Spis treści

[Streszczenie 5](#_Toc405590031)

[Abstract 6](#_Toc405590032)

[Wstęp 7](#_Toc405590033)

[Opis koncepcji gry – high concept document 8](#_Toc405590034)

[1 Wprowadzenie 10](#_Toc405590035)

[1.1 Wprowadzenie do dziedziny pracy dyplomowej 10](#_Toc405590036)

[1.2 Użyte technologie, narzędzia i inne aplikacje 10](#_Toc405590037)

[1.2.1 Unity oraz środowisko MonoDevelop 10](#_Toc405590038)

[1.2.2 System kontroli wersji 10](#_Toc405590039)

[1.2.3 Narzędzia graficzne 11](#_Toc405590040)

[1.2.4 Komunikatory internetowe 11](#_Toc405590041)

[1.3 Automat komórkowy - gra w życie 11](#_Toc405590042)

[1.4 Proces tworzenia pracy dyplomowej 12](#_Toc405590043)

[1.4.1 Harmonogram prac 12](#_Toc405590044)

[1.4.2 Podział prac w zespole 13](#_Toc405590045)

[2 Projekt sytemu 15](#_Toc405590046)

[2.1 Cel i przeznaczenie systemu 15](#_Toc405590047)

[2.1.1 Założenia i cel tworzenia aplikacji 15](#_Toc405590048)

[2.1.2 Planowany sposób rozpowszechnienia aplikacji 15](#_Toc405590049)

[2.1.3 Ograniczenia dotyczące projektu aplikacji 15](#_Toc405590050)

[2.2 Specyfikacja wymagań użytkowych 16](#_Toc405590051)

[2.2.1 Ogólna charakterystyka systemu 16](#_Toc405590052)

[2.2.2 Zbiór wymagań użytkowych 16](#_Toc405590053)

[2.2.3 Zakres funkcjonalny (wymagania funkcjonalne) 16](#_Toc405590054)

[2.2.4 Środowisko pracy systemu 17](#_Toc405590055)

[2.2.5 Wymagania jakościowe 17](#_Toc405590056)

[2.2.6 Wymagania projektowo-wdrożeniowe 17](#_Toc405590057)

[2.2.7 Kryteria akceptacji 17](#_Toc405590058)

[2.3 Projekt systemu 18](#_Toc405590059)

[2.3.1 Projekt architektury systemu 18](#_Toc405590060)

[2.3.2 Przypadki użycia 18](#_Toc405590061)

[2.3.3 Diagram klas 18](#_Toc405590062)

[2.3.4 Projekt interfejsu użytkownika 18](#_Toc405590063)

[2.3.5 Wybór środowiska implementacji 18](#_Toc405590064)

[2.3.6 Projekt danych 18](#_Toc405590065)

[2.3.7 Rozszerzalność aplikacji 18](#_Toc405590066)

[2.3.8 Wymagania dla generatora map 18](#_Toc405590067)

[2.3.9 Zastosowane algorytmy 19](#_Toc405590068)

[2.3.10 System RPG w grze The Mighty Marian - mechanika postaci 19](#_Toc405590069)

[2.3.11 Ekwipunek 21](#_Toc405590070)

[3 Implementacja 23](#_Toc405590071)

[3.1 Implementacja generatora map 23](#_Toc405590072)

[3.1.1 Etapy procesu generowania mapy 23](#_Toc405590073)

[3.1.2 Efekt końcowy 28](#_Toc405590074)

[3.2 Rozmieszczenie gracza i wrogów 29](#_Toc405590075)

[3.2.1 Wyznaczenie początku i końca poziomu 29](#_Toc405590076)

[3.2.2 Pozycje początkowe wrogów 29](#_Toc405590077)

[3.3 Implementacja umiejętności 29](#_Toc405590078)

[3.3.1 Implementacja Ekwipunku 36](#_Toc405590079)

[3.3.2 Implementacja rodzajów broni 37](#_Toc405590080)

[3.4 Implementacja wrogów 39](#_Toc405590081)

[3.4.1 Rodzaje wrogów 39](#_Toc405590082)

[3.5 Animacja 40](#_Toc405590083)

[3.6 Tryby rozgrywki 40](#_Toc405590084)

[3.6.1 Mini gra 40](#_Toc405590085)

[3.6.2 Standardowy poziom labiryntu 41](#_Toc405590086)

[3.6.3 Poziom walki z bossem 42](#_Toc405590087)

[3.6.4 Poziom finałowy 43](#_Toc405590088)

[4 testowanie, walidacja i weryfikacja 45](#_Toc405590089)

[4.1 Testowanie 45](#_Toc405590090)

[4.2 Walidacja 45](#_Toc405590091)

[4.3 Weryfikacja założeń użytkowych aplikacji 45](#_Toc405590092)

[5 Podsumowanie 46](#_Toc405590093)

[5.1 Podsumowanie i wnioski 46](#_Toc405590094)

[5.2 Możliwość dalszego rozwoju 46](#_Toc405590095)

[6 Generowanie proceduralne grach 46](#_Toc405590096)

[6.1 Geneza 46](#_Toc405590097)

[6.2 Najczęściej używane algorytmy 46](#_Toc405590098)

[6.2.1 Labirynty 46](#_Toc405590099)

[6.2.2 Gra w życie 46](#_Toc405590100)

[6.3 Przykładowe zastosowania 46](#_Toc405590101)

[Wykaz Literatury 48](#_Toc405590102)

[Wykaz rysunków 48](#_Toc405590103)

[Wykaz tabel 48](#_Toc405590104)

[Dodatek A: Plakat gry 48](#_Toc405590105)

# Streszczenie

Niniejsza praca dyplomowa zawiera opis procesu projektowania, implementacji i testowania gry komputerowej z gatunku RPG/Survival.

# Abstract

# Wstęp

Autorzy pracy, przy wyborze tematu, kierowali się swoimi osobistymi preferencjami oraz zainteresowaniami związanymi z dziedziną gier komputerowych, a w szczególności gier RPG i proceduralnego generowania terenu. Wreszcie, taka forma realizacji pracy dyplomowej wydawała się być idealną okazją do sprawdzenia posiadanych umiejętności technicznych w praktyce, a jednocześnie umożliwiała zdobycie nowego doświadczenia w zakresie tworzenia gier komputerowych, współpracy w zespole przy większym projekcie i środowiska Unity.

Celem pracy jest stworzenie gry łączącej w sobie cechy gatunku survival, jak i klasycznego RPG. Do zrealizowania pracy wybrano środowisko Unity 3D, ze względu na to, że w ostatnim czasie zyskuje ono na popularności i liczba tutoriali, poradników, oraz aktywnych for z nim związanych jest bardzo duża. Ponieważ dwie z trzech osób realizujących projekt wcześniej nie miały styczności z silnikami graficznymi 3D, wybór padł na Unity. Porównując je z innymi darmowymi rozwiązaniami dostępnymi na rynku, jak np. UDK, stosunkowo łatwo jest opanować jego podstawy, głównie dzięki mnogości dostępnych materiałów szkoleniowych.

Definicja gatunku gier RPG (Role-playing game) jest bardzo szeroka. Właściwie może to oznaczać dowolną grę, w której gracz wciela się w pojedynczą postać i w miarę upływu czasu rozwija ją. Nie inaczej jest w grze The Mighty Marian. Gracz wciela się w postać tytułowego bohatera i razem z nim próbuje wydostać się z lochów pełnych potworów. Doświadczenie i przedmioty, które Marian uzyskuje zabijając wrogów, pozwalają mu się rozwijać i odblokowywać nowe umiejętności. The Mighty Marian, ze względu na mapy generowane proceduralnie, można określić jako Rouge RPG, ale posiada również elementy gry survivalowej.

Trudno również jawnie określić granice gatunku survival. Zwykle określa się tym mianem gry, w których głównym celem jest pozostanie przy życiu, a inne aktywności, jak walka z wrogami, czy rozwiązywanie zagadek są jedynie środkiem, lub zadaniem dodatkowym. Jedyny waleczny czyn, jakiego bohater musi dokonać, aby wygrać grę, to zabicie ostatniego, finałowego przeciwnika. Marian może przebiegać przez wszystkie poziomy, starając się unikać wrogów, jednak gracz musi pamiętać, że z każdym poziomem wrogowie pojawiają się coraz silniejsi i liczniejsi, a bez zdobywania doświadczenia, Marian nie zyskuje nowych możliwości. Lochy, w których znajduje się bohater, są toksyczne. Ściany pomalowano trującą farbą. Jeżeli bohater pozostanie na danym poziomie dłużej niż jest to absolutnie konieczne, szybko zacznie opadać z sił. Nieprzyjazne środowisko jest kolejną cechą charakterystyczną dla gatunku survival.

