Uniwersytet Warmińsko-Mazurski w Olsztynie
Wydział Matematyki i Informatyki

Michał Tadeusz Ruść Miłosz Proc

Kierunek: Informatyka

Implementacja gry RPG przy użyciu środowiska Unity

Praca inżynierska wykonana
w Katedra Analizy Zespolonej
pod kierunkiem
dr Piotra Jastrzębskiego

Olsztyn, 2022 rok

University of Warmia and Mazury in Olsztyn
Faculty of Mathematics and Computer Science

Michał Tadeusz Ruść Miłosz Proc

Field of Study: Computer Science

Implementation of RGP game with Unity editor

Engineer's Thesis is performed
in Chair of Complex Analysis
under supervision of
Piotr Jastrzębski PhD

Olsztyn, 2022

Spis treści

Streszczenie

Celem pracy jest stworzenie gry z elementami RPG, do czego głównie zostało użyte środowisko Unity. Będzie ona działać na systemie operacyjnym Windows, firmy Microsoft. Osobie, która gra w zamiarze ma być dostarczana rozrywka i odpowiednia z niej satysfakcja po jej ukończeniu. W założeniu ma ona posługiwać się pixel-artami, jednocześnie przez tą prostą grafikę być atrakcyjną dla swojego grona docelowego.

Abstract

Implementation of RGP game with Unity editor

Our goal is to make a game with some elements of an RPG, which is why we used Unity for the most part of it. The game will run on Windows operating system from Microsoft. We want the Player to be entertained and feel satisfied after finishing the game. Pixel-art and simple graphics were used for the game to be more appealing for it's target group.

Wstęp

Tematem pracy, którą wykonaliśmy była implementacja gry fabularnej (ang. Role-Playing Game) przy użyciu środowiska Unity. Jako twórcy od dłuższego czasu interesowaliśmy się tematyką tworzenia tego typu aplikacji desktopowej, stąd naszej uwagi przy wyborze tematu pracy inżynierskiej nie mogło przykuć nic innego.

Zdecydowaliśmy się na środowisko Unity, ponieważ jest ono powszechnie znanym narzędziem do tworzenia oprogramowania, w szczególności gier komputerowych, i pozwala na szybką implementację gry. Praca nad grą była wykonywana w dwie osoby ze względu na duży nakład pracy, jaki pochłania proces tworzenia takiego oprogramowania. Ponadto zdecydowaliśmy się na własnoręczną produkcję wymaganych plików graficznych, co znacząco powiększyło ilość pracy, którą trzeba było wykonać.

Michał Ruść przy tworzeniu aplikacji zajmował się implementacją generatora świata, bohatera, kart i przeciwników oraz działaniem świata gry i systemu walki, obsługą interakcji we wcześniej wspomnianych elementach, UI w scenach mapy świata oraz walki, obsługą zdarzeń na mapie, tworzeniem części kart i wydarzeń a także obsługą muzyki i zapisem oraz wczytywaniem gry, podczas gdy Miłosz Proc zajmował się serializacją i układem plików xml zawierających dane aplikacji (przeciwnicy, wydarzenia i karty), menu głównym, przedstawienie graficzne części obiektów, balsem gry i przeprowadzeniem testów.

Rozdział 1

Karciana gra fabularna pt. Ukryta Talia

1.1. Komputerowe gry fabularne i karciane

Komputerowa gra fabularna[1], gatunek wśród innych gier komputerowych, w którym rola gracza obejmuje kontrolę nad głównym bohaterem lub czasem drużyną bohaterów umieszczonych w fikcyjnym świecie. Z biegiem czasu główne postacie awansują lub dokonują jakiegokolwiek progresu stając się silniejszymi, jednocześnie dalej doświadczając świata zbudowanego wokół nich. Dobrym przykładem takiej aplikacji jest Wiedźmin 3: Dziki Gon, stworzony przez polskie studio CD Projekt Red. Jedne z pierwszych z fabularnych gier komputerowych powstały już w latach siedemdziesiątych dwudziestego wieku. Najczęściej tworzone na podstawie niecyfrowych wersji gier fabularnych. Komputerowe gry karciane, inaczej aplikacje, w których główną rolę grają przeniesione na ekran powszechnie znane prostokątne kartki papieru. Wersję namacalną gier interesujących się tym tematem znamy już od ponad tysiąca lat. Dzisiaj karty w dziale rozrywki cyfrowej wykorzystywane są na wiele sposobów, dobrze to widać na przykładzie Hearthstone, które jest w posiadaniu Microsoftu, czy też Gwinta, za którego odpowiada CD Projekt Red.

1.2. Fabuła

Rysunek 1.1: Logo gry

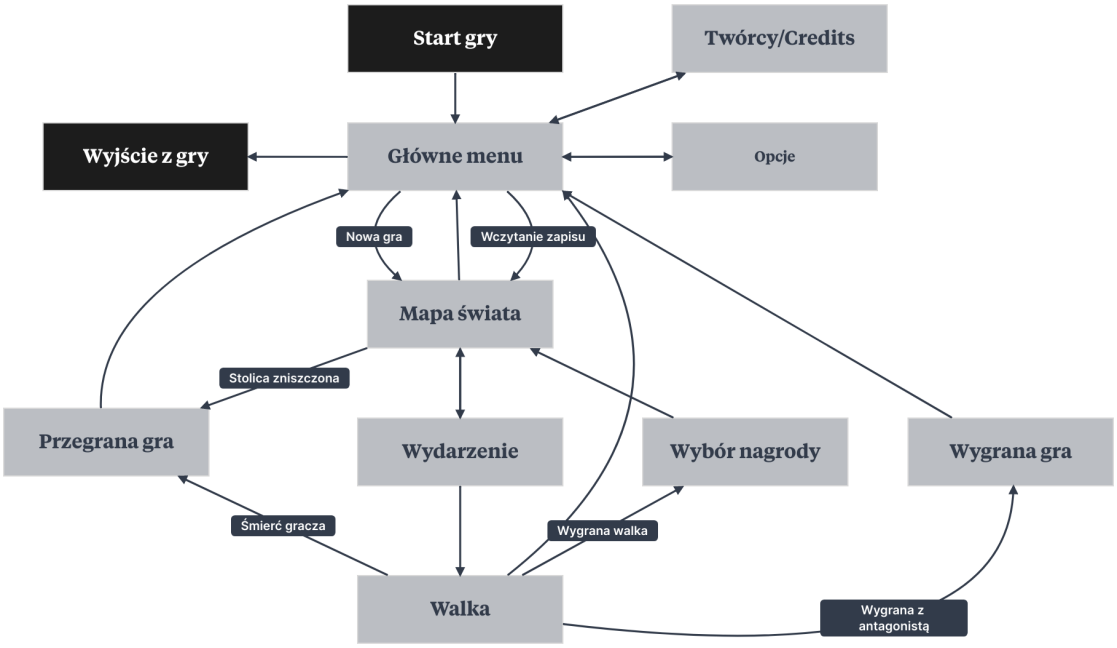
Ukryta Talia, to jedno osobowa gra, w której gracz wciela się w bezimiennego bohatera, który przez lata zgłębiał moc magicznych kart, a teraz dzięki nim posiada moc, którą może wykorzystać w walce. Owe karty można znaleźć na całym świecie, ale tylko nieliczni umieją z nich korzystać. Wyrusza on w podróż by zatrzymać inwazję jaka dokonuje się w regionie świata w którym żyje. Bohater nie jest w stanie walczyć z całą armią, dlatego działa w ukryciu czasem prowadząc bój z maksymalnie kilkoma przeciwnikami. Na celu ma dostanie się do głównodowodzącego wrogich sił i pokonanie go, aby przeciwnicy opuścili ich świat. Bohater zaczyna w stolicy regionu i skrupulatnie, dzięki swojej mapie, będzie dążył do głównej fortecy złego przywódcy.

1.3. Docelowa grupa odbiorców

Aplikacja według rankingu PEGI klasyfikowałaby się do wieku od siedmiu lat[2]. Chętniej zabiorą się do niej osoby lubiące proste grafiki, takie które nie oczekują wymagającej fabuły, a chcą spędzić relaksując się kilka godzin swojego wolnego czasu.

Rozdział 2

Rozgrywka



Rysunek 2.1: Diagram przepływu gry Ukryta Talia

2.1. Cel gry

Celem gry jest pokonanie antagonisty znajdującego się w jego zamku, po przeciwnej stronie mapy od stolicy, w której zaczyna gracz. By tego dokonać gracz musi wpierw dostać się we wcześniej wspomniane miejsce przechodząc między lokacjami. Gra odbywa się w koncepcji trwałej śmierci (ang. permadeath) tj. gdy gracz przegrywa, jego postęp oraz sam zapis gry zostają usunięte, zmuszając gracza do ponownego rozpoczęcia gry.

2.1.1. Warunki wygranej

Aby wygrać rozgrywkę gracz musi pokonać bardzo silnego przeciwnika znajdującego się w zamku wroga.

2.1.2. Warunki przegranej

Gracz przegrywa, gdy jego punkty życia podczas walki spadną do zera, lub gdy stolica zostanie zniszczona przez oblężenie.

2.2. System walki

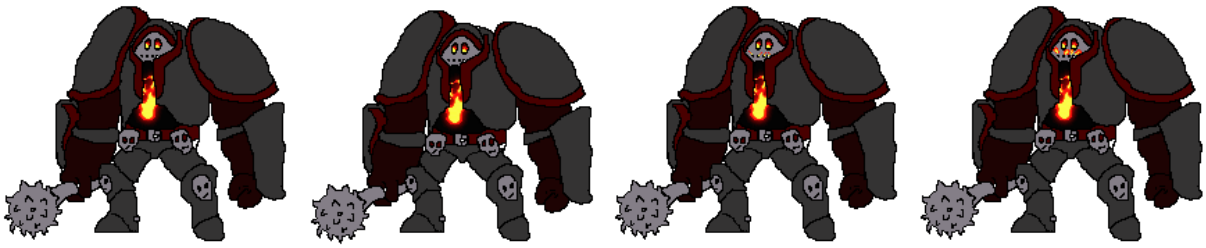
Walka opiera się o karty które posiada gracz. Karty po użyciu nakładają efekty. Aby użyć karty gracz musi posiadać w momencie rzucenia karty odpowiednią ilość punktów akcji. Punkty akcji oraz karty są przyznawane graczowi co pewną ilość czasu. Zmusza to gracza do podejmowania decyzji, której karty użyć w danym momencie. By użyć karty, trzeba ją przeciągnąć na przeciwnika na którego chcemy ją użyć.

2.2.1. Gracz

Gracz posiada dek kart który można rozwijać poprzez zdarzenia na mapie, oraz poprzez karty nagradzane za wygranie walki. Postać gracza jest opisywana przez jego zdrowie oraz punkty akcji które można rozwijać poprzez wydarzenia na mapie świata.

2.2.2. Przeciwnicy

Przeciwnicy są opisywani przez punkty życia, których każdy typ przeciwnika posiada zawsze tyle samo, oraz inicjatywę, która pozwala przeciwnikowi użyć jednej losowo wybieranej karty którą posiada za każdym razem gdy zostanie naładowana. Inicjatywa wrogów zwiększa się z czasem i jest stała dla danego typu przeciwnika. W początkowych fazach projektowania naszej gry planowaliśmy dać przeciwnikom sztuczną inteligencję, lecz po dłuższych przemyśleniach doszliśmy do wniosku, iż prostszy system jest lepszym rozwiązaniem, ponieważ ułatwia to walkę dla gracza.



