

**UNIWERSYTET GDAŃSKI**  
**Wydział Matematyki, Fizyki i Informatyki**

**Mateusz Kleina, Adrian Podlawski**

nr albumu: 231 068, 236 232

**Sieciowa fabularna gra  
komputerowa - The Trinity**

Praca magisterska na kierunku:  
**INFORMATYKA**  
Promotor:  
**dr W. Bzyl**

Gdańsk 2017



## **Streszczenie**

Zaimplementowaliśmy wieloosobową grę w wersji Beta, która bazuje na silniku graficznym Unity.

Tryb multiplayer zbudowaliśmy na podstawie framework'a Photon Unity 3D Networking, który został pobrany ze sklepu Asset Store, a następnie zmodyfikowany i zaprogramowany zgodnie z wymaganiami gry.

Scenę po której poruszają się postaci utworzyliśmy dzięki silnikowi gry Unity. Krajobrazy oraz budynki zostały pobrane z Assets Store, a następnie zmodyfikowane. Wszystkie skrypty odpowiadające za sterowanie postaci, łączenie z serwerem, czy walkę z przeciwnikiem, zostały utworzone przez nas w języku programowania C Sharp. Postaci oraz ich animacje zostały utworzone przy pomocy storny [www.Mixamo.com](http://www.Mixamo.com), a następnie odpowiednio zaimplementowane przy pomocny skryptów.

W trakcie rozwijania aplikacji, pliki były przechowywane we wspólnym repozytorium na stronie <http://www.github.com>. Zbudowana wersja gry, którą można pobrać oraz uruchomić znajduje się na repozytorium Git (<https://github.com/kirin1994/ug-projekt-gra-rpg>). Projekt był testowany manualnie poprzez sterowanie postaciami oraz przy użyciu logów przesyłanych do silnika Unity. Za środowisko testowe posłużył nam osobny serwer, który nie miał wpływu na wydaną wersję projektu, znajdująca się na repozytorium.

## Słowa kluczowe

Gra, RPG, Multiplayer, Unity, PhotonUnityNetwork

# Spis treści

<b>Wprowadzenie</b> . . . . .	8
<b>1. Projektowanie oraz przygotowanie środowiska</b> . . . . .	11
1.1. Projekt Terenu . . . . .	12
1.2. Projekt Postaci . . . . .	14
1.3. Projekt systemu multilayer . . . . .	15
1.4. Instalacja niezbędnych narzędzi . . . . .	16
<b>2. Implementacja</b> . . . . .	18
2.1. Budowa terenu . . . . .	19
2.1.1. Przygotowanie terenu . . . . .	19
2.1.2. Formowanie kształtu . . . . .	20
2.1.3. Nakładanie tekstur . . . . .	20
2.1.4. Rozmieszczanie obiektów . . . . .	21
2.1.5. Oddziaływanie terenu na inne obiekty . . . . .	22
2.2. Tworzenie Postaci . . . . .	25
2.3. Ruch kamery . . . . .	30
2.3.1. Pozycja kamery . . . . .	30
2.3.2. Mechanika obrotu kamery . . . . .	31
2.3.3. Celownik . . . . .	33
2.4. Fizyka . . . . .	35
2.4.1. Mechanika skakania postaci . . . . .	35
2.5. Animacje . . . . .	37
2.6. Poruszanie bohaterów . . . . .	41
2.7. Poruszanie postaci wrogów . . . . .	42
2.8. Umiejętności Bohaterów . . . . .	45
2.8.1. Wojownik . . . . .	45
2.8.2. Mag . . . . .	46
2.9. Multiplayer . . . . .	50

**3. Bibliografia . . . . .** 54

**Bibliografia . . . . .** 55

# Wprowadzenie

Podczas minionych semestrów na uczelni poznaliśmy wiele nowych technologii, Framework-ów oraz języków programowania. Mieliśmy do czynienia z wieloma ich odmianami, poznaliśmy ich różne zastosowania. Niektóre z nich służyły nam do pisania prostych programów tekstowych, które korzystały tylko i wyłącznie z wiersza poleceń. Z drugiej strony były też obszerne aplikacje, połączone z bazą danych oraz utworzone według wzorców projektowych przy użyciu języków programowania takich jak Ruby, czy Java.

Chcąc rozwijać swoją wiedzę oraz poszerzać swoje umiejętności, jako programiści postanowiliśmy napisać nasz projekt w języku programowania, z którym nie mieliśmy styczności na uczelni. Uważaliśmy, że będzie to dla nas dobra okazja do sprawdzenia tego, czego do tej pory się nauczyliśmy, gdyż nauka programowania, to nie tylko nauka składni danego języka. W wielu przypadkach to głównie nauka logicznego myślenia, wiązania ze sobą faktów, szukania oraz korygowania własnych błędów i wyciągania z nich wniosków.

Szukaliśmy języka programowania, który jest wszechstronny i może być wykorzystany w różnych projektach min. przy budowaniu dużych stron internetowych, gier, serwisów. Po namyśle zdecydowaliśmy się na język C Sharp. Jest on zorientowany obiektowo, dzięki temu mogliśmy wykorzystać nasze doświadczenie min. z języka Java. Ma on bardzo szerokie zastosowanie, może być wykorzystany do tworzenia aplikacji desktopowych, Windows Service, dzięki niemu możemy również tworzyć rozbudowane strony internetowe przy użyciu technologii ASP.NET opartej na .NET Framework. Wybierając jednak nowy język programowania, chcieliśmy również skorzystać z technologii, która umożliwi nam przetestowanie naszych umiejętności zupełnie na innej płaszczyźnie.

Aktualnie na świecie miliony osób grają w gry przeróżnego typu, a my należymy do tej bardzo licznej społeczności. Do tej pory jednak nie mieliśmy

przed sobą zadania, by stworzyć jedną z nich od podstaw. Chcieliśmy podjąć się tego wyzwania i za silnik graficzny obraliśmy Unity, który współpracuje z językiem C Sharp, a do tego obsługuje aż 22 platformy sprzętowe i co raz częściej wykorzystywany jest również w przeglądarkach internetowych oraz wirtualnej rzeczywistości. Wiedzieliśmy, że będzie dla nas wyzwanie, ale zarazem wielki krok w początkach naszej kariery, jako młodych programistów.

Unity daje możliwość tworzenia gry bez wdrażania się w szczegółowe techniczne dotyczące renderowania obrazu i obsługi różnych kart graficznych, obliczania skomplikowanych równań fizycznych czy korzystania z protokołów sieciowych. Tworzenie gier nie wymaga już niskopoziomowego kodu, stało się przyjaźniejsze i bardziej dostępne dla programistów chcących skupić się na samej mechanice gry.

Skloniło nas to do stworzenia nieco odmiennego projektu, w którym rozgrywka polega na wspólnym pokonywaniu przeszkód i odkrywaniu dalszego ciągu tej samej historii, a gracze zamknięci zostają we wspólnej ramie czasowej. Dotąd gry tego typu występowały głównie w postaci gier jednoosobowych, takich jak Far Cry, Tomb Raider, czy Grand Theft Auto.

W odróżnieniu od innych gier wieloosobowych takich jak np. Diablo 3, gdzie całą fabułę możemy ukończyć sami, w The Last Trinity położyliśmy główny nacisk na współpracę. Do ukończenia rozgrywki niezbędne są wszystkie postaci, które sterowane są przez różnych graczy.

Aktualnie gry Multiplayer opierają się głównie na mechanice zbierania co raz lepszego ekwipunku, jak np. w World of Warcraft. Często doprowadza to również do wielu konfliktów pomiędzy graczami. W naszej stawiamy na wspólną zabawę, nie będzie w niej ciągłego zbierania ekwipunku, a jedynie pokonywanie rozmaitych i bardziej skomplikowanych przeszkód razem ze swoją drużyną.

Początkowym założeniem naszej pracy było utworzenie gry wielooso-

bowej, której świat jednocześnie będzie mogła eksplorować trójka graczy. Każdy z nich sterowałby inną, unikatową postacią. Gra miała posiadać przykładowe misje oraz stanowić fundament do dalszego jej rozwoju na tle fabularnym. Wszystkie powyższe założenia udało nam się zrealizować.

## **ROZDZIAŁ 1**

# **Projektowanie oraz przygotowanie środowiska**

## 1.1. Projekt Terenu

Projektowanie naszego terenu zaczęliśmy od lokacji startowej, czyli miasta, w którym rozpoczyna się gra. Stworzyliśmy spustoszone miasto, które miało oddawać klimat, w jakim został osadzony cały projekt.



Rysunek 1.1: Początkowa lokacja na scenie, w której bohaterowie rozpoczynają przygodę

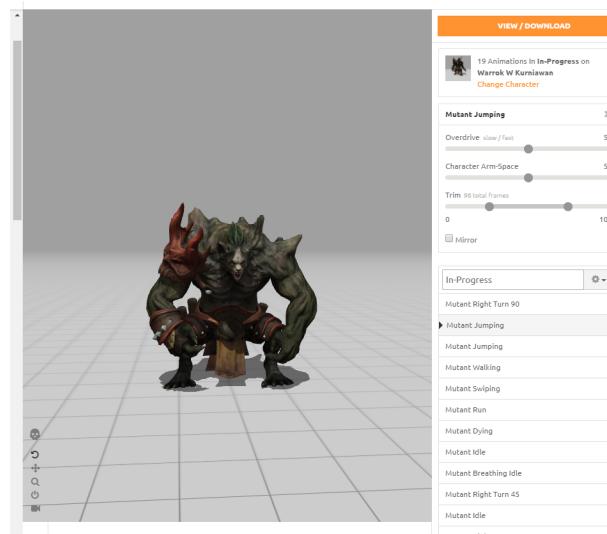
Na początku rozgrywki napotkamy kilku przeciwników oraz przerażoną kobietę, która wprowadzi nas w świat gry. Postaci kierowane przez graczy posiadają określoną rolę, dlatego dalsza część mapy została zaprojektowana z myślą o specjalnych umiejętnościach każdego z nich. Po drodze możemy spotkać niezliczone grupy przeciwników oraz przeszkody w postaci ukształtowania terenu, które możemy pokonać tylko przy pomocy jednego z bohaterów. Przez resztę rozgrywki poruszamy się krętą ścieżką, która prowadzi do oazy. Jest to płaski otwarty teren, na którym bohaterowie będą mogli stoczyć ostateczną walkę ze źródłem swoich problemów.