Tytułowy bohater, Marian, to zwykły chłopak spragniony chwały. Gra rozpoczyna się w momencie, w którym nasz młody entuzjasta przygód trafia do najgłębszego lochu. Rozgrywa się wtedy najstraszniejsza minuta jego życia. Widzi tor przeszkód, kamienie zawieszone w nicości nad lawą. Próbuje przedostać się na drugą stronę toru, czując, że za każdym razem, kiedy uda mu się dotrzeć trochę dalej, staje się odrobinę silniejszy. Niestety nie jest to możliwe. Jego dążenie do pokonania wszystkich przeszkód i dojścia do domniemanego końca jest skazane na porażkę. Po minucie prób i kilku kąpielach w lawie, Marian przenosi się do głębokich lochów, gdzie jego zadaniem będzie wspinanie się na kolejne poziomy i walka z wieloma potworami. Wiedziony tęsknotą za światłem słonecznym bohater pnie się w górę po zaplątanych, ciemnych lochach. Z każdym pokonanym przeciwnikiem bohater zyskuje nieco doświadczenia, które może zainwestować w wybraną statystykę, siłę, zręczność lub inteligencję. Gra kończy się, kiedy Marian wyjdzie na słońce i pokona ostatniego potwora.

## Opis koncepcji gry – high concept document

*Motywacje gracza:*

Gracz wciela się w postać tytułowego Mariana, który został uwięziony w mrocznych lochach. Jego zadaniem jest wydostanie się na powierzchnię i pokonanie ostatecznego bossa. Podczas rozgrywki bohater, wiedziony tęsknotą za światłem słonecznym, zdobywa nowe przedmioty, odblokowuje umiejętności i zyskuje na sile. Ale wrogowie również nie próżnują, z każdym poziomem są silniejsi i jest ich coraz więcej.

*Gatunek:* Survival/RPG z elementami Rougelike RPG i gry zręcznościowej.

*Licencja:* CC BY-NC-SA 4.0: Attribution – NonCommercial - ShareAlike 4.0 International*, czyli* Uznanie autorstwa – Użycie niekomercyjne – Na tych samych warunkach 4.0 Międzynarodowe.

*Potencjalni odbiorcy:* Gra skierowana jest raczej do starszych odbiorców, powyżej czternastego roku życia, ze względu na wysoki poziom trudności i bardzo dynamiczną rozgrywkę. Ponieważ grafika gry utrzymana jest raczej w mrocznej i ciemnej kolorystyce, która nie będzie odpowiednia dla młodszych odbiorców. Mini gra, od której odbiorca zaczyna swoje doświadczenie z The Mighty Marian, nawiązuje to klasycznej produkcji Quake 3, co może pozostać niezauważone przez graczy młodszych. W grze pojawia się przemoc i dużo krwi.

*Sytuacja rynkowa – podobne rozwiązania:* Obecnie na rynku pojawia się ogromna ilość produkcji niezależnych, znanych równierz jako Indie. Jedną z pierwszysch gier niezależnych, która niejako rozpoczęła modę na gry Indie był wydany w 2011 roku Minecraft, stworzona przez niewielkie studio produkcja której jednym z głównych założeń jest otwarty, generowany proceduralnie świat. Gra posiada też tryb Survival, który wymaga, by postac gracza przetrwała w nieprzyjaznym jej świecie. O sukcesie tego pomysło z całą pewnością świadczy fakt, iż firma Mojang, wraz z grą, została zakupiona przez Microsoft za 2,5 miliarda dolarów we wrześniu 2014 roku. Według danych z lutego 2014 sprzedano łącznie ponad 35 milionów kopi gry.[TODO wiki]

Podobną do naszej produkcji grą jest Spelunky, wydana w 2008 roku platformówka, której sposób generowania kolejnych poziomów z podziałem na pokoje, przedstawiony w prezentacji dostępnej pod linkiem [5], był inspiracją dla hybrydowego algorytmu generowania terenu w grze The Migthy Marian. Jednak gra ta jest grą platformową 2D, a nasza produkcja będzie w 3D.

W 2011 roku została wydana kolejna produkcja na której się wzorujemy: The Binding Of Isaac, gra niezależna z gatunku Roguelike. Spotkała się ona z bardzo pozytywnym odbiorem, do kwietnia 2013 rozeszła się w dwóch milionach kopii. [TODO wiki] Naszą produkcję odróżnia generowanie poziomów, w Binding of Isaac mapa jest złożona z predefiniowanych pokoi, w The Mighty Marian każdy korytarz jest generowany proceduralnie

Obecnie gry typu Roguelike i Survival RPG przeżywają drugą młodość, a liczba sprzedanych kopii wzrasta z dnia na dzień. [TODO http://uk.ign.com/articles/2014/07/04/roguelikes-the-rebirth-of-the-counterculture]

*Cechy wyróżniające produkt*

* Mapy generowane proceduralnie
* Permadeath – brak zapisu gry
* Wysoki poziom trudności

*Platformy sprzętowe:* Unity umożliwia zbudowanie projektu na wiele platform. Sterowanie odbywa się za pomocą klawiatury i myszki, zatem wspierane platformy to komputery PC z systemami operacyjnymi Windows i Linux.

*Cele implementacyjne*

* Krótka, bardzo dynamiczna rozgrywka
* Mapy generowane proceduralnie, gra za każdym razem jest inna
* Niewielkie mapy
* Mechanizm nagłej śmierci poganiający gracza
* Brak zapisu gry, śmierć oznacza konieczność zaczęcia od początku

# Wprowadzenie

## Wprowadzenie do dziedziny pracy dyplomowej

## Użyte technologie, narzędzia i inne aplikacje

### Unity oraz środowisko MonoDevelop

Przy tak małym zespole projektowym i rozmiarze projektu, oczywistym wydaje się użycie gotowego silnika gry to zestaw modułów odpowiedzialnych między innymi za wyświetlanie grafiki, obsługę wejścia, odtwarzanie dźwięku czy symulację zjawisk fizycznych, jak grawitacja czy kolizje pomiędzy obiektami. Elementy te są uniwersalne i stworzone tak, aby można je było wielokrotnie wykorzystać w różnych projektach. Implementacja własnego silnika gier być może ma swoje plusy, jak optymalizacja pod konkretny projekt czy implementacja tylko tych funkcjonalności, które są konieczne. Jednak ze względu na ograniczoną ilość czasu na realizację projektu zdecydowaliśmy się użyć gotowego rozwiązania.

Wybór Unity był przemyślaną decyzją. W kategorii darmowych dla małych projektów, zaawansowanych narzędzi do tworzenia gier, jego jedyną konkurencją może być UDK. Zdecydowaliśmy się jednak wykorzystać Unity, ponieważ posiada bogatą bazę materiałów dostępnych w Internecie i wielu aktywnych użytkowników, którzy chętnie udzielają odpowiedzi na forach. Unity w darmowej wersji udostępnia okrojony zestaw funkcjonalności w porównaniu z wersją Pro, jednak na potrzeby tego projektu okazał się wystarczający.

Unity to zintegrowane środowisko do tworzenia gier i innych materiałów multimedialnych. Wspierane języki programowania to C#, UnityScript oraz Boo, o składni podobnej do Pythona. Do realizacji projektu został wybrany język C#, ponieważ zespół ma z nim najwięcej doświadczenia.

Dostarczane wraz z Unity środowisko deweloperskie MonoDevelop okazało się, być w wersji darmowej, pozbawione wielu wygodnych funkcjonalności, do których przyzwyczaja użytkowników Visual Studio. Zatem aby nie tracić czasu na zapoznawanie się z nowym środowiskiem, część projektu związana z kodem realizowana była przy użyciu Visual Studio 2012 lub 2013 wraz z zainstalowaną darmową wtyczką UnityVS, która umożliwia debugowanie skryptów Unity.

### System kontroli wersji

Na początku realizacji projektu używaliśmy systemu kontroli wersji SVN z darmowym repozytorium na serwerach code.google. Do wprowadzania zmian służył nam program TortoiseSVN. Rozwiązanie to miało wiele wad, było mało intuicyjne i generowało dużo problemów przy scalaniu konfliktów.

Mniej więcej w połowie listopada repozytorium projektu zostało przeniesione na GitHub. Po etapie zapoznawania się z nowym narzędziem, zostało ono zaakceptowane przez zespół. Bardziej intuicyjny interfejs oraz P4Merge wpłynęły pozytywnie na tempo prac i zniwelowały problemy związane z kontrolą wersji.

### Narzędzia graficzne

Część grafik, które zostały wykorzystane w grze, została znaleziona w Internecie jako darmowa do użytku niekomercyjnego. Pozostałe zostały stworzone przez członków zespołu na potrzeby projektu. Do tego celu użyto programów Paint.NET oraz IrfanView.