Rysunek 2.2: Grafiki przeznaczone do animacji ostatecznego przeciwnika

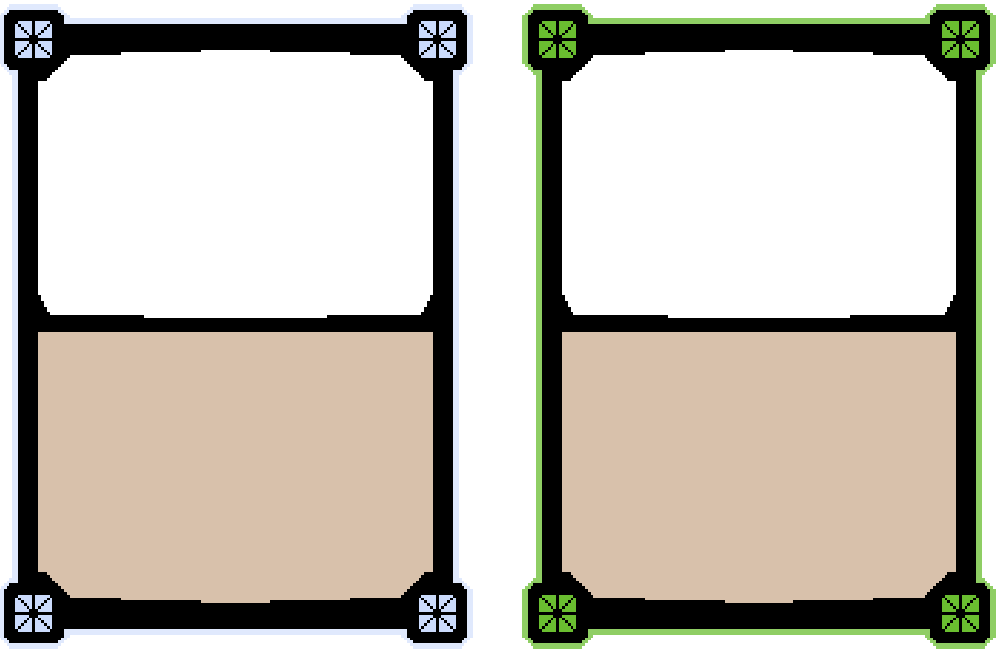
(a) Grafiki przeznaczone do animacji przeciwnika nazwanego Orkiem

(b) Grafiki przeznaczone do animacji przeciwnika nazwanego Krasnoludem

Rysunek 2.3: Grafiki przeciwników wprowadzonych na ten moment w grze

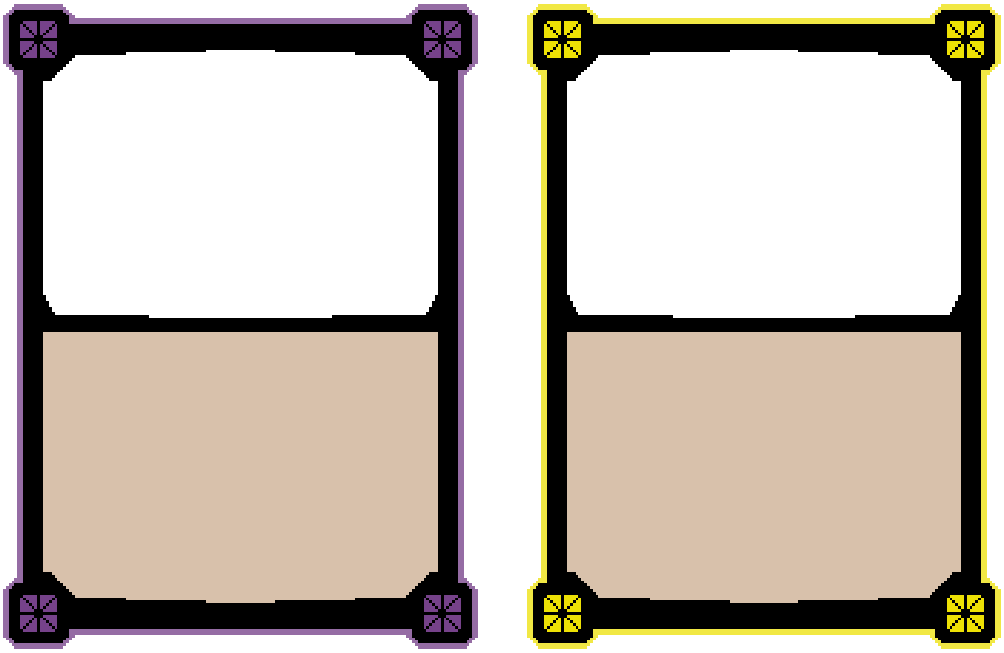
2.2.3. Karty

Karty są głównym elementem, przy użyciu którego gracz toczy walkę. Są one podzielone na kilka poziomów rzadkości, który wyznacza ich siłę i jest reprezentowany przez odpowiedni wzór wokół karty. Każda karta posiada swój koszt bez zapłacenia którego nie może być rzucona i listę efektów. Wrogowie również posiadają swoją listę kart, ale ignorują oni koszt karty.



(a) Wzór zwyczajnej karty (b) Wzór rzadkiej karty

Rysunek 2.4: Grafiki obramowań kart dla najmniejszych rzadkości



(a) Wzór epickiej karty (b) Wzór legendarnej karty

Rysunek 2.5: Grafiki obramowań kart dla największych rzadkości

2.3. Świat

Mapa świata reprezentuje różnorakie miasta, wsie i budowle znajdujące się na terenie na którym odbywa się gra. Gracz może poruszać się między nimi, o ile istnieje stosowne połączenie. Podróż jest natychmiastowa, zabiera jeden dzień niezależnie od odległości i rozpoczyna wydarzenie po dotarciu do celu. Wydarzenia nie pojawiają się dwa razy w tym samym miejscu.

2.3.1. Rodzaje lokacji

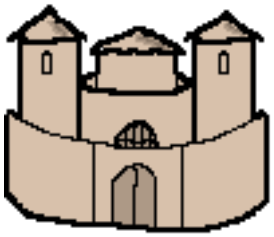
	Stolica, miejsce startowe gracza. Jeżeli oblężenie tego miejsca się zakończy, gra kończy się porażką
	Zamek wroga, tu musi dotrzeć gracz aby rozpocząć walkę z ostatecznym przeciwnikiem. Dodatkowo jest to lokacja która rozpoczyna oblężenia na mapie.
	Forteca,jest ona bardzo wytrzymała na oblężenia. Dwie zawsze są połączone z zamkiem wroga.
	Ruiny, miejsce w którym gracz może odnaleźć rzadkie karty.
	Miasto, gracz może napotkać tu różnorakie wydarzenia, choć mało z nich zmusza gracza do walki
	Wioska, słabo chroniony obszar, dlatego oblężenia niszczą wioski w jeden dzień. Występuje często i daje słabe nagrody

Tabela 2.1: Lokacje na mapie w grze

2.3.2. Oblężenia

Oblężenie jest prostą mechaniką gry, która zmusza gracza do podejmowania decyzji co do trasy jaką będzie się poruszał. Przy każdym ruchu przeciwnika mija jeden dzień. W efekcie rozpoczynając oblężenia od zamku wroga kolejne lokacje słabną i są podbijane. Różne miejsca mają różne czasy po których upadają. Po zniszczeniu, oraz w trakcie oblężenia większość lepszych wydarzeń staje się niedostępna, drastycznie zmniejszając wartość przy odwiedzeniu takiej lokacji. Każde miejsce które zostało zniszczone rozszerza oblężenia dalej. By dać na początku gry trochę więcej czasu przed zamkiem wroga znajdują się zawsze fortece które mają długi czas potrzebny by oblężenie je zniszczyło.

Rozdział 3

Implementacja

3.1. Specyfikacja systemowa

Grę `Ukrtyta Talia` stworzyliśmy na systemy operacyjne `Windows 7 (SP1)`,`Windows 10` i `Windows 11`. Wymaga ona procesora o architekturze x86 lub x64 ze wsparciem zestawu instrukcji `SSE2`, obsługi DirectX 10,11 lub 12 oraz wpieranych przez dostawcę sprzętu sterowników. Gra wspiera rozdzielczości w standardzie 16:9 z rozmiarami od 1366x768 do 3840x2160, a dodatkowo dzięki odpowiednim ustawieniom dostarczanym przez Unity obsługuje również pozostałe rozdzielczości w standardzie 16:9, lecz ze względu na rzadkość występowania ekranów o innych rozmiarach w tej proporcji, nie zostały one przetestowane.

3.2. Sterowanie

Do obsługi gry potrzebna jest albo mysz, albo tablet graficzny albo touchpad. Wszystkie akcje w grze zostały stworzone tak, aby dało się je wykonać przy użyciu lewego przycisku myszy (lub jego odpowiednika na wcześniej wymienionych urządzeniach) oraz poprzez przesuwanie kursora, czy też akcję "przeciągnij i upuść". Zdecydowaliśmy się na takie sterowanie z myślą o rozszerzeniu w przyszłości obsługiwanych systemów głównie o urządzenia Android, na których nie ma ogólnego dostępu do klawiatury w trakcie gry.

3.3. Wykorzystane technologie

3.3.1. Unity

Unity to środowisko używane do tworzenia gier (dwuwymiarowych lub trójwymiarowych) i innych interaktywnych, działająca na systemach operacyjnych, Microsoft Windows, macOS, Linux. Daje możliwość tworzenia aplikacji na przeglądarki internetowe, desktopy, konsole i urządzenia mobilne. W przypadku naszej pracy to unity było w niej najważniejsze, dzięki niemu mogliśmy połączyć wszystkie skrypty i grafiki, w przyjemny i prosty sposób modyfikować stronę wizualną naszej gry. Z funkcjonalności dostarczanych przez to środowisko bardzo często korzystaliśmy między innymi z obiektów gry, które stanowią reprezentację pewnego obiektu w przestrzeni, do którego można dodawać komponenty, wykonujące kod. Korzystaliśmy również wielokrotnie z samej możliwości tworzenia skryptowalnych komponentów, a także z wielu przydatnych klas zawieranych w bibliotekach Unity.

3.3.2. C#

Język programowania obiektowego, stworzony na rzecz firmy Microsoft. Często wykorzystywany przy tworzeniu aplikacji desktopowych, webowych czy mobilnych. Wieloplatformowy

(Linux, Windows lub też macOS). W naszym projekcie wykorzystaliśmy go ponieważ jest to jeden z języków, który obsługuje unity w przypadku pisania skryptu, jednocześnie jest to dla nas język który najlepiej rozumiemy.

3.3.3. Visual Studio 2022

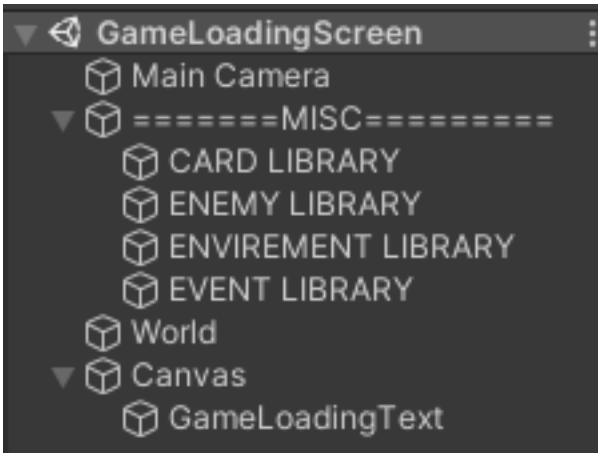
Najnowsza wersja środowiska programistycznego firmy Microsoft. Jest ono wielojęzykowe, to znaczy, że można przy jego pomocy programować np. w C#, C++, Python. Działa tylko na systemach ich twórców. Dzięki dobremu współdziałaniu środowiska z edytorem Unity postanowiliśmy właśnie nim się posłużyć w przypadku naszej aplikacji. Dodatkowo za nim stał czas jaki już spędziliśmy programując w Visual Studio, dokładniej towarzyszył on nam przez cały okres studiów.

