

Sieciowa fabularna gra komputerowa - The Trinity

Kleina Mateusz, Podlawski Adrian

12 kwietnia 2017

Spis treści

1	Streszczenie	1
2	Wstęp i opis	2
3	Projektowanie oraz przygotowanie środowiska	4
3.1	Projekt Terenu	4
3.2	Projekt Postaci	5
3.3	Projekt systemu multilplayer	6
3.4	Instalacja niezbędnych narzędzi	7
4	Implementacja	8
4.1	Budowa terenu	8
4.2	Tworzenie Postaci	13
4.3	Ruch kamery	14
4.4	Fizyka	17
4.5	Animacje	19
4.6	Poruszanie bohaterów	20
4.7	Poruszanie postaci wrogów	21
4.8	Umiejętności Bohaterów	23
4.9	Multiplayer	24
5	Bibliografia	25

1 Streszczenie

Zaimplementowaliśmy wieloosobową grę, która bazuje na silniku graficznym Unity. Serwer, z którym łączą się gracze znajduje się w chmurze, którą oferuje Framework Photon Unity 3D Networking. Mapa została utworzona dzięki technologii Unity, która pozwala na dowolne kształtowanie terenu. Krajobrazy oraz budynki zostały pobrane z Assets Store oraz odpowiednio zmodyfikowane tak, by spełniały wymagania gry. Wszystkie skrypty odpowiadające za sterowanie postaci, łączenie z serwerem, czy walkę z przeciwnikiem, zostały utworzone na potrzeby gry w obiektowym języku programowania C Sharp. W trakcie rozwoju aplikacji, pliki były przechowywane we wspólnym repozytorium na stronie Github.com. Początkowym założeniem naszej pracy było utworzenie gry wieloosobowej, 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, w późniejszym czasie na tle fabularnym. Wszystkie powyższe założenia udało nam się zrealizować. W poniższej pracy przedstawimy ze szczegółami cały proces powstawania naszej produkcji.

2 Wstęp i opis

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 naszą pracę 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 umożliwia 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 do dla nas wyzwanie, ale zarazem wielki krok w początkach naszej kariery, jako młodych programistów.

Unity daje możliwość tworzenia gry bez wdrażania się w szczegółów technicznych dotyczących 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 przyjazniejsze i bardziej dostępne dla programistów chcących skupić się na samej mechanice gry.

Skloniło nas to do stworzenia nieco odmiennego projektu, w którym roz-

grywka 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 postacie, 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.

3 Projektowanie oraz przygotowanie środowiska

3.1 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. Na początku rozgrywki napotkamy kilku przeciwników oraz przerażoną kobietę, która wprowadzi nas w świat gry. Postaci kierowane przez graczy posiadają określona 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.

3.2 Projekt Postaci

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

3.3 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. 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.

3.4 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.

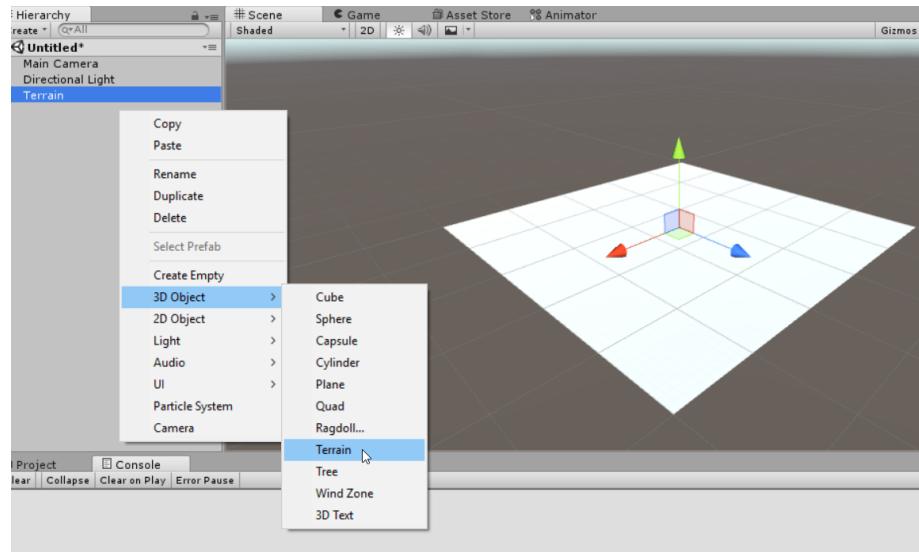
Podczas pracy nad naszym projektem często działałyśmy na tych samych, obszernych plikach np. w przypadku edycji mapy, po której poruszają się gracze. Przez to podczas łączenia naszych zmian często dochodziło do konfliktów w kodzie, których system kontroli wersji nie mógł sam rozwiązać. W tym wypadku niezbędne okazało się narzędzie UnityYAMLMerge, które znacznie usprawniło łączenie dwóch kopii, bez konieczności manualnego rozwiązywania konfliktów.