Paint.NET to darmowy program graficzny, który obsługuje przeźroczystość oraz warstwy. Jest dobrym narzędziem do tworzenia grafik w stylu pixel art. IrfanView zawiera szereg ciekawych filtrów graficznych i pozwala na szybką i intuicyjną edycję obrazów.

### Komunikatory internetowe

Pomimo tego, że członkowie zespołu widują się często na uczelni, cześć komunikacji odbywała się drogą internetową. Najczęściej do tego celu służyła konferencja w komunikatorze Skype oraz wiadomości e-mail.

## Automat komórkowy - gra w życie

W projekcie The Mighty Marian do generowania map został użyty automat komórkowy. Automat komórkowy o odpowiednich parametrach i kryteriach przeżycia komórek pozwala w niewielu krokach wygenerować ciekawe struktury podobne do jaskiń. Niestety to rozwiązanie ma również swoje ograniczenia.

Automat komórkowy to model matematyczny, w którym komórki znajdują się w jednym z określonych stanów. System składa się z pojedynczych komórek, znajdujących się obok siebie. Każda z komórek może przyjąć jeden ze stanów, przy czym liczba stanów jest skończona. Plansza, na której znajdują się komórki może być w dowolnej skończonej liczbie wymiarów.

Inicjalnie, w czasie t = 0, każda z komórek znajduje się w jednym z możliwych stanów. Ich stan w czasie t+1, nowa generacja komórek, określony jest pewną funkcją matematyczną, zwykle zależną od stanu jej sąsiadek.

Automaty komórkowe można podzielić na wiele rodzajów. Jedną z cech je odróżniających jest wykorzystywany rodzaj sąsiedztwa. Sąsiedztwo możemy definiować dowolnie, jednak istnieją dwa podstawowe rodzaje sąsiedztwa, Von Neumanna i Moora. Na rysunku 2.2 zaprezentowano te podstawowe rodzaje dla sąsiedztwa na promienia *r=1* i *r=2*.

|  |  |  |  |
| --- | --- | --- | --- |
| a) | b) | c) | d) |
|  |  |  |  |
| Rysunek 2.1 Różne rodzaje sąsiedztwa  a) sąsiedztwo Neumanna dla r=1 b) Neumana dla r=2 c) sąsiedztwo Moora dla r=1 d) Moora dla r=2 | | | |

Na potrzeby tej pracy rozpatrywać będziemy automaty komórkowe w dwóch wymiarach, w których komórki mogą przyjąć jeden z dwóch stanów {żywa, martwa}. Oczywiście, automaty komórkowe mają o wiele więcej możliwych zastosowań i są używane chociażby przy symulowaniu ewolucji czy proceduralnym generowaniu tekstur.

## Proces tworzenia pracy dyplomowej

### Harmonogram prac

| Tabela 1.1 Harmonogram prac | |
| --- | --- |
| Harmonogram prac | |
| Data | Podjęte działania |
| 10 marca 2014 | Wybór tematu pracy inżynierskiej |
| Marzec – Czerwiec | Przedmiot Realizacja Projektu Informatycznego, gdzie naszym tematem była Gra The Migty Marian |
| 20 marca | Złożenie deklaracji przystąpienia do projektu |
| Marzec | * Rozpoczęcie pracy nad algorytmem do generowania map * Zebranie pierwszych grafik koniecznych do stworzenia gry |
| Pierwsza połowa lipca | Implementacja demo generatora map w Windows Forms C#, algorytm okazał się nieodpowiedni do potrzeb projektu, pomysł odrzucono. |
| Druga połowa lipca | * Pierwsza scena stworzona w Unity * Poruszanie bohaterem po mapie |
| Wrzesień | * Dodanie klasy enemy * Implementacja pierwszej wersji generatora map opartego o automat komórkowy (stworzenie klasy Map) |
| Pierwsza połowa października | * Stworzenie tekstur podłogi, nicości, wody oraz trawy. * Zapis wygenerowanej mapy do pliku graficznego. * Pierwsza implementacja wrogów podążających za bohaterem oraz kamery. |
| Druga połowa października | * Kolejne wersje skryptów dla wrogów i kamery. * Wrogowie odsuwają się od ścian i powiadamiają się nawzajem * Pierwsza wersja intro do gry z kamerą najeżdżającą na stół * Połączenie automatu komórkowego z labiryntem (stworzenie klas Maze, Room) * Optymalizacje i udoskonalenia w klasie Map * Dodanie efektów czarów do projektu |
| Pierwszy tydzień listopada 1-7 | * Dodanie ścian * Stworzenie klasy Projectile * Implementacja atakujących przeciwników * Implementacja ataku dla bohatera, klasa Attack * Dodanie klasy repezentującej bron Weapon * Wykorzystanie systemu inventory * Stworzenie prototypu gry * Początek implementacji drzewka umiejętności * Dodanie drabiny umożlwiającej przejścia między poziomami * Implementacja filtra erozyjnego, poszerzenie korytarzy |
| Drugi tydzień listopada 8-14 | * Drabina i Marian są ustawiani na przeciwnych końcach labiryntu * Wejście bohaterem w drabinę powoduje przejście między poziomami * Wrogowie krwawią i rozpadają się na kawałki * Wrogowie stają się różnorodni, pojawiają się kolorowe duchy i animacje * Coś z broniami tu się dzieje, nie wiem co * No i pojawiają się nowe czary z tego co widze, tu tez trzeba uzupelnic * Zmiana repozytorium na GitHub |
| Trzeci tydzień listopada 15-21 | * Implementacja walki mieczem * Dodanie nowych wrogów – pająków * Implementacja zaklęć erupcja i piorun * Dodanie trybu mapy do walki z bossem * Krawędzie ekranu stają się krwiste kiedy Marian przyjmuje obrażenia * Dodano menu pomocy * Poprawa licznych błędów i drobne optymalizacje |
| Czwarty tydzień listopada 22-31 | * Implementacja shadera, który działa z animacją, przeźroczystością i dobrze przyjmuje światło * Dodanie muzyki, która zmienia się zależnie od sytuacji w grze * Implementacja systemu ekwipunku albo raczej przyznaj się skad go mamy * Po zabiciu wroga jest szansa na zdobycie broni * Implementacja poziomu finałowego * Implementacja kolejnych umiejętności, przyspieszenia i wzmocnienia * Dodanie dwóch nowych rodzajów wrogów * Implementacja mini-gry na początku rozgrywki * Tworzenie dokumentacji |
| Początek grudnia | * Uzupełnianie dokumentacji * Finalne testowanie i doskonalenie aplikacji * Praca nad balansem i grywalnością |

Tabela 1.1 przedstawia harmonogram prac wykonanych przy projekcie The Mighty Marian. Oczywiście wiele podjętych działań, które nie przekładały się bezpośrednio na funkcjonalność w grze lub były związane z poprawianiem błędów, drobnymi optymalizacjami, zdobywaniem wiedzy, czy scalaniem modułów razem nie zostało wylistowanych.

### Podział prac w zespole

|  |  |
| --- | --- |
| Tabela 1.2 Podział prac w zespole | |
| Tobiasz Biernacki | * Kamera podążająca za bohaterem * Sztuczna inteligencja wrogów * Poruszanie się bohatera * Implementacja mini-gry * Efekty czarów i zdolności * Dodanie dynamicznej muzyki |
| Krzysztof Jasiak | * System ekwipunku * Początkowa animacja i menu * Balans i grywalność * Muzyka w menu głównym |
| Dominika Sokołowska | * Algorytmy i impelementacja generacji terenu * Rozmieszczenie wrogów na mapie * Implementacja przejścia pomiędzy poziomami * Animacja wrogów i bohatera |

W tabeli 1.2 wypisano ogólnikowo podział pracy w zespole. Projekt realizowano w grupie trzyosobowej, w atmosferze ścisłej, przyjacielskiej współpracy. Ogólna zasada panująca w projekcie to, że za jakość i poprawność poszczególnych modułów odpowiada osoba, która je zaimplementowała. Role w projekcie zostały przyjęte naturalnie, zgodnie z umiejętnościami i zainteresowaniami danej osoby.

# Projekt sytemu

## Cel i przeznaczenie systemu

### Założenia i cel tworzenia aplikacji

Celem naszego projektu jest stworzenie gry przy użyciu silnika Unity, która łączyć będzie elementy survivalu oraz klasycznego RPG. Planujemy stworzyć produkt, który zapewni użytkownikowi wiele godzin niezapomnianej rozrywki i wyjątkową grywalność. Grafika będzie łączyć w sobie elementy 2D i 3D. Głównym elementem naszej gry, który ma znacząco poprawić grywalność i wydłużyć czas przyjemnej i pełnej niespodzianek rozgrywki, jest stworzenie map generowanych losowo.