3.3.4. Aseprite

Edytor obrazów, który stworzony został do rysowania i animowania w pikselach, działa na wielu systemach. Wspiera tworzenie i pracę na warstwach. Aplikację poznaliśmy na pracowni dyplomowej, na której jedna z osób prowadzących aktualny temat zajęć wspomniała o jego prostocie. Umiejętności graficzne nie stanowiły przeszkody w tworzeniu zarówno animacji jak i grafik, dlatego nasza decyzja padła właśnie na Aseprite.

3.4. Sceny gry środowiska Unity

W naszej grze wykorzystujemy wbudowane w Unity sceny.[3] Sceny w Unity są rodzajem assetu, czyli obiektu, który jest wykorzystywany w projektach tego środowiska, i stanowią pewien rodzaj podziału gry. Jedną z najważniejszych z ich funkcji, jest to, że obiekty gry (`GameObject`) są niszczone przy zmianie sceny, a przy ładowaniu obiekty w danej scenie ładowane są ponownie. Dzięki temu w naszej grze nie musieliśmy się zajmować ponownym tworzeniem wielu obiektów UI przy np. przejściu do ekranu walki i powrocie do ekranu świata. Ponadto sceny pozwalają na organizację projektu, co bardzo ułatwia pracę.

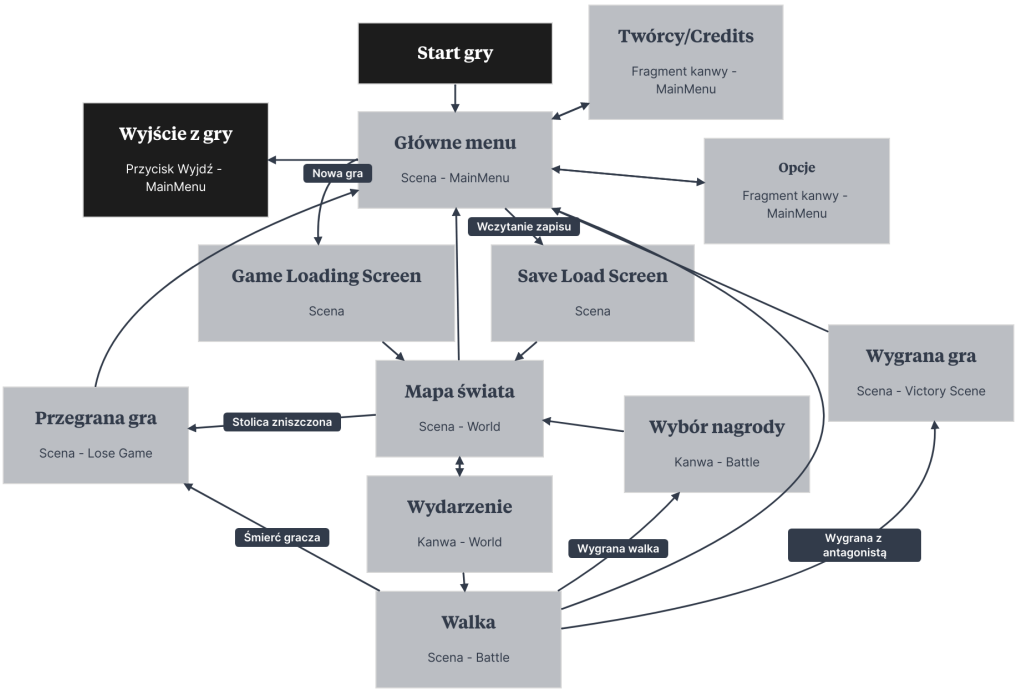


Rysunek 3.1: Organizacja obiektów gry w naszej scenie ładowania

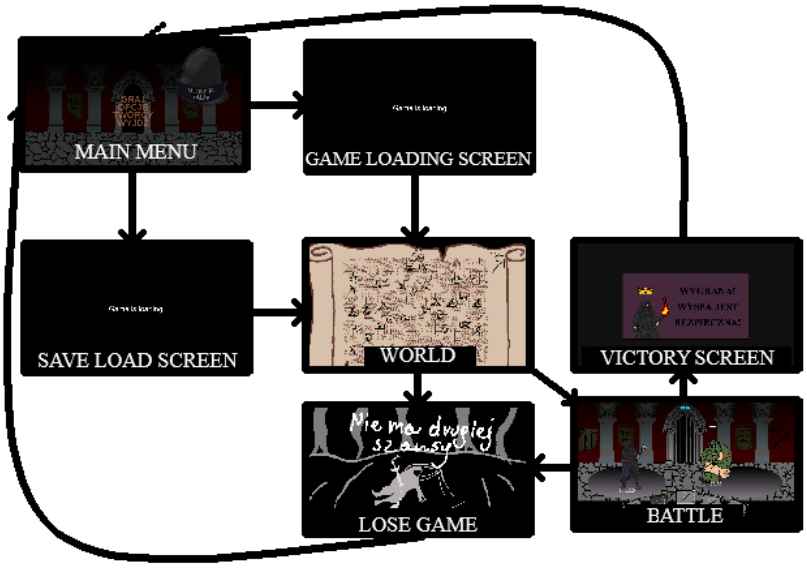
W całym projekcie naszej gry wykorzystaliśmy 7 scen.

1. Menu główne, które dawało możliwość rozpoczęcia gry, zmiany ustawień, czy też wyjścia z gry.

2. Ładowanie gry i wczytywanie zapisu, które pozwalały nam rozróżnić, czy tworzyć nowy świat, czy też wczytywać zapis.
3. Świat w którym pokazywana jest mapa świata i gracz może się poruszać oraz podejmować decyzje przy wydarzeniach.
4. Walka w której gracz walczy o nagrody z przeciwnikami.
5. Przegrana i wygrana gry, które dają znać graczowi czy wygrał, czy też przegrał i resetują grę do stanu początkowego, pozwalając na rozpoczęcie kolejnej rozgrywki.



Rysunek 3.2: Diagram przepływu gry z podpisanym elementami które realizują dany fragment



Rysunek 3.3: Diagram scen znajdujących się w naszej grze

3.5. Generowanie świata

Generowanie świata odbywa się w kilku krokach wewnątrz sceny `Game Loading Screen`. Obiektem gry który jest odpowiedzialny za wygenerowanie świata jest "World", a dokładniej klasa-komponent `GameManager` przypisana do niego, która jest odpowiedzialna za sterowanie dużą częścią gry poprzez łączenie jej elementów. Najpierw generowane są obiekty służące do reprezentacji miejsc. Następnie gra tworzy połączenia między lokacjami i rysuje drogi między nimi. Ostatecznie tworzona jest dekoracyjna mapa znajdująca się na zwoju widocznym na scenie. W postaci listy kroków tworzenie świata wygląda tak:

1. Tworzona jest nowa, pusta lista do obiektów `WorldMapNode`, jest to klasa stworzona do przechowywania danych na temat lokacji i przechowuje takie dane jak pozycja, typ, czy też połączenia wychodzące z danego miejsca. `WorldMapNode` jak można zauważyć po nazwie, realizuje strukturę wierzchołka grafu.
2. Tworzone i stawiane są, na wcześniej w kodzie ustalonych w statycznych koordynatach, lokacje zamku antagonisty i stolicy. Dzięki temu pozycje na których są stawiane są rezerwowane i w pobliżu nie zostaną postawione inne typy lokacji.
3. Podobnie jak wcześniej stawiane są dwie fortece, od razu połączone z zamkiem wroga, rezerwując miejsce we wcześniej ustalonych w statycznych zmiennych pozycjach.
4. Tworzone są połączenia między lokacjami przy użyciu ich wzajemnych odległości.
5. Każde miejsce sprawdzane jest pod kątem połączenia do reszty mapy. Jeżeli nie zostanie znaleziona odpowiednia ścieżka, to algorytm próbuje połączyć odłączoną grupę połączonych nawzajem lokacji od najbliższej do zamku antagonisty do najbliższej do niej, połączonej do zamku wroga.
6. Tworzona jest dekoracyjna mapa, która będzie wyświetlona pod miejscami.
7. Dekoracyjna mapa zostaje wygładzona
8. Tworzone są obiekty gry do wyświetlenia mapy dekoracyjnej

Wszystkie te dane przechowywane są w komponencie `GameManager`. Lokacje na mapie, razem, realizują strukturę grafu, gdzie ich połączenia są krawędziami. Sam obiekt "World" jest singletonem który nie jest niszczony podczas zmiany sceny, co pozwala nam na łatwe odnalezienie tego obiektu po jego nazwie.

```
void GenerateWorldMap()
{
    worldMap = new List<WorldMapNode>();
    CreateMainLocations();
    CreateWorldLocationsRandom();
    CreateBorderFortresses();
    ConnectNodesToTheirClosestNodes();
    for (int i = 1; i < worldMap.Count; i++)
    {
        ConnectIfNotConnected(worldMap[i]);
    }
    CreateWorldDecorationsArray();
    for (int i = 0; i < decoratorArraySmoothingCount; i++)
    {
        SmoothWorldDecorationArray();
    }
    DrawDecorators();
}
```

Rysunek 3.4: Kod generacji świata

3.5.1. Generowanie lokacji

Lokacje generowane są w odpowiedniej kolejności, pierwszymi generowanymi lokacjami są stolica i zamek wroga, następnie generowane są dwie fortece graniczące z zamkiem wroga. Na koniec gra tworzy losowe lokacje w obszarze który został zdefiniowany wcześniej w kodzie. Dla każdego z miejsc podczas generacji tworzone są obiekty:

1. Główny `GameObject` który służy do przechowywania pozycji na mapie niezależnie od pozostałych parametrów i przechowujący uchwyt służący do obsługi interakcji wskaźnikiem
2. `GameObject` służący do wyświetlania obrazka lokacji
3. `GameObject` służący do wyświetlania stanu oblężenia
4. `GameObject` służący do oznaczenia czy miejsce jest odwiedzone

(a) Oblężona wioska

(b) Odwiedzona wioska

Rysunek 3.5: Przykładowe lokacje stworzone przy użyciu powyższego podziału

```
public WorldMapNode(GameManager _gameManager,string _name,string _type)
{
    gameManager = _gameManager;
    name = _name;
    type = _type;
    connections = new List<WorldMapNode>();
    SetSiegeTime();

    gameObject = new GameObject(name);
    gameObject.transform.parent = gameManager.worldMapObject.transform;
    gameObject.transform.localPosition = Vector3.zero;
    WorldMapNodeHandle handle = gameObject.AddComponent<WorldMapNodeHandle>
    handle.SetWorldMapNodeHandle(this,gameManager);

    CreateSpriteObject();

    SetSprite();

    CreateCollider();

    CreateSiegeIcon();

    CreateVisitedIcon();
}
```

