi.
 - Pierwsze pięć narzędzi licząc od lewej strony odpowiada kolejno za:
 - poruszanie się po scenie,
 - przemieszczanie obiektu po wybranej osi X, Y, Z,
 - obracanie obiektu względem wybranej osi X, Y, Z,
 - skalowanie obiektu względem wybranej osi X, Y, Z,
 - zaznaczenie blokowe.
 - Dalsze dwa przyciski odpowiadają za:
 - wybór, czy kurSOR ma być położony względem środka obiektu (center), czy jego punktu centralnego (pivot),
 - wybór odniesienia współrzędnych podczas rotacji obiektu globalne albo

lokalne dla wybranego obiektu.

- Przyciski z symbolami: play do uruchamiania, stop do zatrzymywania, a ostatni do puszczenia jednej klatki,
- lista layers służy do pokazywania/ ukrywania wybranych warstw, a layouts do wyboru widoku okien.
- **Scene** – widok sceny, gdzie rozmieszcza się obiekty (modele, kamery, światła itd.). Kolorowy element będący w prawym górnym rogu służy do obracania sceny (rys. 3.2) oraz zmiany widoków - rys. 3.3 oraz rys. 3.4. Poniżej przedstawiono podstawową nawigację w widoku *Sceny*:
 - lewy przycisk myszy - zaznaczenie,
 - obrót kółkiem myszy – przybliżenie/oddalenie,
 - przytrzymanie kółka myszy i poruszanie myszy – przesuwanie obrazu,
 - prawy przycisk myszy – obrót.

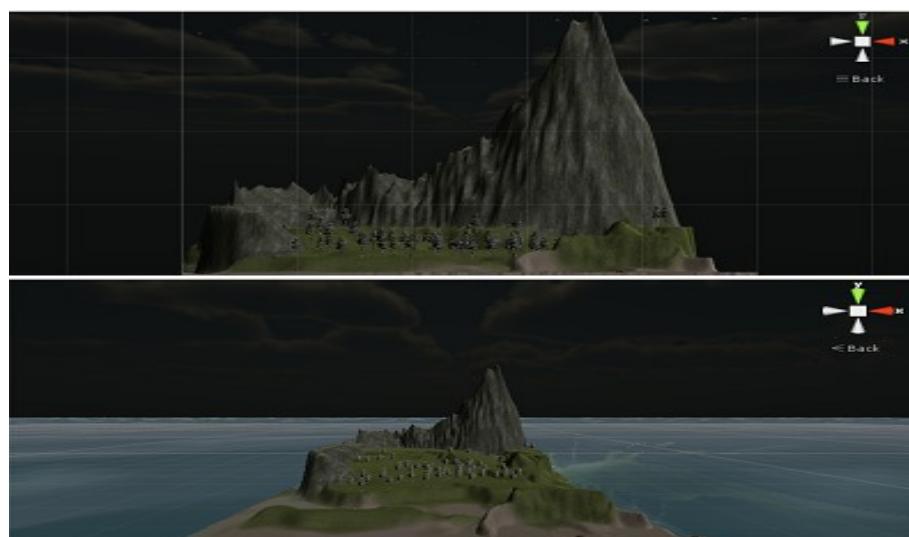


Rys. 3.2. Perspektywa z góry i z lewej strony



Rys. 3.3. Porównanie widoku prawej strony mapy dla rzutu prostokątnego (ISO)

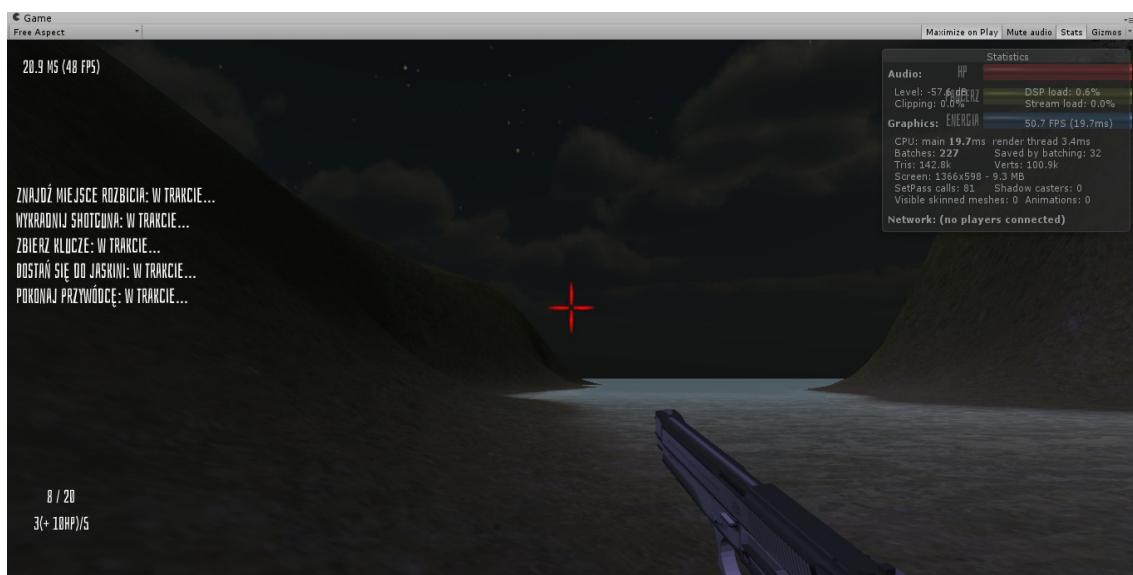
i perspektywicznego



Rys. 3.4. Porównanie widoku z przodu mapy dla rzutu prostokątnego (ISO)

i perspektywicznego

- **Game** – okno gry po wyłączeniu. Z ciekawszych opcji dostępne jest maksymalizowanie okna (domyślnie grę testuje się tylko w tym „małym” oknie) oraz wypisywanie statystyk (rys 3.5). Służy do testowania bieżącej sceny w czym pomaga okno Inspektora, gdzie można zmieniać parametry komponentów, a w oknie Gry obserwować zmiany, jakie zaszły. Należy zapamiętywać, analizować i obserwować jak zmiany wartości parametrów wpływają na rozgrywkę, ponieważ po zakończeniu testowania parametry wracają do wartości początkowych.



Rys. 3.5. Testowanie gry z podglądem statystyk

- **Hierarchy** – podczas umieszczania elementów na scenie należy utrzymywać porządek i obraną konwencję, która będzie widoczna w tym oknie. Znajduje się tu drzewo obiektów i można zarządzać nimi oraz je grupować określając przez to nadrzędnosć (rodziców) oraz składowe (dzieci), co widać na rys 3.6.

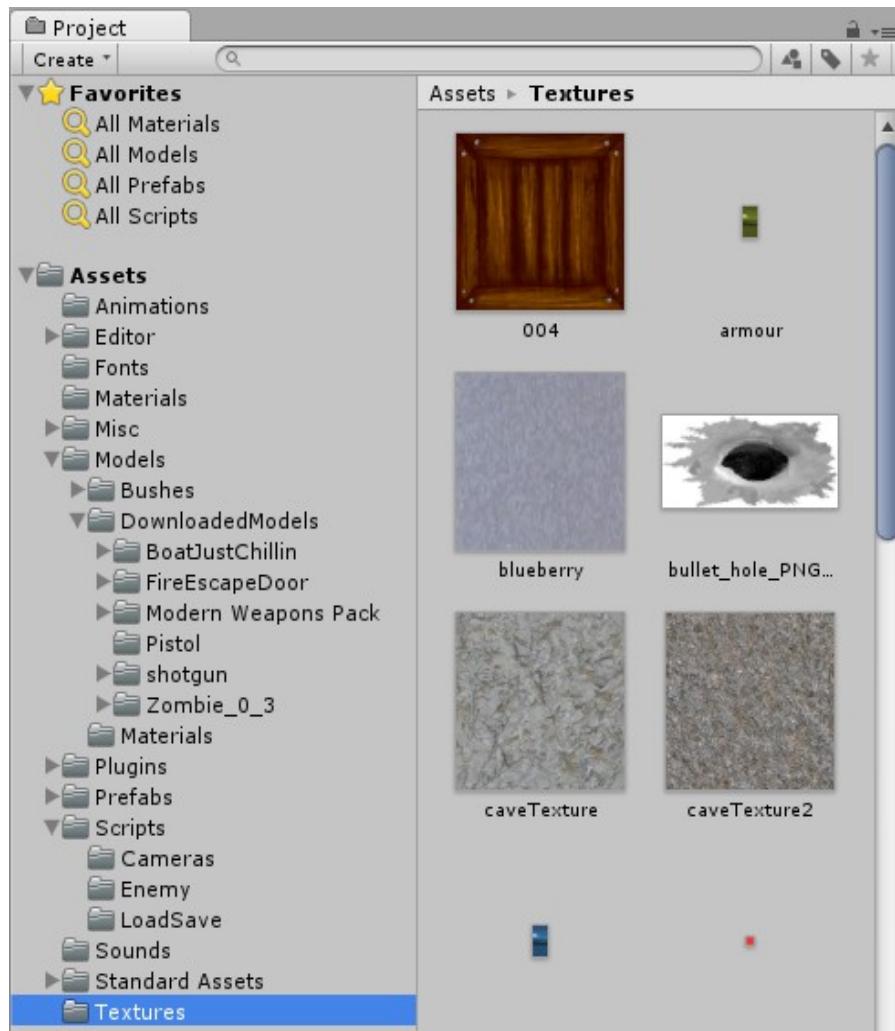


Rys. 3.6. Fragment okna hierarchii

- **Project** – w nim widać foldery, w których trzyma się składowe projektu (*ang. assets*). Są to:
 - modele,
 - tekstury,
 - skrypty,
 - dźwięki,
 - prefabrykaty (zdefiniowane obiekty gry, które można konkretyzować w późniejszym czasie)
 - inne zasoby z których korzysta projekt.

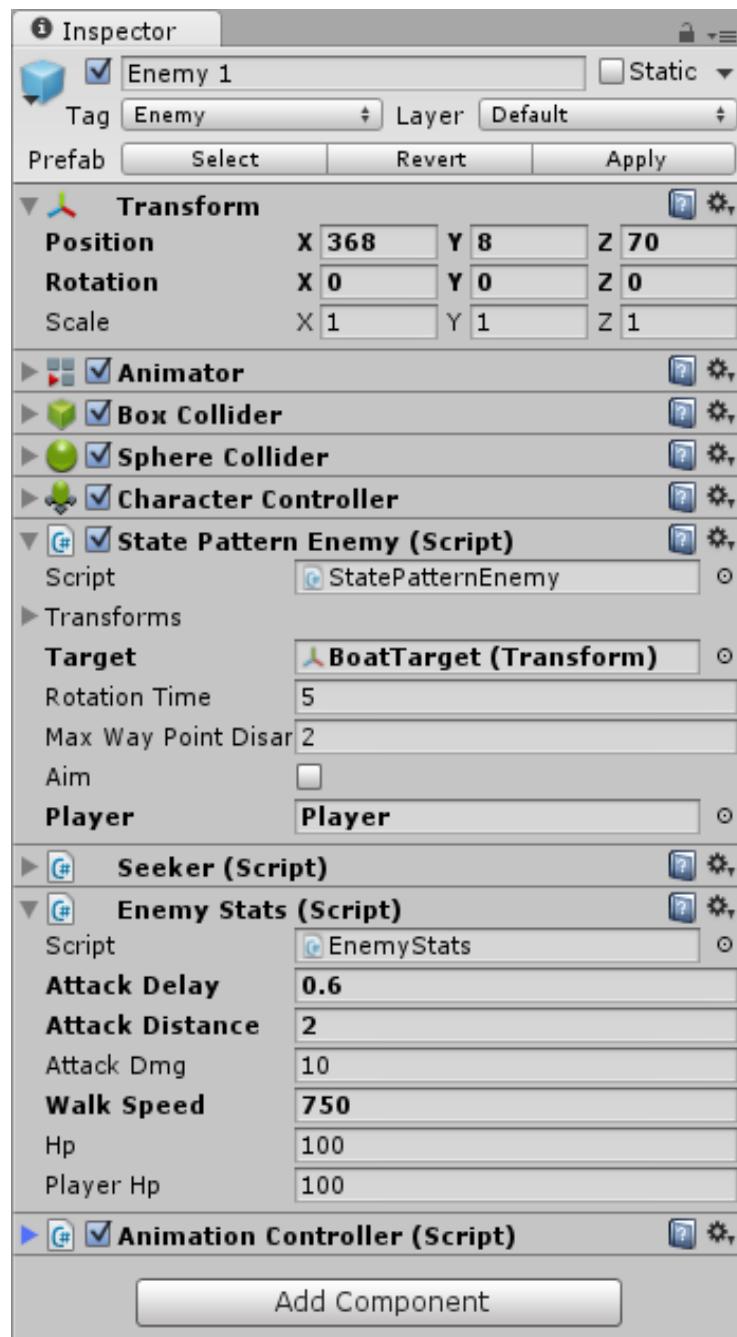
Rys. 3.7 przedstawia podział autora oraz podgląd folderu z teksturami.

W tym oknie dzięki menu **Create** albo kliknięcia prawym przyciskiem myszy można dodać nowy element np. skrypt C#.



Rys. 3.7. Okno projektu

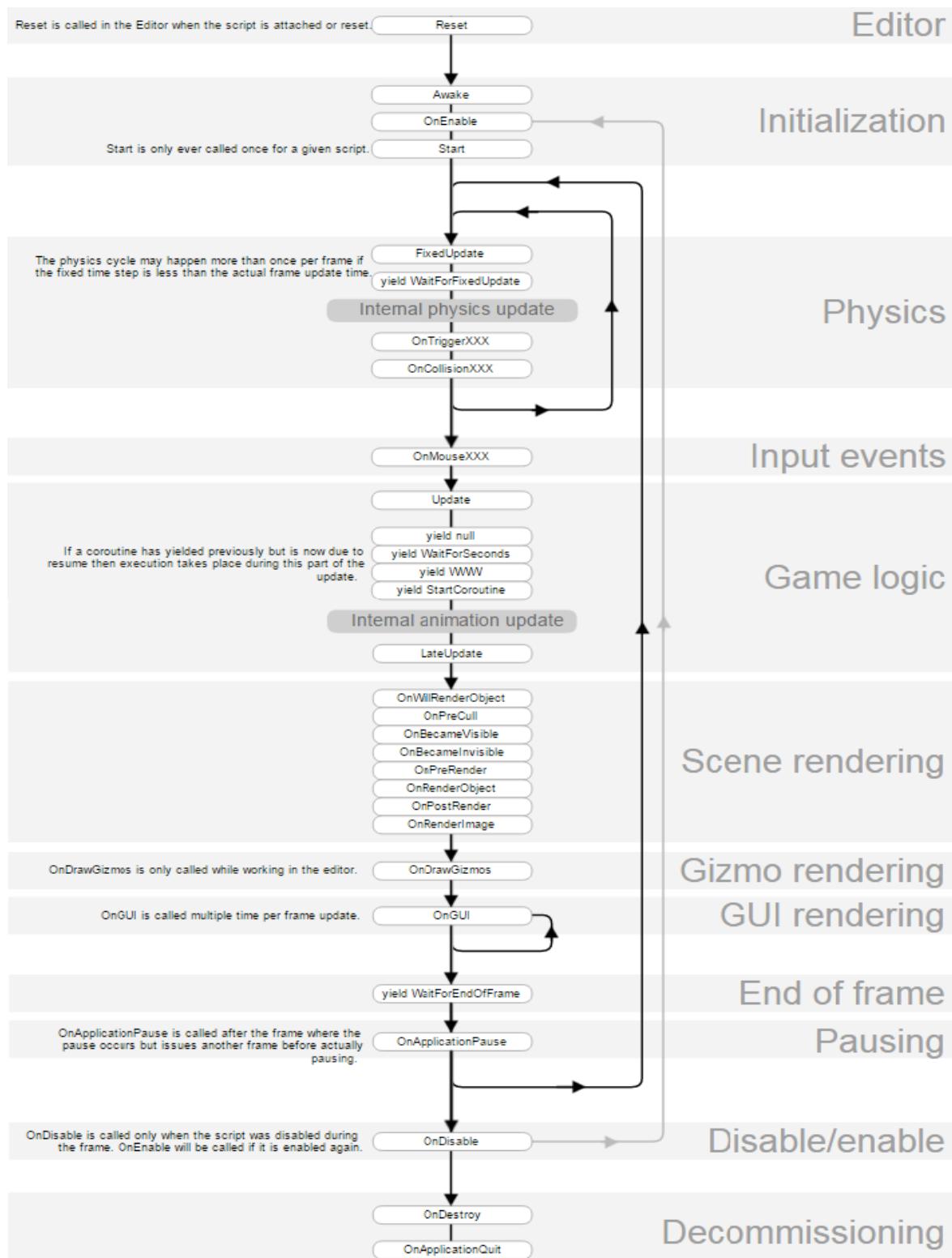
- **Inspector** – w nim wyświetlają się komponenty, o których będzie więcej napisane w punkcie 4.2.1. Pozwala na swobodne zmienianie wartości zmiennych i ustawień pokazywanych w danym komponencie. Jeśli zostanie wybrany obiekt przeciwnika (np. poprzez kliknięcie go w oknie Sceny), to w tym oknie pojawią się wszystkie informacje dotyczące niego w postaci komponentów znajdujących się w oddzielnego zakładkach. Wszystkie zmienne publiczne są widoczne i można je edytować podczas testowania gry w oknie Game.



Rys. 3.8: Okno inspektora

3.1.3 Kolejność wykonywania zdarzeń

Skrypty w Unity 3D korzystają z funkcji zdarzeń, które wykonywane są w poniższej kolejności, co pokazano na rys 3.9.



Rys. 3.9. Cykl życia skryptu [14]

Podczas projektowania rozgrywki i pisania skryptów znajomość tych funkcji i kolejności ich wykonywania jest niezbędna, ponieważ należy ona do specyfikacji technologii Unity 3D. W rozdziale 4 odnoszącym się do implementacji będą omawiane skrypty wykorzystujące funkcje:

- **Awake** - jest zawsze wywoływana przed funkcją Start,
- **OnEnable** - jest wywoływana w momencie gdy stan obiektu będzie ustawiony na dostępny,
- **Start** - jest wywoływana przed pierwszą aktualizacją klatki jeśli instancja skryptu jest włączona,
- **OnLevelWasLoaded** - jest wywoływana gdy nowy poziom będzie ładowany
- **Update** - jest wywoływana raz na klatkę,
- **Fixed Update** - jest wywoływana częściej niż Update, czyli kilka razy w ciągu klatki,
- **OnGUI** - jest wywoływana kilka razy w ciągu klatki w odpowiedzi na zdarzenia GUI,
- **yield WaitForSeconds, yield StartCoroutine** - odpowiadają za odliczanie czasu,
- **OnDestroy** - jest wywoływana po wszystkich aktualnieniach dla ostatniej klatki podczas trwania obiektu,
- **OnApplicationQuit** - jest wywoływana dla wszystkich obiektów gry przed zakończeniem gry.