Docelowa grupa wiekowa

### Planowany sposób rozpowszechnienia aplikacji

Unity udostępnia technologię Unity Web Player, która umożliwia uruchamianie gry w oknie przeglądarki internetowej, jeśli tylko użytkownik ma zainstalowaną odpowiednią wtyczkę. Unity nawet udostępnia generator, który produkuje gotową stronę HTML i odpowiedni kod JavaScript z osadzanym obiektem gry. W przyszłości planujemy stworzyć stronę internetową, na której będzie można zapoznać się z dokumentacją projektu, zagrać w grę, a dla użytkowników, którzy nie będą zainteresowani pobieraniem wtyczki Unity Web Player zostanie udostępniona do pobrania za darmo paczka z grą The Mighty Marian, w wersji na system operacyjny Microsoft Windows i Linux. Obecnie jednak przeszkodą są koszty związane z utrzymaniem serwera.

Ponieważ ten temat projektu został wybrany głównie w celu zdobycia cennego doświadczenia ze środowiskiem Unity i tworzeniem gier komputerowych, a nie stworzenia produktu komercyjnego, produkt końcowy dystrybułowany będzie głownie lokalnie, dla osób zainteresowanych, choć będzie również dostępny do pobrania w Internecie.

### Ograniczenia dotyczące projektu aplikacji

W związku z brakiem grafika w zespole, projekt nie będzie skupiał się na elementach graficznych. Postanowiliśmy w miarę możliwości wykorzystywać gotowe elementy graficzne. Dodatkowo jest to nasz pierwszy projekt tworzony w Unity, przez co część założeń uległa zmianie w miarę poznawania możliwości i ograniczeń tego środowiska.

Projekt jest testowany przez grupę naszych znajomych, przez co testy mogą być niepełne.

W Unity kod jest bardzo silnie powiązany z obiektami w grze, więc diagram klas będzie diagramem obiektów i klas, gdyż tylko w ten sposób można pokazać zależności między nimi

## Specyfikacja wymagań użytkowych

### Ogólna charakterystyka systemu

Gra jest produkcją z pogranicza gatunków Survival, RPG. Stosuje proceduralną generację terenu, a główny nacisk położony będzie na rozgrywkę. Dzięki generowanemu terenowi każde podejście do produkcji oferuje inne doznania. Produkcja jest stworzona w technologii Unity, dzięki czemu posiada zapewnione przez środowisko skalowanie grafiki, w zależności od sprzętu na którym zostanie uruchomiona. Dzięki technologii Unity Web Player można umieścić ją w sieci. Przejście gry powinno zajmować około 30-40 minut, Gra ma działać na systemach operacyjnych wspieranych przez Unity, testowana jest pod systemem MS Windows.

### Zbiór wymagań użytkowych

Gra powinna działać na systemie Windows 7 lub nowszym, jak i na systemach Linux wspieranych przez środowisko Unity.

Sterowanie w grze powinno odbywać się przy pomocy klawiatury i myszki, w możliwie naturalny sposób.

Gra uruchamia się przez standardowy program startowy Unity. Wybór rozdzielczości i jakości efektów powinien mieć odzwierciedlenie w faktycznym wyglądzie gry, jednak powinno być to zapewnione przez Unity.

### Zakres funkcjonalny (wymagania funkcjonalne)

Gracz ma możliwość poruszania się po świecie gry. Kamera powinna pokazywać świat w rzucie izometrycznym lub podobnym. Kamera powinna porusza się za bohaterem. Świat powinien być generowany proceduralnie. Gra będzie posiadać poziomy, celem bohatera jest dojście do ostatniego z nich, ich ilość będzie losowa. W wygenerowanym świecie pojawiają się wrogowie, są oni rozmieszczani losowo, w ilości zależnej od obecnego poziomu gry. Gra powinna posiadać elementy gry RPG: postać gracza powinna być opisana statystykami, których zwiększanie powinno odpowiednio modyfikować rozgrywkę. Postać ma posiadać umiejętności, które gracz będzie mógł aktywować by zyskać przewagę w walce. Umiejętności powinny być odpowiednio zbalansowane. Dodatkowo w grze powinien być zaimplementowany system ekwipunku. Przedmioty powinny odpowiednio modyfikować statystyki postaci. Gra może posiadać mini gry które będą znacząco odbiegać od reszty rozgrywki.

### Środowisko pracy systemu

Dzięki zastosowanemu silnikowi gra jest między-platformowa i można ją uruchomić na najbardziej popularnych systemach: Windows, Linux i Macintosh. Jako że Mac jest najmniej popularnym systemem z powyższej trójki, a także z racji braku maszyny z tym systemem na której moglibyśmy przetestować grę, nasza gra działać będzie na systemie Windows 7 i nowszym, oraz na najpopularniejszych dystrybucjach systemu Linux. Dodatkowo dzięki darmowej wtyczce Unity Web Player gra może działać w przeglądarce. Obecnie Unity Web Player jest kompatybilny z nowymi wersjami przeglądarek Internet Explorer, Firefox, Chrome, Safari, Opera, a wtyczka do instalacji potrzebuje systemu operacyjnego Windows XP/Vista/7/8.

### Wymagania jakościowe

Gra powinna być utrzymana w stylistyce 8-bitowej, więc grafika nie musi być najwyższej jakości. Gra powinna być stabilna, nie zawieszać się, powinna być zapewniona obsługa błędów.

### Wymagania projektowo-wdrożeniowe

Projekt aplikacji powinien być gotowy do 15 października.

Wersja alfa gry, która zostanie rozesłana do wybranej grupy osób, zainteresowanych jej przetestowaniem powinna być gotowa do 31 października.

Wersja beta powinna być gotowa do 15 listopada. Wersja beta powinna już posiadać zaimplementowane wszystkie funkcjonalności gry. Wersja ta ponownie zostanie wysłana do testerów.

Aplikacja powinna być wykonana w terminie do 2 grudnia, dokumentacja powinna być gotowa do 5 grudnia.

Program powinien zajmować mniej niż 0.5 GB miejsca na dysku, dzięki czemu będzie szybki w pobraniu, oraz jego ładowanie przez Unity Web Player zajmie mniej czasu.

## Projekt systemu

### Wymagania dla generatora map

Celem algorytmu jest wygenerowanie mapy, która będąc wystarczająco skomplikowaną, aby gracz mógł się w niej zgubić i jednocześnie spójną, tak, aby wszystkie komórki podłogi, z których składa się mapa były osiągalne przez gracza. Drugi warunek jest konieczny do spełnienia, ponieważ gdyby postać Mariana i drabina umożliwiająca przejście między poziomami gry zostały umieszczone w innych „składowych spójności” mapy, to skończenie gry byłoby niemożliwe.

To, czy mapa jest spójna (nie istnieją w niej komórki podłogi, do których nie da się dotrzeć z każdej innej komórki podłogi) da się stosunkowo prosto sprawdzić algorytmicznie, to określenie stopnia skomplikowania i zawiłości korytarzy jest już zadaniem, którego w ramach tego projektu nie podjęliśmy się zrealizować. Zdecydowaliśmy się określać w początkowym etapie skomplikowane korytarze i traktować je, jako formę, na której działają kolejne kroki algorytmu. Daje nam to gwarancję mapy o pożądanym poziomie skomplikowania i podzielonej na logiczne obszary. Ten proces zostanie dokładniej opisany w dalszej części dokumentu.

|  |  |
| --- | --- |
| a) | b) |
|  |  |
| *Rysunek 2.1: Spójna mapa o niewielkim poziomie skomplikowania. Otwarty pokój, w którym gracz nie będzie miał czego odkrywać* | *Rysunek 2.2: Mapa niespójna, ale ciekawa pod względem grywalności* |

### System RPG w grze The Mighty Marian - mechanika postaci

#### Statystyki opisujące postać

Główny bohater jest opisywany następującymi statystykami: siła, zręczność i inteligencja. Manipulacja nimi wpływa na parametry opisujące postać: punkty życia, manę, regenerację many, szybkość i obrażenia. Dodatkowo postać posiada poziom, punkty doświadczenia i punkty umiejętności.

#### Zdobywanie poziomów

By zdobyć poziom główny bohater musi mieć odpowiednią liczbę punktów doświadczenia, obliczaną według wzoru: Ilość punktów doświadczenia potrzebna by zdobyć następny poziom doświadczenia = 100 \* poprzedni poziom doświadczenia

|  |  |
| --- | --- |
| Poziom | Łączna ilość punktów doświadczenia |
| 1  2  3  4  5  6  7  8  9  ... | 0  100  300  600  1000  1500  2100  2800  3600  ... |
| Tabela 3.1: Tabela doświadczenia | |

Zdobycie każdego poziomu daje bohaterowi 5 punktów umiejętności które może wykorzystać na zwiększenie jednej ze statystyk opisujących postać: siły, zręczności i inteligencji. Dodatkowo zdobycie poziomu dodaje 20 punktów maksymalnego życia, zwraca 20 punktów życia, oraz dodaje 20 punktów maksymalnej many i przywraca 20 many.