Rysunek 3.6: Skrypt tworzenia obiektów lokacji

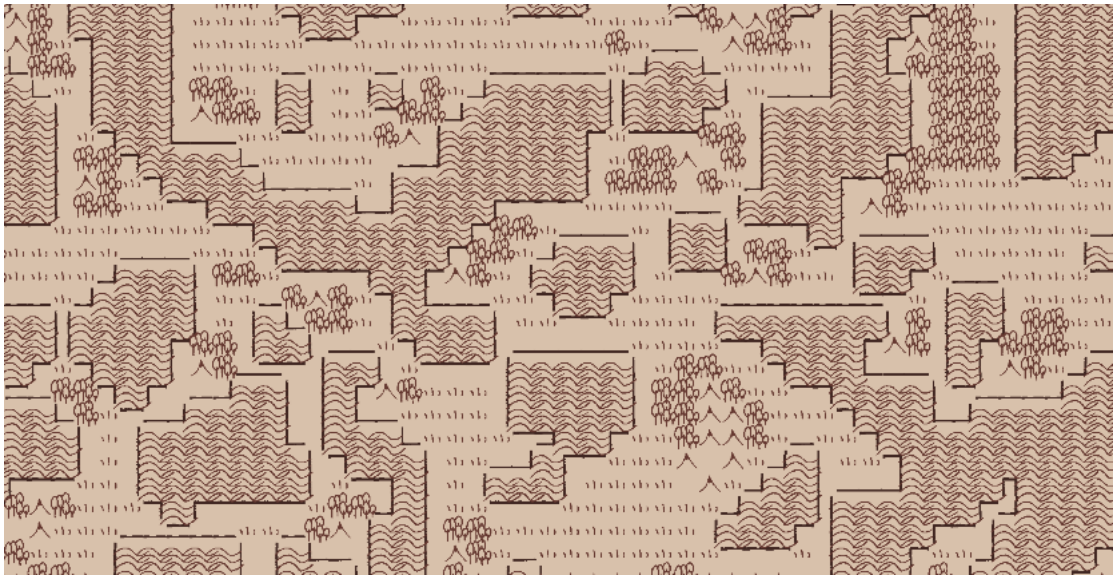
Dzięki takiemu rozwiązaniu dodawanie kolejnych elementów graficznych jest proste i pozwala na proste zarządzanie dodatkowymi elementami. Obiekty tworzone w celu wyświetlania grafiki realizują swoją funkcję poprzez wbudowane w środowisko Unity komponent `SpriteRenderer`

3.5.2. Generowanie połączeń

Połączenia do każdej lokacji generowane są na podstawie odległości do innych miejsc. Ponadto, jeżeli nie zostały znalezione żadne inne lokacje w odpowiedniej odległości, połączenie zostaje utworzone do najbliższego miejsca. Następnie cała mapa zostaje sprawdzona pod względem możliwości dojścia do zamku antagonisty przy użyciu algorytmu przeszukiwaniu grafu wszerz (BFS). W przypadku nie znalezienia możliwej trasy, wybierana jest połączona ze sprawdzanym wierzchołkiem grafu mapy lokacja znajdująca się najbliżej zamku wroga i jest ona łączona z najbliższym miejscem które ma dostęp do celu gry. Wizualnie, drogi są tworzone za każdym razem, gdy dwa punkty są łączone, poprzez użycie komponentu `MeshRenderer` w obiekcie gry niezależnym od lokacji i stworzenie dla każdej drogi dwóch trójkątów i nałożenie na nie grafiki. W tym przypadku nie użyliśmy komponentu `SpriteRenderer` ze względu na to, iż nasze rozwiązanie pozwoliło na dużo lepsze ustawienie odpowiednio tekstury i ustawienie jest pozycji niż to co jest w stanie osiągnąć `SpriteRenderer`.

3.5.3. Tworzenie dekoracyjnej mapy

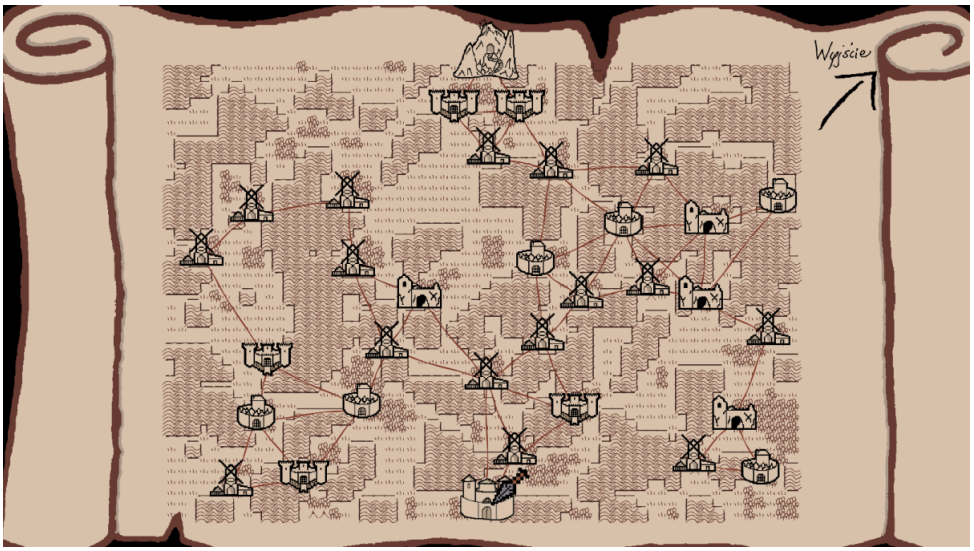
Mapa która znajduje się pod lokacjami tworzona jest poprzez wykorzystanie losowo generowanej mapy wysokości. Ma ona rozmiary większe od pola gry, które zależne są od mnożnika podanego przed kompilacją. Po nadaniu losowych wartości każdemu z punktów, wysokości są kilkukrotnie wygładzane przy użyciu średniej arytmetycznej z 8 sąsiednich pól i danego pola. Podczas wygładzania pominięte zostają wartości pod lokacjami które posiadają stałą wartość i przez to niezależnie od losowości podnoszą wysokość terenu w górę. Następnie w zależności od finałowej wartości, dla każdego pola zostaje przypisana odpowiednia tekstura reprezentująca wodę,pola,lasy,góry, lub jedną z 15 możliwych ustawień granicy z wodą. W ten sposób uzyskujemy naturalnie wyglądający teren, który dobrze prezentuje się na takiej mapie.



Rysunek 3.7: Dekoracyjna mapa

3.5.4. Graficzny styl

Podczas generowania poszczególnych elementów mapy, staraliśmy się zachować wygląd, który przypomina stare, ręcznie rysowane mapy. Z tego powodu mapa zawiera proste grafiki i małą liczbę kolorów.



Rysunek 3.8: Przykładowa mapa wygenerowana w grze

3.6. Implementacja przeciwników i gracza

Zarówno interaktywny obiekt gry gracza jak i przeciwnicy są implementowani w bardzo podobny sposób.

```
public GameObject Hand;
public GameObject playerSpriteObject;

public GameObject uiGameObject;
public GameObject hpTextGameObject;
public GameObject apTextGameObject;
public GameObject spriteObject;

public TextMeshPro hpText;
public TextMeshPro apText;

public GameObject initiativeBar;
```

(a) Elementy tworzące gracza

```
public GameObject enemyObject;
public GameObject spriteObject;
public GameObject uiGameObject;
public GameObject hpTextGameObject;

public TextMeshPro hpText;

public GameObject initiativeBar;
```

(b) Elementy tworzące wroga

Rysunek 3.9: Jak widać elementy które tworzą gracza są bardzo podobne w strukturze do przeciwnika

Podobnie jak w przypadku lokacji rozbiliśmy ich na wiele obiektów gry z których każdy spełnia różne funkcje. Ponadto dodaliśmy do tych obiektów możliwość odtwarzania animacji poprzez komponent `SpriteAnimator`. W naszej grze jest to poważnie zmodyfikowany przez

nas komponent, który był zaproponowany na znanym forum StackOverflow [5]. Rozszerzyliśmy ten komponent o możliwość odtworzenia jednorazowo animacji poprzez podanie listy kolejnych klatek, kontrolę prędkości oraz odtwarzanie całej listy klatek.

Rysunek 3.10: Gracz w trakcie wykonywania animacji ataku

Zarówno przeciwnicy jak i gracz posiadają swoje własne talie kart, jednak w przypadku gracza jest to dużo bardziej rozwinięte. Gracz posiada zaimplementowany stos kart, z którego mogą być pobierane karty do ręki i który może być tasowany, by zmienić kolejność kart.

```
public void CreateNewCardStack(bool random = true)
{
    cardStack = new List<DeckCard>(data.deck);
    foreach (DeckCard item in cardStack)
    {
        item.ResetCard();
    }
    Debug.Log("Created card stack with " + cardStack.Count + " cards.");
    if (random)
    {
        for (int i = 0; i < cardStack.Count -1; i++)
        {

            int swapPosition = UnityEngine.Random.Range(0, cardStack.Count - 1);
            if (swapPosition != i)
            {
                DeckCard card1 = cardStack[i];
                DeckCard card2 = cardStack[swapPosition];
                cardStack[i] = card2;
                cardStack[swapPosition] = card1;
            }
        }
    }
}
```

Rysunek 3.11: Implementacja stosu kart z jego początkowym przetasowaniem

Ponadto sama implementacja gracza realizuje również ”rękę z kartami” znajdującą się na dole ekranu w trakcie walki. Jest ona stworzona tak, by równomiernie rozkładała karty na

kształt przypominający wachlarz. Jest to wykonywane przez funkcję `SetupCardLocation` która rozkłada karty względem pewnego punktu startowego zgodnie ze wzorem -

$$(maxRozpietosc * 2/iloscKart) * (numerKartyWReku + 0.5) - maxRozpietosc$$

Podobne wzory są używane przy określeniu kąta i wysokości karty w ręku.

```
public void SetupCardLocation()
{
    Vector3 BasePosition = Hand.transform.position;
    float maxSpreadDistance = 50f;
    float maxAngle = 30f;
    float maxHeight = 10f;
    if (hand.Count == 1)
    {
        hand[0].card.cardMover.moveCardTo = transform.TransformPoint(BasePosition + new Vector3(0, 0f, 0f));
        hand[0].card.cardMover.rotateTo = Vector3.zero;
        hand[0].card.cardObject.transform.localScale = new Vector3(0.5f, 0.5f);
        hand[0].card.cardObject.transform.parent = Hand.transform;
    }
    else
    {
        for (int i = 0; i < hand.Count; i++)
        {
            float angleHeightInfluence = -Mathf.Abs (((((maxAngle * 2) / hand.Count) * (i + 0.5f)) - maxAngle)/maxAngle);
            hand[i].card.cardMover.moveCardTo = transform.TransformPoint(BasePosition +
                new Vector3((((maxSpreadDistance * 2) / hand.Count) * (i + 0.5f)) - maxSpreadDistance,
                angleHeightInfluence*maxHeight, 0f));
            hand[i].card.cardObject.transform.localScale = new Vector3(0.5f, 0.5f);
            hand[i].card.cardObject.transform.parent = Hand.transform;
            hand[i].card.cardMover.rotateTo = new Vector3(0f,0f,-((((maxAngle * 2) / hand.Count) * (i + 0.5f)) - maxAngle));
            hand[i].card.SetSortingOrder(i * 2);
        }
    }
}
```

Rysunek 3.12: Kod implementujący rozkładanie kart w ręku na kształt wachlarza

3.7. Działanie świata

3.7.1. Realizacja ruchu po mapie

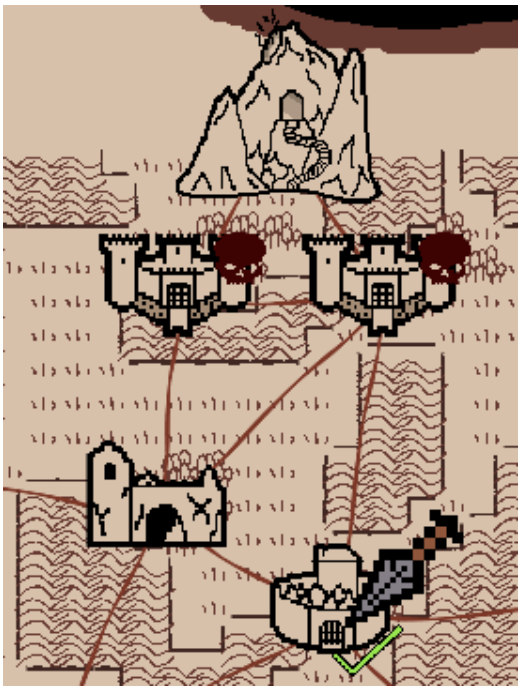
Ruch po mapie jest wykonywany poprzez kliknięcie na lokację bezpośrednio połączoną z obecną pozycją gracza. Gracz na mapie oznaczony jest znacznikiem widocznym na rysunku 3.13.