4 Implementacja

4.1 Budowa terenu

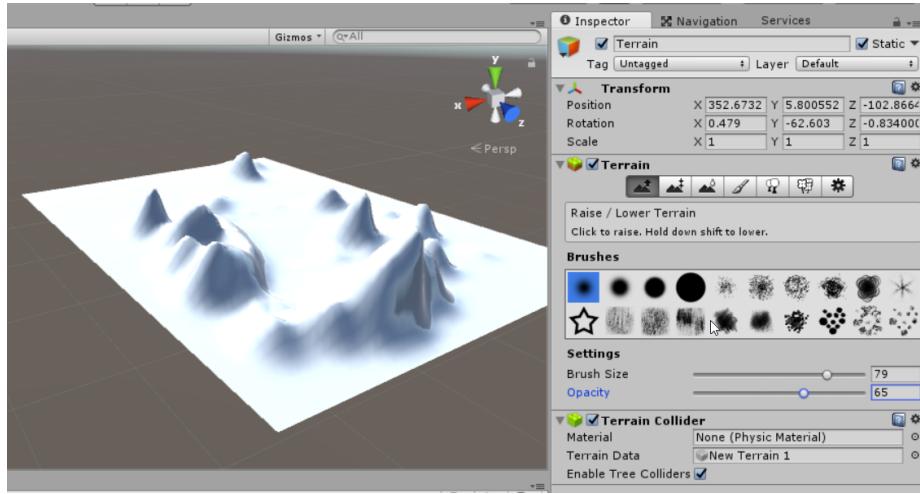
Budowa podstawowej mapy gry jest stosunkowo nieskomplikowanym procesem, dzięki czemu mogliśmy się tutaj skupić głównie na projektowaniu dróg, rozłożeniu obiektów i kształtowaniu mapy na potrzeby rozgrywki.

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 1: Płaszczyzna terenu

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**.

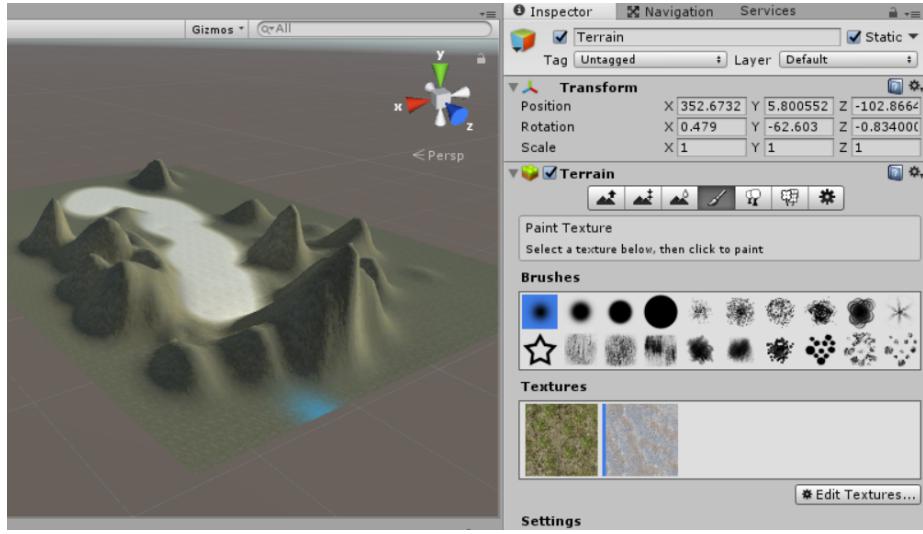


Rysunek 2: Modelowanie nierówności terenu

Nakładanie tekstur Kolejnym krokiem było utworzenie tekstury służącej do nadania 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.