#### Zdobywanie statystyk

Punkty umiejętności można wykorzystać do zwiększenia statystyk. Każdy punkt statystyki zwiększa parametry postaci:

Każdy punkt siły dodaje punkt obrażeń z broni białej, zwiększa maksymalny poziom punktów życia o pięć i przywraca pięć punktów życia.

Punkty zręczności zwiększają obrażenia z broni dystansowej, oraz zwiększają prędkość z jaką postać porusza się po mapie.

Inteligencja zwiększa obrażenia z broni magicznej, zwiększa maksymalną pulę many, przywraca część obecnej many oraz przyspiesza jej regeneracje. Dodatkowo zwiększa punkty obrażeń zadawane przez czary.

#### Zdobywanie umiejętności

W miarę zdobywania statystyk, bohater zyskuje dostęp do umiejętności aktywnych. Postać ma pięć miejsc na umiejętności aktywne, są one uzupełniane pierwszą piątką odblokowanych umiejętności.

|  |  |  |  |
| --- | --- | --- | --- |
| Ilość punktów danej statystyki | Siła | Zręczność | Inteligencja |
| 10 | Wzmocnienie | Przyspieszenie | Erupcja |
| 20 | Szał | Leczenie | Wiązka błyskawic |
| 30 | Ogłuszający krzyk | Przebijający strzał | Kula ognia |
| Tabela 3.2: Tabela umiejętności | | | |

Wzmocnienie: Na dwie sekundy podwaja obrażenia zadawane bronią białą.

Szał: Przez dwie sekundy postać odzyskuje życie równe sile jej ataku broniom białą.

Ogłuszający krzyk: ogłusza przeciwników w otoczeniu bohatera na pięć sekund.

Przyspieszenie: Przez krótki okres czasu postać porusza się szybciej

Leczenie: Przywraca postaci część punktów życia.

Przebijający strzał: Postać wypuszcza wiązkę energii która zadaje poważne obrażenia każdemu przeciwnikowi którego trafi.

Erupcja: Główny bohater przy pomocy swojej umiejętności manipulacji magią powoduje lokalną erupcję lawy która zadaje obrażenia każdemu przeciwnikowi który znajdzie się na jej obszarze.

Wiązka błyskawic: Przez krótki czas podstawowy atak postaci zamienia się w wiązkę błyskawic, która zadaje obrażenia każdemu przeciwnikowi który zostanie nią trafiony.

Kula ognia: Bohater wystrzeliwuje ognisty pocisk który eksploduje po dotarciu do celu, lub po kolizji z pierwszym napotkanym obiektem, zadając poważne obrażenia wszystkim przeciwnikom w polu rażenia.

#### Czasy odnowień

Umiejętności posiadają czasy odnowienia, które są zależne od tego jak wielką przewagę dają w rozgrywce. Dodatkowo czary leczące mają dodatkowo zwiększony czas odnowy, by utrudnić rozgrywkę. Jest to kolejny parametr służący do balansowania gry

#### Obrażenia

Zadawane obrażenia są zależne od rodzaju broni jakiej używa postać gracza, jak i od odpowiedniej statystyki powiązanej z tą bronią. Bazowo bohater zadaje dziesięć obrażeń pojedynczym atakiem, jednak znajdowanie lepszej broni i zwiększanie umiejętności podnosi ta wartość.

W grze są trzy rodzaje przedmiotów: miecz, strzała i różdżka. Każdy wróg po śmierci ma 7.5% szans na upuszczenie jednego z tych przedmiotów. Każdy przedmiot ma inną wartość obrażeń.

### Ekwipunek

W grze będą trzy rodzaje przedmiotów:

Miecz zadaje obrażenia każdemu wrogowi którego trafi, działa na krótkim zasięgu. Jego obrażenia zależą od siły

Strzały zadają obrażenia pierwszemu trafionemu wrogowi, ich obrażenia zależą od wartości zręczności naszego bohatera.

Różdżka zadaje obrażenia na dystansie pierwszemu wrogowi którego trafi, obrażenia zależą od wartości inteligencji głównej postaci. Każdy atak kosztuje punkty many, więc może dojść do sytuacji gdy nasz bohater nie będzie mógł zaatakować, dlatego ważne jest posiadanie dodatkowej broni w ekwipunku.

# Implementacja

## Postać gracza

Tytułowy Marian jest w grze przedstawiany jako gladiator w hełmie z czerwonym pióropuszem. Grafikę przedstawiono na rysunku 3.1.

|  |
| --- |
|  |
| Rysunek 3.1: Grafika głównego bohatera pochodzi ze źródła [6] i udostępniona jest nieodpłatnie do użytku niekomercyjnego |

## Świat gry

Świat, po którym w głównej mierze porusza się bohater to lochy. Marian przemierza zimne i ciemne korytarze, których ściany pokryte są skruszałą cegłą. Za każdym rogiem czekają na niego nowi wrogowie. Otoczenie jest nieprzyjazne, a jeśli za długo zostanie się na jednym poziomie, również toksyczne. Marzeniem Mariana jest wydostać się na powierzchnię i spędzić resztę lat wypoczywając na tropikalnej wyspie. Jednak jest to cel bardzo trudny i ryzykowny do osiągnięcia.

Świat gry składa się z płytek ścian, podłogi i „sufitu”. Na podstawie wygenerowanego kształtu mapy płytki są odpowiednio ustawiane. Celowo w grze jest niewiele światła, aby podnieść poziom trudności i zmusić gracza do korzystania z czaru światło, dostępnego pod przyciskiem Q.

## Implementacja generatora map

W naszym projekcie każda rozgrywka polega na przejściu kilku, dziesięciu do piętnastu, poziomów jaskiń. Każda z jaskiń generowana jest niezależnie, przy użyciu algorytmu stworzonego na potrzeby projektu. Generowany obszar, pomimo tego, że w grze wizualnie reprezentowany jest w trzech wymiarach, na etapie generacji traktujemy jak dwuwymiarowy. Mapa składa się z płytek, zwanych także komórkami, które mogą przyjmować dwie wartości, są podłogą lub nicością. Bohater i wrogowie mogą przebywać i poruszać się jedynie po komórkach podłogi.

Komórki podłogi na rysunkach reprezentowane będą przez jaśniejsze pola, natomiast komórki, po których postacie nie mogą się poruszać, kolorem ciemnym.

### Etapy procesu generowania mapy

Proces powstawania map można podzielić na pięć etapów. Produkt końcowy każdego z etapów jest danymi wejściowymi dla kolejnego etapu.

#### Etap labiryntu

Proces generowania mapy rozpoczyna się od stworzenia labiryntu, który określi czy pomiędzy wybranymi pokojami występuje połączenie. Parametrami istotnymi w tej fazie jest liczba pokojów, jakie chcemy uzyskać. W poniższych przykładach i implementacji w projekcie wybrana liczba to 4 w wymiarze X i 4 w wymiarze Y, więc 16 pokojów. Oczywiście, metoda działa prawidłowo również dla innej liczby pokojów.

Na tym etapie mapę modelujemy za pomocą grafu prostego ważonego w następujący sposób:

*G(V,E) - graf nieskierowany*

*V* - zbiór wierzchołków - pojedynczy wierzchołek reprezentuje jeden pokój

*|V|= 4 \* 4*

E - zbiór krawędzi - krawędź reprezentuje przejście między pokojami

Program generuje początkowy graf przejść między pokojami. Inicjalnie wszystkie możliwe przejścia między sąsiednimi pokojami istnieją.

|  |
| --- |
|  |
| Rysunek 3.1: Początkowy graf połączeń między pokojami |

W tak zamodelowanej przestrzeni wygenerowanie labiryntu łączącego pokoje sprowadza się do znalezienia minimalnego drzewa spinającego w grafie G. Drzewo spinające grafu jest grafem spójnym i acyklicznym, który zawiera wszystkie wierzchołki grafu oraz niektóre z jego krawędzi. Minimalne drzewo spinające jest drzewem spinającym, którego suma wag krawędzi jest najmniejsza ze wszystkich pozostałych drzew rozpinających danego grafu.. W danym grafie może istnieć może istnieć więcej niż jedno drzewo o tych własnościach. Z punktu widzenia grywalności nie ma znaczenia które wybierzemy, zatem wystarczy wskazać jedno z nich, a wagi krawędziom grafu możemy przypisać losowo. Do uzyskania minimalnego drzewa spinającego został użyty został algorytm Prima, wybrany ze względu na łatwość implementacji.

|  |
| --- |
|  |
| Rysunek 3.2: Graf przejść z przyporządkowanymi losowymi wagami na krawędziach |

Krawędziom grafu G przyporządkowane zostają losowe wagi z zakresu (3,660). Wierzchołek startowy dla algorytmu również jest wybierany losowo.