Rysunek 1.2: Przykładowa przeszkoda w postaci ukształtowania terenu

## 1.2. Projekt Postaci

Postaci zostały utworzone przy pomocy strony <https://www.mixamo.com>. Wyбралиśmy modele, w których przeważają ciemne kolory, a ich charakterystyczna współpraca z mrocznym klimatem otoczenia. Następnie do każdej postaci zostały dopasowane animacje, które mogliśmy wykorzystać podczas procesu implementacji.



Rysunek 1.3: Projekt głównego przeciwnika

Ostatnim krokiem było wybranie szczegółów takich jak ilość klatek na sekundę (FPS) z jaką mają być wyświetlane animacje. Po wykonaniu powyższych czynności pakiet postaci mógł zostać pobrany, a następnie dołączony do projektu.

### 1.3. Projekt systemu mulitplayer

Do utworzenia systemu Multiplayer został użyty Framework Photon Unity 3D Networking często nazywany PUN. Photon udostępnia również chmurę, na której znajduje się nasza aplikacja, dzięki temu nie jest wymagane użycie innego, zewnętrznego serwera.

The image displays two screenshots of the Photon dashboard interface. Both screens show a summary of resource usage and plan details.

**Top Screenshot (App ID: apodlawski@gmail.com):**

- Plan:** 20 CCU
- Peak Current Month:** 0 CCU
- Peak Previous Month:** 0 CCU
- Rejected Peers:** 0

Buttons at the bottom: Analyze, Manage, Change CCU, Add Coupon / PUN+

**Bottom Screenshot (App ID: 3dcf15cb-8...):**

- Plan:** 20 CCU
- Peak Current Month:** 0 CCU
- Peak Previous Month:** 0 CCU
- Rejected Peers:** 0

Buttons at the bottom: Analyze, Manage, Change CCU, Add Coupon / PUN+

Rysunek 1.4: Serwery udostępniane przez framework Photon, na których znajdują się nasze środowiska testowe oraz produkcyjne

Gracze po włączeniu gry automatycznie dołączają do rozgrywki. Maksymalnie do jednej rozgrywki może połączyć się trzech graczy. Jeżeli do serwera dołączy czwarty gracz, zostanie utworzony kolejny pokój gry, a wszyscy kolejni użytkownicy będą mogli się z nim połączyć, dopóki nie zostanie osiągnięty limit osób. Każda postać jest unikatowa i poszczególny gracz może sterować tylko jedną z nich. Jeśli jeden gracz opuści grę, kolejny po dołączeniu zajmie jego miejsce.

## 1.4. Instalacja niezbędnych narzędzi

Pierwszym krokiem było zainstalowanie Unity – silnika, na którym utworzona została nasza gra.

Do tworzenia oraz edycji skryptów, używaliśmy dwóch środowisk, Visual Studio IDE oraz Visual Studio Code.

Dla usprawnienia wspólnej pracy użyliśmy rozproszonego systemu kontroli wersji Git. Korzystaliśmy zarówno z programu używając wiersza poleceń jak i aplikacji GitHub Desktop.

Podczas pracy nad naszym projektem często działałyśmy na tych samych, obszernych plikach np. w przypadku edycji mapy, po której poruszają się gracze. Przez to podczas łączenia naszych zmian często dochodziło do konfliktów w kodzie, których system kontroli wersji nie mógł sam rozwiązać. W tym wypadku niezbędne okazało się wbudowane w silnik narzędzie **UnityYAMLMerge**, które znacznie usprawniło łączenie dwóch kopii, bez konieczności manualnego rozwiązywania konfliktów. Do skorzystania z niego należało odpowiednio skonfigurować plik *.gitconfig*, dzięki czemu aby automatycznie naprawić konflikty, należało uruchomić z linii poleceń komendę **git mergetool**.