Oprócz nich na uwagę zasługują **OnTriggerXXX** oraz **OnCollisionXXX**, które są wykonywane, gdy obiekt wejdzie w pole widzenia drugiego obiektu. Pierwsza odpowiada za wyzwalacz, pozwala na przenikanie obiektów, a druga zawiera mechanizm, który nie pozwala obiektem przenikać przez siebie. Rozróżnia się trzy typy funkcji, takich jak:

- **OnTriggerEnter()** - gdzie umieszczany jest kod wykonujący się raz, gdy inny obiekt wejdzie w pole określające przez wyzwalacz,
- **OnTriggerStay()** - w ciele tej funkcji kod wykonuje się ciągle, gdy inny obiekt przebywa w polu określonym przez wyzwalacz,
- **OnTriggerExit()** - zawiera kod wykonujący się raz, gdy inny obiekt opuści pole określające przez wyzwalacz.

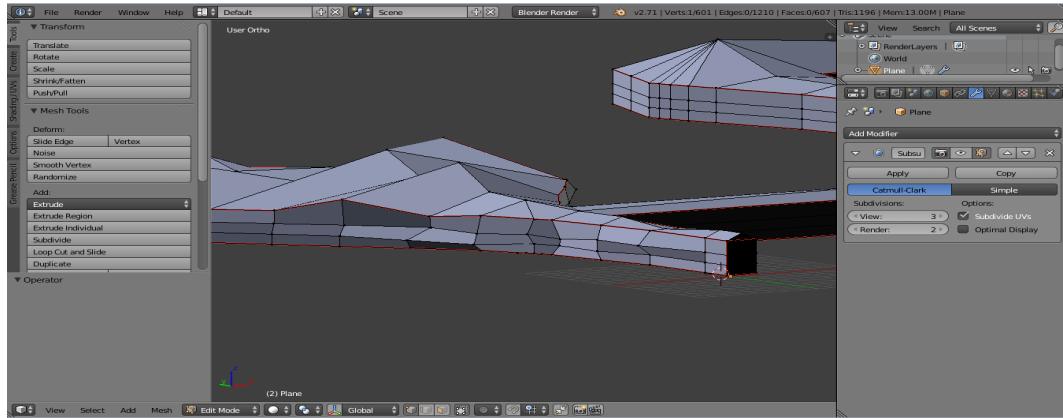
Analogiczne są funkcje odpowiadające za kolizję.

3.2 Blender

Blender jest programem udostępnionym na darmowej licencji służącym do tworzenia grafiki 3D – modelowania, teksturowania, renderowania, a także animacji.

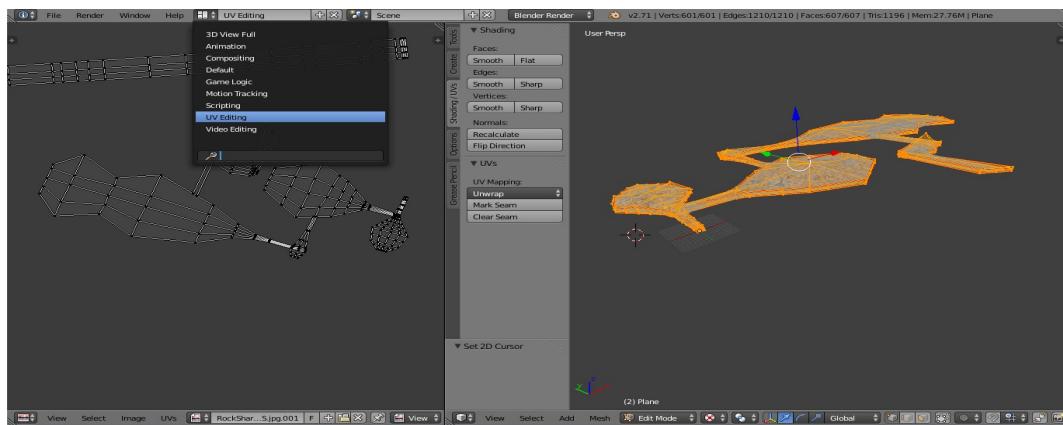
Aby zainstalować ten program należy pobrać plik instalacyjny, który jest dostępny na stronie umieszczonej w załączniku [15]. Proces instalacji przebiega standardowo.

Podstawowy interfejs został przedstawiony na rys 3.10. Wyróżniają się następujące elementy: okien, kontekstów, paneli i przełączników.



Rys. 3.10. Interfejs Blenera

Jak widać jest on rozbudowany i posiada wiele opcji. Dużą zaletą stanowi jego elastyczność i dostosowywanie widoku okien do aktualnych potrzeb. Na rys 3.11 został przedstawiony typowy widok edycji tekstur. Z lewej strony znajduje się obszar przeznaczony do wybrania tekstuury i naniesienia jej na wcześniej utworzoną siatkę obiektu. Z prawej strony znajduje się już wcześniej znany widok podczas którego modelowano obiekt i nanoszono na niego szwy. Więcej informacji o grafice zawiera podrozdział 4.8.



Rys. 3.11. Widok edycji UV

3.3 Programy grafiki rastrowej

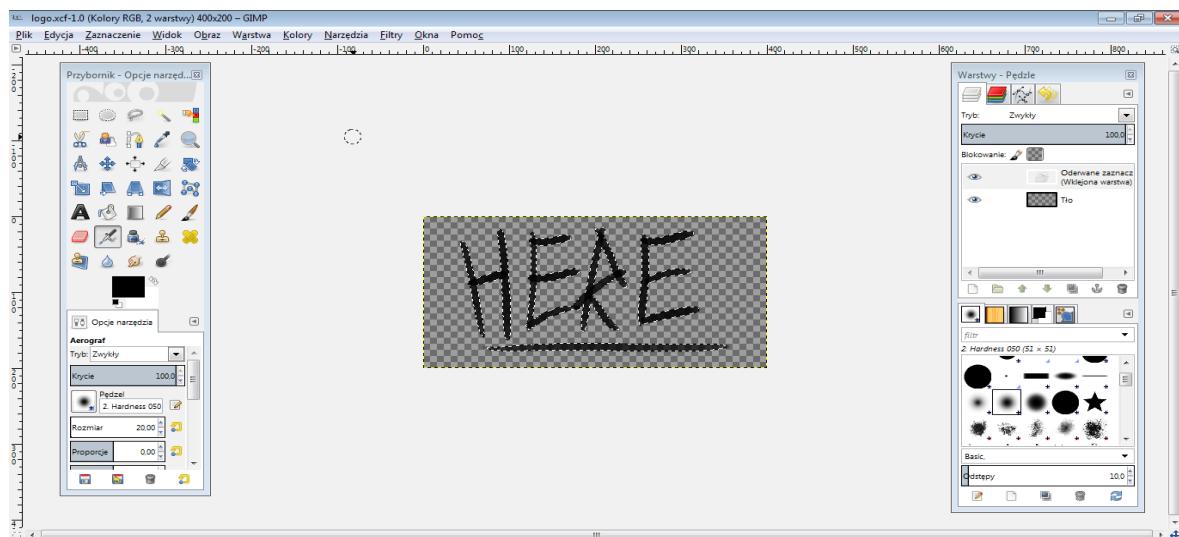
3.3.1 Gimp

Gimp jest darmowym programem służącym do tworzenia grafiki komputerowej bądź obróbki zdjęć. Pozwala na operowanie na warstwach, korzystanie z własnych pędzli, modyfikację kolorów. Wspiera także tablety graficzne.

W celu pobrania pliku instalacyjnego należy wejść na stronę umieszczoną w załączniku [16]. Proces instalacji przebiega standardowo.

Interfejs tego programu jest dość elastyczny, ponieważ można zmieniać położenia poszczególnych okien. Składa się z trzech głównych elementów, które są zaprezentowane na rys. 3.12:

- **Przybornik** – będący zwykle po lewej stronie. Zawiera w górnej części dostępne narzędzia: takie jak pędzle, wypełnianie kolorem, wprowadzanie tekstu, zaznaczanie, pobieranie koloru, rozmycie, a w dolnej opcje tego narzędzia m.in.: tryb, krycie, rozmiar, twardość, dynamika (potrzebna szczególnie przy używaniu tabletu graficznego w celu dobrania odpowiedniego nacisku piórka).
- **Menu wraz z oknem głównym** – na rys. 3.12 jest to zmaksymalizowane okno, gdzie znajduje się obszar roboczy. Na uwagę zasługują zakładki: **kolory** – modyfikacja balansu, barwy i nasycenia, jasności i kontrastu, krzywych, **filtры** – rozmycie, zniekształcenia, światło i cień,
- **Warstwy** – będące zwykle po prawej stronie. Zawierają oprócz utworzonych warstw historię wykonywania grafiki, a także okno kanałów, ścieżek.



3.12. Interfejs Gimpa

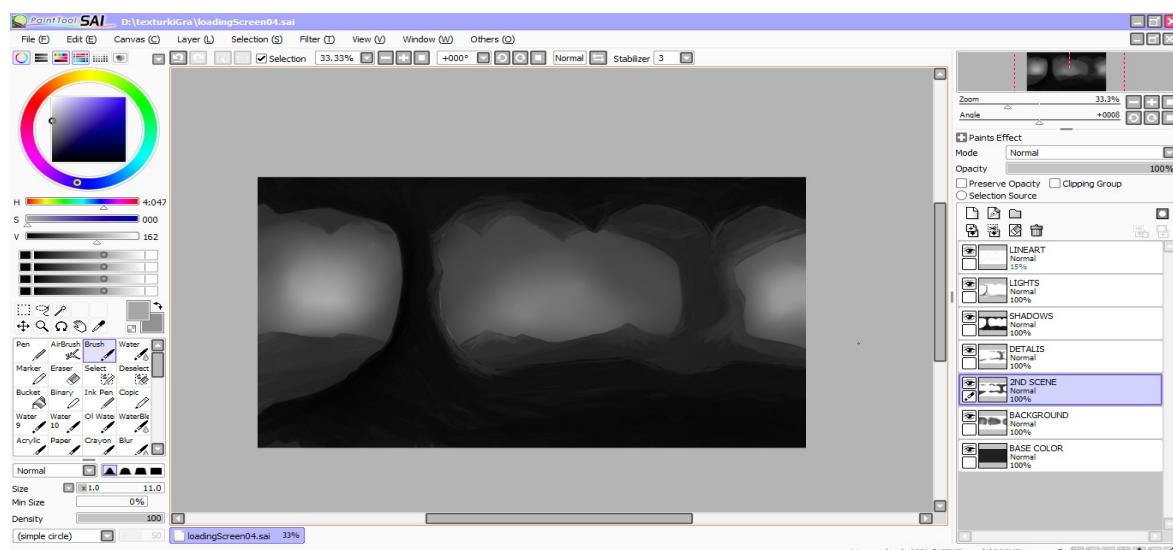
3.3.1 Paint Tool SAI

Paint Tool SAI jest narzędziem stosowanym głównie do rysowania przy pracy z tabletem graficznym.

Plik instalacyjny jest do pobrania ze strony umieszczonej w załączniku [17], a instalacja przebiega standardowo. Należy zaznaczyć, że jest to wersja testowa (tzw. trial) na 31 dni. Licencja na dzień dzisiejszy (styczeń 2016r) kosztuje 5400JPY (ponieważ jest to produkt japoński), czyli około 180 PLN.

Już na pierwszy rzut oka (rys. 3.13) widać uporządkowanie i prostotę jego interfejsu, co jest niewątpliwym plusem. Składa się on z czterech podstawowych części:

- **główne menu** – oprócz standardowych opcji umożliwia pod zakładkami:
 - **Canvas** - zmianę obszaru roboczego albo rozmiaru obrazu,
 - **Filter** - ustawianie parametrów barwy i nasycenia, jasności i kontrastu.
- **lewy panel** – metoda wybierania koloru (między innymi: koło barw, suwaki RGB/HSV, mikser kolorów, przyciski z zapisanymi wcześniej ulubionymi barwami) , poniżej opcje zaznaczania i nawigacji, pod nimi ikony narzędzi rysowania, a ostatnią część zajmują opcje wybranego narzędzia.
- **prawy panel** – zawiera okno nawigacyjne (przybliżenie, oddalenie, obrót) oraz warstwy wraz z ich trybami.
- **środkowy obszar** – jest przeznaczony na widok obszaru roboczego.



Rys. 3.13. Interfejs Paint Tool SAI

4. ZAŁOŻENIA, STRUKTURA I REALIZACJA GRY

4.1 Zalożenia projektowe

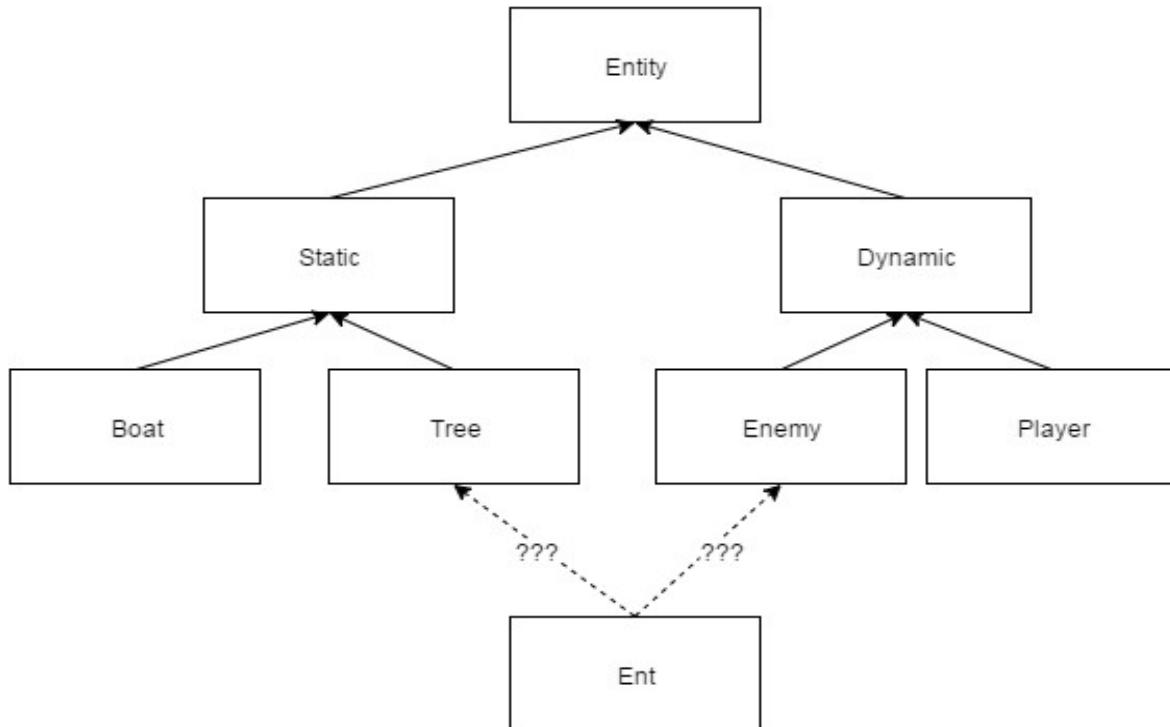
Z racji złożoności projektu oraz rozległości zagadnienia jakim są gry FPS założenia były następujące:

- zaprojektowanie gry komputerowej FPS wzorowanej w mechanice na klasyki, takie jak Fallout, czy Battlefield,
- gra przeznaczona dla jednego gracza,
- czytelny interfejs użytkownika,
- możliwość zbierania i używania przedmiotów z naciskiem na broń,
- zapis i odczyt gry,
- przeciwnicy wykorzystujący sztuczną inteligencję,
- system wskazówek ułatwiający eksplorację otoczenia,
- fabuła liniowa,
- gra składająca się z przynajmniej dwóch poziomów,
- klimatyczna rozgrywka oddziałująca na emocje gracza poprzez akcję, grafikę oraz dźwięk.

4.2 Architektura aplikacji

4.2.1 Model ECS

Tradycyjnym sposobem na implementację encji było kiedyś używanie OOP (*ang. object oriented programming*). Każda encja była obiektem, który intuicyjnie pozwalał na konkretyzację systemu bazującego na klasach i dostępnych encjach. Rozszerzanie funkcjonalności odbywało się poprzez polimorfizm. Doprowadziło to do tworzenia obszernej, sztywnej hierarchii klas. Wraz ze wzrostem ilości encji powstawały problemy z umieszczeniem nowej encji w hierarchii. Szczególnie trudnymi były przypadki, które wymagały od encji posiadania różnych typów funkcjonalności z różnych encji bazowych. Rys. 4.1 przedstawia ten problem [18], gdzie encja Ent powinna być statyczna, ale również posiadać funkcjonalności encji Tree oraz Enemy.

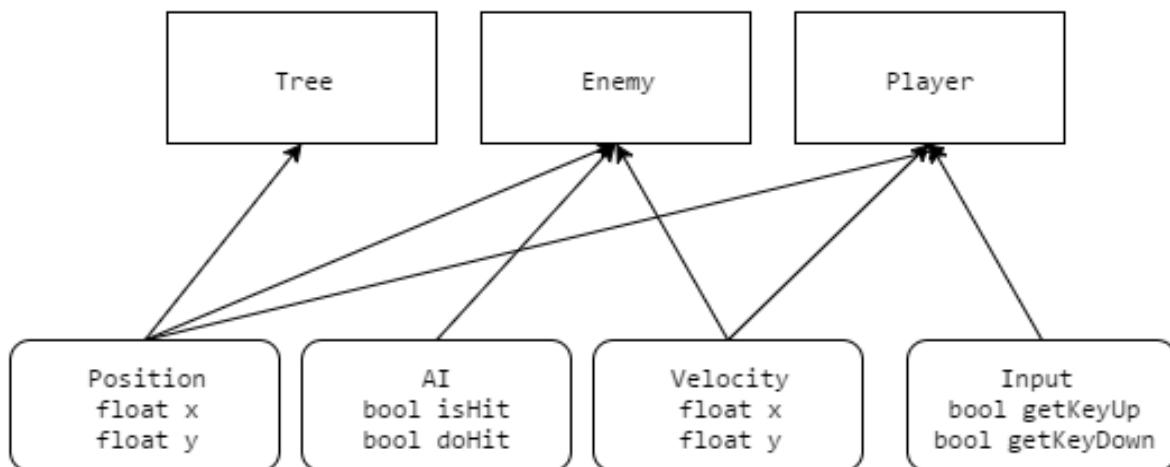


Rys. 4.1. Problem z umieszczeniem statycznej encji przeciwnika

Aby rozwiązać ten problem programiści przestali używać złożonego dziedziczenia, a zaczęli korzystać z kompozycji. Encja jest agregacją komponentów. To rozwiązanie posiada następujące zalety:

- prostota dodawania nowych, złożonych encji,
- prostota definiowania nowych encji w danych,
- większa wydajność.

Rys. 4.2 przedstawia implementację problemu zaprezentowanego na rys 4.1.



Rys. 4.2. Rozwiązanie problemu z wykorzystaniem komponentów

Komponent w najprostszym ujęciu jest to „mała część większej maszyny”. Każdy komponent posiada własne zadanie i optymalnie powinien wykonywać je bez pomocy źródeł zewnętrznych. Komponenty nie zajmują się szerokimi zagadnieniami związanymi z konkretnym systemem, więc są bardzo uniwersalne.

Podsumowując OOP z wykorzystaniem skomplikowanego dziedziczenia i sztywnej hierarchii nie jest tak efektywne, jak użycie komponentów. Ponadto technologia Unity 3D bazuje na CBSE (*ang. component-based software engineering – inżynieria oprogramowania opierająca się na komponentach*).