Algorytm Prima oparty jest o metodę zachłanną. Można opisać go następująco:

1. Rozpoczynamy od grafu składającego się jedynie z wierzchołka startowego.
2. Krawędzie incydentne do wierzchołka umieszczamy na posortowanej wg. wag liście.
3. Zdejmujemy z listy krawędź o najmniejszej wadze i sprawdzamy, czy łączy wierzchołek wybrany z niewybranym. Jeśli tak, to znalezioną krawędź dodajemy do drzewa spinającego.
4. Dodajemy krawędzie incedentne z nowo wybranym wierzchołkiem do posortowanej listy.
5. Powtarzamy kroki 2 - 4 dopóki lista krawędzi nie będzie pusta.

|  |  |
| --- | --- |
| a) | b) |
|  |  |
| Rysunek 3.3: Labirynt uzyskany po zastosowaniu algorytmu Prima na grafie przejść między pokojami | Rysunek 3.4: Przykładowa mapa możliwa do wygenerowania dla uzyskanego labiryntu |

#### Położenie przejść między pokojami

W określonym w poprzednim etapie drzewie spinającym wagi krawędzi zastępujemy losowo wartościami ze zbioru {1, 4}. Ta wartość określa, w którym miejscu pomiędzy pokojami utworzone zostanie przejście. Przejście to prostokąt o szerokości *rozmiar pokoju/4* i długości dwóch komórek. Przejścia są obliczane i umieszczane osobno w każdym pokoju. Dzięki wprowadzeniu różnorodności w położeniu przejścia, pomimo tego, że są one wszystkie tych samych rozmiarów, generowane mapy zyskują nieco na poziomie skomplikowania i dzięki temu urozmaicają rozgrywkę. Takie rozwiązanie pomaga ukryć przed graczem fakt, że poziom, po którym się porusza jest zwykłym labiryntem. Na rysunkach 2.7 i 2.8 porównano przykładową mapę z przejściami w tych samych miejscach w pokoju oraz mapę z czterostopniową różnorodnością w położeniu przejścia.

|  |  |
| --- | --- |
| a) | b) |
|  |  |
| Rysunek 3.5: Przykładowa mapa, w której przejścia między pokojami wygenerowano na szerokości oznaczonej przez wartość 4 | Rysunek 3.6: Mapa, gdzie przejścia między pokojami występują losowo na szerokościach oznaczanych przez wartości {1,4} |

Już cztery stopnie różnorodności przejść tworzą wrażenie różnorodności i pomagają ukryć przed graczem to, że porusza się po prostu po labiryncie. Komórki oznaczone, jako należące do przejścia są zapisywane i wykorzystywane w kolejnym etapie.

#### Etap pokoju

Etap pokoju jest decydujący dla kluczowy dla definicji ostatecznego kształtu korytarzy na mapie. Każdy z pokojów generowany jest osobno, przy pomocy automatu komórkowego. Po każdej iteracji algorytmu komórki należące do przejścia, określonego w poprzedniej fazie stają się podłogą. Na początku plansza pokoju wypełniania jest komórkami nicości. Komórki nieleżące na krawędzi planszy pokoju z prawdopodobieństwem 0,51 zamieniane są w komórki podłogi. Prawdopodobieństwo początkowe zostało wybrane eksperymentalnie, taka wartość daje najciekawsze rezultaty.

Automat komórkowy działa przez sześć iteracji. Żywa komórka, na rysunkach oznaczana kolorem czarnym, to nicość. Komórki jasne oznaczają podłogę. Pierwsze cztery to iteracje tworzące automatu o dość ciekawych regułach przejść między stanami. Jeżeli w sąsiedztwie Moora o promieniu 2 liczba sąsiadów wynosi mniej niż 3 to komórka jest żywa. Jest również żywa, jeśli w sąsiedztwie Moora o promieniu 1 liczba sąsiadów wynosi 5678. W przeciwnym razie komórka zostaje podłogą. Po czterech iteracjach parametry automatu zmieniają się na 5678/5678 z sąsiedztwem Moora o promieniu 1. Zatem pierwsze trzy iteracje sprzyjają powstawaniu nowych komórek w miejscach opustoszałych i zapobiegają powstawaniu map nieciekawych i pustych. Pozostałe iterację pełnią rolę wygładzającą.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
| Rysunek 3.8: Obrazy przedstawiają przebieg formowania się przykładowego pokoju. W lewym górnym rogu znajduje się stan początkowy. Pozostałe obrazy przedstawiają wygląd mapy po kolejnych iteracjach. W dolnym rzędzie znajduje się wynik działania kolejnych iteracji wygładzających | | | |

Po tym następuje sprawdzenie, czy uzyskany układ komórek jest spójny, to znaczy czy wszystkie komórki podłogi są połączone. Sprawdzane jest również, czy komórki podłogi stanowią co najmniej 30% całkowitej powierzchni pokoju. Jeżeli dane ułożenie komórek nie spełnia któregokolwiek z tych warunków rozwiązanie jest odrzucane i proces generowania rozpoczyna się od początku, z prawdopodobieństwem początkowym wystąpienia podłogi większym o 0,1. Proces powtarza się, aż nie powstanie plan pokoju spełniający wspomniane warunki. Dzięki zwiększanej początkowej ilości podłóg prawdopodobieństwo tego, że gracz będzie długo czekał na wygenerowanie mapy, zmniejsza się. Takie zwiększanie prawdopodobieństwa umożliwia również generowanie nie tylko map, gdzie kształt pokoju jest od razu widoczny a komórki podłogi stanowią znaczy procent powierzchni całkowitej, ale również pokojów o kształcie wąskich korytarzy, sterując wspomnianym prawdopodobieństwem początkowym, bez obawy o zbyt długi czas oczekiwania na rozwiązanie.

#### Etap łączenia i wygładzania

Kiedy wszystkie pokoje są wygenerowane, w klasie Map przepisywane są w odpowiednich położeniach do tablicy smallMap. Tak powstała mapa przepisywana jest do tablicy o wymiarach dwukrotnie większych niż podane na wejściu. Każda komórka w tabeli smallMap reprezentowana jest przez cztery komórki w tabeli map. Na tak przygotowanej planszy wykonywana jest jedna iteracja automatu komórkowego o parametrach 5678/5678 w celu wygładzenia powiększonej mapy.

|  |  |
| --- | --- |
| a) | b) |
|  |  |
| Rysunek 2.10: Porównanie mapy po przeskalowaniu przed i po wygładzeniu realizowanym w metodzie CellularSmooth() klasy Map. | |

Jak można zaobserwować na rysunku 2.10 automat komórkowy zrealizowany w tym etapie zachowuje ogólny kształt mapy jednocześnie wygładzając krawędzie.

Mamy gwarancję, że przejścia istniejące w mapie reprezentowanej przez tabele smallMap, po powiększeniu i wygładzeniu nie zamkną się, ale nie wiemy, czy nie zostaną zwężone na tyle, że postać Mariana nie będzie mogła się przez nie przecisnąć.

#### Etap erozji

Ponieważ szerokość bohatera jest większa niż szerokość jednej komórki i koliduje on ze ścianami kołem o promieniu 1,5 komórki spójność mapy nie gwarantuje, że będzie ona grywalna. Wszystkie przejścia muszą być, co najmniej o szerokości trzech komórek, aby gracz mógł się przez nie przecisnąć.

Ponieważ nie znaleźliśmy stosunkowo łatwego w implementacji sposobu, aby upewnić się, że wszystkie przesmyki na mapie mają szerokość co najmniej trzech komórek, zdecydowaliśmy się w końcowym etapie poszerzyć wszystkie korytarze prewencyjnie. Ponieważ komórki mapy mogą należeć jedynie do dwóch kategorii, są albo podłogą, albo nicością, to mapę możemy potraktować jak obraz binarny. Poszerzenie korytarzy realizowane jest dzięki cyfrowemu przetwarzaniu obrazów binarnych, przy pomocy filtra erozyjnego.