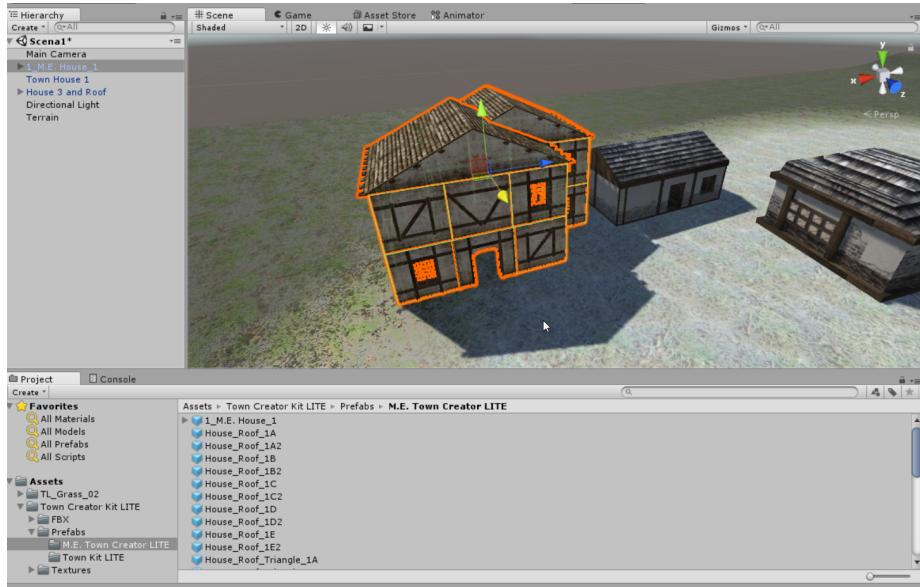
Po pobraniu odpowiedniej tekstury trawy, zostaje ona zainportowana do folderu [Assets](#) znajdującego się w głównym katalogu projektu.

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 3: Nakładanie tekstur na mapę terenu

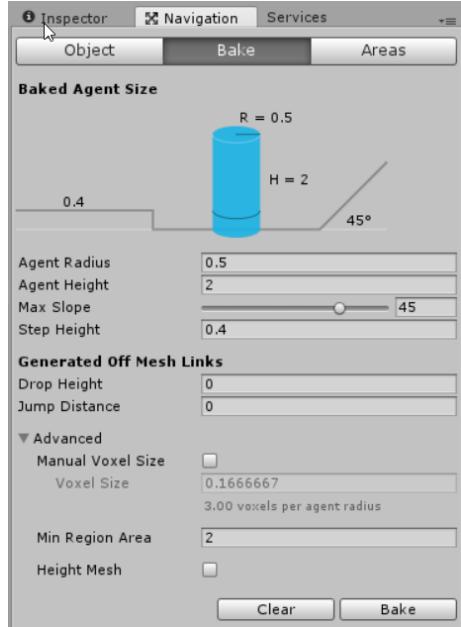
Rozmieszczanie obiektów Kolejnym krokiem w tworzeniu świata gry było umieszczenie na mapie sprefabrykowanych wcześniej obiektów (tzw. *Prefabs*, o tym później). Wszystkie obiekty użyte w pracy są dostępne za darmo w bibliotece obiektów Unity ([Asset Store](#)). Po pobraniu, obiekty znajdują się w podfolderze **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 4: Umieszczanie obiektów na mapie