---

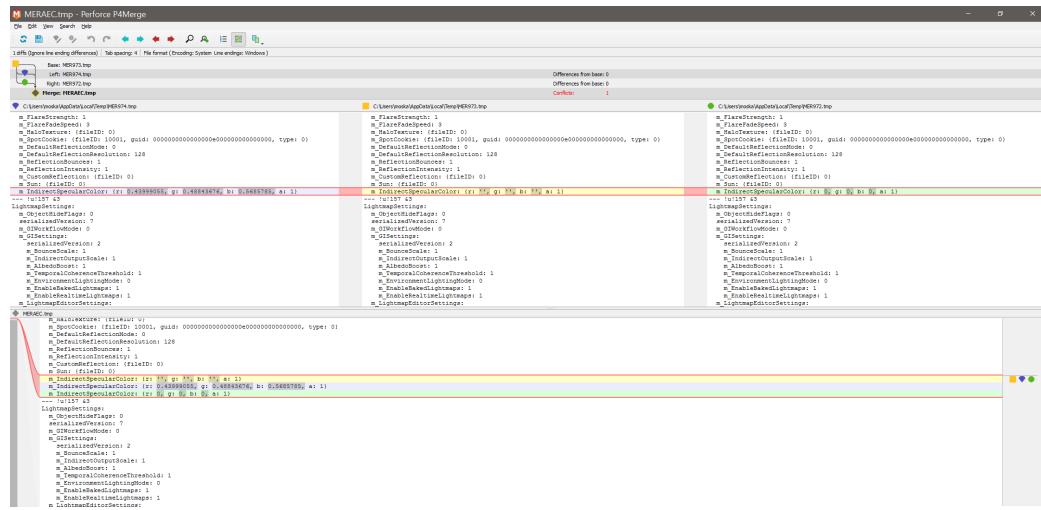
```
[merge]
    tool = unityyamlmerge
[mergetool "unityyamlmerge"]
    cmd = 'C:/Program
        Files/Unity/Editor/Data/Tools/UnityYAMLMerge.exe'
    merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

---

Listing 1.1: Zawartość pliku *.gitconfig* po konfiguracji narzędzia UnityYAMLMerge

UnityYAMLMerge natomiast korzysta domyślnie z darmowej aplikacji

**Perforce P4Merge**, która w przypadku problemów z automatycznym rozwiązyaniem konfliktów, pozwala na ręczne wybranie prawidłowej wersji kodu spośród dwóch konfliktujących (np. w wypadku, gdy podczas pracy nad projektem, bazując na tym samym kodzie, oboje przesunęliśmy ten sam obiekt w różne strony, możemy wybrać jego ostateczną pozycję).



Rysunek 1.5: Ręczne rozwiązywanie konfliktów w projekcie gry

## **ROZDZIAŁ 2**

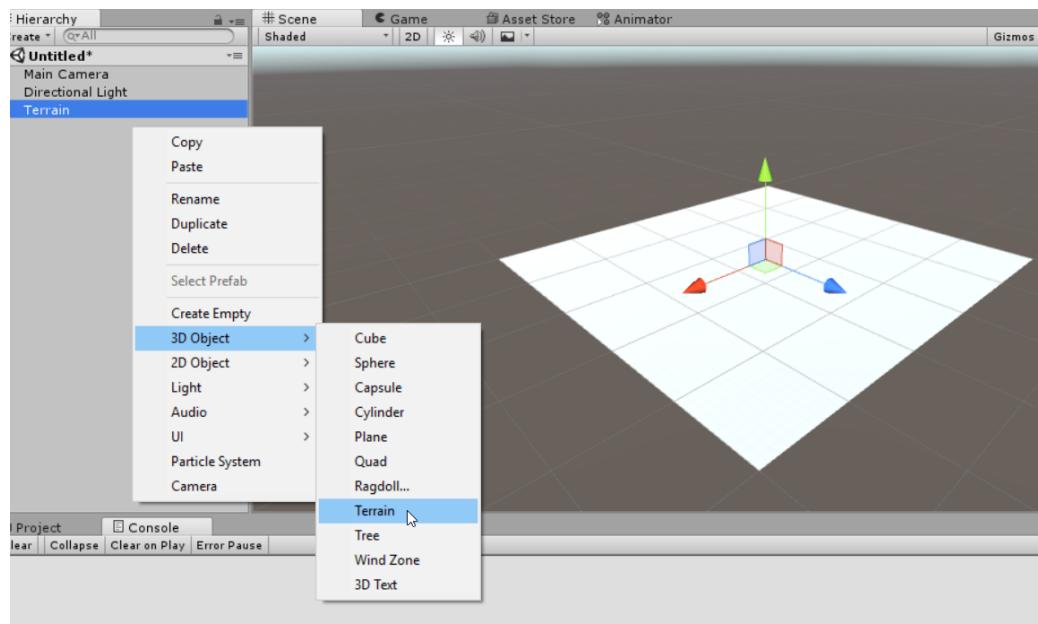
# **Implementacja**

## 2.1. Budowa terenu

Budowa podstawowej mapy gry jest stosunkowo nieskomplikowanym procesem, dzięki czemu mogliśmy się tutaj skupić głównie na projektowaniu dróg, rozłożeniu obiektów i kształtowaniu mapy na potrzeby rozgrywki.

### 2.1.1. Przygotowanie terenu

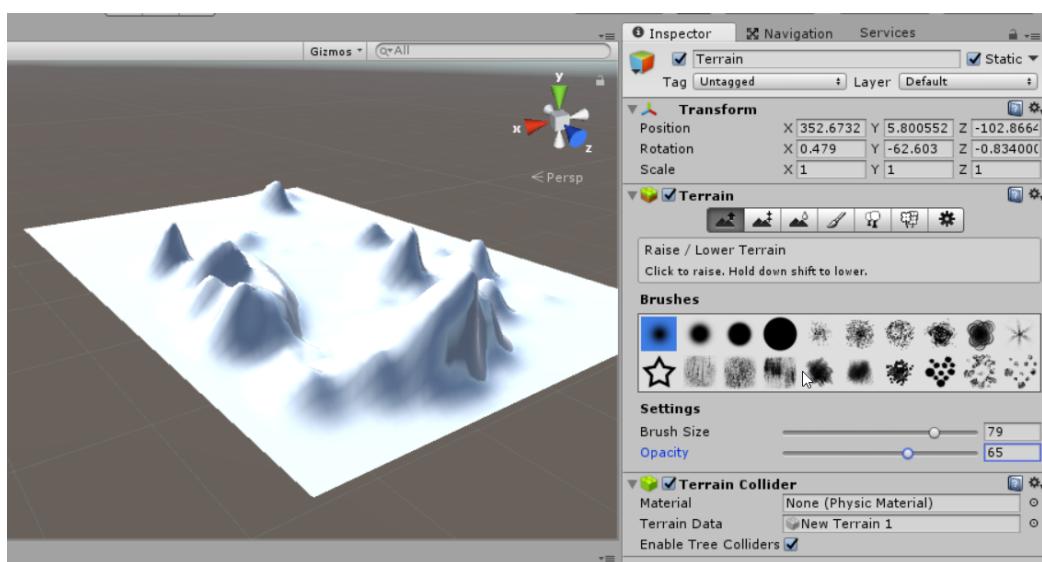
Do przygotowania terenu został użyty specjalnie do tego przeznaczony typ obiektu 3D o nazwie **Terrain**. Po utworzeniu takiego obiektu w hierarchii obiektów na ekranie ukaże się nieoteksturowana płaszczyzna.



Rysunek 2.1: Płaszczyzna terenu

### 2.1.2. Formowanie kształtu

Na tak przygotowanej płaszczyźnie uformowane zostały nierówności przy użyciu palety narzędzi terenu. Używając opcji **Raise / Lower Terrain** utworzone zostały wypiętrzenia nadające kształt mapie gry. Charakter wypiętrzeń dostosowany został używając odpowiedniego pędzla z panelu **Brushes**, natomiast promień zniekształceń oraz siła efektu za pomocą parametrów kolejno **Brush Size** oraz **Opacity**.



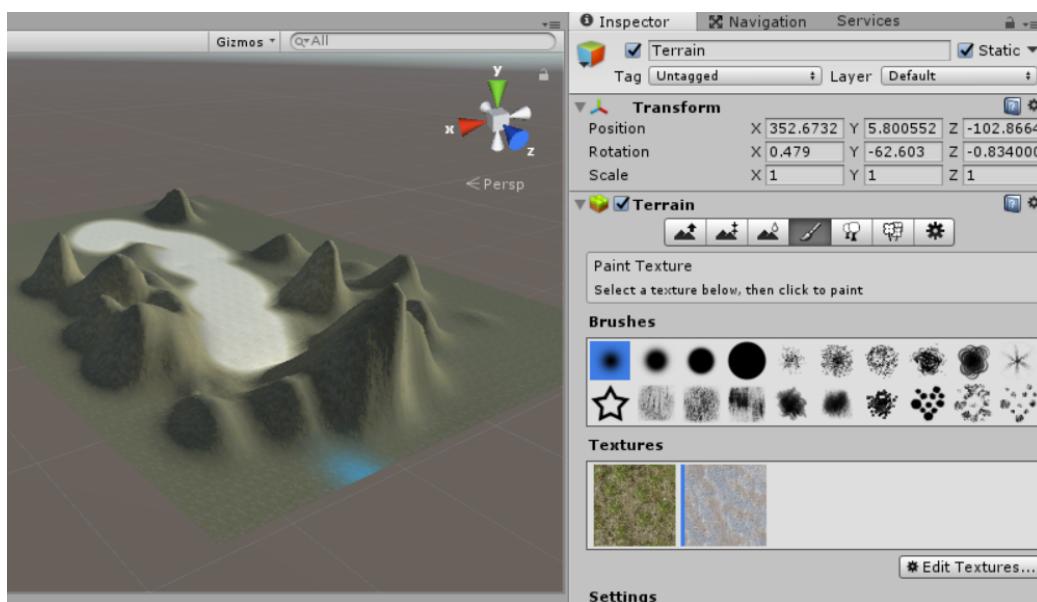
Rysunek 2.2: Modelowanie nierówności terenu

### 2.1.3. Nakładanie tekstur

Kolejnym krokiem było utworzenie tekstury służącej do nadaniu naszemu terenowi koloru oraz faktury. Do tego celu pobrane zostały odpowiednie tekstury z wbudowanego sklepu assetów **Asset Store**. Assety są rodzajem pakietów zawierających różnorakie obiekty, skrypty oraz tekstury, dostępne do pobrania z serwerów Unity.

Po pobraniu odpowiedniej tekstury trawy, zostaje ona zainportowana do folderu **Assets** znajdującego się w głównym katalogu projektu.

Po wybraniu narzędzia **Paint Texture** ukazuje się panel **Textures** pozwalający na skonfigurowanie używanej przez narzędzie tekstury. Znajduje się tam przycisk **Edit Textures...** po kliknięciu którego otwiera się okno konfiguracyjne tekstury pozwalające na wybór tekstury podstawowej oraz tekstury przechowującej dane o chropowatościach. Po skonfigurowaniu tekstur i nałożeniu ich na teren, całość prezentuje się następująco.

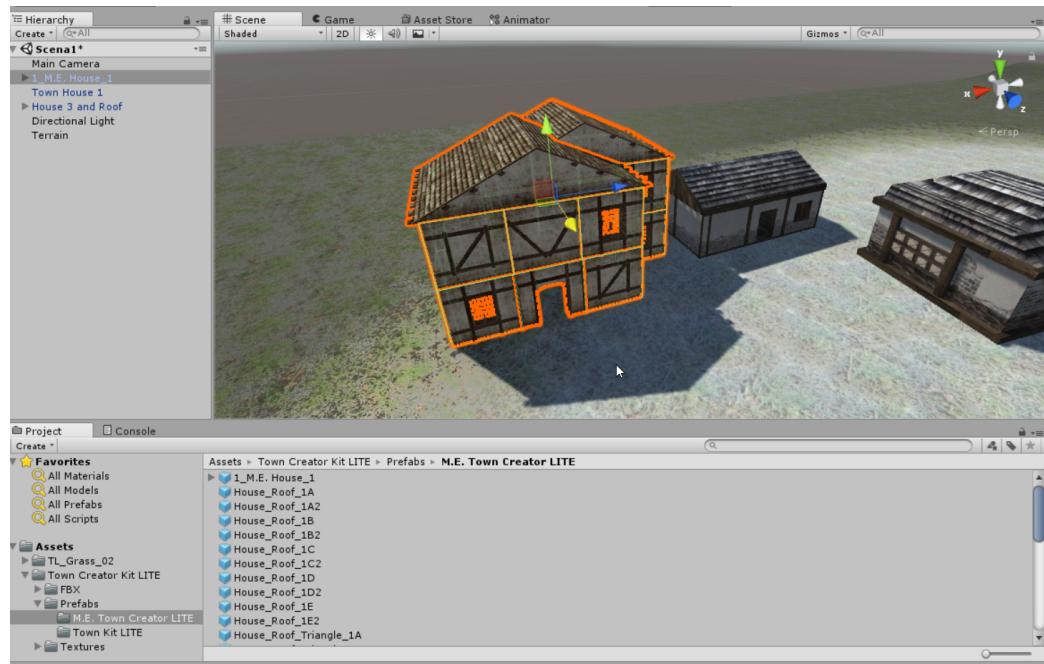


Rysunek 2.3: Nakładanie tekstur na mapę terenu