Niestety, porównując rysunki 2.9 i 2.10 możemy zaobserwować, że zastosowanie filtra erozyjnego ma również swoje wady. Mapa na rysunku 2.10 ma ostrzejsze krawędzie i jest bardziej otwarta. Ponieważ lokalizacja miejsca zwężenia nie jest trywialna i nie została zaimplementowana, filtr nie poszerza jedynie miejsca zwężenia, a ma wpływ na kształt całej mapy.

|  |  |
| --- | --- |
| a) | b) |
|  |  |
| Rysunek 3.7: Mapa przed zastosowaniem filtra erozyjnego. Można zaobserwować przejście o szerokości dwóch komórek, przez które bohater się nie przeciśnie | Rysunek 3.8: Mapa z rysunku 2.9 po zastosowaniu filtru erozyjnego. Problematycznie wąskie przejście zwiększyło swoją szerokość do czterech komórek. |

### Efekt końcowy

Mapy generowane przez zaimplementowany w projekcie algorytm są różnorodne i zapewniają ciekawą rozgrywkę pomimo prostoty graficznej. W grze występują również inne rodzaje map, składające się z jednego pokoju, które występują na poziomach z bossem i są pewnym uogólnieniem stosowanych tutaj zasad. W dokumentacji opisano najbardziej skomplikowany przypadek. Na poniższym rysunku 2.11 widnieje kilka map wygenerowanych przez program. Wszystkie komórki podłogi są osiągalne przez bohatera, mapy są podzielone na pokoje, co pozwala w łatwy sposób równomiernie rozłożyć wrogów.

|  |  |
| --- | --- |
| a) | b) |
|  |  |
| c) | d) |
|  |  |
| Rysunek 2.11: Cztery przykładowe mapy wygenerowane przez program | |

## Rozmieszczenie gracza i wrogów

### Wyznaczenie początku i końca poziomu

Ponieważ generowana mapa ma strukturę labiryntu i znany jest graf przejść między pokojami, za pomocą algorytmu Floyda-Warshalla możemy wyznaczyć długość ścieżki pomiędzy każdymi dwoma pokojami. Pomimo tego, że algorytm Floyda-Warshalla ma złożoność n3 i istnieją lepsze pod tym względem rozwiązania tego problemu, ponieważ graf przejść ma jedynie szesnaście wierzchołków, nie warto jest inwestować w implementację bardziej skomplikowanego algorytmu.

Znając długości ścieżek pomiędzy pokojami, wybieramy te dwa pokoje, pomiędzy którymi odległość ta jest największa i jeden z nich wybieramy jako start i w tym pokoju, na losowej komórce umieszczany jest Marian, a drugi pokój jako koniec i tam umieszczana jest drabina.

### Pozycje początkowe wrogów

W każdym pokoju, oprócz pokoju startowego, rozmieszczani są wrogowie. Ich ilość jest zależna od poziomu na którym znajduje się gracz. Większą szansę wystąpienia mają wrogowie słabsi, których siła płynie z liczności, mniejszą wrogowie, którzy w pojedynkę stanowią wyzwanie dla Mariana. Wrogowie rozmieszczani są osobno w każdym pokoju, pojawiają się na losowych komórkach podłogi. Gdyby rozprowadzać wrogów po prostu losowo po całej mapie, istniałaby możliwość, że rozłożenie wrogów po mapie byłoby bardzo nierównomierne, i mogłyby wystąpić miejsca, gdzie liczba wrogów jest za duża i gra jest za trudna. Zastosowane rozwiązanie jest związane z próbkowaniem w przestrzeni podzielonej na prostokąty, a problem rozłożenia wrogów równomiernie na mapie jest związany z problemem próbkowania losowego, tylko występuje w mniejszej skali. Źródło 1.

## Implementacja umiejętności

### Drzewko umiejętności

|  |
| --- |
|  |
| Rysunek 3.1 |

### Wzmocnienie

|  |
| --- |
|  |
| Rysunek 3.2 |

### Szał

|  |
| --- |
|  |
| Rysunek 3.3 |

### Ogłuszający krzyk

|  |
| --- |
|  |
| *Rysunek 3.4* |

### Przyspieszenie

|  |
| --- |
|  |
| *Rysunek 3.5* |

### Leczenie

|  |
| --- |
|  |
| *Rysunek 3.6* |

### Przebijający strzał

|  |
| --- |
|  |
| *Rysunek 3.7* |

### Erupcja

|  |
| --- |
|  |
| *Rysunek 3.8* |

### Wiązka błyskawic

|  |
| --- |
|  |
| *Rysunek 3.9* |

### Kula ognia

|  |
| --- |
|  |
| *Rysunek 3.10* |

### Implementacja Ekwipunku

### W naszej produkcji zdecydowaliśmy się zastosować gotowy system ekwipunku. Na stronie Unity Asset Store [TODO link] jest dostępny gotowy system, bardzo łatwy w implementacji. Inventory System [TODO link] stworzony przez Brackeys[TODO link] jest najlepszym darmowym rozwiązaniem, niestety z racji braku oficjalnej wersji która była by napisana w C#, zdecydowaliśmy się na zaimportowanie najnowszej wersji (1.2.2) opartej o JavaScript. Wiązało się to z problemami z komunikacją między kodem napisanym w JavaScripcie, a kodem C#. Unity wymaga by w wypadku odwołań między różnymi językami, jeden z nich był skompilowany wcześniej [TODO link]. Niestety zastosowanie tego modułu przeszkodziło w implementacji niektórych funkcjonalności systemu ekwipunku, takich jak pokazywanie wartości obrażeń po najechaniu na przedmiot.

Screeny ekwipunku

|  |
| --- |
|  |
| *Rysunek 3.11* |

### Implementacja rodzajów broni

|  |
| --- |
|  |
| *Rysunek 3.12: Miecz do walki wręcz* |
| Rysunek 3.13: Strzały do walki na dystans | |
|  | |
|  | |
| *Rysunek 3.14: Magiczne pociski do walki na dystans* | |

## Implementacja wrogów

### Sztuczna inteligencja wrogów

Aby zacząć pracę nad szeroko pojętym zachowaniem wrogów postanowiliśmy zastanowić się najpierw jak to robili profesjonaliści z dużym doświadczeniem. Dość oczywistym wyborem było zajrzenie do kodu źródłowego Quake 2. Jest on publicznie dostępny już od 22 grudnia 2001 roku. Poza tym kod pochodzący z id Software będący dziełem John’a Carmack’a jest znany jako jeden z najpiękniejszych kodów. Prawdą jest, że Quake 2 bardzo różni się od naszej gry z wielu powodów, z czego najważniejszym jest fakt, że są to dwa zupełnie odrębne gatunki. Jednak mimo innej perspektywy i typu rozgrywki od zachowania przeciwników oczekujemy niemal tego samego, czego oczekiwalibyśmy w wypadku gry FPS.

Przeglądając kod zauważyliśmy, że przeciwnicy są sterowani różnymi flagami oznaczającymi ich stan i w zależności od stanu wykonują różne czynności takie jak szukanie, gonienie i atakowanie bohatera, czy po prostu stanie w miejscu. Na ich stan mają wpływ bodźce z otoczenia od tak oczywistych jak ujrzenie głównego bohatera, po mniej oczywiste, jak na przykład zaobserwowanie innego zachowania sąsiednich sojuszników.

Podobnie też zostało to zrealizowane przez nas. Domyślnie wrogowie są w stanie spoczynku, który ulega zmianie jeśli wróg ten zostanie zaatakowany, zobaczy głównego bohatera, lub innego wroga, który goni, lub atakuje tytułowego Mariana. W zależności od spowodowało wyjście ze stanu spoczynku, dana jednostka może rozpocząć różne czynności. W wypadku zobaczenia Mariana będzie za nim podążać, lub atakować go jeśli jest w zasięgu ataku. Jeśli zobaczy sojuszniczą jednostkę znającą położenie głównego bohatera będzie podążać w jej kierunku. Natomiast jeśli zostanie zaatakowana będzie podążać w kierunku miejsca z którego zostały zadane obrażenia, chyba że miejsce to nie będzie widoczne z jej perspektywy – wtedy przejdzie w stan zaalarmowany, w którym będzie losowo przeszukiwać otoczenie.

Jednostki podejmują próbę ataku tylko jeśli są w zasięgu charakterystycznym dla rodzaju wrogów, który reprezentują. W podobny sposób ograniczony jest ich zasięg widzenia. Jednostka będąca za daleko by atakować, lecz wystarczająco blisko, aby widzieć Mariana będzie go gonić. Jeśli ten opuści pole widzenia, jednostka będzie podążać do miejsca, w którym był on widziany po raz ostatni. Gdy dojdzie w pobliże tego punkty wciąż nie wiedząc gdzie znajduje się Marian, będzie próbować iść jego śladami. Jednak aby ucieczka przed wrogami była możliwa algorytm, według którego poruszają się wrogowie idąc jego śladami jest napisany w taki sposób, by trop prowadzący do protagonisty był szybko gubiony. Tak też się przeważnie dzieje, po czym dany wróg przechodzi w stan zaalarmowany.

Aby sprawdzić czy dany wróg widzi Mariana sprawdzane jest czy promień od danego wroga do Mariana przecina jakiś obiekt, który mógłby zasłonić widok. Takim obiektem może być ściana, natomiast inni wrogowie, lub latające w powietrzu magiczne pociski nie są w tym wypadku brane pod uwagę. Oczywiście zanim kosztowne wydajnościowo obliczenia promieni zostaną zastosowane najpierw sprawdzona zostaje odległość między wrogą jednostką, a bohaterem. Jeśli jest za duża dalsze obliczenia są zbędne. Podobna technika jest stosowane w wielu innych przypadkach takich jak sprawdzenie czy wróg widzący Mariana może poinformować o tym swoich sąsiednich sojuszników.