Rysunek 3.13: Pozycja gracza onzaczona znacznikem, pokazująca że jest w mieście

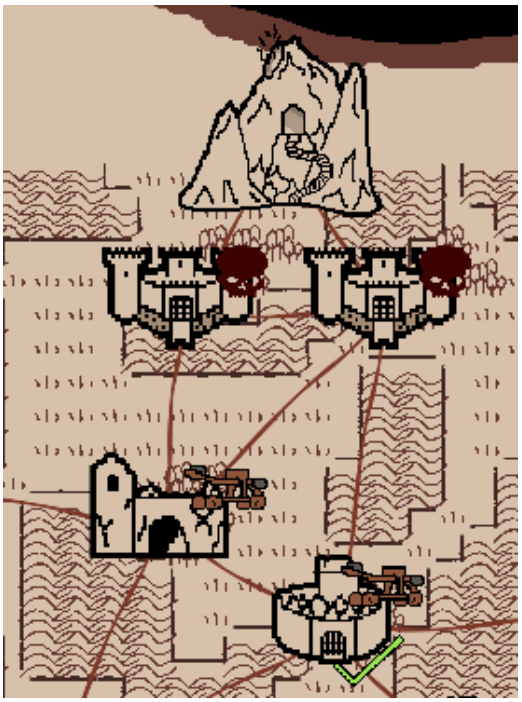
Gdy gracz najedzie kursorem na lokację do której można odbyć podróż, zostaje ona wizualnie powiększona, co potwierdza taką możliwość. Powiększanie oraz obsługa kliknięć działają dzięki komponentowi `WorldMapNodeHandle` podłączonego do głównego obiektowi lokacji. Dotarcie do dowolnej lokacji powoduje aktywację wydarzenia, o ile nie została ona odwiedzona wcześniej, a następnie oblężenia na mapie postępują.

3.7.2. Oblężenia

Mechanika oblężeń została zaimplementowana w prosty sposób. Wewnątrz danych każdej lokacji, w klasie `WorldMapNode` znajduje się zmienna `siegeTime` która zależy od przypisanej lokacji. Wyznacza ona ile "dni", tj. wykonanych ruchów gracza na mapie musi minąć nim lokacja upadnie po rozpoczęciu oblężenia. Oblężenie rozpoczyna się 1 ruch gracza po tym, jak upadnie jedna z połączonych do niej lokacji.



(a) Stan mapy na jeden dzień zanim oblężenia rozpoczną się od podbitych fortec



(b) Stan mapy po następnym ruchu

Rysunek 3.14: Postępujące oblężenia

Wartości siegeTime wynoszą dla lokacji odpowiednio:

1. Stolica - 5
2. Miasto - 3
3. Wioska - 1
4. Forteca - 5
5. Ruiny - 1

3.7.3. Zdarzenia

W wykreowanym przez nas świecie w momencie odwiedzenia nieodwiedzonej wcześniej lokacji gra symuluje wydarzenie które w tym miejscu mogło zajść. Zależnie od lokacji, przez `GameManager` losujemy jedno zdarzenie z listy wydarzeń które mogły zajść w danym miejscu. Specjalnym przypadkiem są podbite i oblężone lokacje, w których mogą wystąpić tylko kilka niezależnych od typu miejsca wydarzeń. Gracz po wylosowaniu i pokazaniu przez grę wydarzenia z jej wyborami (może ich być od 1 do 4) i opisem musi dokonać wyboru. Każda decyzja może dać szeroką ilość możliwych efektów, od nadania nowych kart, do rozpoczęcia walki, aż po odjęcie lub dodanie całkowitego zdrowia. Konsekwencje wydarzenia zależą od tagów które są przypisane do danej decyzji.

Rysunek 3.15: Przykładowe wydarzenie w lokacji "Wioska"

3.8. Implementacja systemu walki

System walki został przez nas zaimplementowany poprzez scenę walki i stworzony przez nas komponent `BattleManager`. Służy on do spięcia całej pojedynczej bitwy w jedną całość oraz przygotowuje wizualne obiekty gracza i przeciwnika oraz pobiera on odpowiednie dane z komponentu `GameManager`, takie jak dane przeciwników którzy są w walce. `BattleManager` obsługuje również nadawanie graczowi zasobów, zarówno kart jak i punktów akcji, oraz wysyła sygnały do odpowiednich przeciwników by wykonali swoją turę.

3.8.1. Karty w walce

Karty w trakcie walki są reprezentowane przez obiekty typu `DeckCard` i będące ich częścią `Card`. `Card` zawiera zestaw funkcji, służący do stworzenia instancji karty jako obiektu gry Unity, a także posiada wszystkie dane potrzebne do wykorzystania karty w walce.

Rysunek 3.16: Instancje kart gracza po utworzeniu w grze.

Działanie karty zależy od jej efektów, implementowanych poprzez klasę `Effect`. Klasa ta pozwala nam na uniwersalną i prostą obsługę wielu różnych wydarzeń podczas walki, oraz ich wpływu na przeciwnika lub gracza. Klasa `Effect` została stworzona z myślą o jej przyszłej rozbudowie, tak by mogła ona działać w przyszłości na podstawie zewnętrznego kodu lub

skryptu, rozszerzając modyfikowalność gry. Karty mogą również posiadać tagi implementowane poprzez klasę `Tag` dzięki którym można określać proste informacje na temat karty. Wśród tych informacji mogą być określone np. to czy jest jednokrotnego użycia, czy też możliwość jej otrzymania z losowych wydarzeń. Karty są przechowywane w deku gracza i ich stan w nim jest implementowany poprzez klasę `DeckCard`. Przechowuje ona samą instancję klasy `Card` oraz informacje na temat stanu karty w deku gracza, takie jak np. liczbę użyć karty, czy karta jest spalona i nie można jej dostać ponownie. Interakcja z kartami odbywa się przez jej łapanie, tj. mechanikę "drag and drop" realizowaną poprzez stworzony przez nas komponent `PointerControl`.

```
private void OnClickPerformed(InputAction.CallbackContext obj)
{
    if (grabbedCard == null)
    {
        Physics.Raycast(mouseWorldPosition, new Vector3(0f, 0f, 1f), out raycastHit, 10f);
        if (raycastHit.collider != null && raycastHit.collider.CompareTag("card"))
        {
            grabbedCard = raycastHit.collider.gameObject.transform.parent.gameObject;
            grabbedCardHandle = grabbedCard.GetComponent<CardHandle>();
        }
    }
    else
    {
        Physics.Raycast(grabbedCard.transform.position, new Vector3(0f, 0f, 1f), out raycastHit, 10f);
        if (raycastHit.collider != null && raycastHit.collider.CompareTag("enemy_sprite"))
        {
            EnemyHandle enemyHandle = raycastHit.collider.gameObject.transform.parent.gameObject.GetComponent<EnemyHandle>();
            battleManager.CardWasMovedOntoEnemy(grabbedCardHandle.card, enemyHandle.enemy);
        }
        grabbedCard = null;
    }
}
```

Rysunek 3.17: Kod realizujący łapanie kart

Gdy użytkownik kliknie gdzieś na ekranie w trakcie walki, wysyłany jest promień (ang. raycast) który podróżuje od kursora wzdłuż osi Z. Jeżeli natrafi on na kartę, ta karta zostaje zapisana i jest przesuwana w wygładzany sposób pod pozycję kursora.

```
static public float speed = 0.2f;
static public float rotationSpeed = 0.1f;
public Vector3 moveCardTo = new Vector3(0f,0f,0f);
public Vector3 rotateTo = new Vector3(0f, 0f, 0f);
Unity Message | 0 references
void Update()
{
    Vector3 position = this.transform.position;
    if (this.transform.position != moveCardTo)
    {
        Vector3 newScreenPosition = Vector3.Lerp(position, moveCardTo, speed);
        this.transform.position = new Vector3(newScreenPosition.x, newScreenPosition.y, newScreenPosition.z);
    }
    if (this.transform.rotation != Quaternion.Euler(rotateTo))
    {
        Quaternion newRotation = Quaternion.Lerp(this.transform.rotation, Quaternion.Euler(rotateTo), rotationSpeed);
        this.transform.rotation = newRotation;
    }
}
```

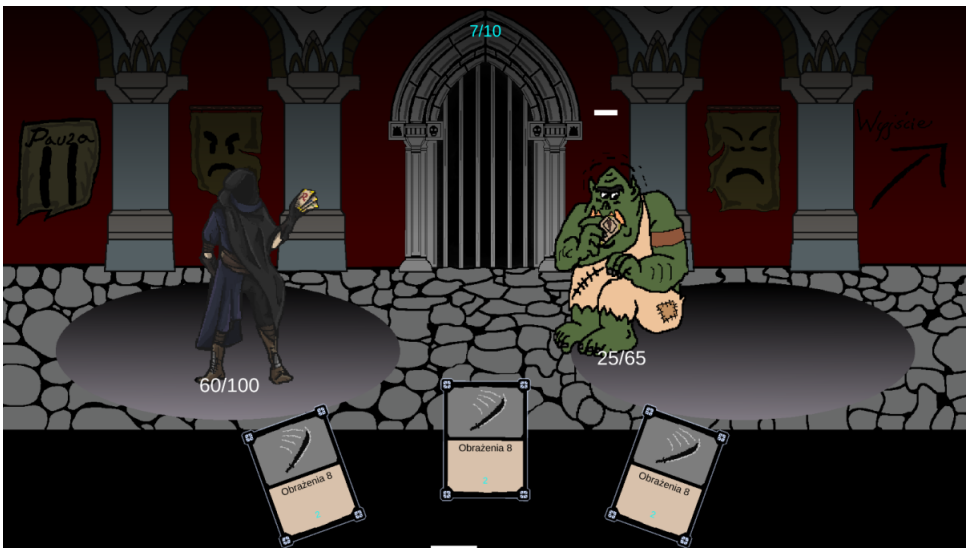
Rysunek 3.18: Skrypt realizujący wygładzanie ruchu kart

Wygładzanie kart odbywa się głównie poprzez wykorzystanie wbudowanej w dostarczane przez Unity klasy do przechowywania pozycji (`Vector3`) i rotacji (`Quaternion`) funkcji Lerp,

która pozwala na interpolacje pomiędzy dwoma stanami co pewną, podaną jako zmienna odległością. Gdy użytkownik upuści kartę, tj. puści lewy przycisk myszy gra sprawdza ponownie promieniem, czy pod kartą znajduje się przeciwnik. W przypadku znalezienia przeciwnika, komponent przesyła odpowiednią informację do klasy `BattleManager` o wykonaniu ruchu przez gracza, przy użyciu odpowiedniej karty. Informację o karcie która została użyta komponent obsługujący wskaźnik otrzymuje poprzez specjalnie do tego stworzonego komponentu `CardHandle`. Gdyby karta została upuszczona poza przeciwnikiem, wraca ona na swoje miejsce.