#### 2.1.4. Rozmieszczanie obiektów

Kolejnym krokiem w tworzeniu świata gry było umieszczenie na mapie spróbowanej wcześniej obiektów (tzw. *Prefabs*, o tym później). Wszystkie obiekty użyte w pracy są dostępne za darmo w bibliotece obiektów Unity (**Asset Store**). Po pobraniu, obiekty znajdują się w podfolderze **Prefabs** zainstalowanej paczki. Obiekty umieszcza się na mapie metodą przeciągnij-

upuść, dostosowując ich koordynaty używając narzędzi transformacji dostępnych w pasku narzędzi znajdującym się w górnej części okna edytora.



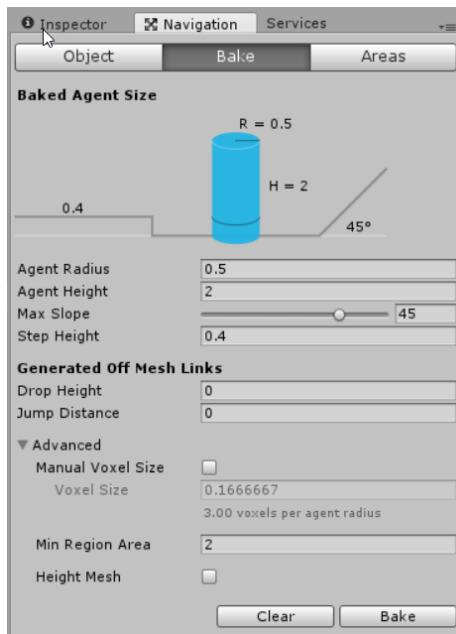
Rysunek 2.4: Umieszczanie obiektów na mapie

### 2.1.5. Oddziaływanie terenu na inne obiekty

Kluczowym elementem tworzenia mapy świata gry są takie elementy jak oddziaływanie na postacie ograniczeń nachylenia terenu typu wzgórza, drzewa, czy budynki, uniemożliwiających dostanie się w niektóre miejsca. Unity w celu uproszczenia obliczeń posiada możliwość wygenerowania uproszczonej mapy dróg (tzw. *NavMesh*) na bazie modelu terenu, pozwalającej na dynamiczne omijanie przeszkód przez wroga (o tym później w sekcji 2.7 na stronie 42).

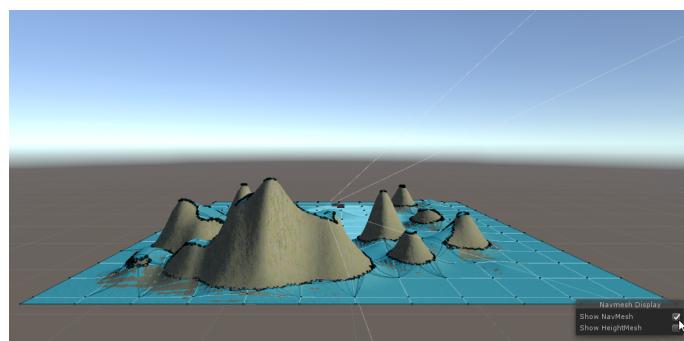
Aby utworzyć taką mapę, należy przejść do zakładki **Navigation**, gdzie w panelu **Bake** znajduje się lista parametrów dotyczących maksymalne-

go kąta nachylenia terenu, czy maksymalnej wysokości uskoku, którą obiekty sterowane przez komputer mogą pokonać.



Rysunek 2.5: Parametry mapy dróg

Po ustaleniu parametrów i kliknięciu przycisku **Bake**, mapa zostaje wygenerowana, natomiast w oknie widoku sceny, obszary dostępne do przemierzania oznaczone zostają niebieskim kolorem. Operację można powtarzać do uzyskania optymalnych efektów.



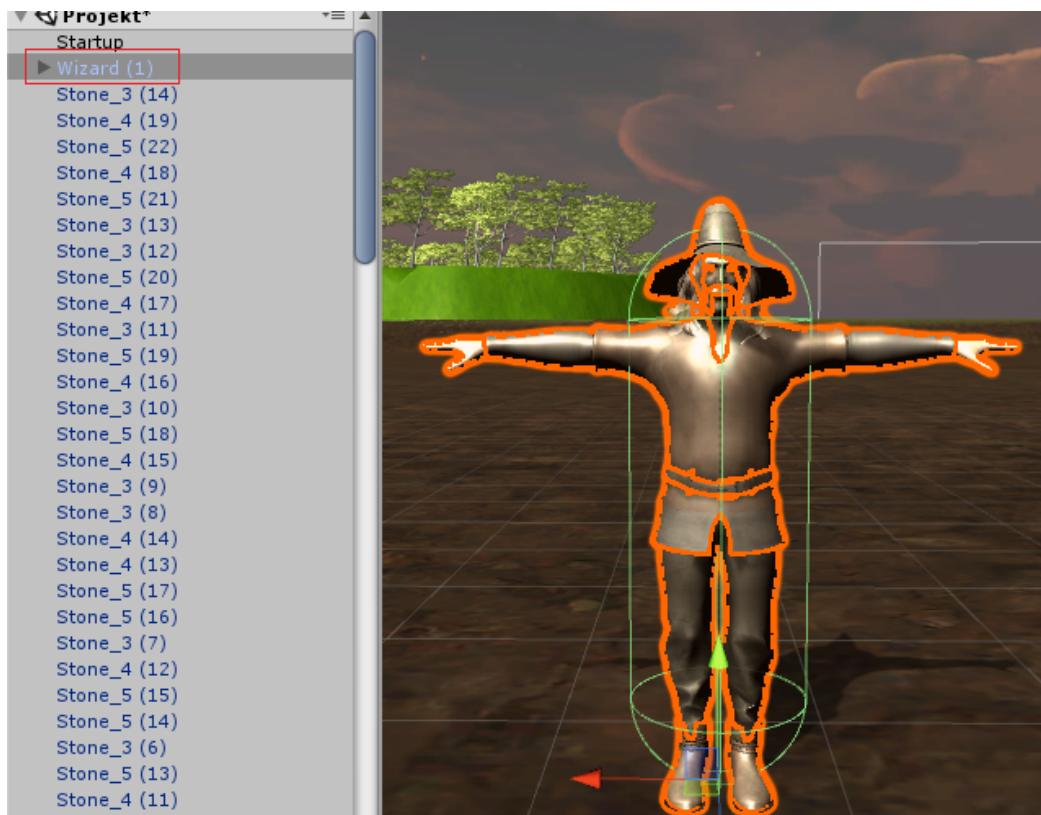
Rysunek 2.6: Poprawnie wygenerowany *NavMesh*

Tak przygotowana mapa posłużyła nam do projektowania dalszej części gry.

## 2.2. Tworzenie Postaci

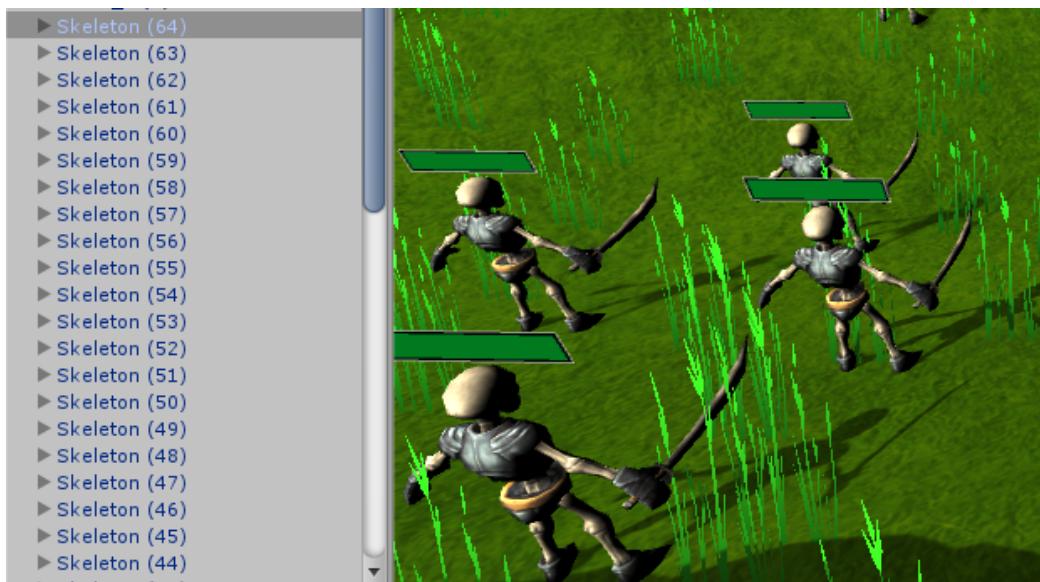
Po zaprojektowaniu postaci dołączymy je do naszego folderu z projektem. Żeby nasza postać pojawiła się na mapie, musimy ją przeciągnąć na wcześniej utworzoną scenę. Tak oto otrzymaliśmy nieruchomy model naszego maga. Aby upewnić się, że jest on poprawnie ulokowany, możemy sprawdzić zakładkę Hierarchy, gdzie znajdziemy listę wszystkich dostępnych obiektów na mapie.

W naszym wypadku nie możemy jednak umieścić postaci od razu w rozgrywce, gdyż nie będzie mogła ona być poprawnie sterowana przez danego gracza. Żeby system Multiplayer działał sprawnie, musimy stworzyć tak zwanego „Prefaba”, czyli obiekt w grze, który ma pewne właściwości, może zostać wielokrotnie użyty, lub pojawić się np. w momencie zalogowania gracza do gry. W tym celu musimy przeciągnąć nasz obiekt, który ma zamienić się w typ Prefab z listy Hierarchy, do naszego folderu (W tym przypadku jest to folder Assets). Teraz w celu ulokowania naszych magów na mapie, wystarczy przeciągnąć obiekt z rozszerzeniem .prefab na naszą scenę. Przy umieszczaniu takich samych obiektów Unity automatycznie będzie dodawało do naszej nazwy numer obiektu np. Wizard (1).



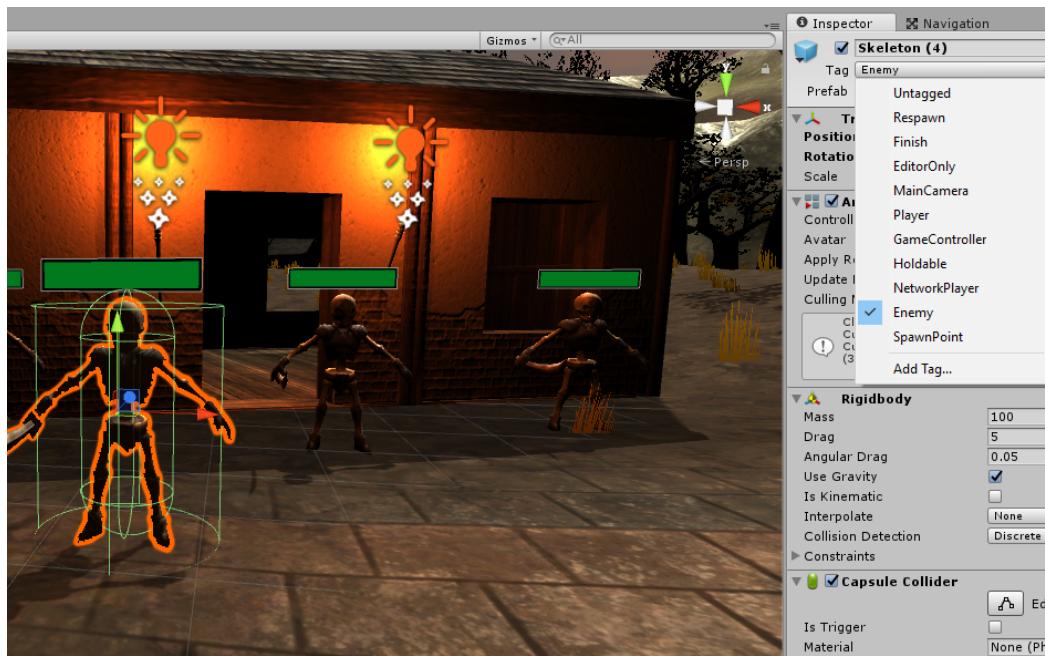
Rysunek 2.7: Ręczne rozwiązywanie konfliktów w projekcie gry