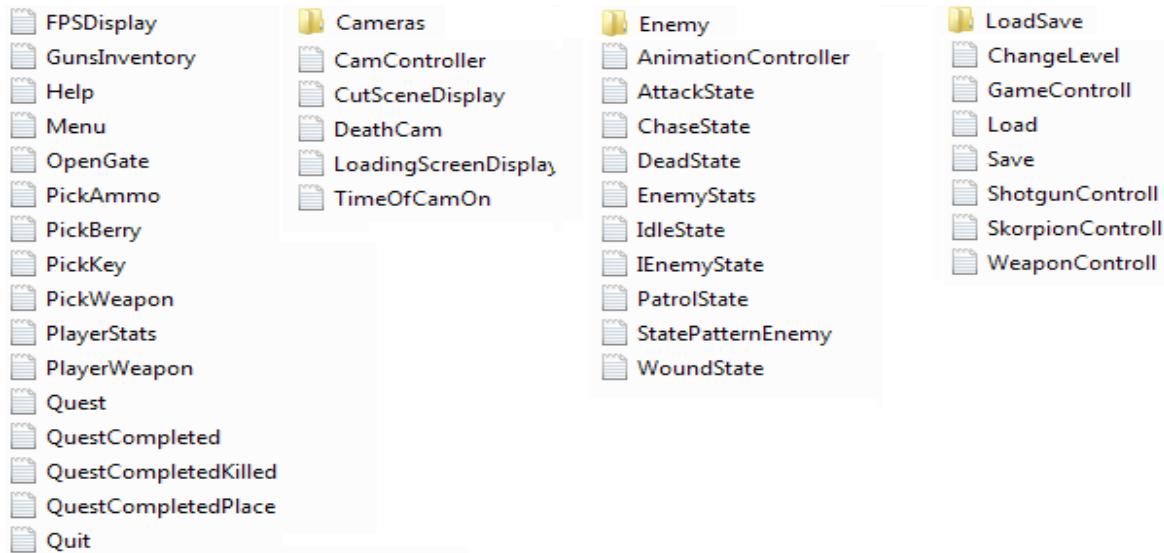
Faworyzacja komponentów w technologii Unity 3D opiera się na konstrukcji jej wizualnego edytora. Podczas pracy pokazuje wszystko na czym się operuje w czasie rzeczywistym. Oznacza to, że można testować projekt w oknie Game (omawianym w punkcie 3.2.1), obserwować jak aplikacja działa oraz edytować kod poprzez okno Inspektora, żeby zaobserwować zmiany.

4.2.2 Pakiety

Wszystkie zasoby jak już wcześniej wspomniano przechowane są w folderze **Assets**. Przy realizacji tego projektu zostały utworzone następujące foldery zawierające:

- **Animations** - animacje kamer wykorzystywane przy przerywnikach filmowych,
- **Editor** - skrypty w nim zawarte mają dostęp do Unity Editor Scripting API, nie są dołączane do tworzonej gry, ale do edytora Unity,
- **FONTs** - czcionki. Zawiera jedną pobraną czcionkę Bahiana-Regular [19],
- **Materials** - materiały, które definiują jak obiekt będzie wyświetlany,
- **Misc** - Astart Pathfinding Projekt [20] oraz plik LightmapSnapshot.asset odpowiadający za oświetlenie w drugim poziomie gry ,
- **Models** - modele wykorzystane do gry:
 - **Bushes** - krzak wykonany w programie Blender,
 - **DownloadedModels** - darmowe, pobrane modele: łódki [21], drzwi [22], broni [23, 24], przeciwnika [25],
 - modele wykonane w Blenderze - jagoda, jaskinia, klucze, wejście do jaskini wraz z kamieniami.
- **Plugins** - natywną wtyczkę do Astar Pathfinding Project, do której mają mieć dostęp skrypty,

- **Prefabs** - prefabrykaty: pudełko z amunicją, krzak, przeciwnik, gracz, ekran ładowania, inne kamery, teksty, o których będzie mowa w kolejnych rozdziałach,
- **Scripts** - skrypty przedstawione na rys. 4.3,



Rys. 4.3. Skrypty

- **Sounds** - dźwięki przeładowań i strzałów [26] oraz ścieżki dźwiękowej będącej w tle gry [27],
- **Standart Assets** - kontroler pierwszoosobowy, kreator umożliwiający utworzenie środowiska, system cząsteczkowy,
- **Textures** – tekstury, czyli obrazki w formacie najczęściej .png, które nanoszone są na modele.

Oprócz wymienionych folderów zawiera również utworzone sceny:

- **Menu** - menu główne gry. Dostępne są tu opcje: *Nowa gra*, *Wczytaj grę*, *Wyjście*. Widoczne także jest logo gry, a tło stanowi widok nocnego nieba.
- **Lvl01** - pierwszy poziom gry. Został zaprojektowany jako wyspa, którą ogranicza z trzech stron morze, a z czwartej wysokie góry. Nanieśione są na nią obiekty, które odpowiadają za ten poziom gry i były już wcześniej wymieniane (m. in. obiekty przeciwników, broni do podniesienia, drzew i krzaków z dostępymi jagodami do zebrania). Aby gracz opuścił ten poziom musi znaleźć klucz i przejść przez wrota, które wprowadzają do jaskini.
- **Lvl02** - drugi poziom gry. Stanowi go jaskinia z licznymi korytarzami i wąskimi przejściami, które przypominają labirynt.

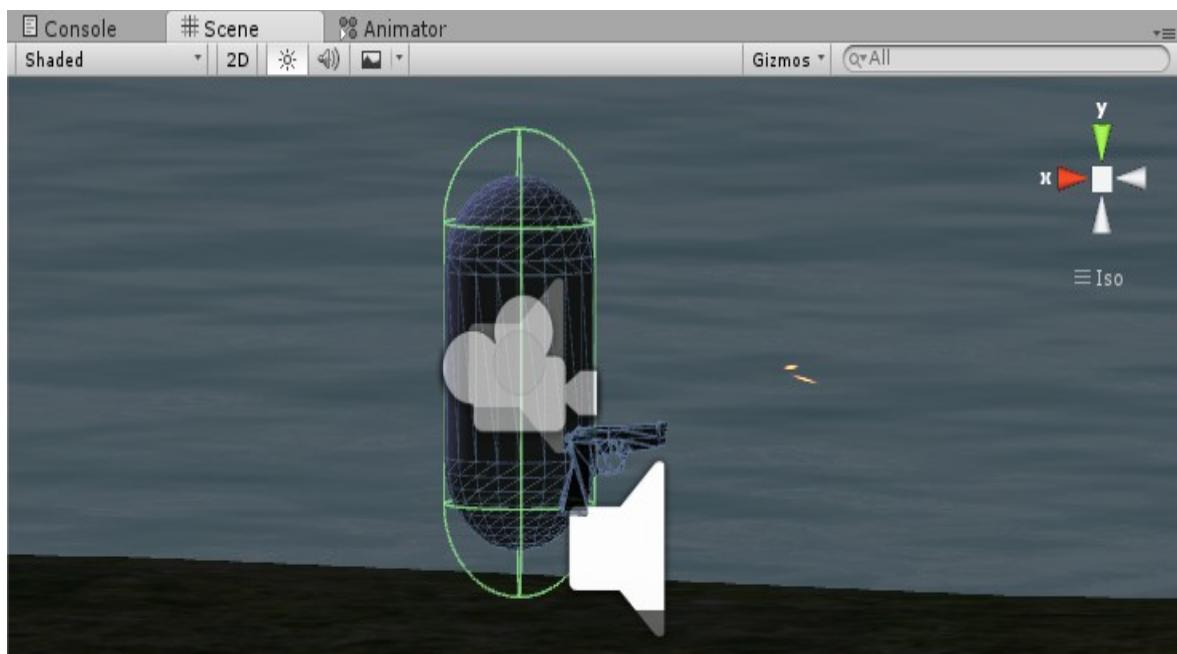
4.3 Obiekt gracza

4.3.1 Omówienie obiektu gracza

Dzięki niemu użytkownik porusza się po mapie i ma do dyspozycji wszystkie funkcjonalności aplikacji.

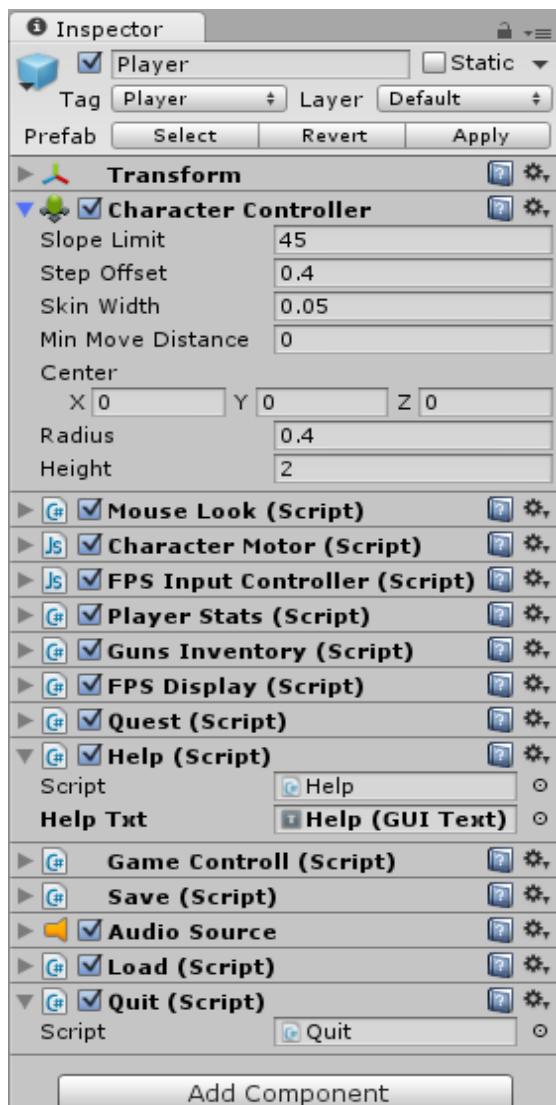
Rys. 4.4 pokazuje jak ten obiekt prezentuje się na scenie. Wizualnie składa się z: cylindra, który odpowiada wymiarom gracza, aby ułatwić projektowanie rozgrywki oraz aktualnie wybranej broni. Widnieją także ikony powiadamiające, że obiekt ten wykorzystuje kamerę oraz źródło dźwięku.

Zielona klatka, która otacza cylinder pochodzi z komponentu *Character Controller*. Odpowiada on za kąt oraz wysokość stopnia, po jakim może wejść gracz, a także za wielkość collaidera (element określający przestrzeń w której dojdzie do kolizji, gdy inny obiekt zajdzie się w tym obszarze). Za poruszanie się względem osi X, Y, Z oraz podstawową fizykę przy skoku oraz biegu odpowiadają komponenty *Mouse Look* oraz *Character Motor*. Te trzy komponenty dostarcza Unity 3D w pakiecie *Standart Assests*.



Rys. 4.4. Obiekt gracza w oknie Sceny

Każdy obiekt gry posiada komponent *Transform*, którego nie da się usunąć. Odpowiada on za pozycję, obrót oraz skalę obiektu względem osi X, Y, Z. Podczas poruszania się wartości *Position* oraz *Rotation* ulegają zmianie, co można kontrolować w oknie Inspektora (rys. 4.5). Na tym rysunku jest również przedstawione ustawienie tagu (czyli etykiety) obiektu na *Player*.



Rys. 4.5. Okno Inspektora obiektu Gracza

Autorskimi skryptami dotyczącymi tego obiektu są: Player Stats, Guns Inventory, FPS Display, Quest, Help, Quit, Game Controll, Save, Load.

4.3.2 Skrypt statystyk gracza

Ten skrypt jest odpowiedzialny za przechowywanie, zarządzanie oraz wyświetlanie stanu statystyk poziomów: zdrowia, pancerza i energii. Ze względu na ścisłe oddziaływanie na poziom pancerza lub zdrowia pełni dodatkowo dwie funkcje. Pierwsza to przechowywanie danych dotyczących ilości zebranych jagód, które przywracają ten pasek, a druga to odbieranie informacji o otrzymanych obrażeniach od przeciwnika i odejmowanie tej wartości najpierw z paska pancerza, a potem z paska życia.

Poziom dostępnej energii maleje po naciśnięciu klawisza *Shift*, a wzrasta, gdy gracz pozostaje w bezruchu przez określoną liczbę sekund.

Paski zdrowia, pancerza oraz energii wyświetcone są w prawym górnym rogu, natomiast liczba zebranych jagód i liczba przyrostu zdrowia po użyciu znajduje się w lewym dolnym rogu, co widać na rys. 4.6.



Rys. 4.6. Interfejs użytkownika

Listing 4.1: Zmienne publiczne widoczne w oknie inspektora:

```
public Texture2D texHealth, texArmour, TexEnergy;
public float walkSpeed = 10.0f, runSpeed = 30.0f;
public GUITexture hitTexture;
public GUIText potionTxt;
public int sectToSpawn = 5;
public int deadTime=0;
public int maxPot=5;
public int nowPots = 4;
public bool haveKey;
public GUIText HpTxt;
public GUIText ArmourTxt;
public GUIText EnergyTxt;
public float nowHealth = 100.0f, nowArmour = 100.0f, nowEnergy =
100.0f;
public float nowHealth = 100.0f, nowArmour = 100.0f, nowEnergy =
100.0f;
```

Zmienne te mają odzwierciedlenie w oknie Inspektora (rys. 4.7). Aby program zadziałał należy przygotować tekstury oraz obiekty GUI Text oraz przypisać je do odpowiednich zmiennych.



Rys. 4.7. Zmienne publiczne skryptu Player

Stats widoczne w oknie Inspektora

Listing 4.2 Funkcja Awake():

```
void Awake() {
    barH = Screen.height * 0.03f;
    barW = barH * 10.0f;
    fontSize= (int) (Screen.height * 0.035f);
    charMotor = GetComponent<CharacterMotor>();
    startPosition = transform.position;
    spawnPosition = transform.position;
    spawnRotation = transform.rotation;
}
```

Przed pokazaniem pierwszej klatki programu funkcja ta ustawia:

- zmienne barH i barW odpowiadające za wysokość i szerokość pasków życia, pancerza oraz energii. Przypisanie do tych zmiennych konkretnych wartości byłoby nierozsądne, ponieważ przy różnych rozdzielcościach ekranu proporcje byłyby zachwiane. W ten sposób pasek, który miałby 10px wysokości na ekranie o wysokości 100px stanowiłby 10%, a na ekranie o wysokości 1 000px stanowiłby tylko 1%. Dzięki temu rozwiązaniu będzie to zawsze 3%,
- rozmiar czcionki w stosunku do wysokości ekranu,

- pobiera komponent *CharacterMotor*,
- przypisuje do zmiennych spawnXXX aktualne transformacje gracza. Będzie to wykorzystywane przy śmierci, aby przywrócić gracza do pozycji startowej.

Listing 4.3 Funkcja Update():

```
void Update()
{
    if (usingPot) {
        float sum =startedArmour+potion;
        if(nowArmour<suma) {
            nowArmour += potion * 0.01f ;
            nowArmour=Mathf.Clamp(nowArmour, 0, suma);
        }
        if (nowArmour>=suma) {
            usingPot=false;
        }
    }

    if(nowPots > 0.0f && Input.GetKeyDown(KeyCode.P) &&
nowArmour < maxArmour && waitingTime<=0) {
        usingPot=true;
        nowPots -= 1;
        startedArmour=nowArmour;
        waitingTime=10f;
    }
    timer(ref waitingTime);

    if(restPossible > 0.0f) {
        restPossible -= Time.deltaTime;
    }

    if(restPossible <= 0.0f && nowEnergy < maxEnergy) {
        nowEnergy += maxEnergy * 0.003f ;
        Mathf.Clamp(nowEnergy, 0, maxEnergy);
    }
}
```

Funkcja ta sprawdza, czy użytkownik posiada przynajmniej jeden przedmiot leczący, nacisnął przycisk P, jego stan pancerza jest mniejszy od maksymalnego poziomu oraz, czy już nie użył przed chwilą danego przedmiotu. W przypadku spełnienia wszystkich warunków ustawia zmienną usingPot na true, odejmuje jeden od bieżącej liczby przedmiotów leczących (zmienna nowPots), ustawia wartość zmiennej startedArmour na nowArmour, a waitingTime na 10. Timer posiada referencję do zmiennej waitingTime. Domyślnie wynosi ona 0, aby użycie przedmiotu było możliwe. Funkcja timera odlicza czas (w tym przypadku 10 sekund) do możliwości ponownego

użycia przedmiotu. Proces zwiększania zmiennej nowArmour następuje w każdej klatce o małą wartość, przez co gracz widzi jak sukcesywnie jego pasek pancerza się zwiększa. Zmienna restPossible odpowiada za możliwość regeneracji poziomu energii, gdy gracz przestaje się ruszać na 3 sekundy.

Listing 4.4: Funkcja drawInterface() umieszczona w funkcji OnGUI():

```
void drawInterface() {
    GUI.DrawTexture(new Rect( Screen.width - barW - (float)(0.5*barH) ,
                                barH + (float)(0.5*barH) ,
                                nowHealth * barW / maxHealth ,
                                barH ) ,
                    texHealth);
    (...)

    HpTxt.fontSize = (int)( barH + 0.1 * barH );
    HpTxt.text = "hp";
    HpTxt.pixelOffset = new Vector2(Screen.width - barW - (float)(1.9
        *barH) , Screen.height - (float)(2*barH));
    (...)

    potionTxt.fontSize = fontSize;
    potionTxt.text = nowPots + "(" + potion + "hp) / " + maxPot;

}
```

W pierwszej kolejności funkcja rysuje teksturę w kształcie prostokąta o wymiarach odpowiednich dla aktualnego stanu zdrowia oraz rozdzielczości ekranu. Analogicznie postępuje dla pancerza oraz energii. Potem dobiera wielkość czcionki, ustawia tekst, a także przesunięcie względem ekranu i kolejnego paska. Ostatnia linijka kodu ustawia treść, jaka będzie wyświetlona w miejscu informującym o ilości przedmiotów przywracających życie.

Ważna jest także funkcja `takeHit()`, która ustawia czerwony ekran przy trafieniu przez przeciwnika oraz odejmuje życie gracza.

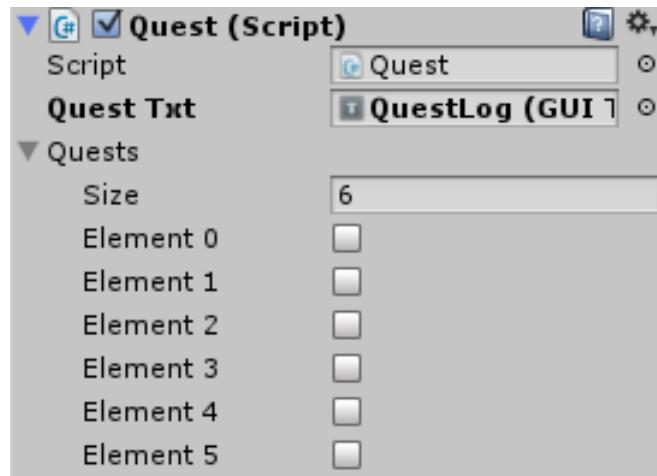
4.3.3 System zadań

Założenie, że gra powinna posiadać fabułę liniową spowodowało zaprojektowanie systemu zadań. Gracz pod przyciskiem *Tab* ma do dyspozycji podgląd misji, które może wykonywać i które pomagają czerpać rozrywkę z użytkowania aplikacji (rys. 4.8).



Rys. 4.8. Podgląd zadań

Komponent *Quest* w obiekcie *Player* zawiera tablicę wszystkich dostępnych zadań wraz z ich statusem oraz aktualizuje go po ukończeniu misji. Rys. 4.9 przedstawia jego zmienne publiczne.