Rysunek 3.19: Karta podniesiona przez gracza w trakcie walki

3.8.2. Walka



Rysunek 3.20: Przykładowa scena walki

Walka odbywa się w czasie rzeczywistym. W jej trakcie gracz jest reprezentowany przez postać która znajduje się po lewej stronie ekranu (rysunek 3.21), przy której znajdują się punkty życia gracza. Gracz przegrywa walkę gdy jego punkty życia spadną poniżej 1. Przegranie walki oznacza porażkę w całej grze i zmienia aktywną scenę na scenę porażki (`LoseGame`).

Rysunek 3.21: Postać reprezentująca gracza

Wrogowie w walce reprezentowani są podobnie jak gracz, przez postacie znajdujące się po prawej stronie ekranu (rysunek 3.22). Martwi przeciwnicy są obróceni tak by wyglądali jakby leżeli na ziemi i nie da się z nimi wejść w interakcję. Każdy żywy przeciwnik ma swój własny element UI przedstawiający ich punkty życia, oraz znajdujący się powyżej nich pasek inicjatywy.

Rysunek 3.22: Postacie reprezentujące przeciwników, jeden z przeciwników jest martwy

Każdy przeciwnik, za każdym razem gdy jego inicjatywa się wypełni, wykonuje losowy ruch na podstawie kart które posiada. Przeciwnik może posiadać karty podobnie do gracza i może też nakładać poprzez nie bardziej skomplikowane efekty. W obecnym stanie przeciwnicy są tworzeni by tylko wykonywać proste ruchy, gdyż jedną z dopiero planowanych funkcjonalności jest dodanie wizualnego pokazania obecnie nałożonych efektów.

```
public void MakeAMove()
{
    List<DeckCard> possibleCard = new List<DeckCard>();
    for (int i = 0; i < deck.Count; i++)
    {
        if (!deck[0].destroyed &&
            deck[0].exhausted == 0)
        {
            possibleCard.Add(deck[0]);
        }
    }
    int pickRandom = UnityEngine.Random.Range(0, possibleCard.Count);
    Tag selfTag = possibleCard[pickRandom].card.FindCardTag("self");
    if (selfTag!=null)
    {
        battleManager.ApplyCard(possibleCard[pickRandom], this);
    }
    else
    {
        battleManager.ApplyCardToPlayer(possibleCard[pickRandom]);
    }
}
```

Rysunek 3.23: Kod wykonywany by dokonać ruchu przeciwnika

3.8.3. Wygrana

Gracz wygrywa walkę gdy punkty życia wszystkich przeciwników spadnie poniżej 1. Z wyjątkiem finałowej walki, po każdej walce pokazuje się ekran wyboru nagrody w którym gracz może wybrać 1 z 3 losowo wybranych kart.

Rysunek 3.24: Okno wyboru nagrody

Po wyborze karty lub pominięciu nagrody gra powraca do sceny mapy świata.

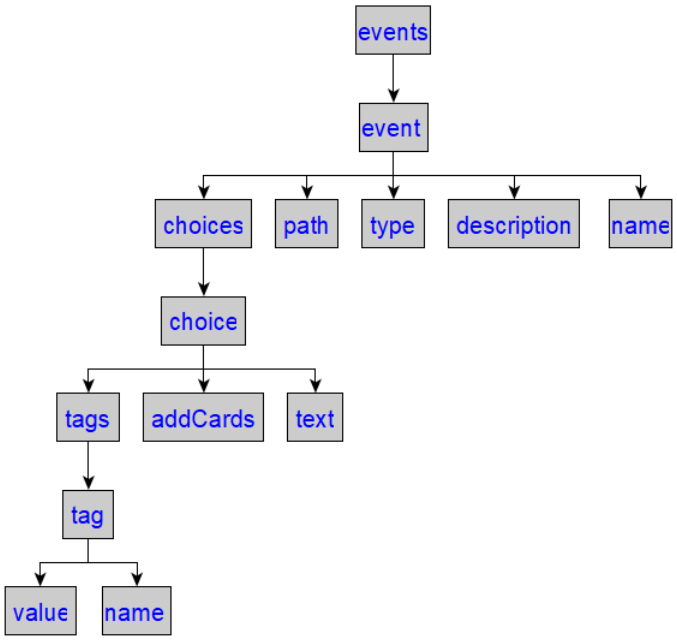
3.9. Wczytywanie i tworzenie obiektów przy pomocy XML

3.9.1. Budowa plików XML

W przypadku naszej pracy, większość obiektów wczytujemy z odpowiednich plików XML, do nich należą wydarzenia ze świata gry, karty, a nawet przeciwnicy. W plikach XML każdy obiekt ma odpowiednie dane by skrypt przerobił je w pełni na obiekty istniejące w grze.

```
<event>
    <name>Ciemny zakątek świata</name>
    <description>Wchodzisz w głębszą część ruin, co może Ciebie spotkać? Robi się kompletnie ciemno, co zrobisz?</description>
    <type>AcientRuins</type>
    <path>/CoreGame/EventGFX/Najciemniejszy_z_ciemnych.png</path>
    <choices>
        <choice>
            <text>Spróbuj zapalić pochodnie, unikając kłopotów psychicznych.</text>
            <addCards>
            </addCards>
            <tags>
                <tag>
                    <name>BattleRandom</name>
                    <value>1</value>
                </tag>
            </tags>
        </choice>
        <choice>
            <text>Kłopoty psychiczne to dla Ciebie pestka, tak samo jak rozciągająca się ciemność.</text>
            <addCards>
            </addCards>
            <tags>
                <tag>
                    <name>IncreaseMaxHp</name>
                    <value>-10</value>
                </tag>
            </tags>
        </choice>
    </choices>
</event>
```

Rysunek 3.25: Budowa obiektu wydarzenia w pliku XML.



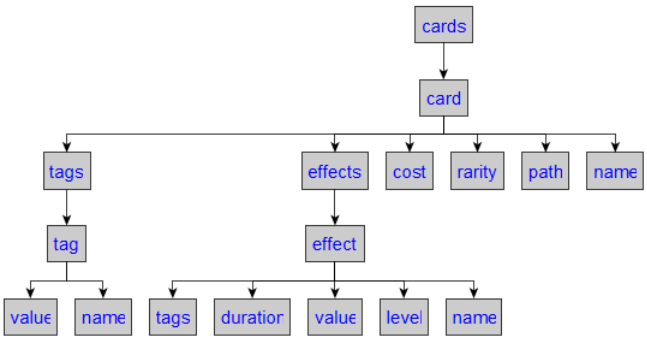
Rysunek 3.26: Schemat graficzny events.xml.

Jeśli mówimy wydarzeniach, obiekt w XML zawiera kolejno nazwę wydarzenia, jego opis, miejsce w którym występuje, ścieżkę do grafiki, listę możliwych wyborów, a w niej dla każdego

wyboru, opis, możliwe karty jakie otrzymamy, tagi dzięki którym ustalamy, czy wybór ciągnie za sobą walkę z przeciwnikiem, otrzymanie losowej karty lub też zmianę zdrowia bohatera.

```
<card>
  <name>Ostatnia_szansa</name>
  <path>CoreGame/CardGFX/</path>
  <rarity>2</rarity>
  <cost>8</cost>
  <effects>
    <effect>
      <name>Ostatnia szansa</name>
      <level>0</level>
      <value>0</value>
      <duration>0</duration>
      <tags>
      </tags>
    </effect>
  </effects>
  <tags>
      <tag>
          <name>NoRandom</name>
          <value>0</value>
      </tag>
  </tags>
</card>
```

Rysunek 3.27: Budowa obiektu karty w pliku XML.

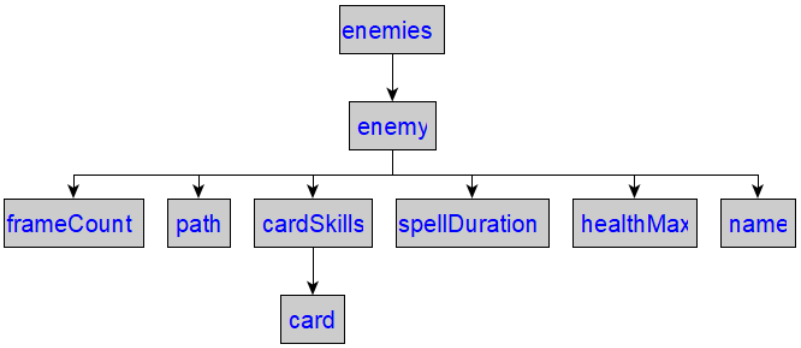


Rysunek 3.28: Schemat graficzny cards.xml.

Kiedy przyjrzymy się kartom, obiekt w XML zawiera w sobie nazwę karty, kolejno ścieżkę do grafiki używanej do stworzenia karty, rzadkość, koszt, listę efektów, a w niej dla pojedynczego elementu listy, nazwę, poziom, wartość, czas i listę tagów. Prócz listy efektów pojawia się również lista tagów, z nazwą i wartością w każdym elemencie.

```
<enemy>
    <name>Orc</name>
    <healthMax>50</healthMax>
    <spellDuration>5</spellDuration>
    <cardSkills>
        <card>orkDamage</card>
    </cardSkills>
    <path>CoreGame/EnemyGFX/</path>
    <frameCount>4</frameCount>
</enemy>
```

Rysunek 3.29: Budowa obiektu przeciwnika w pliku XML.



Rysunek 3.30: Schemat graficzny enemies.xml.

W przypadku przeciwników, obiekt w XML jest krótszy od pozostałych, oczywiście ma to swoje odwzorowanie w klasie zawartej w skrypcie. W jednym obiekcie mamy nazwę przeciwnika, jego maksymalne zdrowie, czas odstępu między jego atakami, listę jego umiejętności, a w każdym jej elemencie tylko nazwę, ścieżkę prowadzącą do jego animacji oraz liczbę grafik zawartych w tej animacji, inaczej listę klatek do wyświetlenia.