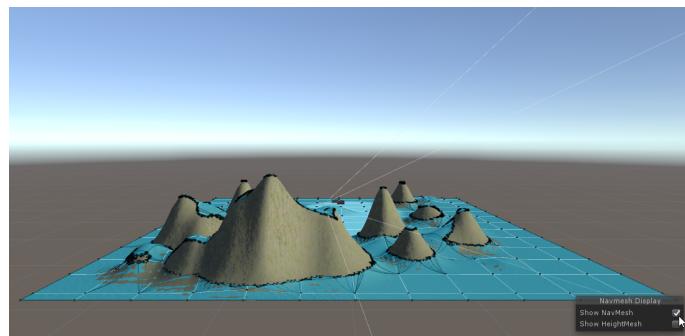
Oddziaływanie terenu na inne obiekty Kluczowym elementem tworzenia mapy świata gry są takie elementy jak oddziaływanie na postacie ograniczenie nachylenia terenu typu wzgórza, drzewa, czy budynki, uniemożliwiających dostanie się w niektóre miejsca. Unity w celu uproszczenia obliczeń posiada możliwość wygenerowania uproszczonej mapy dróg (tzw. *NavMesh*) na bazie modelu terenu, pozwalającej na dynamiczne omijanie przeszkód przez wroga (o tym później w sekcji 4.7 na stronie 21).

Aby utworzyć taką mapę, należy przejść do zakładki **Navigation**, gdzie w panelu **Bake** znajduje się lista parametrów dotyczących maksymalnego kąta nachylenia terenu, czy maksymalnej wysokości uskoku, którą obiekty sterowane przez komputer mogą pokonać.



Rysunek 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 6: Poprawnie wygenerowany *NavMesh*

Tak przygotowana mapa posłużyła nam do projektowania dalszej części gry.

4.2 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 (3). Jest to również bardzo dobre rozwiązanie, gdy w projekcie musimy użyć wielu takich samych obiektów. W naszej grze wykorzystaliśmy to między 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.

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.

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.

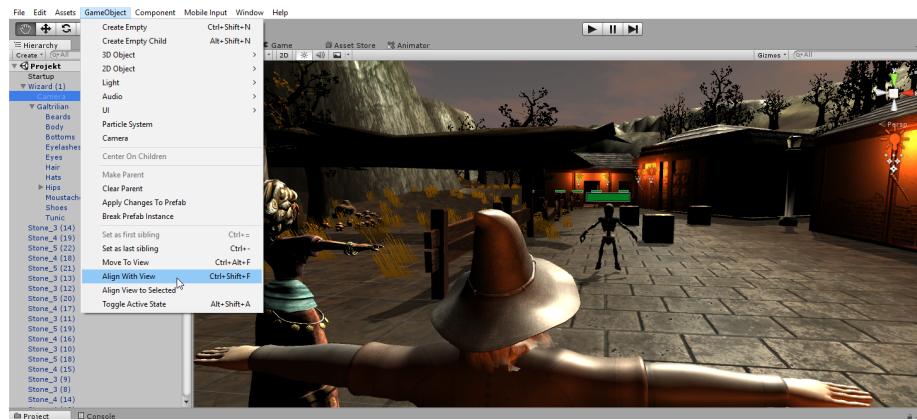
Niektóre obiekty mogą składać się z kilku innych obiektów. Dobrym przykładem jest rycerz, który poza modelem postaci, posiada również obiekt tarczy, hełmu oraz miecza. Każdy z tych obiektów posiada również swoje własne komponenty. W przypadku miecza może być to komponent odpowiadający za wykrywanie kolizji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

4.3 Ruch kamery

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*.