Rys. 4.9 Zmienne publiczne skryptu *Quest*

widoczne w oknie Inspektora

Listing 4.5: Fragmenty skryptu *Quest*:

```
public GUIText QuestTxt;
public bool[] quests = new bool[] {false, false, false, false,
false, false};
(...)
string status(bool ifCompleted) {
```

```

        if (ifCompleted) {
            return "skończony!";
        } else {
            return "w trakcie...";
        }
    }

public void ifDone(int numer){
    for (int i=1; i<quests.Length; i++) {
        if (i == numer) {
            quests [i] = true;
        }
    }
}

void OnGUI(){
    QuestTxt.text="Znajdź miejsce rozbicia: " +
status(quests[1]) +
"\n" + "Wykradnij shotguna: " +status(quests[2])+
"\n" + "Zbierz klucze: " +status(quests[3]) +
"\n" + "Dostań się do jaskini: " + status (quests[4])+
"\n" + "Pokonaj przywódcę: " + status (quests[5]);
}

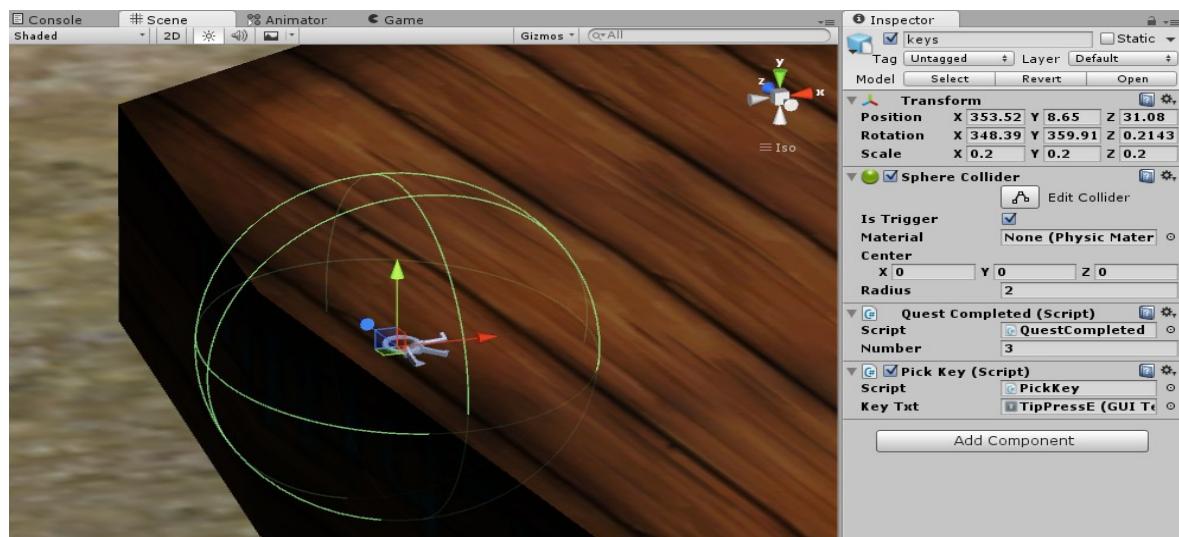
void Update () {
    Timer += Time.deltaTime;
    if (Input.GetButton("QuestLog") && Timer > 1){
        if (!QuestTxt.enabled){
            QuestTxt.enabled = true;
            Timer =0;
        }else{
            QuestTxt.enabled =false;
            Timer =0;
        }
    }
}

```

Cały system jest bardziej skomplikowany, ponieważ ten skrypt komunikuje się z trzema innymi skryptami, które powiadamiają go o wypełnieniu zdania, czyli wywołują funkcję `ifDone (int numer)`. Ich liczba wynosi właśnie trzy ze względu na trzy możliwości zakończenia zadania poprzez:

- podniesienie przedmiotu,
- dotarcie na miejsce,
- zabicie przeciwnika.

Pierwszy przypadek, czyli podniesienie przedmiotu dotyczy komunikacji skryptu *Quest* będącego komponentem obiektu *Player* i skryptu *QuestCompleted* będącego komponentem podniesionego obiektu. rys. 4.10 przedstawia klucze, do których jest dołączony skrypt *QuestCompleted*, a rys. 4.11 widok podczas gry, gdy gracz wejdzie w wyzwalacz. Komunikacja odbywa się poprzez funkcję `OnTriggerStay(Collider other)`. Ta funkcja odpowiada za sprawdzenie, czy tag obiektu to *Player*, a potem czy został naciśnięty klawisz *E*. Jeśli tak, to wysyła się za pomocą funkcji `other.SendMessage ("ifDone", number)` numer ukończonego zadania. Przy tym sposobie komunikacji oba skrypty nie muszą wiedzieć o swoim istnieniu i ich byty są od siebie niezależne. Możliwe jest to dzięki funkcji `OnTrigger`, która sama odnajduje poprzez `Collider` adresata, który wszedł w pole widzenia obiektu.

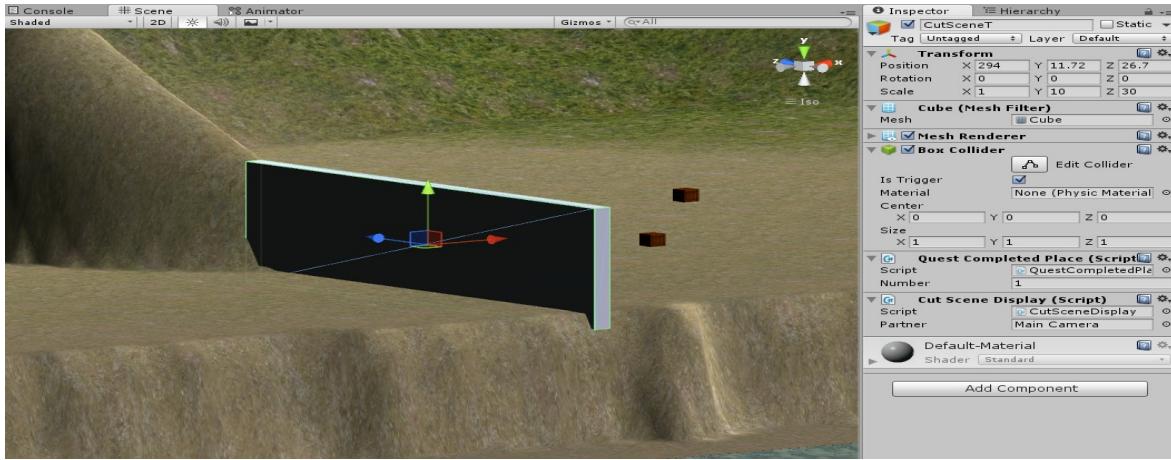


Rys. 4.10: Widok wyzwalacza obiektu Keys oraz jego okna Inspektora

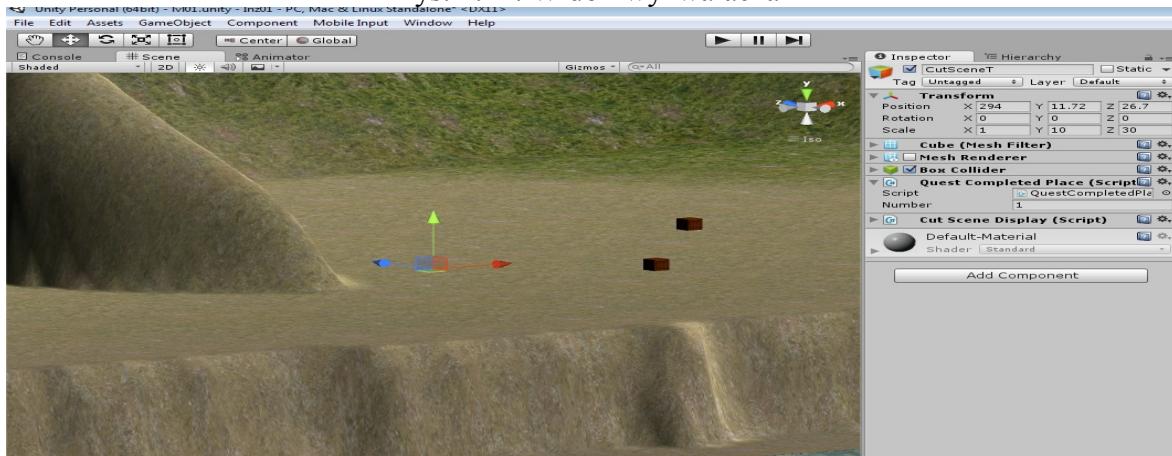


Rys. 4.11. Zrzut ekranu z gry podczas wypełniania zadania

Drugi przypadek pod względem kodu jest rozwiązyany analogicznie do pierwszego, ale dotyczy sytuacji jedynie wejścia przez gracza w wyzwalacz. Renderowanie obiektu jest wyłączone poprzez odznaczenie komponentu *Mesh Rendering*. Mesh oznacza kształt 3D, który zawiera wierzchołki, krawędzie oraz ściany (rys. 4.12 oraz rys. 4.13).



Rys. 4.12. Widok wyzwalacza



Rys. 4.13. Widok bez renderowania wyzwalacza

Trzeci natomiast nie bazuje na systemie wyzwalaczy, ponieważ dotyczy spełnienia warunku – życie przeciwnika musi być mniejsze bądź równe 0.

Listing 4.6: Skrypt *QuestCompletedKilled*:

```
public class QuestCompletedKilled : MonoBehaviour {
    public GameObject partner;
    public int number;
    void killedCompleted () {
        Quest quest = partner.GetComponent<Quest>();
        quest.ifDone(number);
```

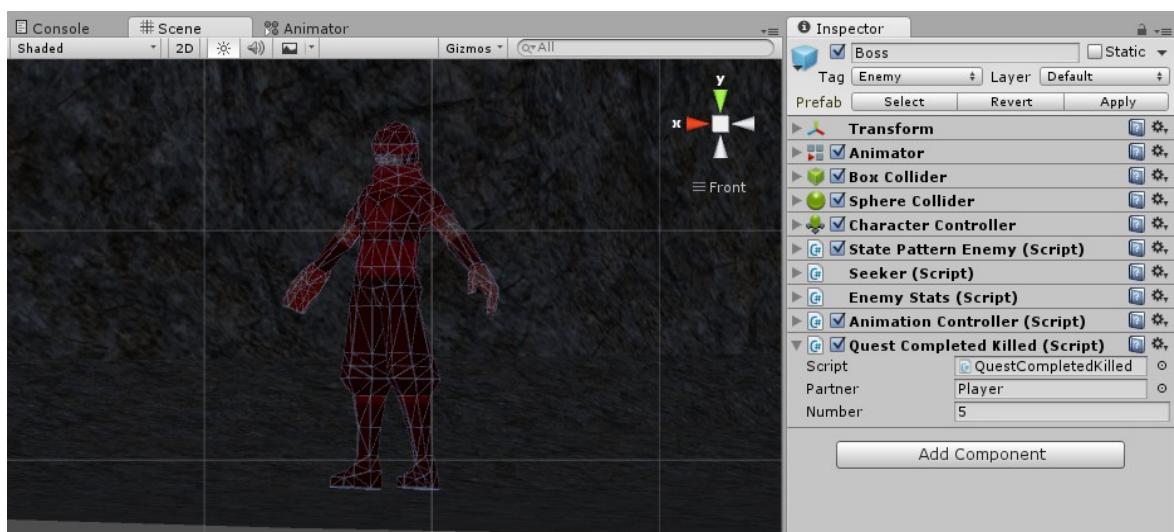
```

        }
    void Update() {
        EnemyStats enemy = GameObject.GetComponent<EnemyStats>();
        if (enemy.hp <= 0) {
            killedCompleted ();
        }
    }
}

```

Publiczna zmienna `partner` typu `GameObject` oznacza, że obiekt, do którego zostanie dołączony ten skrypt będzie musiał ustawić za parametr inny obiekt gry. W tym wypadku będzie to obiekt *Player* (rys. 4.14).

Zapis `Quest quest = partner.GetComponent<Quest>()` umożliwia komunikację skryptów poprzez pobranie komponentu *Quest* i wywołaniu funkcji `quest.ifDone(number)`. Odpowiadający temu widok z gry prezentuje rys. 4.15.



4.14. Widok obiektu Bossa oraz jego okna Inspektora



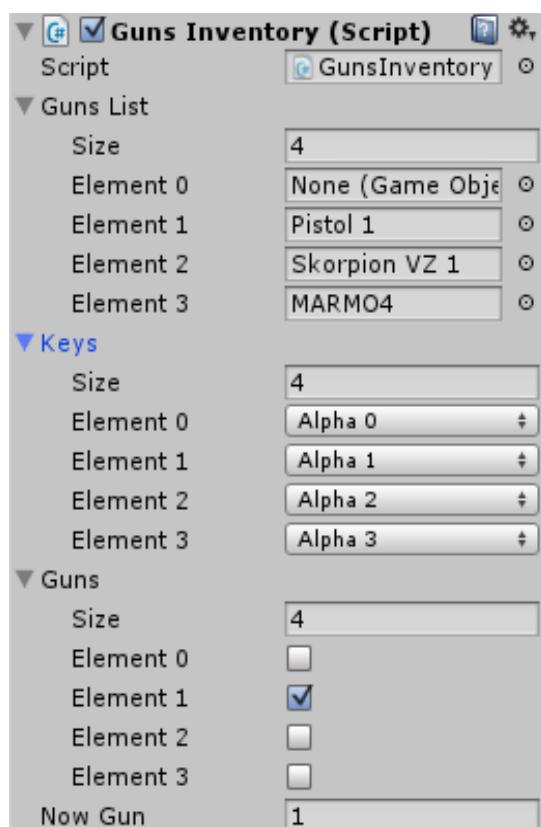
4.15. Zrzut ekranu z gry po wypełnieniu wszystkich zadań

4.3.4 Pozostałe skrypty

W tym punkcie zostaną jeszcze przedstawione skrypty *Guns Inventory* oraz *Help*. Opis działania skryptów: *Game Control*, *Save*, *Load* zawiera rozdział 4.7.

Skrypt *Guns Inventory*:

Jego zadaniem jest przechowywanie listy wszystkich dostępnych broni w grze, przypisanie im klawiszy, przechowywanie danych dotyczących które z broni są dostępne dla gracza oraz która broń jest aktualnie używana. rys. 4.10 przedstawia zmienne publiczne tego skryptu.



Rys. 4.10: Zmienne publiczne skryptu *Guns Inventory* widoczne w oknie Inspektora

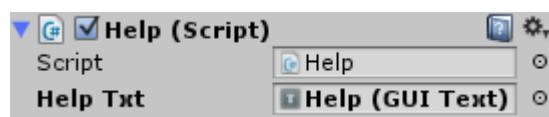
W funkcji `Update()` wywołuje się funkcja `changeGun()`. Zmiana broni polega na naciśnięciu przez gracza klawisza od 0 do 9 oraz spełnieniu warunku posiadania danej broni (wartość w tablicy `guns[]` o indeksie z wybranego klawisza klawiatury ustawiona na `true`). Wykonuje wówczas kod odpowiadający za chowanie aktualnej broni (funkcja `hideGuns()`), ustawienie na `true` indeksu w tablicy `guns[]` oraz przypisanie do zmiennej `nowGun` tego numeru. Posiada też funkcje odpowiedzialne za dodawania nowej broni i dodawanie amunicji.

Skrypt Help:

Jego zadaniem jest ustawienie treści pomocy i wyświetlanie jej podczas rozgrywki (rys. 4.11) po naciśnięciu klawisza *F1*. Jako zmienną publiczną przyjmuje obiekt typu GUI Text (rys. 4.12), na której jest wyświetlana treść pomocy.



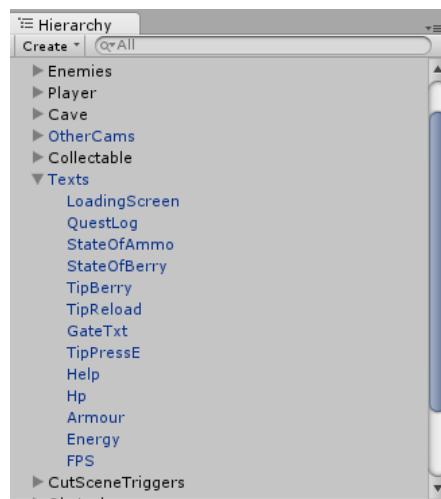
Rys. 4.11. Zrzut ekranu z gry z włączoną pomocą



Rys. 4.12. Zmienna publiczna skryptu *Help* widoczna w oknie Inspektora

Obiekty GUI Text:

Wszystkie obiekty typu GUI Text zostały zgrupowane w jednym pustym obiekcie (*ang. Empty GameObject*) o nazwie Texts (rys. 4.13). To rozwiązanie zapewnia łatwy dostęp do poszczególnych pól tekstowych oraz porządek w strukturze aplikacji.



Rys. 4.14. obiekty GUI Text widoczne w oknie Hierarchii

4.4 System broni

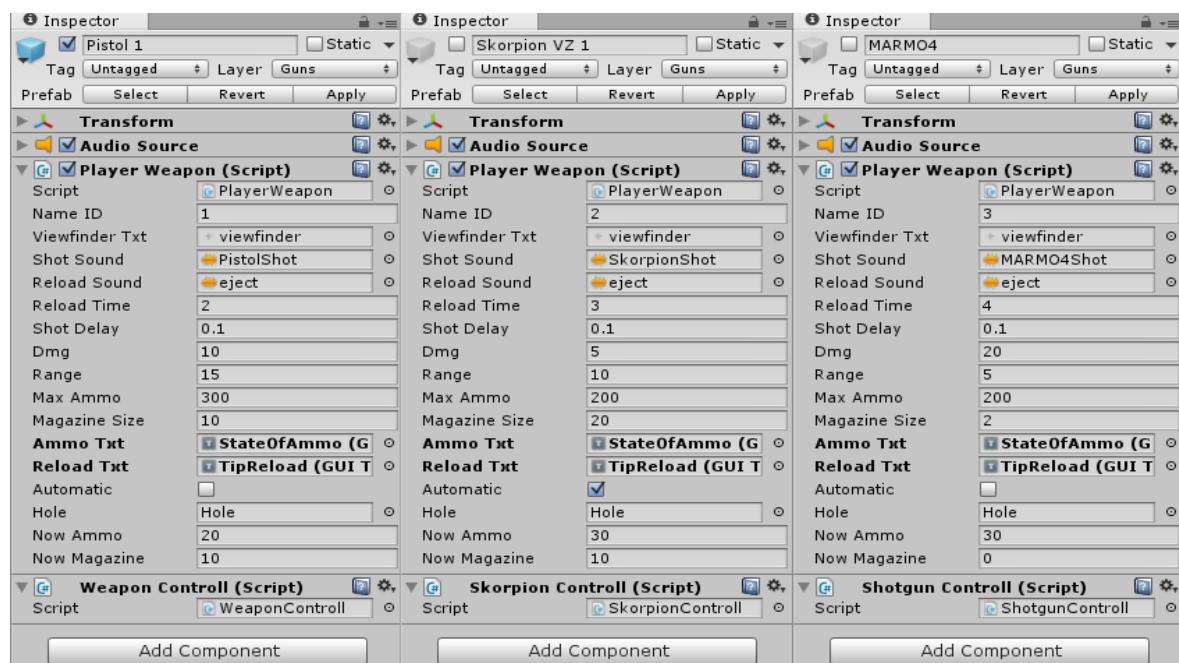
4.4.1 Używanie broni

Komponent Player Weapon:

Jest on odpowiedzialny za zarządzanie możliwościami broni. Każda broń posiada: własne ID, celownik, dźwięki wystrzału oraz przeładowania, czas przeładowania, czas pomiędzy kolejnymi wystrzałami, liczbę zadanych obrażeń przez jeden pocisk, zasięg, maksymalną liczbę amunicji, rozmiar magazynka (liczba dostępnych pocisków po przeładowaniu), obiekty GUI Text odpowiedzialne za wyświetlanie stanu broni oraz podpowiedzi w przypadku konieczności przeładowania, informację, czy jest to broń automatyczna, obiekt typu prefab przedstawiający ślad po pocisku, obecny stan dostępnych magazynków oraz pocisków znajdujących się w aktualnie wybranym. W grze dostępne są trzy rodzaje broni różniące się nie tylko wyglądem (rys. 4.15), a także parametrami przedstawionymi na rys. 4.16. Są to pistolet, pistolet maszynowy oraz shotgun.