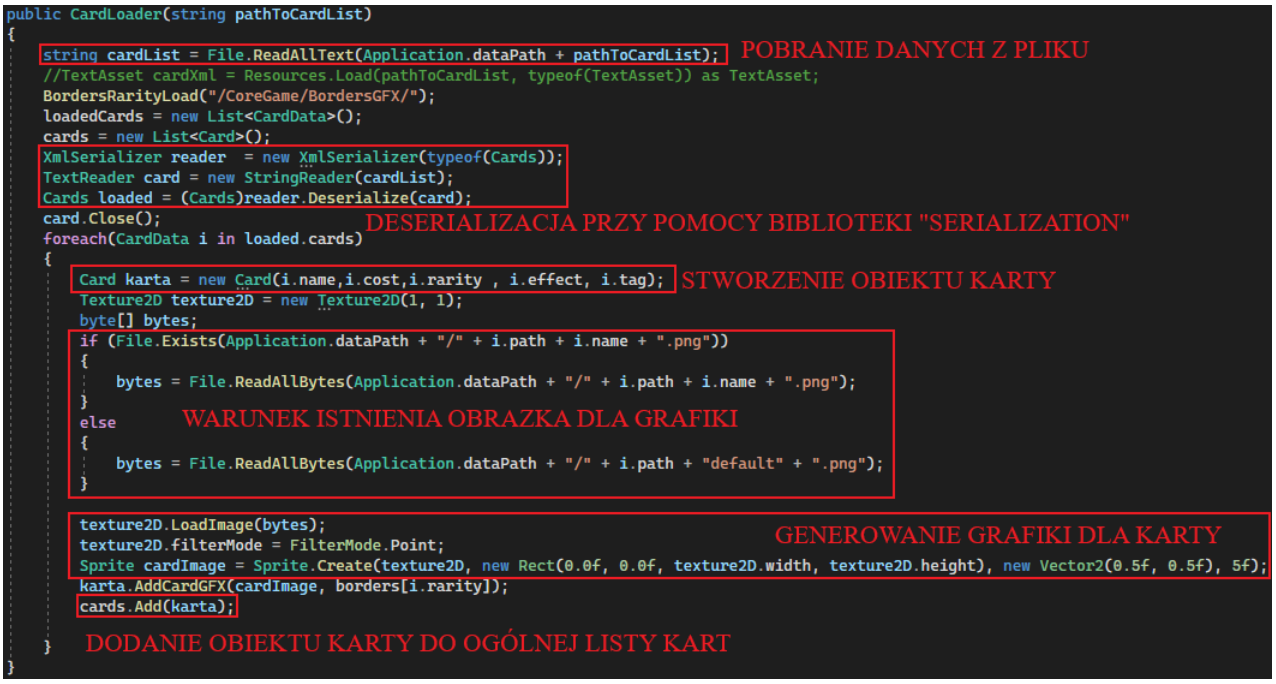
3.9.2. Deserializacja obiektów zawarta w skrypcie

Do deserializacji[4] plików XML użyliśmy biblioteki dostarczonej nam wraz z możliwościami języka programowania C# (`System.Xml.Serialization`), w naszym przypadku przebiega ona w trzech skryptach wczytaniu kart`card.xml`, wczytaniu wydarzeń`event.xml` i wczytaniu przeciwników`enemies.xml`. Samo wczytywanie odbywa się poprzez 3 obiekty gry:

1. `CardLibrary`
2. `EnemyLibrary`
3. `EventLibrary`

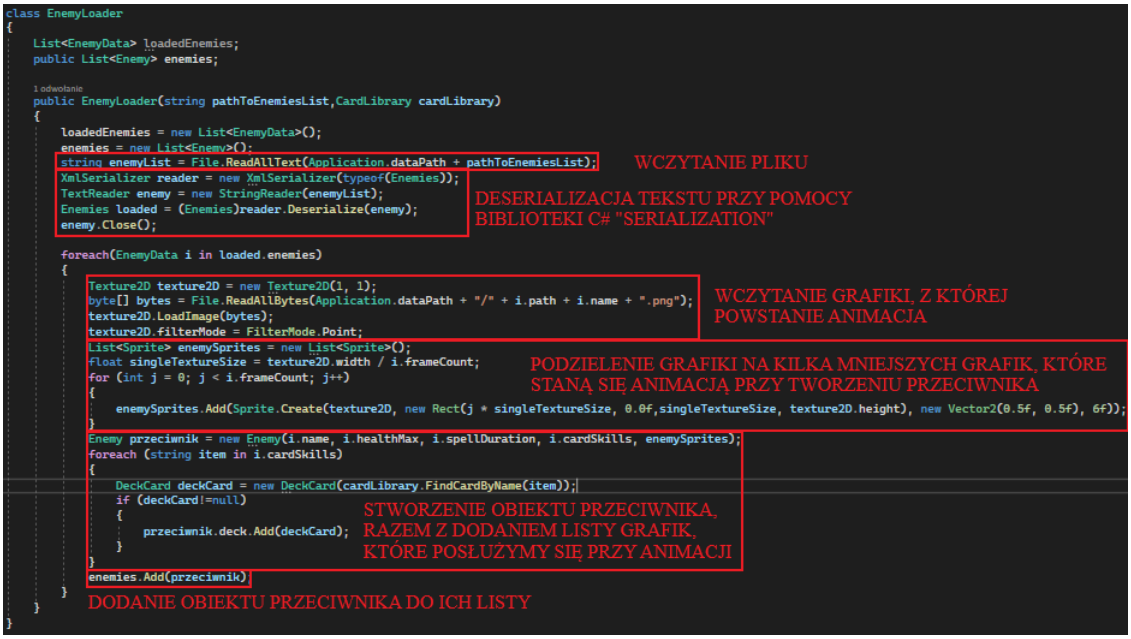
Każdy z nich ma do siebie przypisany, identycznie nazwany, komponent Unity. Dzięki takiemu rozwiązaniu i faktowi, iż te obiekty są tworzone tylko podczas ładowania gry, te obiekty są łatwe do odnalezienia poprzez dostarczoną przez środowisko Unity funkcję `GameObject.Find()`.

Dobrym przykładem działania naszego skryptu jest klasa `"CardLibrary"`, w której to zawiera się publiczny `"CardLoader"`, a to właśnie w nim przy pomocy klasy do deserializacji `CardData` odczytujemy z pliku XML jej obiekty aby kolejno wrzucać je do listy znajdującej się w wrapperze. Wcześniejsza lista, właściwie jej argumenty są odpowiednio preparowane, a w przypadku ścieżki tworzone są od razu z jej pomocą grafiki, i tworzone z ich pomocą nowe obiekty klasy `Card`.



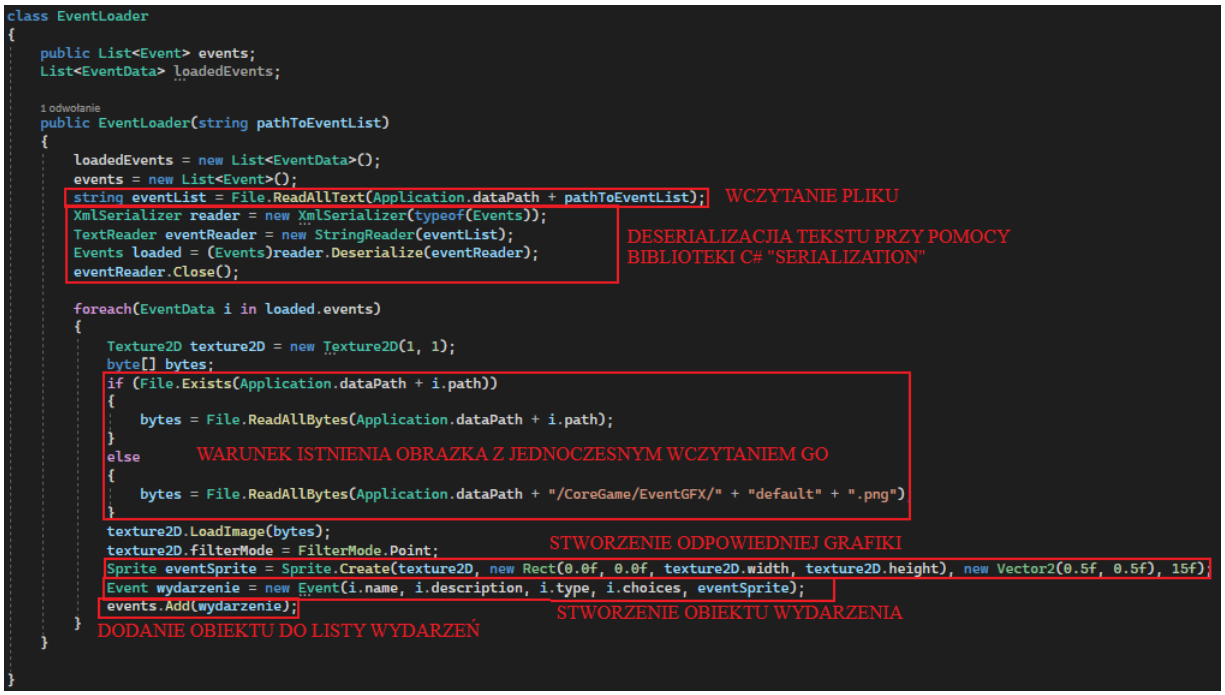
Rysunek 3.31: Budowa w skrypcie CardLoadera.

Kolejnym przykładem tym razem rozbudowanym o dzielenie grafiki na listę mniejszych grafik, które posłużą do animowania postaci jest `EnemyLoader`.



Rysunek 3.32: Budowa w skrypcie EnemyLoadera.

Ostatni przykład deserializacji plików xml w naszym projekcie pojawia się w przypadku `EventLibrary` gdzie to `EventLoader`, który tworzy nam listę obiektów klasy `Event`.



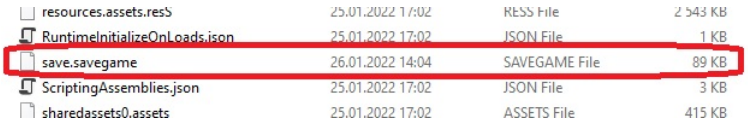
Rysunek 3.33: Budowa w skrypcie EventLoadera.

3.9.3. Zapis i wczytywanie gry

Zapisywanie i wczytywanie gry odbywa się poprzez plik xml nazwany `save.savegame`. Generuje go statyczna klasa `SaveManager` posiadająca funkcje do zapisu, wczytania i usunięcia pliku zapisu. Zapisywanie i wczytywanie gry odbywa się przez serializator xml wbudowany w C# poprzez bibliotekę `System.Xml.Serialization`. Klasa zapisuje tylko najważniejsze, niezbędne do zapisania stanu gry dane, takie jak dane potrzebne do ponownego wygenerowania lokacji, wysokości na mapie dekoracyjnej, oblężone lokacje oraz informacje na temat gracza. Plik zostaje zapisany w folderze `UKRYTATALIA_Data` dołączonym do gry. Stan gry zostaje zapisany za każdym razem gdy gracz ukończy wydarzenia i walkę w danej lokacji.

```
1 reference
public static void SaveGame(
    List<WorldMapNode> worldMap,
    int[,] worldDecoratorArray,
    List<WorldMapNode> siegedLocations,
    PlayerData player,
    WorldMapNode playerLocation)
{
    SaveGameData saveGameData = new SaveGameData(worldMap,worldDecoratorArray,siegedLocations,player,playerLocation);
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(SaveGameData));
    TextWriter writer = new StreamWriter(Application.dataPath + "/save.savegame");
    xmlSerializer.Serialize(writer, saveGameData);
    writer.Close();
}
1 reference
public static SaveGameData LoadGame()
{
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(SaveGameData));
    TextReader reader = new StreamReader(Application.dataPath + "/save.savegame");
    SaveGameData saveGameData = (SaveGameData)xmlSerializer.Deserialize(reader);
    reader.Close();
    return saveGameData;
}
```

Rysunek 3.34: Skrypt służący do wczytywania i zapisywania gry



Rysunek 3.35: Utworzony plik zapisu znajdujący się w folderze `UKRYTA TALIA_Data`

Rozdział 4

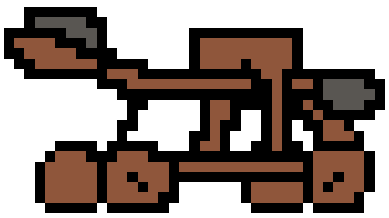
Testowanie

4.1. Zakres osób przystępujących do testowania oraz metodologia

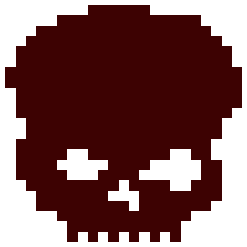
Testowania naszej aplikacji podjęły się osoby dorosłe, zainteresowane tematem gier fabularnych. Osoby przystąpiły do testu czarnej skrzynki, nie znały one kodu gry, miały dostęp jedynie do części wizualnej produktu. Zostali poproszeni o przekazanie swojej opinii na temat gry oraz zgłaszanie odpowiednich bugów.