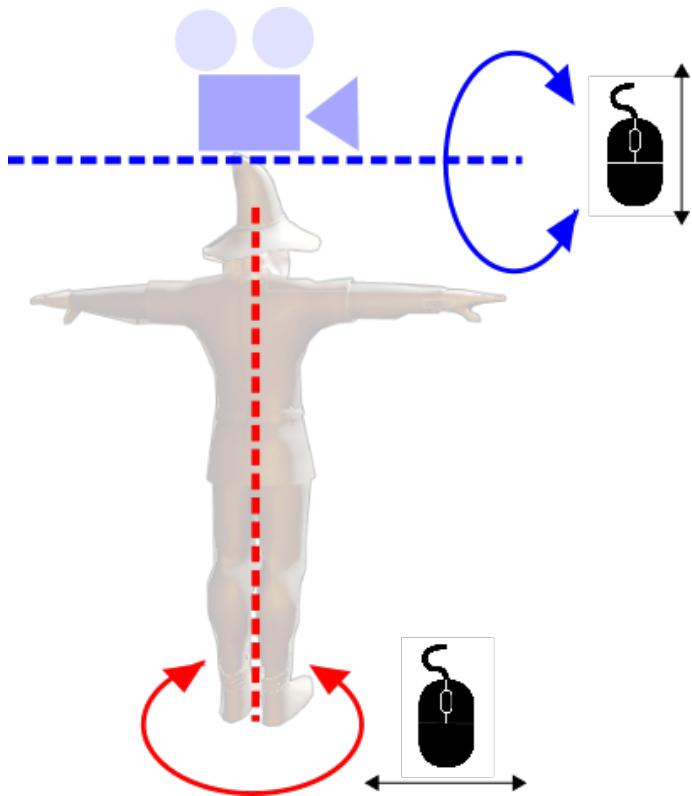
Aby uzyskać efekt kamery podążającej za graczem, obiekt kamery powinien zostać umieszczony wewnętrz obiektu gracza w hierarchii obiektów. W ten sposób koordynacje kamery będą ustawiane względem gracza, a sama kamera poruszać się będzie i obracać wraz z nadziednym obiektem.

Koordynacje kamery ustawiśmy na (0, 0, 0), w ten sposób kamera jest w środkowym punkcie obiektu nadziednego i wytarczy 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 7: Dostosowanie pozycji kamery gracza względem aktualnego widoku sceny

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 rozglądzania się, modyfikujemy jedynie jej obrót w pionie, natomiast ruch myszy w poziomie obraca poziomo całą postać wraz z kamerą.