Rys. 4.15. Różne rodzaje dostępnych broni



Rys. 4.15 Porównanie parametrów broni

Skrypt zawiera funkcje odpowiadające za przeładowanie broni (widok komunikatu rys. 4.16), zostawianie śladów po kulach (rys. 4.17), dodawanie amunicji, które można zebrać z paczek, a także najważniejszą – strzelanie. Odbywa się ono wtedy, gdy gracz posiada więcej niż 0 pocisków oraz nie jest w trakcie przeładowywania broni, a także nacisnął lewy przycisk myszy i zmienna `shot_dealay` jest mniejsza bądź równa 0.

Za pomocą raycastingu (metoda polegająca na „wysłaniu wiązki” z konkretnego punktu na daną odległość) sprawdza się, czy doszło do kolizji z obiektem o tagu *Enemy* oraz czy ta odległość jest mniejsza bądź równa zasięgowi broni. Rys. 4.18 pokazuje trafienie przeciwnika wraz z elementem cząsteczkowym towarzyszącym podczas wystrzału.



Rys. 4.16. Komunikat o przeładowaniu



Rys. 4.17. Ślady po kulach na pniu drzewa



Rys. 4.18. Trafienie przeciwnika

4.4.2 Zbieranie przedmiotów

Skrypt **Pick Weapon** jest dołączany do każdego obiektu broni, który można podnieść z ziemi. Gdy obiekt o tagu *Player* wejdzie w jego wyzwalacz wyświetli się komunikat o możliwości podniesienia broni (rys. 4.19). Po naciśnięciu klawisza *E* do obiektu o tagu *Player* zostanie wysłana wiadomość, aby wywołać funkcję `addGun (int gunID)` z przekazanym parametrem odpowiadającym ID podniesionej broni oraz obiekt broni zostanie usunięty ze sceny.



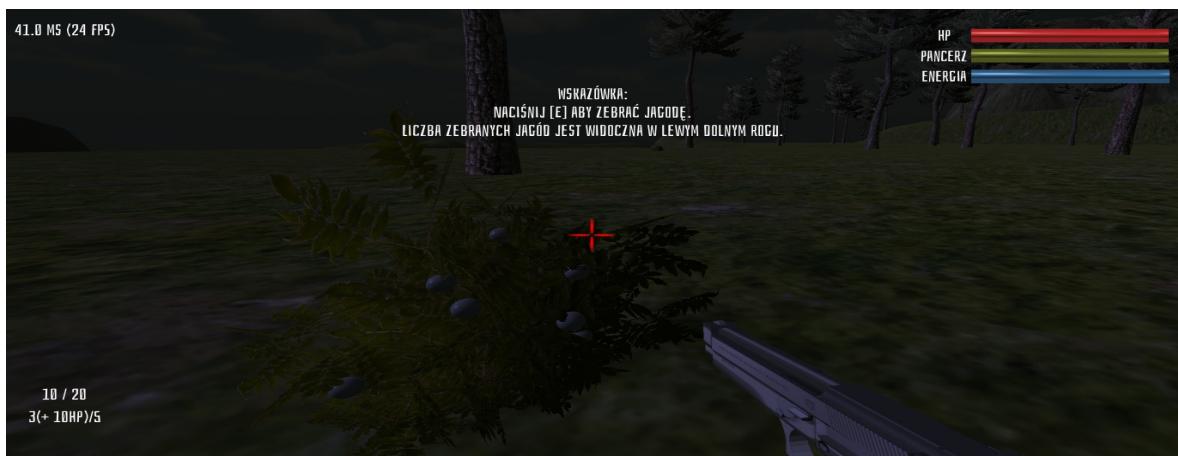
Rys. 4.19. Podniesienie broni

Wszystkie obiekty, które można zebrać, są zgrupowane jako dzieci pustego obiektu o nazwie **Collectable** (rys. 4.20). Dzięki temu w oknie Hierarchii łatwiej jest zarządzać nimi oraz dodawać kolejne obiekty z danych grup.

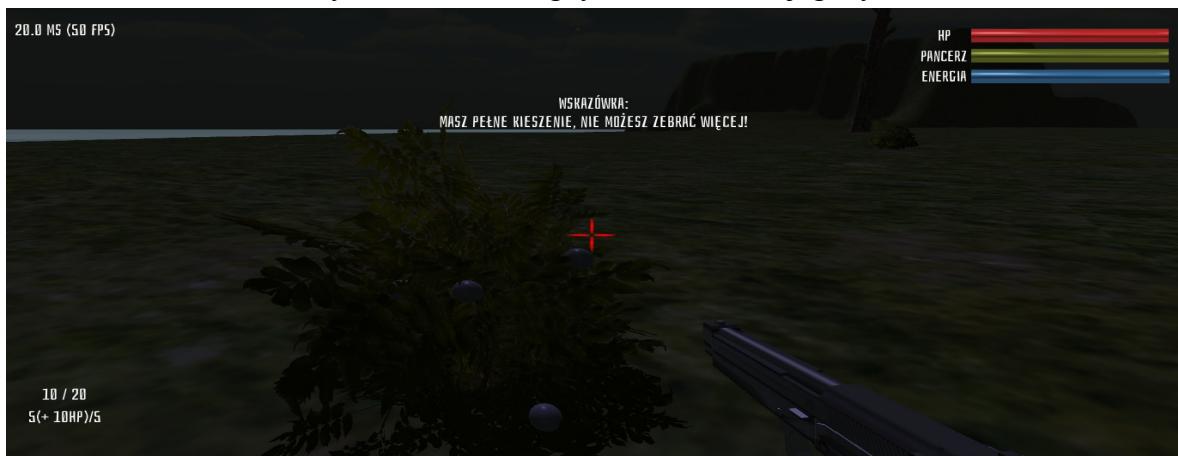


Rys. 4.20. Widok okna Hierarchii przedstawiający obiekty, które można zbierać

Na podobnej zasadzie, co skrypt **Pick Weapon**, działają inne skrypty pozwalające zbierać przedmioty. Dla jagód jest to **Pick Berry** sprawdzający, czy maksymalna ilość jagód nie została już osiągnięta. Jeśli nie, wówczas pokazuje komunikat ze wskazówką dotyczącą zebrania przedmiotu (rys. 4.21), a po naciśnięciu klawisza *E* m.in. wywołuje funkcję `addPot()`, która w obiekcie gracza zwiększa ilość posiadanych jagód i usuwa obiekt jagód. Jeśli limit został już osiągnięty, pokazuje komunikat odnośnie posiadania już maksymalnej ilości danego przedmiotu (rys. 4.22).



Rys. 4.21. Widok, gdy można zebrać jagody



Rys. 4.22. Widok, gdy nie można zebrać więcej jagód

Skrypt **PickAmmo** odpowiada za dodawanie amunicji. Zebranie takiego pudełka odbywa się, podobnie jak we wcześniej omawianych przypadkach, poprzez wejście w jego wyzwalacz. Skrypt posiada dwie zmienne publiczne odpowiadające za ilość pocisków w paczce oraz za rodzaj broni. Są one wysyłane jako parametr `Vector2 (ammo, gunID)` do funkcji `addAmmo (Vector2 vec2)` będącej w skrypcie *Guns Inventory* obiekcie gracza. Liczba zebranej amunicji jest wyświetlana w lewym dolnym rogu dla aktualnie wybranej broni.



Rys. 4.23. Pudełko z amunicją

4.5 Przeciwnicy

4.5.1 Omówienie obiektu przeciwnika

Projektowanie przeciwników w grach jest ściśle związane z zagadnieniem sztucznej inteligencji. Aplikacja wykorzystuje dwa rozwiązania typowe dla gier FPS. Pierwszym jest automat stanów skończonych, a drugim inteligentne wyznaczanie ścieżek algorymem A*.

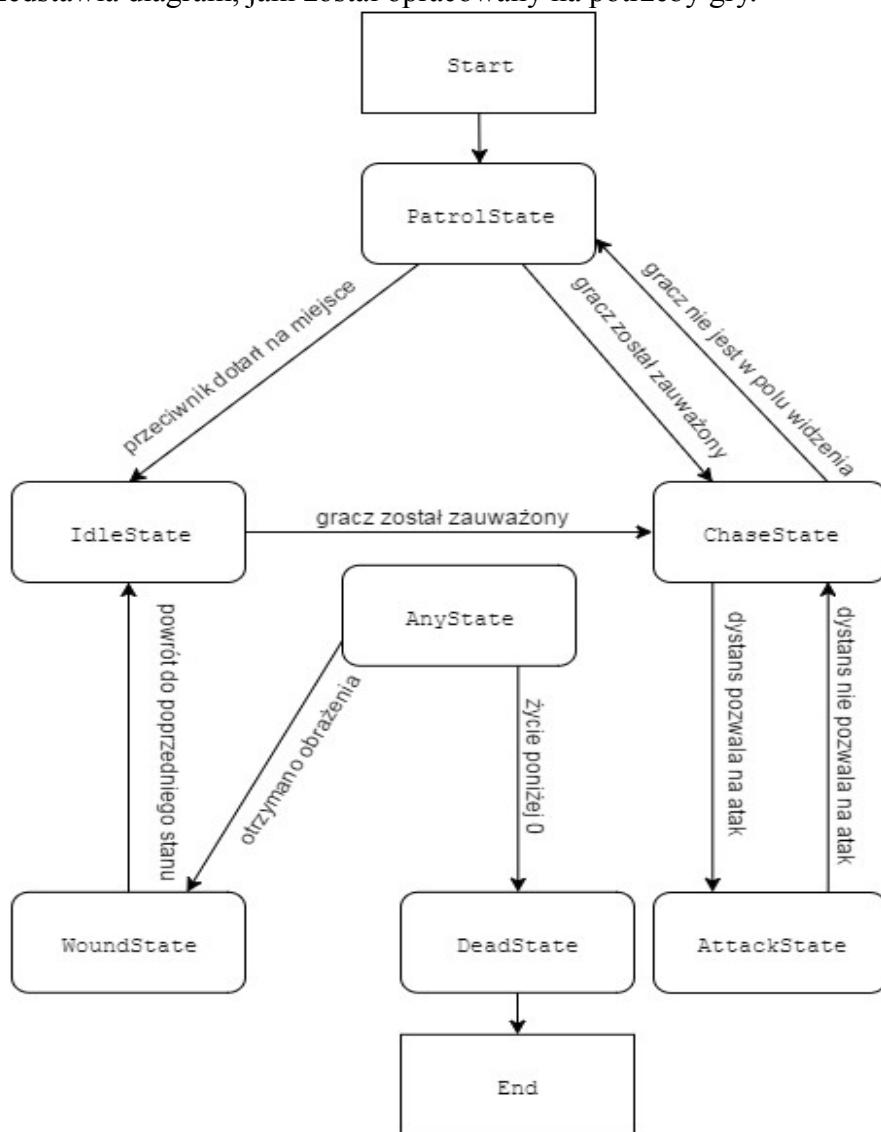
Omawiani przeciwnicy dzięki tym rozwiązaniom nie wchodzą w obiekty, które są przeszkodami (np. łódki), a omijają je podczas podążania do swojego punktu kontrolnego. Ponadto posiadają funkcje odpowiedzialne za:

- pozostawanie w czuwaniu,
- podążanie za graczem,
- atakowanie go,
- otrzymywanie obrażeń,
- śmierć.

4.5.2 Automat stanów skończonych

Podczas implementacji AI (*ang. artificial intelligence*) napotkano problemem przejść pomiędzy jednym stanem a drugim. Przykładowo, przeciwnik mógł podążać do wyznaczonego mu celu, podczas gdy gracz atakował go, a dodatkowo wszedł w jego obszar widzenia. Wraz z powiększającą się liczbą możliwych stanów, w jakich znajduje się przeciwnik wzrastają warunki określające, kiedy należy ustawić odpowiednią animację oraz kiedy udostępnić funkcje np. od ataku lub biegu.

Do zaprojektowania różnych stanów i przejść pomiędzy nimi wykorzystano wzorzec stanu w rozbudowanej formie, jaką jest FSM (*ang. finite state machine*). Jest to matematyczny model określający na podstawie diagramu stanów zachowanie obiektu. Rys. 4.23 przedstawia diagram, jaki został opracowany na potrzeby gry.



Rys. 4.23. Diagram przejść pomiędzy stanami

Implementacja automatu stanów skończonych zaczęła się od określenia wspólnego interfejsu dla klas odpowiadających za poszczególne stany *IEnemyState*. Zawiera on nazwy funkcji, jakie mają implementować klasy, które będą z niego korzystać. Zalicza się do nich standardowe funkcje na wyzwalaczach (wejście, przebywanie, wyjście), a także funkcje wykonujące przejścia pomiędzy stanami – o nazwach analogicznych do void *ToAttackState()*.

Klasa *EnemyStats* jest dołączona do obiektu przeciwnika i zawiera zmienne, które definiują jego parametry, a także pobiera informację o poziomie życia przeciwnika.

Listing 4.7: Klasa *EnemyStats*:

```
public class EnemyStats : MonoBehaviour {  
    public float attackDelay = 0.5f;  
    public float attackDistance = 3.0f, attackDmg = 10.0f;  
    public float walkSpeed = 200.0f, hp = 100.0f;  
    public float playerHp = 100.0f;  
    void getPlayerHp(float pHp){  
        playerHp=pHp;  
    }  
}
```

Klasa **FSM** także jest dołączona do obiektu przeciwnika i odpowiada za zarządzanie stanami agenta. W niej zawarty jest kod odpowiedzialny za aktualizację oraz obsługiwanie wyzwalaczy dla aktualnie wykonywanego stanu.

Klasa **PatrolState** implementuje (tak jak pozostałe klasy, których nazwy kończą się na State) interfejs *IenemyState*. Ta klasa odpowiada za stan, w którym obiekt, dzięki funkcji *Patrol()* podąża do celu.

Sam ruch postaci jest istotną kwestią. Aby postać się poruszała pobierany jest komponent AnimationController, określany obrót od punktu w którym się znajduje cel do postaci, następnie stopniowo zmienia się wartość rotacji postaci uzależniając ją od czasu, potem ustawia się animację na bieg, a dzięki kontrolerowi postaci nakazuje postaci przemieszczanie się o wektor, który jest na bieżąco wyliczany od punktu docelowego do postaci.

Do jej zadań należy również przekierowanie do innego stanu – w tym wypadku ChaseState albo IdleState w zależności od warunków.

Listing 4.8 Wybrane elementy klasy *PatrolState*:

```
private readonly FSM enemy;

public PatrolState (FSM fsm)
{
    enemy = fsm;
}

public void OnTriggerEnter (Collider other)
{
    EnemyStats es = enemy.gameObject.GetComponent<EnemyStats> ();
    if (other.tag.Equals ("Player") && (es.hp > 0)) {
        ToChaseState ();
    }
}
void Patrol () {
(...)
    if (currentWayPoint >= path.vectorPath.Count) {
        ToIdleState ();
    }
}
public void ToChaseState ()
{
    enemy.currentState = enemy.chaseState;
}
public void ToIdleState () {

    enemy.currentState = enemy.idleState;

}
```

Klasa ***ChaseState*** działa na podobnej zasadzie, co klasa odpowiadająca za patrolowanie, ponieważ też dotyczy przemieszczania się, jednak nie do statycznego obiektu, a do dynamicznego *Gracza*. Przy wyznaczaniu obrotu oraz wektora odległości korzysta się z komponentu transform obiektu *Gracza*. Gdy odległość dzieląca przeciwnika od gracza jest mniejsza bądź równa od zmiennej `attackDistance`, wówczas następuje zmiana stanu na atak. W razie, gdy gracz wyjdzie z wyzwalacza przeciwnika, następuje powrót do stanu patrolowania, a ścieżka jest wyznaczana na nowo.

Klasa ***IdleState*** odpowiada za ustawienie animacji stania w bezruchu. Jest konieczna, gdy przeciwnik dociera do swojego punktu kontrolnego. Gdy gracz wejdzie w jego pole widzenia, następuje zmiana stanu na podążanie.

Klasa ***AttackState*** posiada funkcję ataku. Gdy dystans jest większy od wartości zmiennej `attackDistance`, następuje zmiana stanu na podążanie.

Listing 4.9: funkcja odpowiedzialna za atak:

```
void Attack() {  
  
    EnemyStats es =  
    enemy.gameObject.GetComponent<EnemyStats> ();  
    AnimationController sc =  
    enemy.gameObject.GetComponent<AnimationController> ();  
  
    float distance = Vector3.Distance  
(enemy.transform.position, enemy.player.transform.position);  
    if (timer <= 0 && es.playerHp>0) {  
        sc.animationSet ("attack0");  
        enemy.player.SendMessage ("takeHit",  
        es.attackDmg);  
        timer = es.attackDelay;  
    }  
    else if (distance > es.attackDistance) {  
        ToChaseState();  
    }  
    if (timer > 0) {  
        timer -= Time.deltaTime;  
    }  
}
```

Klasa **WoundState** odpowiada za zmianę animacji postaci na otrzymywanie obrażeń. Jest dostępna z każdego stanu, ponieważ nie jest wymagane, aby przeciwnik widział gracza, gdy ten do niego strzela. Po jej wykonaniu następuje przekierowanie do stanu bezczynności (animacja stania) i w razie konieczności (gdy przeciwnik otrzymał obrażenia ścigając gracza) dalszej zmiany stanu na podążanie.