Jest to również bardzo dobre rozwiązanie, gdy w projekcie musimy użyć wielu takich samych obiektów. W naszej grze wykorzystaliśmy to miedzy innymi podczas tworzenia przeciwników dla naszych graczy. Dzięki temu tworząc jeden model wroga, mogliśmy go użyć w wielu miejscach, bez konieczności tworzenia od nowa tego samego szkieletu postaci.



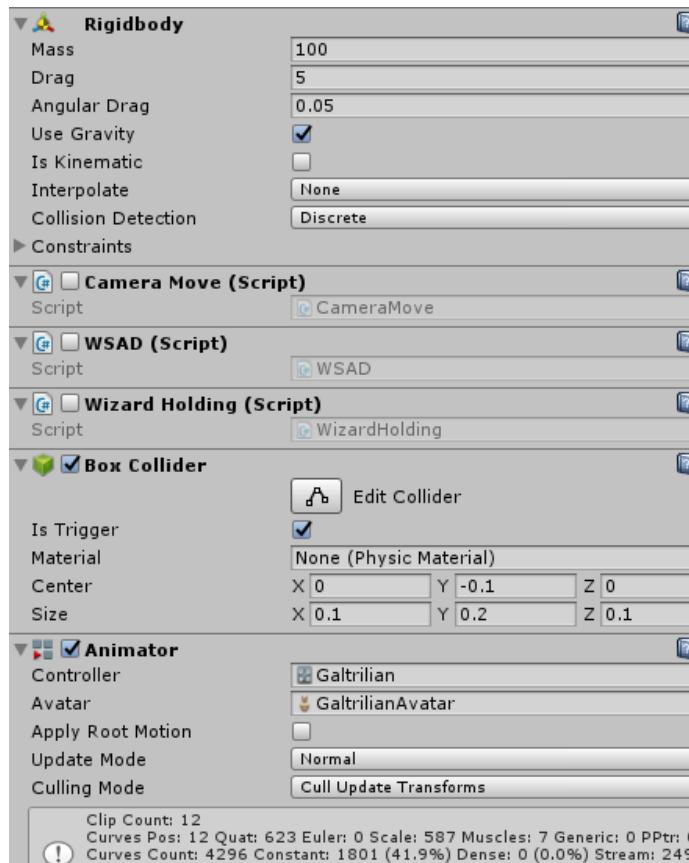
Rysunek 2.8: Armia szkieletów utworzona przy pomocy jednego Prefaba

Każdy obiekt poza nazwą posiada również swój Tag, możemy dzięki temu definiować np. grupę wrogów, przedmiotów o specjalnych właściwościach lub graczy. Na przykładzie naszej gry sprawdziło się to przy wykonywaniu misji. Szkielety w mieście posiadają specjalny Tag, który wyróżnia je od pozostałych. Jeśli zostaną one pokonane, dopiero wtedy kobieta w mieście przekaże nam kolejne informacje.



Rysunek 2.9: Tagi zdefiniowane w naszym projekcie, które możemy przyporządkować do danego obiektu

Bardzo istotne w obiekcie są komponenty. To one odpowiadają za to jak zachowuje się dany obiekt. Wszystkie skrypty odpowiadające za poruszanie się, grawitację oraz inne czynności ulokowane są na obiekcie pod postacią komponentów.



Rysunek 2.10: Przykładowe komponenty postaci maga odpowiadające min. za poruszanie się, kolizje z otoczeniem oraz animacje

Niektóre obiekty mogą składać się z kilku innych obiektów. Dobrym przykładem jest rycerz, który poza modelem postaci, posiada również obiekt tarczy, hełmu oraz miecza. Każdy z tych obiektów posiada również swoje własne komponenty. W przypadku miecza może być to komponent odpowiadający za wykrywanie kolizji z przeciwnikiem i tym samym odbieraniem mu odpowiedniej ilości zdrowia, gdy zostanie trafiony.

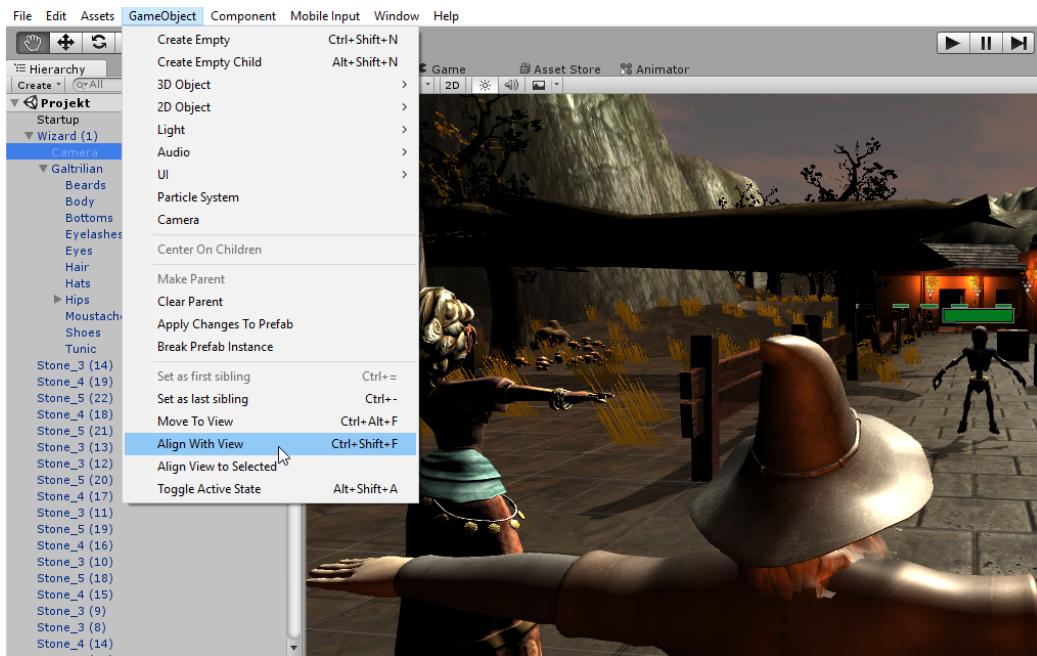
## 2.3. Ruch kamery

### 2.3.1. Pozycja kamery

Widok w grze jest trzecioosobowy, kamera obejmuje zarówno widzialny obszar jak i samego gracza. Pod tym względem jest to gra TPP (*Third Person Perspective*). Przykładowymi grami tego typu są bardzo znane produkcje, takie jak seria *Wiedźmin*, *Tomb Raider*, czy *GTA*.

Aby uzyskać efekt kamery podążającej za graczem, obiekt kamery powinien zostać umieszczony wewnątrz obiektu gracza w hierarchii obiektów. W ten sposób koordynacje kamery będą ustawiane względem gracza, a sama kamera poruszać się będzie i obracać wraz z nadziednym obiektem.

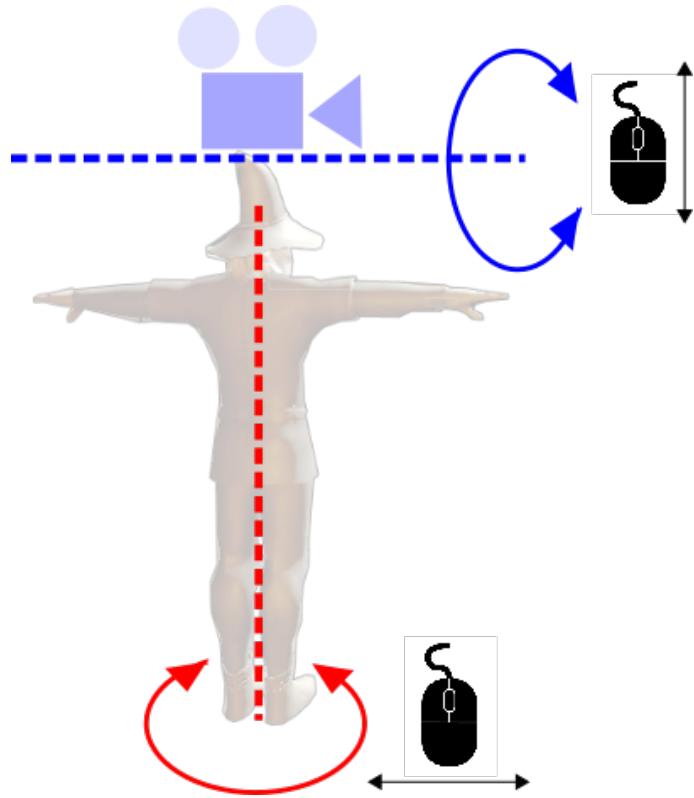
Koordynacje kamery ustawiliśmy na  $(0, 0, 0)$ , w ten sposób kamera jest w środkowym punkcie obiektu nadziednego i wystarczy za pomocą narzędzi transformacji przesunąć ją do oczekiwanej pozycji (tak, aby obejmowała obraz zza pleców postaci). Inną, bardziej poręczną metodą jest odpowiednie dostosowanie widoku sceny i skopiowanie jego koordynatów do zaznaczonej w hierarchii kamery.



Rysunek 2.11: Dostosowanie pozycji kamery gracza względem aktualnego widoku sceny

### 2.3.2. Mechanika obrotu kamery

Kolejnym etapem jest oprogramowanie ruchu kamery za pomocą ruchów myszy. Mysz jest używana również do obrotu postacią. Jako, że kamera ?przyklejona? jest do postaci, aby uzyskać efekt rozglądania się, modyfikujemy jedynie jej obrót w pionie, natomiast ruch myszy w poziomie obraca zarówno całą postać wraz z kamerą.



Rysunek 2.12: Schemat działania systemu sterowania kamerą

---