4.2. Instrukcje wprowadzające do gry dostarczone testerom

Przed rozpoczęciem testowania naszej gry, odpowiednio dobrani testerzy otrzymali niedługą instrukcję dotyczącą rozgrywki: Podczas gry, przy każdej zmienionej lokacji etapy podboju świata pogłębiają się, tym samym nakładając na gracza presję czasu. Na mapie przy odpowiednich lokacjach z czasem rozwoju gry pojawiają się nowe ikony, sygnalizujące kolejno:



(a) Ikona oblężonego miejsca



(b) Ikona podbitego miejsca

W momencie kiedy nasz Kapitol (start) dostanie się w ręce przeciwnika, gra zostaje zakończona. Naszym celem jest dotarcie do ostatniej lokacji wystarczająco silnym by pokonać ostatniego przeciwnika. Nasz bohater ma trzy wartości jakie nas interesują, są to: zdrowie (białe wartości na naszym bohaterze), punkty ruchu (niebieska wartość na górnej części ekranu) oraz posiadane karty. Zdrowie regeneruje się co walkę, a wartość punktów ruchu powiększana jest o jeden za każde trzy sekundy prowadzenia walki, za to do naszej dłoni nowe karty z posiadanej talii dostajemy co pięć sekund. Przeskakując między kolejnymi jeszcze nieodwiedzonymi lokacjami trafiamy na nowe wydarzenia, od których może sporo zależeć w sytuacji naszego gracza.

Często od twojej decyzji zależeć będzie, czy unikniesz walki, czy przebijesz się do finału pozbywając się małej części armii wroga. Do walki posłużysz się magicznymi kartami, określanymi kosztem (niebieski napis na karcie), wartościami obrażeń oraz kilkoma wyjątkowymi kartami nakładającymi specjalne efekty. Nie dajcie im zdobyć Kapitolu i zdobyć tym całego regionu, jego mieszkańce Ciebie potrzebują! Powodzenia!

4.3. Pierwsze wiadomości zwrotne od grupy testerów

Zaledwie po kilku minutach od pierwszego uruchomienia gry wiele osób poinformowało nas o potrzebie opisu w przypadku kart ze specjalnymi efektami. Jakiś czas później dostaliśmy bardziej szczegółowe informację, dowiedzieliśmy się o problemie z balansem i prawie niemożliwym podejściem do finałowego przeciwnika, był on za silny na możliwości jakie daje gra. Gracze duży nacisk przeznaczyli jednak na pozytywny odbiór wizualny aplikacji. Odpowiadało im przedstawienie jej w dany przez nas sposób.

4.4. Zmiany od strony kodu gry

Otrzymane informacje od naszych graczy zmobilizowały nas do działania i reakcji względem niedociągnięć gry, postawiliśmy na balans, utrudnienie walki z podstawowymi przeciwnikami, co zdecydowanie wydłuży grę.

4.5. Zmiany balansu

Gra przez dłuższy okres czasu zmagała się z problemem balansu, przeciwnicy byli zbyt prości, za to ostateczny przeciwnik był zbyt potężny. Problemy się zapętlały i często wracały do początku. Ostatecznie do samych testów trafiła wersja w pełni nie zbalansowana. Dopiero wykonane testy na większym gronie osób pozwoliły odpowiednio ustabilizować możliwości graczy do stawianego poziomu rozgrywki. Dzięki wczytywaniu większości danych z plików XML, modyfikacja balansu była szybka i kończyła się na modyfikacji właśnie zawartych w nich danych. Stąd też walka z balansem opiera się na zmianach siły kart, zdrowia antagonistów oraz ich siły ataku.

4.6. Zmiany jakich byśmy się podjęli w przypadku większej ilości czasu

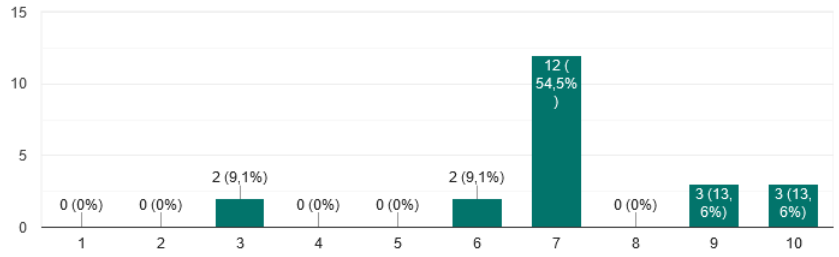
Po dłuższym obcowaniu z grą, można zauważyć potrzebę ingerencji w opisy kart, zwłaszcza w przypadku kart z najwyższych szczebli rzadkości, przez ich specjalne efekty. W przypadku pracy z ograniczonym czasem musieliśmy zostawić produkt mający braki, ale mając ich świadomość w przyszłym czasie zostałyby one zmodyfikowane i poprawione.

4.7. Efekty ankiety wykonanej przez grono testujących

Osoba, która podchodziła do testowania jednocześnie dostawała ankietę stworzoną na `Google Forms`, większość odpowiedzi, prócz nacisku na niekoniecznie idealne UI, była pozytywna.

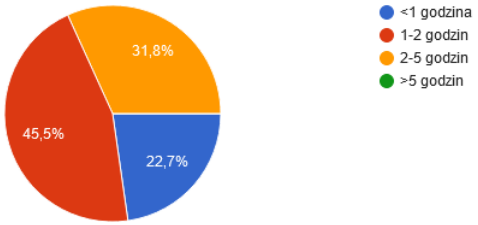
Jak bardzo podobała ci się nasza gra

22 odpowiedzi



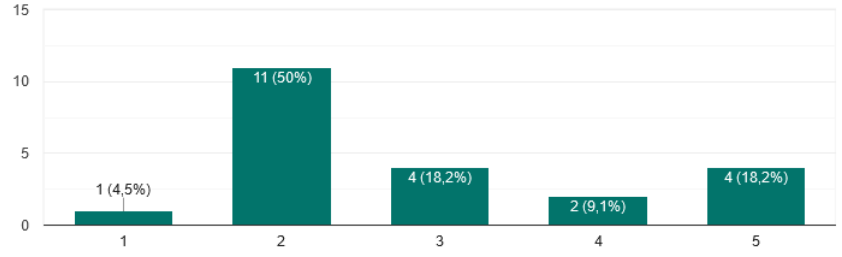
Jak długo zajęło ci wygranie gry

22 odpowiedzi



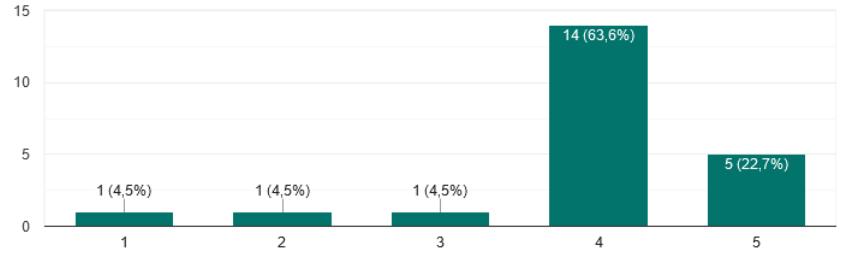
Czy UI zawarte w grze było zrozumiałe

22 odpowiedzi



Czy gra wydawała się uczciwa wobec gracza

22 odpowiedzi



Rozdział 5

Możliwości rozwoju gry

5.1. Dodanie nowych kart,wydarzeń i przeciwników

Obecnie, ze względu na nakład pracy, jaki został wykonany, by gra działała poprawnie, w grze obecnie znajduje się stosunkowo mało kart, zdarzeń i wrogów, więc jednym z najprostszych opcji rozwoju byłoby ich dodanie.

5.2. Rozszerzenie o nowe platformy

Dzięki edytorowi Unity nie dużym kosztem można rozwinąć naszą aplikację o kolejne platformy, dobrym pomysłem jest zbudowanie nowych wersji w postaci aplikacji mobilnych, które na ten moment zrobiły się bardzo popularne.

5.3. Wsparcie dla modyfikacji tzw. Modów

W naszej grze bardzo dużą wagę przyłożyliśmy, aby karty, przeciwnicy i wydarzenia pozwalały na bardzo łatwe i szybkie dodawanie nowych obiektów. Dzięki przechowywaniu wszystkich informacji wewnątrz plików xml dodanie np. nowego przeciwnika sprowadza się do skopiowania wpisu i wypełnienia go nowymi danymi. Takie podejście pozwala w przyszłości na bardzo łatwe dodanie wsparcia dla modyfikacji tworzonych przez użytkowników.

5.4. Powiększenie mapy świata

Mapa świata w naszej grze zależy od statycznych parametrów podanych na stałe w generatorze ze względu na to, że nie mieliśmy potrzeby tworzyć większych map, gdyż w grze obecnie jest za mało unikatowych lokacji i obszarów, by większa mapa miała sens. Powiększenie mapy byłoby jednak bardzo łatwe i zdecydowanie zwiększyło by czas gry i liczby decyzji oraz możliwych strategii gry.

5.5. Dodanie nowych lokacji

Obecnie gra posiada sztywną listę możliwych lokacji, więc dobrym pomysłem na rozwój gry byłoby dodanie możliwości dodawania nowych miejsc lub samo dodanie większej ilości unikatowych miejsc do odwiedzenia przez gracza.

5.6. Dodanie możliwości dodawania nowych efektów

Przy obecnym stanie gry efekty są czymś co wymaga uprzedniego napisania kodu do ich działania. Tę funkcję można by wymienić na możliwość ładowania zewnętrznego kodu lub pisania skryptu który gra jest w stanie interpretować.

Bibliografia

[1] *Komputerowa Gra Fabularna według wikipedii* https://pl.wikipedia.org/wiki/Komputerowa_gra_fabularna *z dnia 25.01.2022*

[2] *Klasyfikacja PEGI* https://pegi.info/pl/node/59 *z dnia 26.01.2022*

[3] *Sceny w Unity* https://docs.unity3d.com/ *z dnia 26.01.2022*

[4] *Serializacja XML w dokumentacji microsoft* https://docs.microsoft.com/pl-pl/dotnet/standard/serialization/examples-of-xml-serialization *z dnia 26.01.2022*

[5] *Animacja sprite w Unity* https://stackoverflow.com/questions/37805594/unity-sprite-two-images-animation-at-runtime *z dnia 26.01.2022*

Spis rysunków