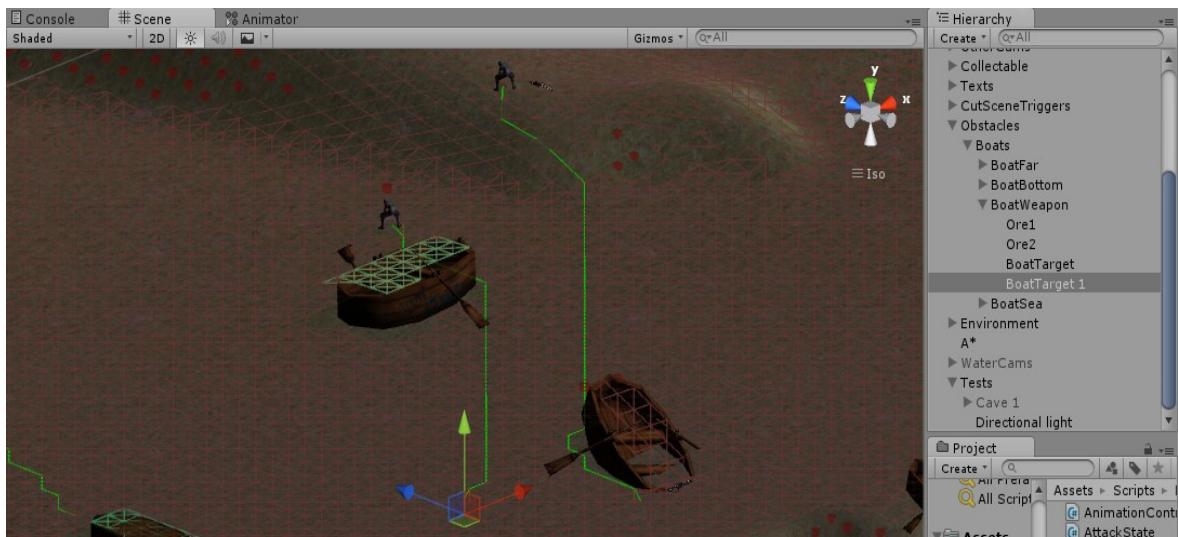
Do stanu **DeadState** można również być przekierowanym z każdego innego położenia. Ustawia on animację na śmierć. Przeciwnik będąc w tym stanie nie może brać dalej udziału w innych interakcjach z graczem.

4.5.3 Inteligentne wyznaczanie ścieżek algorytmem A*

Pojęcie wyznaczania ścieżek odnosi się do obliczania najkrótszej ścieżki pomiędzy punkami A oraz B. Algorytm A* jest optymalny – dzięki zastosowaniu go uzyskuje się najkrótszą ścieżkę pod warunkiem jej istnienia.

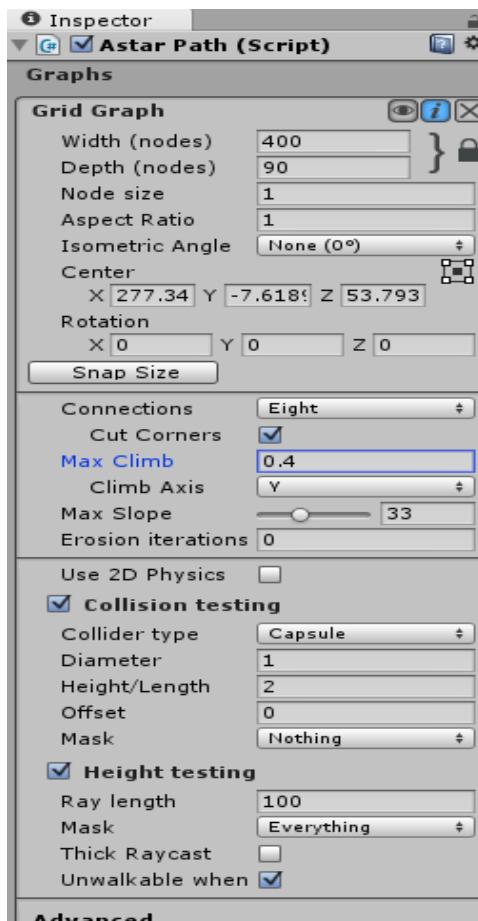
Zasada działania opiera się na minimalizacji funkcji $a(x)$, która jest sumą funkcji $b(x)$ i $c(x)$. Funkcja $b(x)$ to droga dzieląca wierzchołek startowy od aktualnego wierzchołka x , czyli jest to rzeczywista długość ścieżki już wygenerowanej, natomiast $c(x)$ to przewidywana heurystycznie droga od aktualnego wierzchołka x do punktu końcowego.

Rys. 4.24 pokazuje wyznaczone ścieżki przez algorytm A* - są to zielone linie. Obiekty łódek zostały ominięte, ponieważ są zbyt wysokie, by postać mogła na nie wejść.



Rys. 4.24. Algorytm A*

Maksymalna wysokość możliwa do pokonania jest wartością, która podlega modyfikacji w oknie inspektora komponentu Astar (rys. 4.25).



Rys. 4.25. Fragment okna Inspektora dla obiektu Astar

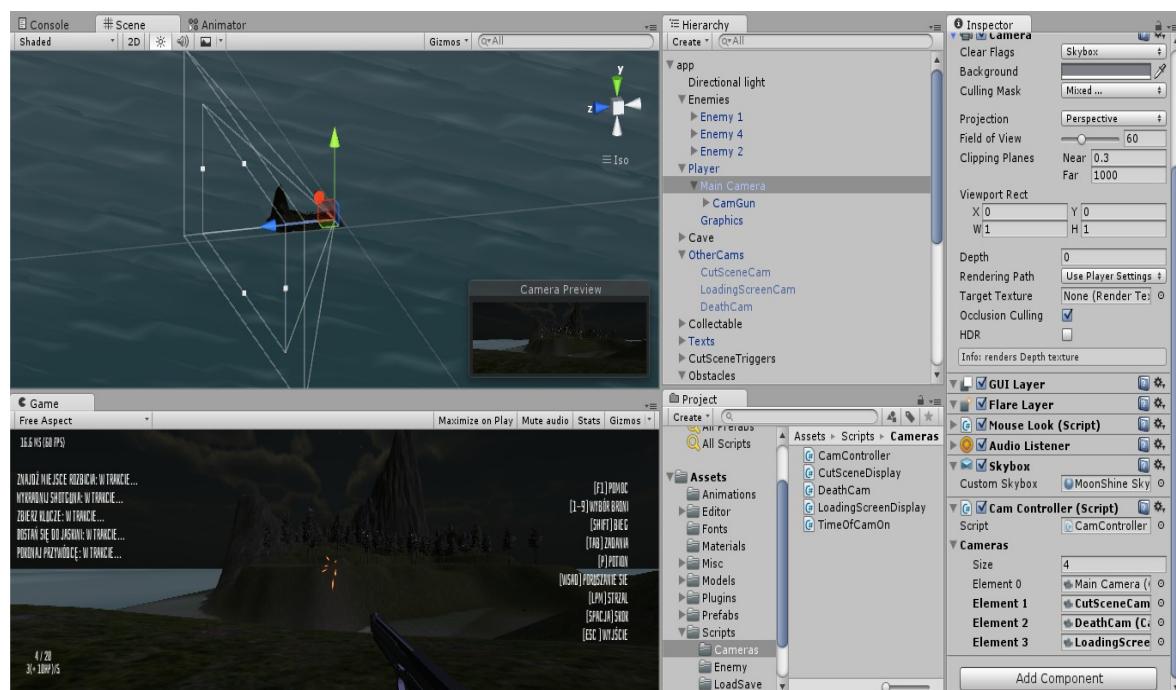
4.6 System kamer

4.6.1 Kamery podczas gry

W każdej aplikacji 3D jest konieczne używanie przynajmniej jednej kamery. Różne wymagania odnośnie renderowania obrazu wymuszają operowanie na kilku kamerach. W aplikacji znajduje się pięć kamer, z własnoręcznie zaprojektowanym systemem ich zarządzania, a są to:

- Main Camera,
- Cam Gun,
- DeathCam,
- Cut Scene Cam,
- Loading Screen Cam.

Main Camera to główna kamera, która odpowiada za renderowanie obrazu, gdy gracz się porusza. Dlatego też jest to dziecko obiektu *Player*, aby jej położenie było uwarunkowane położeniem gracza. Nie wyświetla jednak warstw oznaczanych jako broń. W oknie *Scene* widać, jaki zasięg posiada ta kamera, a w oknie *Camera Preview*, co renderuje. Skrypt *Cam Controller* zawiera w tablicy wszystkie wykorzystywane w aplikacji kamery oraz zarządza nimi – funkcja *CamOn (int number)* ustawia na aktywną wybraną kamerę, a funkcja *hideCams ()* chowa wszystkie kamery.



Rys. 4.25. Widok pracy z kamerami

Kamera **CamGun** renderuje jedynie bronie. Obiekt ten jest dzieckiem kamery **MainCamera**, aby zawsze jego położenie było uzależnione od głównej kamery. Z kolei jej dziećmi są obiekty broni. Aby te obiekty zawsze było widać na pierwszym planie, został zwiększyły parametr `depth` odpowiadający za głębię widzenia. Zapobiega to sytuacji, gdy w polu widzenia znajduje się lufa pistoletu oraz inny obiekt np. łódka i renderowana jest łódka, a lufa pistoletu jest ucięta i wtapia się w ten model.

Gdy gracz umrze, widok jest przełączany na kamerę **DeathCam**. Dla tej kamery został napisany skrypt, który symuluje upadek gracza na bok, przez co rozgrywka wygląda bardziej realistycznie (rys. 4.26). Pobiera on z obiektu gracza czas, przez jaki gracz ma być martwy, a po jego zakończeniu zostaje przywrócony widok z poprzedniej kamery.



Rys. 4.26. Animacja kamery podczas umierania

Listing 4.9 Skrypt DeathCam:

```
public class DeathCam : MonoBehaviour {

    float rotationTime = 1f;
    int deadTimeCam= 0;
    public GameObject partner;

    void Start () {

        PlayerStats player =
partner.GetComponent<PlayerStats>();
        transform.position = player.transform.position;
        transform.rotation=player.transform.rotation;

    }

    void Update () {

        PlayerStats sc = partner.GetComponent<PlayerStats>();
        if (sc.deadTime > deadTimeCam) {
            transform.position = partner.transform.position;
            transform.rotation=partner.transform.rotation;
            deadTimeCam+=1;

        }
    }
}
```

```

        Quaternion targetRotation = Quaternion.Euler(0, 0, 90);
        Quaternion finalRotation = Quaternion.Slerp
(transform.rotation, targetRotation, rotationTime *
Time.deltaTime);
        transform.rotation = finalRotation;

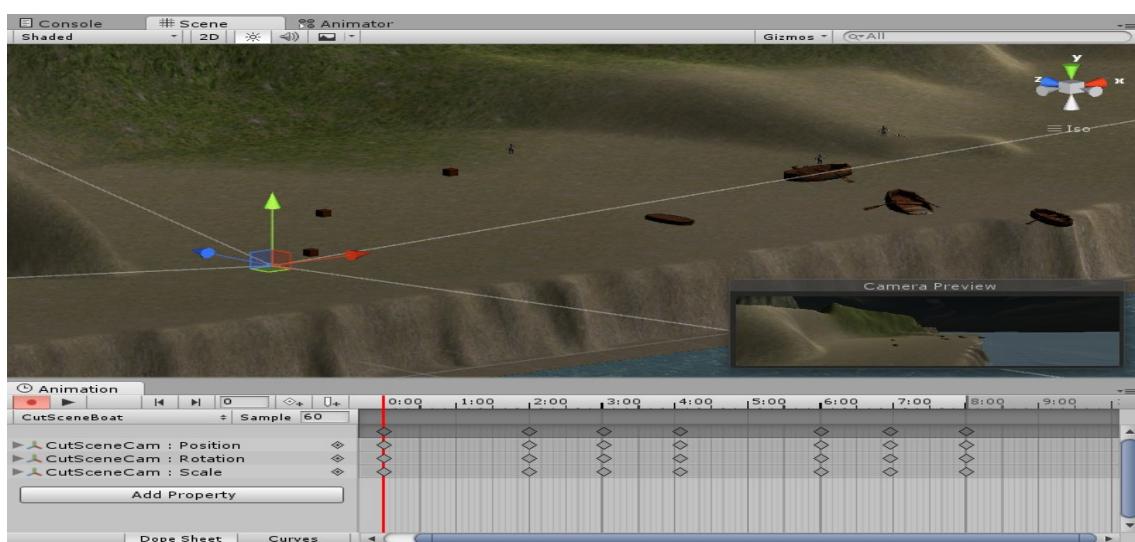
    }
}

```

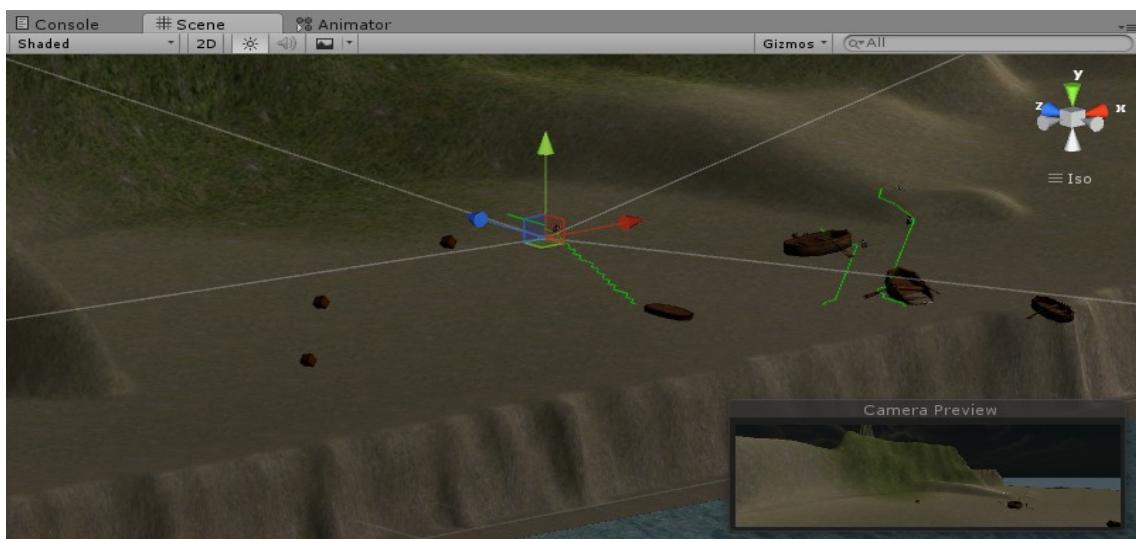
4.6.2 Przerywnik filmowy

W celu urozmaicenia rozgrywki zostały wprowadzone przerywniki filmowe (*ang. cut scene*). Podczas ich trwania użytkownik nie ma możliwości kontrolowania swojej postaci. Jednak jest to istotny element przy budowaniu klimatu gry, wprowadzaniu do fabuły. W projekcie zostało to wykorzystane do pokazania graczowi miejsca na mapie, gdzie znajduje się broń, która jest potrzebna do wykonania zadania. Dodatkowo widać, jak przeciwnicy podążają w jej kierunku, co buduje napięcie.

Kamera **Cut Scene Cam** korzysta ze skryptu *TimeOfCamOn* odpowiedzialnego za odliczanie czasu, po którym do kontrolera kamery zostanie wysłana wiadomość o przełączeniu tej kamery na główną kamerę (co skutkuje końcem trwania przerywnika filmowego i zwróceniem sterowania graczowi). Została opracowana animacja kamery, korzystająca z kolejnego okna Unity, jakim jest *Animation*. Ta animacja została dołączona do komponentu *Animator*. Ruch kamery zaczyna się w miejscu, gdzie gracz wchodzi w wyzwalacz wywołujący wyświetlenie się przerywnika filmowego, a kończy na miejscu, gdzie leży broń, co pokazują poszczególne ujęcia klatek na rys. 4.27 do rys. 4.30.



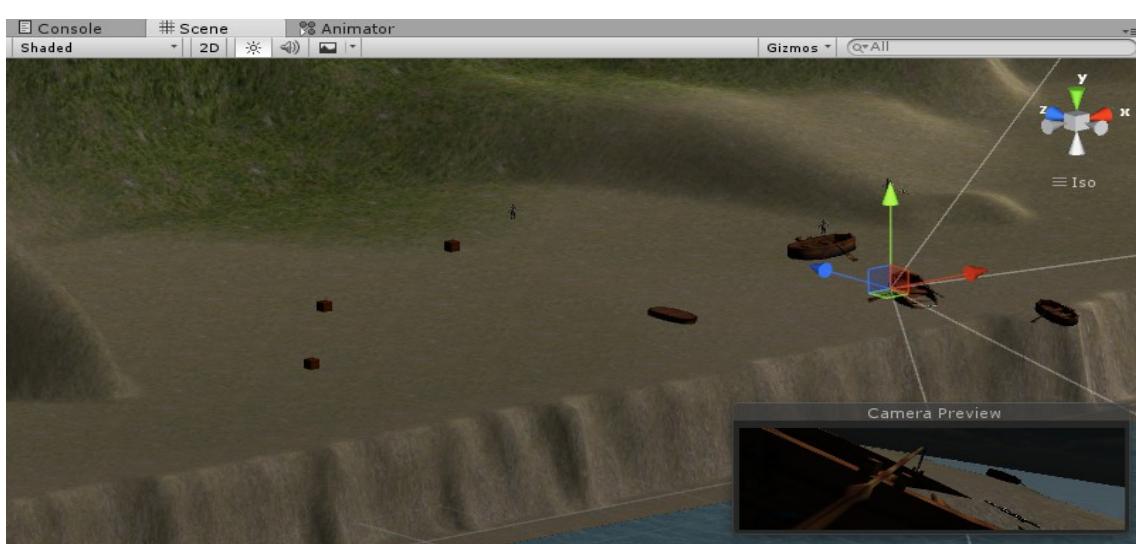
Rys. 4.27. Początkowa klatka z animacji kamery



Rys. 4.28. Jedna z środkowych klatek z animacji kamery (spojrzenie w lewo)



Rys. 4.29. Jedna z środkowych klatek z animacji kamery (spojrzenie w prawo)



Rys. 4.30. Końcowa klatka z animacji kamery

4.6.3 Ekran ładowania

Pomiędzy zmianą jednego poziomu na drugi, czyli zmianą scen, konieczne było schowanie aktualnego widoku sceny oraz ustawienie grafiki powiadającej użytkownika, o procesie zapisu i ładowania kolejnego poziomu.

Skrypt **ChangeLevel** zostaje wywołany, gdy gracz wejdzie w wyzwalacz umieszczony w obiekcie odpowiadającym za przejście do kolejnego poziomu gry. Zmienia on kamerę na **Loading Screen Cam**, która wyświetla grafikę umieszczoną na rys. 4.31, a także odlicza potrzebny czas – minimum 3 sekundy do załadowania kolejnego poziomu gry.



Rys. 4.30. Ekran ładowania

Listing 4.10 Klasa *ChangeLevel*:

```
public class ChangeLevel : MonoBehaviour {
    public GameObject loadingScreen;
    public GameObject partner;

    void Start () {
        loadingScreen.SetActive(false);
    }

    void OnTriggerEnter(Collider other) {
        CamController cc = partner.GetComponent<CamController> ();
        if (other.tag.Equals ("Player")) {
            cc.CamOn(3);
            StartCoroutine (loadNextLvl("lvl02"));
            other.SendMessage("saveLvl");
            other.SendMessage ("hideUI", true);
        }
    }
}
```

```

        }
    }

    IEnumerator loadNextLvl(string lvlName) {
        loadingScreen.SetActive(true);
        yield return new WaitForSeconds (3.0f);
        AsyncOperation async = Application.LoadLevelAsync
(lvlName);
        while (!async.isDone) {
            yield return null;
        }
    }
}

```

4.7 System zapisu i odczytu

W punkcie 4.6.3 już wspomniano o zapisie gry. W aplikacji przyjęto, że zapis gry nastąpi przy ładowaniu kolejnego poziomu gry. Odczyt następuje w momencie wyboru z głównego menu opcji „*Wczytaj Grę*”.

Wykorzystano serializację, aby przekonwertować obiekty na strumień bajtów. **Zapis gry** polega na zapisaniu na dysku w plikach o rozszerzeniach .dat: stanu statystyk gracza, czyli aktualnych poziomów życia, pancerza, energii, liczbie posiadanych jagód, wiadomości, czy gracz posiadał klucz, numery posiadanych broni, numer aktualnie wybranej broni, wiadomości, które zadania zostały ukończone, a także w osobnych plikach stany magazynków oraz ilość amunicji danej broni.