```

if (!photonView.isMine) return;
if (!stopCamera)
{
    Camera.main.transform.Rotate(new
        Vector3(-Input.GetAxis("Mouse Y") *
        rotationSensitivity * Time.deltaTime, 0, 0));
    rb.transform.Rotate(new Vector3(0,
        Input.GetAxis("Mouse X") * rotationSensitivity *
        Time.deltaTime, 0));
}

```

---

Listing 2.1: Obrót kamery za pomocą myszy

Obrót kamery polega na dodaniu wektora obrotu do odpowiedniej osi kamery, natomiast wielkość wektora określona jest przez wartość chwilowego

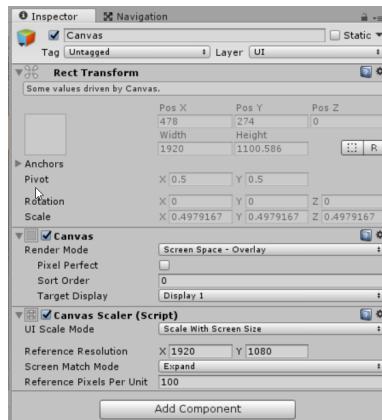
przesunięcia myszy, skalowanego parametrem *rotationSensitivity* w celu określenia czułości myszki. Parametr *Time.deltaTime* zwraca czas od ostatniej wyrenderowanej klatki, co pozwala na uzyskanie jednolitej czułości myszki bez względu na ilość generowanych klatek na sekundę.

### 2.3.3. Celownik

Aby ułatwić celowanie w przeciwników oraz interakcję z otoczeniem umieścimy w grze celownik, który w dalszym etapie tworzenia gry zmienia kolor informując o możliwej interakcji.

Celownik jest obiektem 2D nałożonym na ekran w formie *Sprite'a*. *Sprite'y* to 2-wymiarowe, wcześniej przygotowane obiekty graficzne. W naszym wypadku posłużyliśmy się bezstratnym formatem PNG.

Do sceny został dodany obiekt, na który nałożyliśmy komponenty **Canvas** oraz **CanvasScaler**. Nakładają one na ekran dwuwymiarowe płótno oraz pozwalają na skalowanie go względem podanej rozdzielczości referencyjnej. Wewnątrz takiego płotna można umieszczać obrazy, elementy interfejsu oraz różnego rodzaju wskaźniki (np. Pasek postępu życia bohatera).



Rysunek 2.13: Konfiguracja płynna 2D do wyświetlania interfejsu gry

Wewnątrz płótna został umieszczony obrazek (komponent **Image**) celownika, wyrównany do środka ekranu.



Rysunek 2.14: Gotowy celownik gracza

## 2.4. Fizyka

Ważnym elementem gry jest fizyka, aby postaci mogły poruszać się, podskaikiwać i reagować na kolizje.

Podstawowym komponentem zapewniającym obliczenia fizyczne jest **Rigidbody**. Komponent ten przyjmuje parametry takie jak wartość masy, tarcia, czy wyłączenie grawitacji.

Aby zdarzenia fizyczne mogły mieć miejsce, na elementy otoczenia oraz samą postać gracza nałożone powinny zostać komponenty **Collider**. Są to obszary określające granice kolizji. Mogą one mieć kształt kapsuły, sześcianu, bądź uproczzonego modelu samego obiektu. Pozwalają one silnikowi gry na przyspieszenie obliczeń i płynne reakcje na zdarzenia kolizji.



Rysunek 2.15: Przykład – collider mapy terenu

W przypadku postaci poruszających się po mapie, zablokowane zostały obroty we wszystkich osiach (parametr **Freeze Rotation** komponentu Rigidbody), dzięki czemu nie przewracają się one na bok przy nierównościach terenu i kolizjach z otoczeniem. Zdarzenia fizyczne wyliczane są jedynie w pionie (spadanie, skok itp.).

### 2.4.1. Mechanika skakania postaci

Mając już zaimplementowaną fizykę, dodaliśmy możliwość skoku. Utworzona została metoda *jump()*, która dodaje do zdarzeń fizycznych naszej postaci siłę w kierunku pionowym o wektorze ustalonym parametrem *jumpPower*. Natomiast na podstawie kolizji z terenem ustalone jest, czy postać stoi na

ziemi, dzięki czemu nie można wykonać kolejnego skoku będąc już w powietrzu.

---

```
void Start()
{
    rb = GetComponent<Rigidbody>();
    anim = GetComponent<Animator>();
}

void jump()
{
    if (onGround)
    {
        onGround = false;
        rb.AddRelativeForce(new Vector3(0, jumpPower, 0));
        anim.SetTrigger("Jump");
    }
}

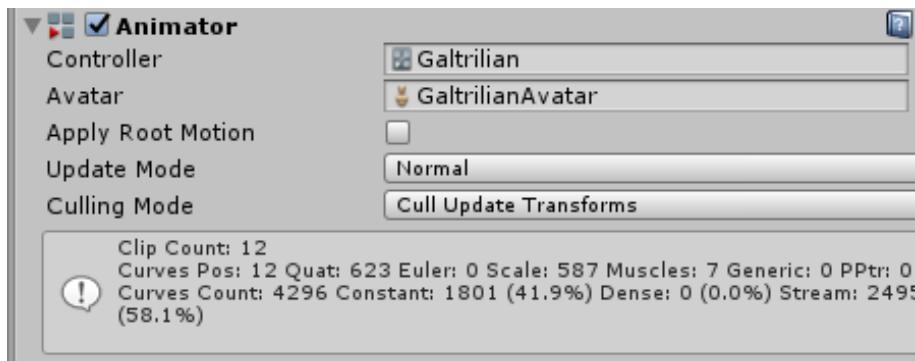
void OnTriggerEnter(Collider other)
{
    onGround = true;
}
```

---

Listing 2.2: Fragment algorytmu skoku postaci

## 2.5. Animacje

Jednym z głównych komponentów na obiekcie jest Animator. To dzięki niemu postacie odpowiednio poruszają się podczas zmieniania swojej pozycji lub ataku. Animator jest odpowiedzialny za wszystkie animacje odgrywane przez nasze postaci, zarówno te sterowane przez komputer, jak i graczy.

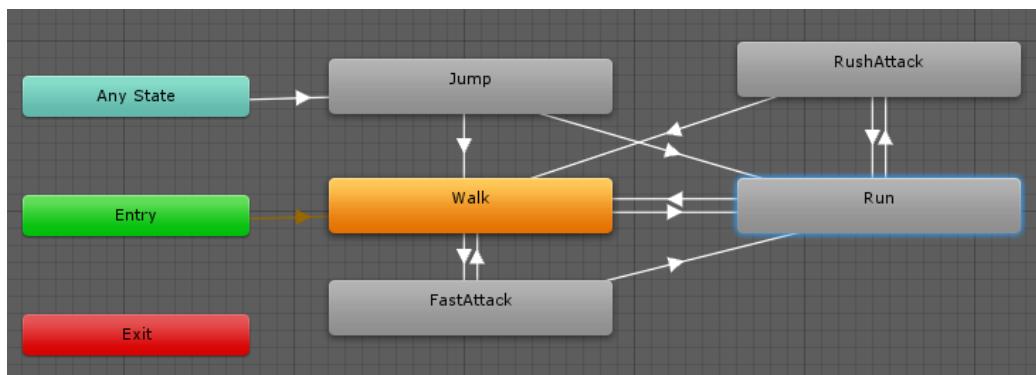


Rysunek 2.16: *Animator*

Składa się on z dwóch ważnych elementów. Jednym z nich jest kontroler, który odpowiada za wszystkie przejścia pomiędzy naszymi animacjami. Odbiera on parametry, które na bieżąco aktualizowane są przez odpowiednie skrypty zaimplementowane dla danego obiektu. Wykrywają one wszystkie sygnały z urządzeń wejścia, odbierają informacje od innych graczy, postaci niekontrolowanych przez użytkowników oraz otoczenia.

Ważną kwestią jest odpowiednie zaprogramowanie kontrolera, gdyż na jego podstawie postać będzie odpowiednio reagować i przemieszczać się. Jeśli jeden z warunków zawiedzie nasz gracz może pozostać w fazie ataku, która nigdy się nie skończy, przez co będzie blokowała wszystkie inne ruchy dla danej postaci.

Podstawą w kontrolerze są stany, a ich głównym parametrem jest Motion. Zawiera informacje o animacji, która ma się wykonać, gdy postać znajduje się w określonym momencie.

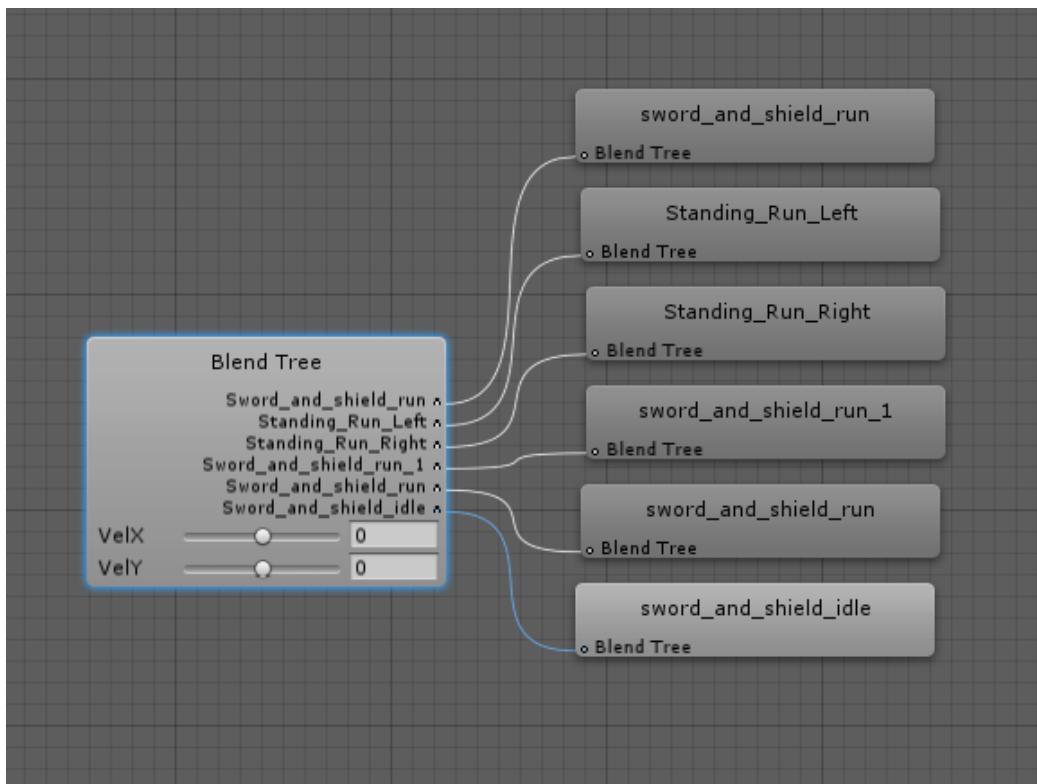


Rysunek 2.17: Stany odpowiadające za animacje wojownika

Żeby postać mogła przejść z jednego stanu na drugi niezbędne są transakcje. To w nich ustalamy zmianę animacji względem poszczególnych parametrów.

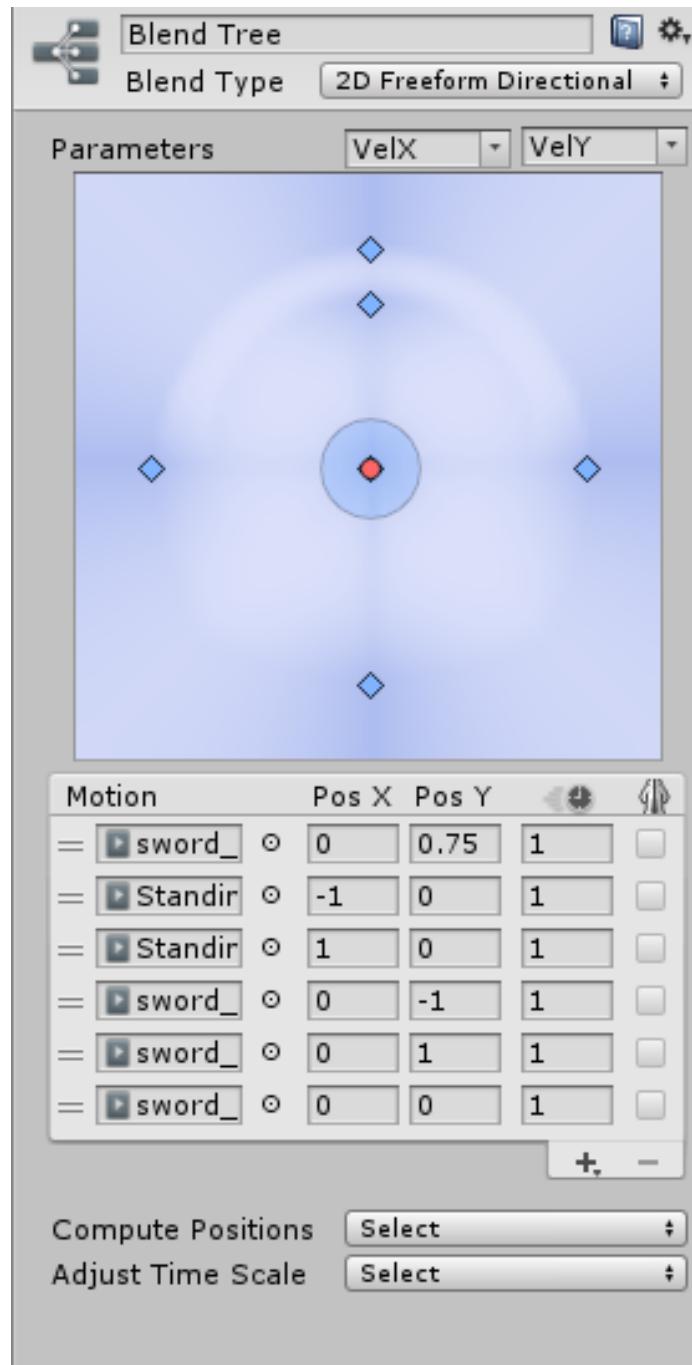
Zwykłe stany posiadają jednak jedną znaczącą wadę, brak płynnych przejść pomiędzy poszczególnymi animacjami. Postać w jednym momencie po prostu zmienia animację na kolejną, co bardzo psuje efekt wizualny. Aby uniknąć tego w naszym projekcie użyliśmy bardziej zaawansowanej opcji – Blend Tree.

Blend Tree to bardziej rozbudowane stany, które pozwalają na płynne przejścia pomiędzy animacjami oraz łączenie ich. Dzięki czemu nasza postać może połączyć np. bieg w przód z poruszaniem się na boki. Daje to również odczucie, jakby nasza postać posiadała o wiele więcej animacji.



Rysunek 2.18: Stan "Run" zaimplementowany jako Blend Tree

W Blend Tree możemy wykorzystać również stopniowe zwiększenie się naszych parametrów. Najlepszy efekt wspomnianej funkcjonalności możemy zaobserwować w momencie, gdy nasz użytkownik zdecyduje się na sterowanie kontrolerem z gałkami analogowymi, które są czułe na siłę nacisku. Im bardziej gracz będzie przesuwał analog w przód, tym bardziej nasza postać będzie się pochylać i przechodzić płynnie do kolejnej animacji.



Rysunek 2.19: Przejścia pomiędzy animacjami biegu w Blend Tree, na podstawie parametrów X oraz Y

## 2.6. Poruszanie bohaterów

Poruszanie bohaterów działa na podobnej zasadzie jak mechanika skoku. Do zdarzeń fizycznych bohatera dodawana jest siła o wektorze skierowanym w kierunku określonym przez wciskany klawisz (W, S, A lub D).

W przypadku chodzenia, animacja zależna jest od kierunku, w którym porusza się postać (cofanie, chodzenie bokiem).

Dodatkowym elementem mechaniki poruszania jest bieg, który aktywowany jest poprzez wcisnięcie lewego klawisza Shift.

---