Rysunek 8: 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));
}

```

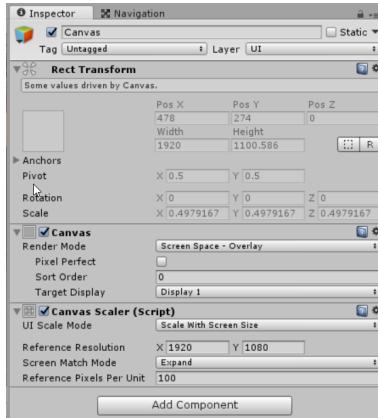
Listing 1: Obrót kamery za pomocą myszy

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.

Celownik Aby ułatwić celowanie w przeciwników oraz interakcję z otoczeniem umieściliśmy 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*. Sprity' 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. We wnętrzu takiego płotna można umieszczać obrazy, elementy interfejsu oraz różnego rodzaju wskaźniki (np. Pasek postępu życia bohatera).



Rysunek 9: Konfiguracja płotna 2D do wyświetlania interfejsu gry

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



Rysunek 10: Gotowy celownik gracza

4.4 Fizyka

Ważnym elementem gry jest fizyka, aby postaci mogły poruszać się, podskakiwać 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, sześciangu, bądź uproczczonego modelu samego obiektu. Pozwalają one silnikowi gry na przyspieszenie obliczeń i płynne reakcje na zdarzenia kolizji.



Rysunek 11: 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.).

Mechanika skakania postaci Mając już zaimplementowaną fizykę, dodaliśmy możliwość skoku. Utworzona została metoda *jump()*, która dodaje do zdań fizycznych naszej postaci siłę w kierunku pionowym o wektorze ustalonym parametrem *jumpPower*. Natomiast na podstawie kolizji z terenem ustalane 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: Fragment algorytmu skoku postaci

4.5 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.

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.

Żeby postać mogła przejść z jednego stanu na drugi niezbędne są transakcje. To na nich ustalamy, na jakie parametry oraz ich wartości ma dojść do zmiany animacji.

Zwykle 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.

Są to o wiele 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 niż rzeczywiście ma zaimplementowanych.

W Blend Tree możemy wykorzystać również stopniowe zwiększenie się naszych parametrów, co można doskonale zobaczyć 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 pełnego sprintu.

4.6 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 aktywowy 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 3: Fragment kodu mechaniki biegania – reakcja na wcisnięcie klawisza **W**

4.7 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 4.1 na stronie 8). 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 wróg 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 4: Algorytm zwracający wskaźnik do obiektu znajdującego się najbliżej, przy określonym maksymalnym promieniu poszukiwań



Rysunek 12: 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 5: Algorytm zwracający wskaźnik do obiektu znajdującego się najbliżej, przy określonym maksymalnym promieniu poszukiwań

4.8 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.

Wojownik słynie z ogromnej siły, jego zadaniem jest rozprawianie się z jak największą ilością przeciwników. Jego postać posiada specjalny komponent, która umożliwia mu odbieranie życia swoim przeciwnikom. Przy każdym uderzeniu, gdy tylko skrypt wykryje kolizję pomiędzy mieczem naszego bohatera, a przeciwnikiem, odbiera mu określoną ilość punktów życia. Nasi wrogowie oznaczeni są odpowiednim Tagiem, przez co nie możemy zranić innych graczy. Wojownik posiada dwa różne ataki. Jeden szybki, tym samym zadający mniej obrażeń, który wywoływany jest lewym przyciskiem myszy oraz mocny, zamazzysty atak, który odbiera znacznie większe ilości życia. Drugi atak może zostać użyty tylko podczas sprintu, po kliknięciu prawym przyciskiem myszy.

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 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.

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.

4.9 Multiplayer

Ostatnim elementem implementacji naszego projektu był system Multiplayer. Stworzyliśmy Menadżera naszej sieci, który odpowiednio łączył graczy z serwerem i kontrolował ilość osób w danej rozgrywce. Napotkaliśmy jednak wiele problemów, gdyż wszystkie funkcje były wykonywane tylko lokalnie u danego gracza.

Jeśli wojownik atakował, potwory traciły życie, lecz ta informacja wyświetlała się tylko u jednego z graczy. To samo miało miejsce w przypadku animacji i poruszanych obiektów.

Z drugiej strony pojawił się problem sterowania daną postacią, gdyż Unity nie odróżniało, do którego gracza należy dana postać i jeden z trzech graczy poruszał wszystkimi bohaterami.

Najpierw zajęliśmy się problemem poruszania postaciami. Photon nadaje każdemu obiekowi posiadającemu komponent PhotonView odpowiednie ID. W skryptach, które odczytywały wszystkie sygnały z wejścia. Mogliśmy wykorzystać ten fakt i przed wykonaniem jakiejkolwiek operacji sprawdzaliśmy, czy dany widok, który wyświetla dany gracz jest zgodny z jego numerem ID.

Do rozwiązania problemu z przesyłaniem animacji oraz pozycji pomiędzy graczami użyliśmy na obiektach komponentów PhotonRigidbodyView oraz PhotonAnimatorView, które musiały zostać połączone z komponentem PhotonView przechowującym nasze unikalne ID.

Finalnym krokiem było poprawne wykonywanie pozostałych funkcji, takie jak przenoszenie przedmiotów, atakowanie potworów, czy rozmowa z NPC. Rozwiążanie w tym wypadku zaimplementowaliśmy w naszym Menadżerze połączeń. Domyślnie wszystkie skrypty, które kolidowały pomiędzy postaciami zostały na obiektach wyłączone. W momencie, gdy użytkownik łączy się do gry, wszystkie komponenty zostają aktywowane konkretnie dla danego obiektu. Dzięki tym zabiegom Framework odróżnia, który gracz wywołał daną funkcję oraz synchronizuje wszystkie dane u pozostałych użytkowników.

5 Bibliografia

Photon Documentation <https://doc-api.photonengine.com/en/pun/current>

Unity Documentation <https://docs.unity3d.com/Manual/index.html>

C Sharp Documentation <https://msdn.microsoft.com/en-us/library/67ef8sb.aspx>

Stack Overflow <https://stackoverflow.com>

Mixamo <https://www.mixamo.com>