W skrypcie ***Game Controll*** utworzono klasę *PlayerData*, która podlega serializacji i zawiera potrzebne pola z racji przechowania tych danych w przez różne komponenty.

Listing 4.11: Klasa *PlayerData*:

```

[Serializable]
class PlayerData{
    public float hp;
    public float armour;
    public float energy;
    public int nowPots;
    public bool haveKey;
    public bool[] guns;
    public int nowGun;
    public bool[] quests = new bool[] {false, false, false,
false, false, false};
}

```

Funkcja *Save()*: pobiera potrzebne komponenty z obiektu gracza, tworzy plik, instancję klasy *PlayerData* o nazwie *data*, przypisuje do jej pól dane pobierane z wcześniej pobranych komponentów, serializuje te dane w pliku oraz zamyka plik. Funkcja *Load()*

działa podobnie, ale najpierw sprawdza, czy plik o wskazanej nazwie istnieje, potem otwiera go, deserializuje (proces polegający na konwersji strumienia danych do klasy - w tym wypadku *PlayerData*), zamyka plik i ustawia wartości poszczególnych komponentów obiektu gracza na odczytane z pliku wartości.

Listing 4.12: funkcje *Save()* oraz *Load()* skryptu *GameControll*:

```
public void Save() {
    PlayerStats ps =
gameObject.GetComponent<PlayerStats>();
    GunsInventory gi =
gameObject.GetComponent<GunsInventory>();
    Quest q = gameObject.GetComponent<Quest>();

    BinaryFormatter bf = new BinaryFormatter ();
    FileStream file = File.Create
(Application.persistentDataPath + "/playerInfo.dat");
    PlayerData data = new PlayerData ();
    data.hp = ps.nowHealth;
    data.armour = ps.nowArmour;
    data.energy = ps.nowEnergy;
    data.nowPots = ps.nowPots;

    data.haveKey = ps.haveKey;
    data.guns = gi.guns;
    data.nowGun=gi.nowGun;

    for (int i=1; i<q.quests.Length; i++) {
        data.quests [i] = q.quests[i];
    }

    bf.Serialize (file, data);
    file.Close ();
}

public void Load() {
    PlayerStats ps =
gameObject.GetComponent<PlayerStats>();
    GunsInventory gi =
gameObject.GetComponent<GunsInventory>();
    Quest q = gameObject.GetComponent<Quest>();

    if (File.Exists (Application.persistentDataPath +
"/playerInfo.dat")) {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file =
File.Open(Application.persistentDataPath + "/playerInfo.dat",
 FileMode.Open);
        PlayerData data =
(PlayerData)bf.Deserialize(file);
        file.Close();
    }
}
```

```

        ps.nowHealth=data.hp;
        ps.nowArmour = data.armour;
        ps.nowEnergy=data.energy;
        ps.nowPots = data.nowPots;

        ps.haveKey=data.haveKey;
        gi.guns= data.guns;
        gi.nowGun=data.nowGun;

        for (int i=1; i<q.quests.Length; i++) {
            q.quests [i] = data.quests[i];
        }
    }

}

```

Skrypty ***ShotGunControll***, ***SkorpionControll*** oraz ***WeaponControll*** działają analogicznie, jednak w klasach podlegających serializacji posiadają jedynie dwie zmienne odpowiadające za liczbę pozostałej amunicji w magazynku oraz liczbę magazynków.

Skrypt ***Save*** zawiera funkcję zapisu poziomu gry. Wykorzystując konstruktor klasy *GameControll* wywołuje jej funkcję dotyczącą zapisu statystyk gracza. Potem sprawdza po wszystkich indeksach, czy gracz posiadał daną broń, jeśli tak, to: ustawia ją na aktywną, wywołuje jej funkcję zapisu i ustawia ją na nieaktywną. Na koniec ustawia broń będącą wybraną przez gracza w momencie zapisu gry.

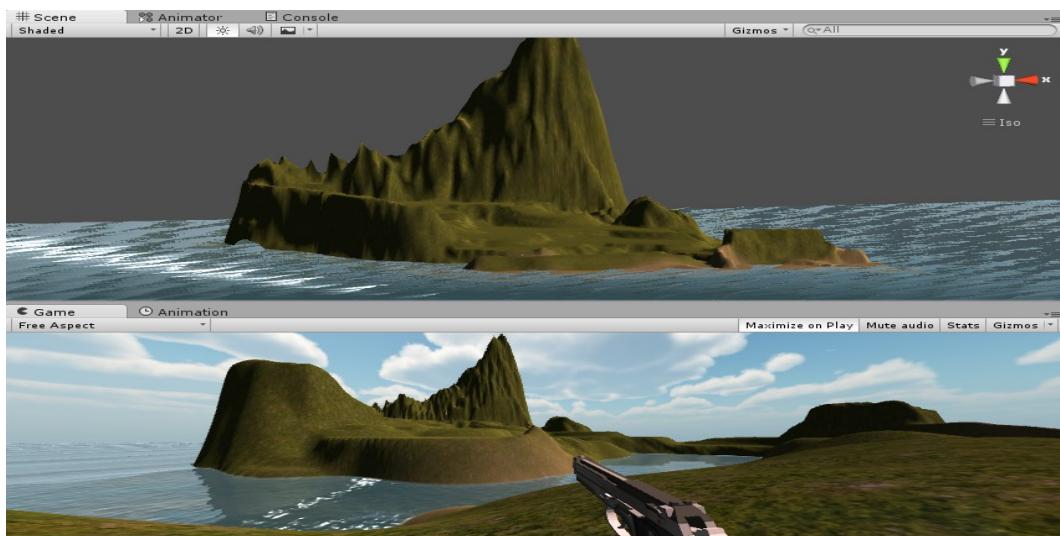
Skrypt ***Load*** jest podobnie jak skrypt dołączony do obiektu *Player*. W funkcji *Start()* sprawdza, czy poziom gry wymaga ładowania danych. Jeśli tak, to postępuje analogicznie do skryptu *Save()* z tą różnicą, że wywołuje poszczególne funkcje odpowiadające za odczyt.

4.8 Grafika i dźwięk

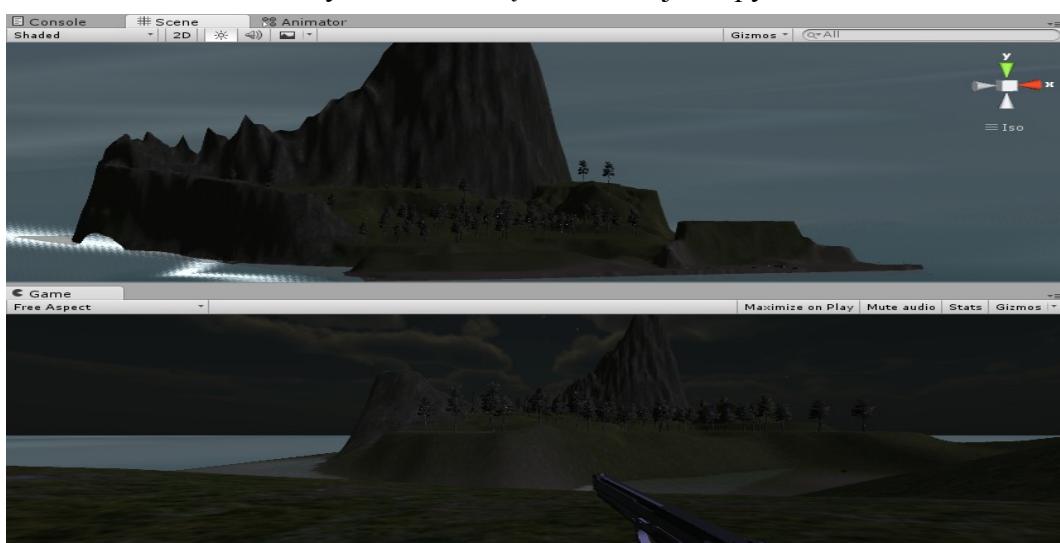
4.8.1 Projektowanie map

Gra składa się z dwóch poziomów, a więc dwóch map. Pierwszą stanowi teren będący wyspą, która jest ograniczona od jednej strony górami, a z pozostałych wodą, natomiast drugą mapą jest jaskinia. Aby rozgrywka była zróżnicowana pierwsza część rozgrywa się na otwartej przestrzeni, gdzie gracz widzi rozgwieżdżone niebo oraz otoczenie – wodę, drzewa, krzaki, porusza się po pagórkach i brzegu morza. Drugi etap gry odbywa się w zamknięciu, labiryncie, gdzie trzeba zapamiętywać przebytą trasę, aby się nie zgubić i odnaleźć cel.

Pierwszy poziom gry został zaprojektowany w oparciu o narzędzie, jakie dostarcza Unity 3D – terrain. Umożliwia ono modyfikację geometrii podstawowego płaskiego terenu. Proces ten odbywa się poprzez używanie pędzli o różnych parametrach – rozmiarze, kształtach, przeźroczystości, maksymalnej wysokości. Pozwala także na łagodzenie utworzonej siatki poprzez narzędzie smooth height. Kolejnym narzędziem jest pędzel pozwalający na nanoszenie tekstur. Można używać kilku tekstur i określić ich przenikanie się. Dostępne jest również narzędzie, dzięki któremu można nanosić drzewa oraz operować na ich wysokości, gęstości rozmieszczenia. W celu utworzenia rozgrywki trzymającej w napięciu zmieniono porę dnia z poranka (rys. 4.32) na noc (rys 4.33). Ustawiony został obiekt Skybox na odpowiednią teksturę oraz dostosowane parametry głównego światła (mniejsze natężenie, chłodniejsza barwa).

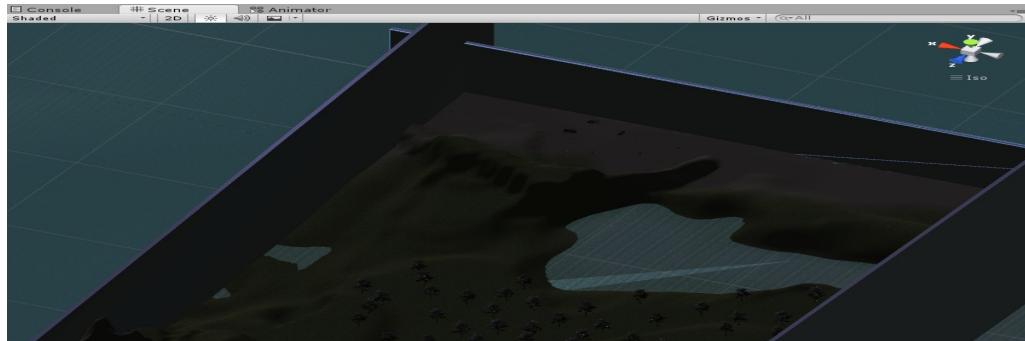


Rys. 4.31. Początkowa wersja mapy



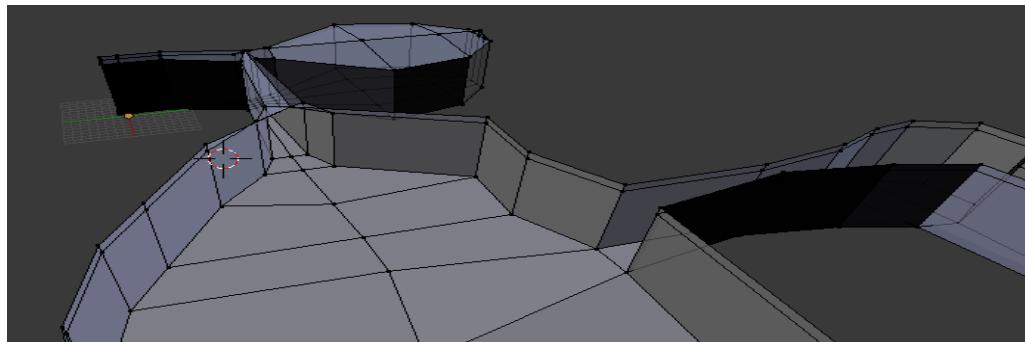
Rys. 4.32. Finalna wersja mapy

Ze względu na otwartą przestrzeń, a szczególnie miejsca, gdzie kończy się ląd, a zaczyna woda, zostały dodane tzw. „niewidzialne ściany” (rys. 4.33). Są to obiekty, które nie pozwalają użytkownikowi na przedostanie się za dany obszar. Ich rendering podczas gry jest wyłączony.

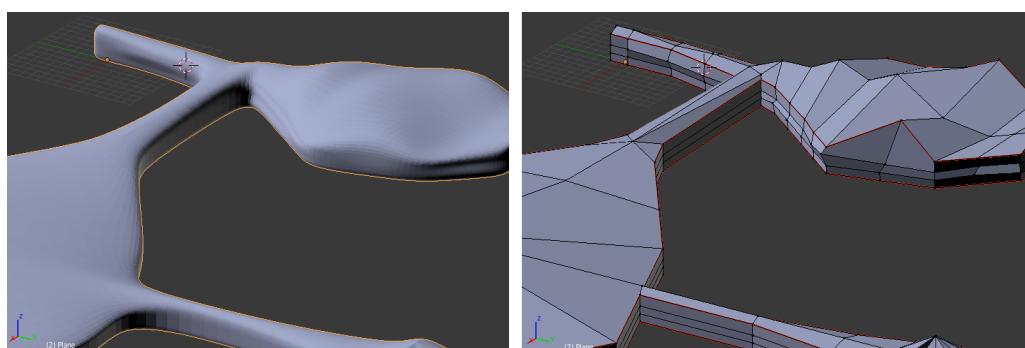


Rys. 4.33. Ograniczenia mapy

Drugi poziom gry został wymodelowany w Blenderze. Jest to model pusty w środku, a rys. 4.34 pokazuje moment, gdy już został opracowany plan mapy i zaczęto łączyć wierzchołki górnych ścian, aby utworzyć strop. W celu zaokrąglenia ostrych krawędzi oraz wzbogacenia siatki (rys. 4.35) zastosowano modyfikator podziału powierzchni. Z lewej strony widoczny jest fragment modelu w oknie *Object Mode* z włączonym modyfikatorem, a z prawej okno *Edit Mode* bez włączenia jego podglądu.

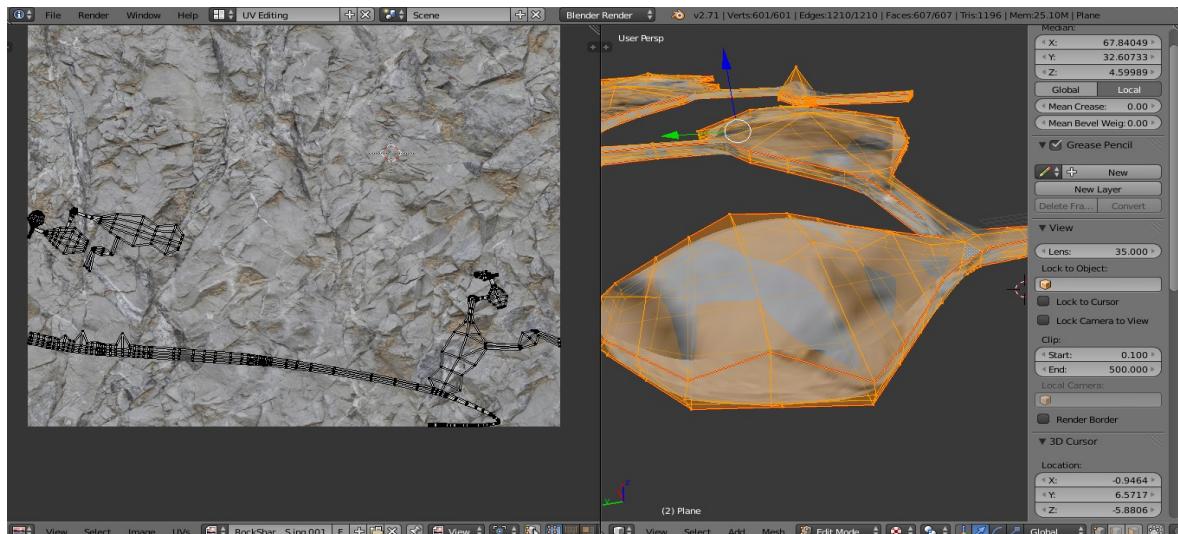


Rys. 4.34 Proces tworzenia modelu jaskini



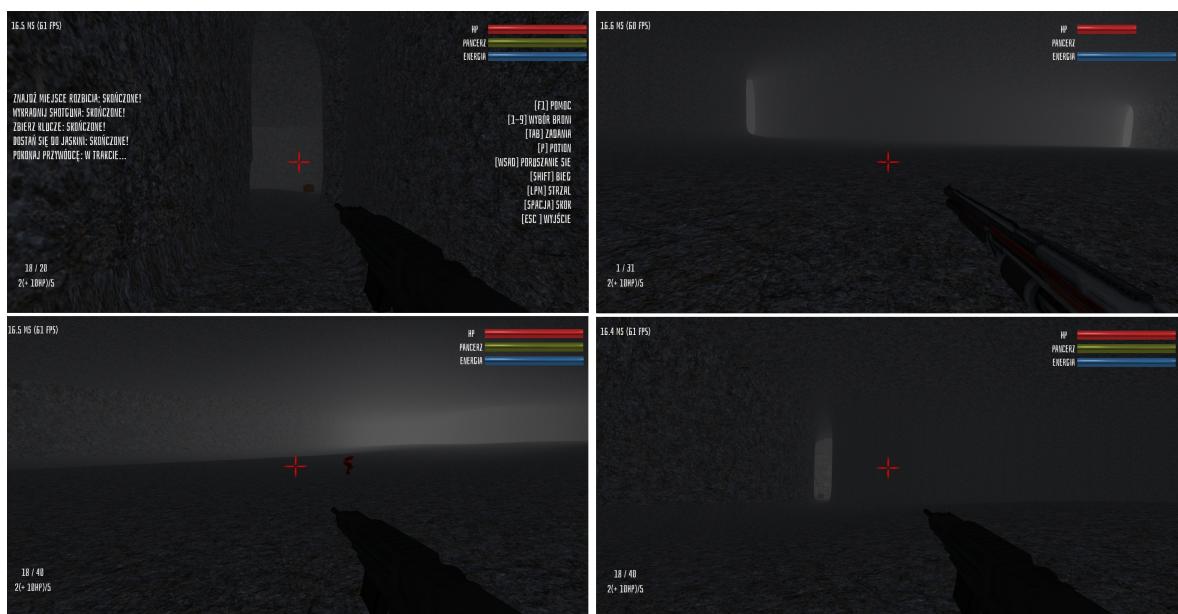
Rys. 4.35. Porównanie wyglądu geometrii

Czerwone linie widoczne na modelu to szwy. Zostały naniesione ręcznie poprzez zaznaczanie wybranych krawędzi, aby podzielić model na trzy zasadnicze części – górną oraz dolną podstawę oraz pasmo ścian. Po użyciu opcji unwrap została utworzona siatka modelu na podstawie tych cięć. Następnie został przypisany materiał oraz tekstura do modelu oraz zmieniony widok okien na *UV Editing*. Zaprezentowano na rys. 4.36 początkową fazę teksturowania. Rozmiar siatki należało przeskalaować, aby wzór skał był szczegółowy, a nie rozciągnięty po całej powierzchni.