```
run = Input.GetKey(KeyCode.LeftShift);
anim.SetBool("Sprint", run);
float speed = run ? runSpeed : walkSpeed;

...
if (Input.GetKey(KeyCode.W))
{
    rb.AddRelativeForce(Vector3.forward * speed,
        ForceMode.VelocityChange);
}
```

---

Listing 2.3: Fragment kodu mechaniki biegania – reakcja na wcisnięcie klawisza **W**

## 2.7. Poruszanie postaci wrogów

Poruszanie wrogów odbywa się autonomicznie. Droga jest wyliczana, a koordynacje postaci zmieniane są zgodnie z wyliczoną drogą, dzięki czemu postacie wroga przemieszczają się płynnie w kierunku zdefiniowanego celu.

Unity posiada mechanizm wyznaczania dróg na mapie terenu (więcej informacji na ten temat w dziale 2.1 na stronie 19). Aby móc z niego skorzystać, dodaliśmy do postaci wroga agenta nawigacji **NavAgent**. Jest to komponent zajmujący się obliczeniami drogi i zarządzaniem przemieszczeniem i obrotem postaci w czasie. Podstawowym parametrem jest 3-wymiarowy wektor określający docelową pozycję, silnik gry zajmuje się wyliczeniem trasy prowadzącej do punktu.

Mechanika wrogów polega na wyszukiwaniu najbliższej znajdującej się postaci gracza, a następnie ustaleniu jej jako cel podążania wroga. W momencie, gdy wrog znajduje się odpowiednio blisko, rozpoczyna atak.

---

```
GameObject findNearestPlayer(float maxDistance)
{
    GameObject[] players =
        GameObject.FindGameObjectsWithTag("Player");
    GameObject nearestPlayer = null;
    float distance = maxDistance;
    foreach (GameObject player in players)
    {
        float distanceFromEnemy =
            Vector3.Distance(player.transform.position,
                transform.position);
        if (distanceFromEnemy < distance)
        {
            distance = distanceFromEnemy;
            nearestPlayer = player;
        }
    }
}
```

```
    }  
  
    return nearestPlayer;  
}
```

Listing 2.4: Algorytm zwracający wskaźnik do obiektu znajdującego się najbliżej, przy określonym maksymalnym promieniu poszukiwań



Rysunek 2.20: Obszary interakcji postaci wroga

W przypadku, gdy wróg „widzi” postać gracza, jest w jego stronę stale odwrócony. Służy do tego algorytm, który najpierw wylicza kierunek w postaci wektora wartości w skali 0.0f..1.0f w przestrzeni 3D, a następnie zamienia wektor na kąt, do którego stopniowo dąży. Daje to efekt płynnego obracania postaci wroga z możliwością dostosowania prędkości obrotu, aby ruchy wyglądały naturalnie, a wróg dawał możliwość np. zajścia go od tyłu.

---

```
private void rotateTowards(Transform target)
{
    Vector3 direction = (target.position -
        transform.position).normalized;
    Quaternion lookRotation =
        Quaternion.LookRotation(direction);
    transform.rotation =
        Quaternion.Slerp(transform.rotation, lookRotation,
        Time.deltaTime * 5);
}
```

---

Listing 2.5: Algorytm obracania postaci wroga w kierunku gracza

## 2.8. Umiejętności Bohaterów

Zgodnie z naszymi założeniami każdy bohater miał być unikalny i odgrywać ważną rolę w całej rozgrywce. Wszystkich nadaliśmy różne umiejętności, a każda z nich jest niezbędna, by wspólnie ukończyć grę.

### 2.8.1. Wojownik

Wojownik słynie z ogromnej siły, jego zadaniem jest rozprawianie się z jak największą ilością przeciwników. Do jego postaci napisaliśmy skrypt, który pobiera listę obiektów znajdujących się w jego obrębie a następnie każdemu z nich odbiera życie. Skrypty dodawane są do postaci na podobnej zasadzie, jak **Komponenty**. W jednej z klatek animacji uderzenia mieczem ustawiłyśmy tzw. **Trigger**, który uruchamia odpowiednie zdarzenie. Każda z postaci znajdujących się na planszy posiada specjalny **Tag**, jest to dodatkowa informacja o obiekcie w postaci ciągu znaków, w wypadku wroga jest to *Enemy*. Siła rażenia zależna jest od rodzaju ataku (Wojownik posiada 2 rodzaje ataku - standardowy szybki oraz silny, który jednak trwa nieco dłużej).

---

```
// HitEvent is animation triggered method
void HitEvent(int type)
{
    if (!photonView.isMine) return;

    Collider[] hitColliders =
        Physics.OverlapSphere(transform.position,
        attackDistance);
    foreach (var enemyCollider in hitColliders)
    {
        if (enemyCollider.tag == "Enemy")
        {
            float damageValue = 0.0f;
            ...
        }
    }
}
```

```

        switch (type)
    {
        case 0: damageValue = 20.0f; break;
        case 1: damageValue = 50.0f; break;
    }
    enemyCollider.gameObject.GetPhotonView().RPC("damage",
        PhotonTargets.All, damageValue);
}
}

```

---

Listing 2.6: Algorytm zadawania obrażeń wrogom

### 2.8.2. Mag

Mag posiada umiejętność podnoszenia przedmiotów, którą możemy wykorzystać na dwa sposoby. Jednym z nich jest pokonywanie przeszkód, drugim atak. Specjalny skrypt umieszczony w komponentach wykorzystuje celownik, za którym podąża kamera. Gdy najedziemy celownikiem na obiekt o odpowiednim oznaczeniu, możemy go podnieść. Gracz może obracać przedmiot wokół własnej osi, oddalać go od siebie i przybliżać. Przy pomocy odpowiedniego przycisku może również rzucić trzymanym przedmiotem w przeciwnika. Dzięki wykorzystaniu grawitacji i kolizji z danymi przedmiotami możemy w ten sposób zabrać życie naszym przeciwnikom.

Aby odnaleźć interesujący nas obiekt, tworzymy wiązkę (*Ray*), przechodzącą z pozycji kamery przez środek ekranu, gdzie znajduje się celownik, a następnie pobieramy listę wszystkich obiektów, które ta wiązka przecina, przy zadanej maksymalnej odległości. Następnie iterujemy poprzez wszystkie znalezione obiekty, i gdy znajdziemy obiekt posiadający tag *Holdable*, wyłączamy grawitację tego obiektu, wyliczamy odległość od niego, a następnie pobieramy jego wskaźnik do zmiennej globalnej, w celu możliwości manipu-

lowania tym obiektem:

---

```
ray = Camera.main.ScreenPointToRay(new
    Vector3(Screen.width / 2, Screen.height / 2));
hits = Physics.RaycastAll(ray);
foreach (var hit in hits)
{
    if (hit.transform.gameObject.tag == "Holdable" &&
        hit.distance < objectDistanceMax)
    {
        turnSpecialCrosshair(true);

        if (Input.GetMouseButtonDown(0) && holdedThing ==
            null)
        {
            holdedThing = hit.transform.gameObject;
            objectDistance = Mathf.Clamp(hit.distance,
                colliderSize(holdedThing) / 2 +
                objectDistanceMin, objectDistanceMax);
            holdedThing.GetComponent<PhotonView>
                ().RequestOwnership ();
            photonView.RPC("setGravity",
                PhotonTargets.All,
                hit.transform.gameObject.GetComponent<PhotonView>().viewID,
                false);
            break;
        }
    } else
    {
        turnSpecialCrosshair(false);
    }
}
```

---

Listing 2.7: Algorytm wykrywania obiektów w obrębie Maga, które mogą zostać przeniesione

Po wykryciu obiektu, który da się przenieść, zmienia się również kolor wskaźnika (na zasadzie podmiany grafiki celownika). Nie wdrażając się we wszystkie szczegóły (oddalanie, przybliżanie, rzucanie - są to kolejne manipulacje obiektem), fragment kodu przemieszczania obiektów wylicza punkt w przestrzeni 3D oddalony o odpowiednią wartość od widoku kamery, a następnie na tej podstawie wylicza trójwymiarowy wektor prędkości, dzięki czemu przenoszony obiekt dąży w każdej z osi do docelowego punktu:

---

```
// Moving object in space
if (heldedThing != null)
{
    // Get point in front of camera
    var newObjectPosition = ray.GetPoint(objectDistance);

    // Move helded thing to new destination in front of
    // camera
    var heldedThingDestination =
        Vector3.ClampMagnitude((newObjectPosition -
            heldedThing.transform.position) * objectMoveSpeed,
            objectMoveSpeed);
    heldedThing.GetComponent<Rigidbody>().velocity =
        heldedThingDestination;
    ...
}
```

---

Listing 2.8: Algorytm przenoszenia obiektów

Wartość prędkości podążania obiektu za punktem docelowym jest ustawiona z góry i dopasowana tak, aby przenoszony obiekt sprawiał wrażenie delikatnej bezwładności.

Łotr jest przebiegły, potrafi pozyskiwać informacje oraz dostrzega znacznie więcej niż inni. Posiada skrypt, który umożliwia mu rozmowę z innymi osobami, która udzielają mu cennych informacji, gdy tylko znajdzie się

w odpowiedniej odległości. Może też wyszukiwać przedmioty, które pozostają niewidoczne dla pozostałych graczy.

## 2.9. Multiplayer

Ostatnim elementem implementacji naszego projektu był system Multiplayer, który zbudowaliśmy na podstawie popularnego framework'u **Photon Unity Network**. Stworzyliśmy *Menadżera* naszej sieci (komponent **NetworkManager**), który odpowiednio łączy graczy z serwerem i kontroluje ilość osób w danej rozgrywce. Problemem na tą chwilę okazały się skrypty, wykonywane dotąd lokalnie.

Jeśli wojownik atakował, potwory traciły punkty życia, lecz ta informacja wyświetlała się tylko u jednego z graczy, natomiast u pozostałych graczy, mimo zadawania obrażeń, pasek punktów życia wrogów wracał w ułamku sekundy do poprzedniego stanu. To samo miało miejsce w przypadku animacji i poruszanych obiektów.

Działo się tak, ponieważ synchronizacja danych polega na wysyłaniu ich przez jednego gracza i odbieraniu przez pozostałych. Jeśli dany gracz nie jest w danym momencie nadawcą, może jedynie odbierać dane o obiektach, efekty jego interakcji z otoczeniem zastępowane są stale poprzez odbierane dane.

Problem został rozwiązany na dwa sposoby. Zdarzenia jednorazowe, takie jak obrażenia, włączanie odpowiednich animacji, czy przejmowanie podnoszonego obiektu są wysyłane jednorazowo przez gracza wywołującego daną interakcję. Odbywa się to za pomocą **funkcji RPC**. Są to funkcje specjalnego przeznaczenia, które zachowują się jak zwykłe lokalne funkcje, lecz w momencie ich wywołania, uruchamiane są także u wszystkich pozostałych graczy, podłączonych do gry. Dzięki temu uniknęliśmy całkowicie kolizji związanych z przesyłaniem danych.

Do zdarzeń ciągłych, takich jak przenoszenie obiektów przez Maga, użyliśmy komponentu **PhotonView** i przypisanych jako jego parametry modułów **PhotonTransformView** oraz **PhotonRigidbodyView**, dzięki czemu obserwowane są określone parametry obiektu i nadawane w sposób cią-

gły do wszystkich odbiorców. Naprawia to także problemy z opóźnieniami w sieci, ponieważ framework wylicza płynną drogę obiektu z jednego miejsca do drugiego, dzięki czemu ruch nie jest poklatkowy. W tym wypadku jednak konieczne jest płynne przełączanie między nadawcami, np. w przypadku podniesienia obiektu, nadawcą jego pozycji staje się gracz, który ten obiekt podniósł (jest on określany automatycznie na podstawie klienta, który wywołał funkcję przejęcia kontroli nad obiektem):

---

```
heldedThing.GetComponent<PhotonView> ().RequestOwnership  
();
```

---

Listing 2.9: Funkcja przejmowania funkcji nadawania danych

Z drugiej strony pojawił się problem sterowania daną postacią, gdyż Unity nie odróżniało, do którego gracza należy dana postać i wciskanie klawiszy poruszało wszystkimi bohaterami (jako, że każdy z nich przypisany ma ten sam kod odpowiedzialny za sterowanie).

Podczas łączenia z grą, Photon inicjalizuje wybraną postać i nadaje jej odpowiedni numer ID, powiązany z nadawcą. Dzięki temu postać gracza jest rozróżniana spośród innych i w prosty sposób można sprawdzić, czy funkcja została wywołania lokalnie, czy po stronie odbiorcy. W każdej lokalnej funkcji, powiązanej z postacią gracza umieściliśmy kod, który opuszcza funkcję, jeśli nie została ona wywołania dla postaci, którą sterujemy:

---

```
if (!photonView.isMine) return;
```

---

Listing 2.10: Opuszczanie funkcji w przypadku, gdy postać gracza nie należy do nas

Postać gracza wybierana jest na podstawie postaci, które są już podłączone, w określonej przez nas kolejności. Tworzymy listę dostępnych postaci, następnie iterujemy po podłączonych już graczach i usuwamy z listy kolejne elementy. Jeśli lista po przefiltrowaniu nie pozostaje pusta, wybieramy jej

pierwszy element i na tej podstawie umieszczamy gracza na planszy:

---

```
// All available players with spawn order
List<string> availPlayers = new List<string>();
availPlayers.Add("Wizard");
availPlayers.Add("Warrior");
availPlayers.Add("Archer");

foreach (PhotonPlayer otherPlayer in
    PhotonNetwork.otherPlayers)
{
    availPlayers =
        availPlayers.Remove(otherPlayer.NickName);
    Debug.Log("Found other player: " +
        otherPlayer.NickName);
}
if (availPlayers.Count > 0)
{
    PhotonNetwork.player.NickName = availPlayers[0];
}
else
{
    Debug.Log("No more players left");
    return;
}
Debug.Log("Let begin as " + PhotonNetwork.player.NickName);
```

---

Listing 2.11: Algorytm wybierania postaci gracza

Następnie obiekt postaci jest inicjalizowany w określonej pozycji (określonej poprzez pusty, otagowany **GameObject** umieszczony na mapie), jego kamera jest włączana oraz przejmowane są przez niego operacje nadawania

informacji o sobie:

---

```
// Instantiate player
var currentPlayer =
    (GameObject)PhotonNetwork.Instantiate(spawnParams.prefab.name,
    spawnPoint.transform.position,
    spawnPoint.transform.rotation, 0);
currentPlayer.transform.FindChild("Camera").gameObject.SetActive(true);
currentPlayer.GetComponent<PhotonView>().RequestOwnership();
```

---

Listing 2.12: Algorytm inicjalizacji postaci gracza

## **ROZDZIAŁ 3**

# **Bibliografia**

# Bibliografia

[1] Photon Documentation

<https://doc-api.photonengine.com/en/pun/current> (dostęp 22.04.2017)

[2] Unity Documentation

<https://docs.unity3d.com/Manual/index.html> (dostęp 22.04.2017)

[3] Unity Forums

<https://forum.unity3d.com/> (dostęp 22.04.2017)

[4] C Sharp Documentation

<https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx> (dostęp 22.04.2017)

[5] Stack Overflow

<https://stackoverflow.com> (dostęp 22.04.2017)

[6] Mixamo

<https://www.mixamo.com> (dostęp 22.04.2017)