Rys. 4.36. Początkowa faza teksturowania

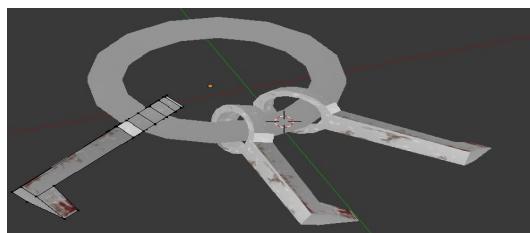
W aplikacji scena składa się z wykonanego modelu oraz dodanego efektu mgły, co widać na zrzutach ekranu z rys 4.37.



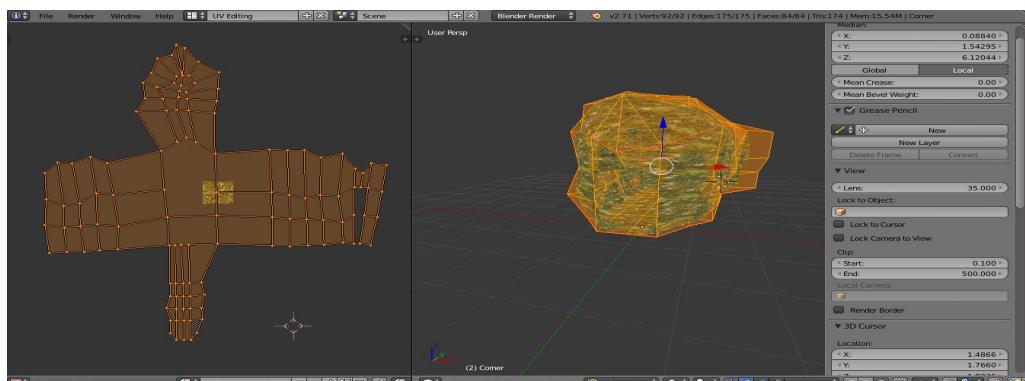
Rys 4.37 Zrzuty ekranu z drugiego poziomu gry

4.8.2 Projektowanie modeli oraz grafiki 2D

Modele zostały zaprojektowane w Blenderze. W podręczniku do Unity 3D [14] jest zawarta odnosiąca się do nowoczesnych kart graficznych – radzą sobie dobrze dużą liczbą wielokątów, a podczas renderowania obiekty składające się ze 100 trójkątów są tak samo kosztowne, jak obiekty złożone z 1500 trójkątów. Dla optymalnego renderingu zaleca się 1500-4000 trójkątów przypadających na jeden mesh. Wynika z tego, że nie trzeba upraszczać siatek, które mają poniżej 1500 trójkątów. Obiekty eksportowane do Unity poddawane są automatycznie procesowi triangulacji, zatem tworząc obiekt można wykorzystywać prostokąty. Używanie ich jest zalecane ze względu na utrzymywanie prostej siatki – co widać na rys. 4.38 oraz rys. 4.39. Rys 4.40 pokazuje gotowy model z rys 4.49 w grze otoczony wymodelowanymi i oteksturowanymi głazami.



Rys. 4.38. Model kluczy



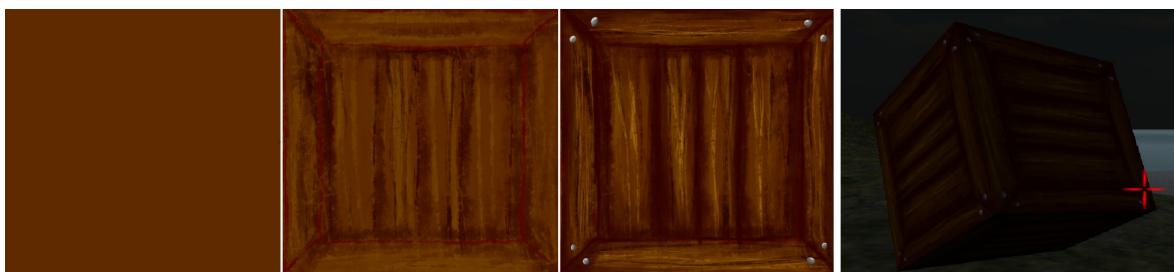
Rys. 4.39. Proces teksturowania wejścia do jaskini



Rys. 4.40. Widok modelu w grze

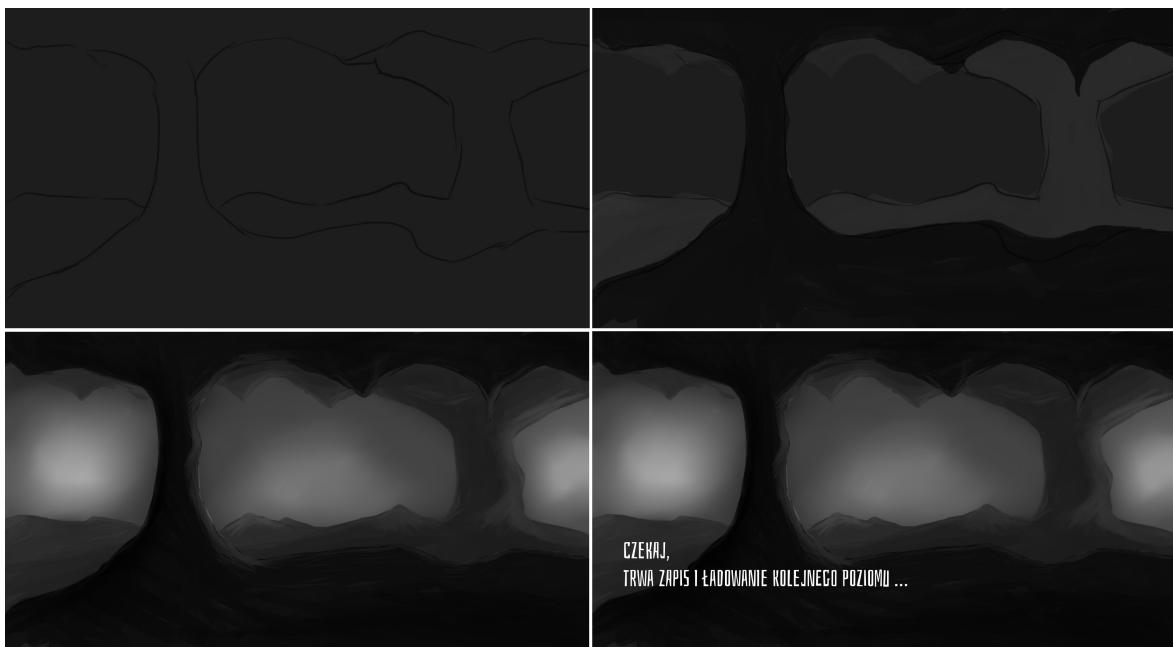
Podczas projektowania tekstur zostały użyte narzędzia omawiane w podrozdziale 3.3 oraz tablet graficzny Wacom Bamboo Pen & Touch A6.

Gdy istnieje możliwość obciążenia modelu, to nanoszone tekstury powinny tworzyć efekt trójwymiarowości na płaskich ścianach. Uproszczona siatka powoduje, że karta graficzna wykonuje mniej obliczeń wpływając korzystne na płynności rozgrywki. W przypadku pudełka na amunicję modelem był sześcian. Tekstura dzięki zastosowaniu cieniowania i rozjaśniania nadała głębie deskom (rys. 4.41). Gwoździe również nie zostały ujęte w geometrii modelu, natomiast dzięki narysowaniu ich na tekstuze wydają się być elementami wystającymi (rys. 4.42).



Rys. 4.41. Projektowanie tekstuury pudełka

Projektowanie ekranu lądowania również odbywało się w kilku krokach odpowiadających poszczególnym warstwom. Były to: szkic, podstawowe kolory, cienie, światła, detale i napis. Program Paint Tool SAI nie posiada możliwości wprowadzania tekstu, więc został on dodany w programie Gimp.



Rys. 4.43. Proces rysowania ekranu lądowania

Logo jest również ważnym elementem projektowania gry jako produktu na sprzedaż. Ten znak graficzny stanowi źródło informacji na temat danego produktu. Kierując się zasadami, że logo powinno być proste, czytelne, rozpoznawalne oraz posiadać jak najmniej kolorów zostało opracowane jako napis tytułu gry - *Here* z pociskiem w miejscu jednej z linii prowadzonej od litery *E*. Tło jest przezroczyste, a więc można je nanosić na inne grafiki.



Rys. 4.44. Logo

Przy doborze muzyki kierowano się jej wpływem na emocje. Wybrany utwór stanowiący ścieżkę dźwiękową podczas trwania całej rozgrywki pasuje do klimatu nocy oraz wprowadza dozę niepokoju. Dodatkowo zostały dołączone odpowiednie dźwięki wystrzału broni dla każdego typu oraz przeładowania, aby gracz lepiej wczuł się w rolę strzelca.

5. TESTY GRY KOMPUTEROWEJ

5.1 Warunki techniczne przeprowadzonych testów

Gra testowana była pod systemem Windows 7. Testy zostały przeprowadzone na dwóch komputerach. Pierwszym był laptop Lenovo Y580 z procesorem Intel Core i7-3630QM @ 2.40 GHz wyposażonym w 8.0GB pamięci RAM i dwie karty graficzne: NVIDIA GeForce GTX 660M i zintegrowaną Intel HD Graphics 4000. Drugi to PC z procesorem AMD Phenom II X4 955 3.2 GHz, pamięcią RAM 8.0GB i kartą graficzną NVIDIA GeForce GTX 950.

Podczas testowania gry skupiono się na liczbie fps (*ang. frame per second*), czyli liczbie klatek wyświetlanych w ciągu jednej sekundy. Jest to miara płynności renderowania obrazu. Dla gier z gatunku FPS przyjęto, że wystarczająca liczba fps wynosi 30 dla płynnej rozgrywki. Zalecane jest 60 albo 120+ fps, natomiast obraz odświeżany z częstotliwością poniżej 20 fps oko ludzkie z reguły postrzega jako pokaz slajdów

Włączając grę opartą na silniku Unity 3D wyświetla się okno, w którym jest do wyboru: rozdzielcość ekranu, jakość grafiki: *Fastest, Fast, Simple, Good, Beautiful, Fantastic*.

5.2 Wyniki testów wydajnościowych

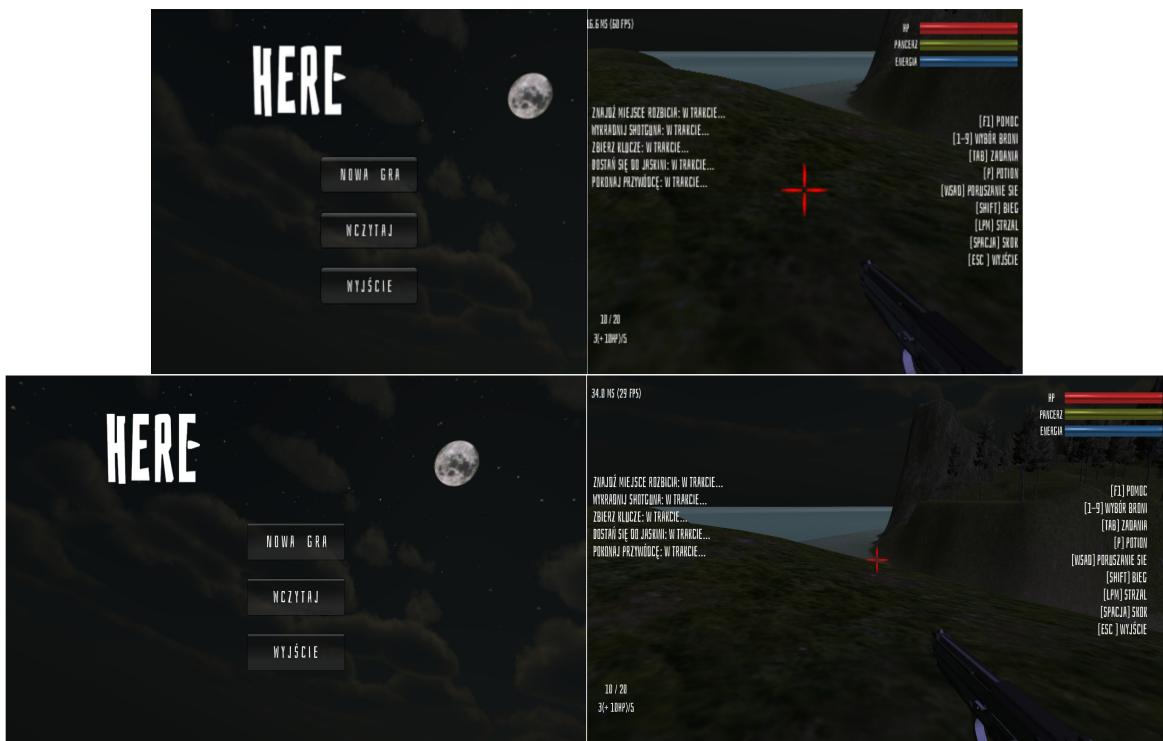
Parametry takie jak: procesor, karta graficzna, rozdzielcość ekranu, jakość grafiki wpływają na liczbę fps (tabela 5.1). Im mniejsza rozdzielcość ekranu, tym jest ich więcej, a im wyższe wymagania odnośnie detali graficznych, tym jest ich generowane mniej.

Tabl. 5.1. Zestawienie wpływu parametrów na liczbę fps

Procesor	Karta graficzna	Rozdzielcość	Jakość	Min fps	Max fps
Intel Core i7	zintegrowana	1366x768	Fastest	30	60
Intel Core i7	zintegrowana	1366x768	Good	60	60
Intel Core i7	zintegrowana	1024x768	Fantastic	40	60
Inter Core i7	NVIDIA GeForce GTX 660M	512x384	Fastest	300	500
Inter Core i7	NVIDIA GeForce GTX 660M	1366x768	Fanstastic	60	60
AMD Phenom II X4 955	NVIDIA GeForce GTX 950	2560x1600	Fanstastic	60	60
AMD Phenom II X4 955	NVIDIA GeForce GTX 950	1920x1080	Fantastic	60	60

Porównując zintegrowaną kartę graficzną, która posiada słabszą moc obliczeniową z kartą graficzną serii NVIDIA widać wyraźnie, że ta druga jest wydajniejsza. Dla ustawień o maksymalnej rozdzielczości ekranu i najlepszej grafice otrzymuje się stałe 60 fps. Aby ten wynik był zbliżony w przypadku zintegrowanej karty należało albo ustawić nieco niższą jakość grafiki albo zmniejszyć rozdzielczość ekranu. Jednak minimalna ilość otrzymywanych klatek na sekundę – 30 – nie powodowała zacinania się obrazu podczas renderowania, gra wciąż działała płynnie.

Rozmiary napisów, przycisków oraz innych grafik zostały ustalone z myślą o różnych rozdzielczościach ekranów i nie zostały ustalane na sztywno, a w oparciu o wysokość i szerokość ekranu, aby zapewnić dobrą skalowalność i zachowane proporcje. Zrzuty ekranów o rozdzielczości 512x384 i 1366x768 zostały zaprezentowane na rys. 5.1.



Rys. 5.1. Zrzuty ekranów z gry o różnych rozdzielczościach

6. ZAKOŃCZENIE

Celem projektu było opracowanie gry komputerowej FPS z wykorzystaniem silnika Unity 3D. Niniejsza praca stanowi opis zagadnień zarówno teoretycznych, jak i praktycznych dotyczących realizacji projektu. Proces projektowania gry obejmował różne aspekty. Począwszy od napisania fabuły, poprzez projektowanie poziomów, modelowanie i teksturowanie obiektów, dobór muzyki oraz dźwięków, skończywszy na programowaniu i testowaniu, na co został położony największy nacisk.

Główną zaletą projektu jest jego skalowalność. Dzięki zastosowaniu modelu ECS napisany kod w postaci komponentów można używać wiele razy w różnych obiektach. Zwiększa to też elastyczność, ponieważ można swobodnie dodawać i usuwać wybrane skrypty, co pomaga w wprowadzaniu nowych funkcjonalności oraz konserwacji. Wpływ na łatwość rozszerzania aplikacji ma również jego hierarchia. Obiekty są pogrupowane według określonych wytycznych i każdy nowy potencjalny obiekt gry ma już swoje wydzielone miejsce. Opracowany obiekt gracza i jego możliwości interakcji z otoczeniem – zbieranie przedmiotów, wybór broni, przechodzenie przez kolejne etapy gry też stanowią mocną stronę projektu. Skrypty odpowiadające za działania przeciwnika również zasługują na uwagę, ponieważ zostały napisane w oparciu o FSM i algorytm A*. Ponadto zostały opracowane dwa systemy: zapisu i odczytu oraz kamer.

Wadą jest brak w głównym menu *Opcji*, gdzie użytkownik mógłby dostosowywać wybrane parametry gry, takie jak natężenie oświetlenia oraz głośność do własnych preferencji. Następną słabą stroną jest miejscami uboga szata graficzna oraz model przeciwnika. Z racji niedostępności animowanego modelu postaci, która posługiwałaby się bronią strzelającą, gracz walczy jedynie z przeciwnikami bijącymi wręcz. Nie ma jednak przeszkód, by napisany skrypt odpowiadający za atak wykorzystać w takim modelu.

Patrząc szerzej na projekt widać potencjał. Aktualna wersja gry jest już w pełni grywalna. W planowanej kontynuacji pierwszym z pomysłów byłoby rozbudowanie głównego menu, fabuły oraz zaprojektowanie kolejnych poziomów gry. Wprowadzenie możliwości gry przez sieć stanowi również dobrą ideę.

LITERATURA

- [1] Entertainment Software Association.: „Essential facts about the computer and video game industry 2015” [online]. <http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf>
- [2] <http://www.valvesoftware.com/games/csgo.html>
- [3] <http://bgs.bethsoft.com/>
- [4] <http://www.infinityward.com/games>
- [5] <http://www.dice.se/games/>
- [6] <https://unity3d.com/legal/eula>
- [7] Will Goldstone.: „Projektowanie gier w środowisku Unity 3.x”, Helion, 2012
- [8] Alan Thorn.: „Pro Unity Game Development with C#”, Apress, 2014
- [9] Szymon Radziewicz.: „Nie musisz już płacić ani grosza za Unreal Engine 4” [online]. <http://www.spidersweb.pl/2015/03/unreal-engine-4-za-darmo.html> (dostęp 11.01.2015)
- [10] <https://unity3d.com/unity>
- [11] <http://unity3d.com/unity/system-requirements>
- [12] <http://unity3d.com/unity/download>
- [13] <http://www.monodevelop.com/>
- [14] <http://docs.unity3d.com/Manual/>
- [15] <https://www.blender.org/download/>
- [16] <https://www.gimp.org/downloads/>
- [17] <https://www.systemax.jp/en/sai/>
- [18] Boreal Games.: „Understanding Component-Entity-Systems” [online]. http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013 (dostęp 13.01.2015)
- [19] <http://www.fontsquirrel.com/fonts/bahiana>
- [20] <http://arongranberg.com/astar/>

[21] <https://www.assetstore.unity3d.com/en/#!/content/30272>

[22] <https://www.assetstore.unity3d.com/en/#!/content/3316>

[23] <https://www.assetstore.unity3d.com/en/#!/content/14233>

[24] <https://www.assetstore.unity3d.com/en/#!/content/26685>

[25] <https://www.assetstore.unity3d.com/en/#!/content/19256>

[26] <http://soundbible.com/tags-gun.html>

[27] <https://www.assetstore.unity3d.com/en/#!/content/19233>

ZAŁĄCZNIKI

Do pracy dyplomowej została załączona płyta CD przymocowana do wewnętrznej strony okładki.