Poza tym zaimplementowany jest też prosty mechanizm, dzięki któremu jednostki nie skupiają się grupami w jednym punkcie, a zamiast tego gdy jest ich dużo mają tendencję do tworzenia formacji przypominającej tłum.

U przeciwników występują trzy rodzaje ataków. Jeden z nich to atak wręcz, którego działanie jest trywialne. Kiedy przeciwnik walczący wręcz jest w zasięgu ataku zadaje obrażenia co określony czas. Kolejnym, również trywialnym sposobem ataku są ataki samobójcze stosowane przez jeden z rodzajów wrogów, które na zetknięciu się z głównym bohaterem eksplodują zadając obrażenia. Ostatnim i najciekawszym pod względem algorytmicznym sposobem ataku przeciwników jest atak dystansowy. Jest on konfigurowalny kilkoma parametrami. Prócz uniwersalnych takich jak częstotliwość ataków, czy wartość obrażeń są też parametry definiujące prędkość magicznych pocisków, które wykorzystuje atak dystansowy, jak również sposób określenia kierunku w którym mają one lecieć. Przeciwnicy w przybliżony sposób przewidują gdzie należy wystrzelić pocisk by trafić w Mariana na podstawie jego prędkości i kierunku poruszania się. Wspomniane powyżej parametry definiują z jakim prawdopodobieństwem ma następować to przewidywanie i jaki wpływ na ostateczny kierunek lotu pocisków ma aktualna pozycja Mariana, oraz ta która została przewidziana. Poza tym jest jeszcze parametr określający rozrzut pocisków wycelowanych w Mariana.

Opisane powyżej algorytmy dają pozytywne wrażenia z gry, ponieważ jednostki zachowują się dość naturalnie reagując na zdarzenia w grze i informując się nawzajem, co powoduje ich grupowanie się gdy gra jest na bardziej zaawansowanym etapie i zagęszczenie wrogów jest większe. Ich ataki cechują się wysoką różnorodnością, dzięki czemu gracz, aby uniknąć obrażeń musi skorzystać z wielu różnych technik, w zależności od szybko zmieniającej się sytuacji.

### Rodzaje wrogów

#### Duchy

Duchy to słaby przeciwnik, którego siła pochodzi z tego, że jest ich bardzo dużo. Ma wysokie prawdopodobieństwo pojawienia się. Występuje w trzech kolorach, jak na rysunku 5.1, które różnią się od siebie wytrzymałością na obrażenia. Duchy atakują dystansowo.

|  |
| --- |
|  |
| Rysunek 5.1 Grafika duchów. Źródło: [2] |

#### Pająki

Pająki występują z mniejszym prawdopodobieństwem niż duchy. Ich ataki są silne i bezpośrednie. Na rysunku 5.2 zaprezentowano ich cykl poruszania się. Podczas wykonywania ataku bezpośredniego wrogowie roztaczają dookoła siebie pomarańczowe koła. Grafika pająka jest bardzo ciemna, więc przeciwnik jest praktycznie nie do zauważenia dopóki nie zacznie atakować gracza.

|  |
| --- |
|  |
| Rysunek 5.2 Animacja chodzącego pająka, grafikę udostępniono na licencji CC-sa-3.0, umożliwiającej wykorzystanie nawet do celów komercyjnych pod warunkiem podania źródła. Źródło: [4] |

#### Tanki

Ich rolą jest przyjmowanie dużej ilości obrażeń. Są najbardziej wytrzymałymi ze wszystkich wrogów. Atakują bezpośrednio. Występują ze stosunkowo niskim prawdopodobieństwem. Ich wygląd przedstawiono na rysunku 5.3.

|  |
| --- |
|  |
| Rysunek 5.3 Grafika tanka została stworzona specjalnie na potrzeby gry The Mighty Marian |

#### Zielone kamikadze

Wróg o bardzo niewielkiej wytrzymałości, przedstawiony na rysunku 5.4, zainspirowany stworem baneling z gry Starcraft 2. Kiedy zauważy bohatera wbiegaja w niego z dużą prędkością. Przy zderzeniu z Marianem wybucha i zadaje obrażenia.

|  |
| --- |
|  |
| Rysunek 5.4 Ten pozornie uroczy stworek potrafi zrobić dużą krzywdę |

#### Boss

Boss jest większą i o wiele silniejszą wersją tanka. Oba te stwory mają zbliżoną grafikę, jednak boss ma bardziej bojowe barwy, rysunek 5.5. Jest jedynym przeciwnikiem spotykanym na etapie finałowym i wcześniejszych poziomach specjalnych. Jego rozmiar jest o wiele większy niż zwykłego wroga, żeby podkreślić jego siłę.

|  |
| --- |
|  |
| Rysunek 5.5: Grafika bossa pojawiającego się w grze. Podobnie jak grafika tanka również została stworzona na potrzeby projektu |

## Animacja

Do animacji wrogów użyto operacji odpowiedniego obracania powierzchni, na której rysowany jest sprite (zestaw obrazów przedstawiających kolejne klatki animacji postaci). Takie rozwiązanie daje iluzję ruchu nawet dla statycznych obrazów. W przypadku wrogów, których animacja ma więcej niż jedną klatkę zaimplementowano technikę animacji polegającą na przesuwaniu offsetu tekstury, opisaną szerzej w źródle [3].

## Tryby rozgrywki

### Mini gra

Swoją przygodę z grą The Mighty Marian, użytkownik rozpoczyna w trybie mini gry. Trwające minutę wyzwanie polega na doskoczeniu jak najdalej po planszach unoszących się nad lawą. Im dalej gracz doskoczy, tym większy bonus doświadczenia otrzyma na start właściwej rozgrywki. Rysunek 5.6 przedstawia zrzut ekranu podczas początkowych chwil rozgrywki. Po minucie gracz przenoszony jest do standardowego trybu gry.

|  |
| --- |
|  |
| Rysunek 5.6 Zrzut ekranu z mini gry w The Mighty Marian. Obrazy są wyostrzone, rozjaśnione i ze zmniejszonym kontrastem, żeby w wydrukowanej wersji pracy były czytelne. |

#### Strafe-jumping

Najważniejszym elementem mini-gry jest niestandardowy sposób poruszania się inspirowany tym znanym z Quake 3 Arena. Jego założenia mogą wydawać się dziwne, jednak po nabraniu wprawy są bardzo przyjemnym elementem urozmaicającym grę. Charakteryzuje się tym, że mimo limitu prędkości, po osiągnięciu którego przestaje się przyśpieszać w kierunku, w którym ta prędkość jest skierowana, można wciąż przyśpieszać w innym kierunku. Dodatkowo ważnym jest fakt, że tarcie podczas poruszania się w powietrzu nie występuje, a skacząc tuż po zetknięciu z ziemią pomija się tarcie z ziemią. W efekcie możliwe jest ciągłe przyśpieszanie jeśli grasz skacze i rytmicznie zmienia kierunek wektora przyśpieszenia, co osiąga się zgrywając ruchy myszą z kierunkiem wybieranym klawiaturą. Technika ta jest znana jako „strafe-jumping”.

Nasza implementacja znacznie różni się od tej z gry id Software, jednak wrażenia z poruszania się są dość podobne. W naszym projekcie zgranie ruchów myszy i klawiatury jest nieco łatwiejsze. Jeśli kierunek odczytany z wejścia myszy i klawiatury jest taki sam, możliwe jest uzyskanie dodatkowego przyśpieszenia w locie, jednak to bonusowe przyśpieszenie działa tylko przez pierwszy ułamek sekundy po wyskoku, aby tak ograniczyć jego działanie do racjonalnych wartości. Siła tarcia jest znacznie zwiększana jeśli gracz nie wybiera kierunku poruszania się klawiaturą, tak aby ten był w stanie szybko zahamować. Skok zostanie wykonany automatycznie po dotknięciu ziemi jeśli przycisk skoku został naciśnięty tuż przed tym momentem. Ma to na celu uniknięcie spowolnienia poprzez tarcie. Podobnie jeśli skoczymy tuż po zetknięciu z ziemią również nie stracimy prędkości, ponieważ tarcie zaczyna działać dopiero po krótkiej chwili od wylądowania. Aby odróżnić dotykanie ziemi od dotykania ścian w dolnej części postaci został dodany dodatkowy „collider”, a więc wykrywacz zderzeń, który ustawiony jest jako tak zwany „trigger”, dzięki czemu nie wpływa na obliczenia fizyki a jedynie informuje o tym, że jakiś obiekt zawiera się w jego przestrzeni.

Ogólnie rzecz biorąc skok odbywa się poprzez zmianę prędkości w pionie, nie jest to jednak tak trywialne jak można by się spodziewać, ponieważ zmiany te muszą być różne w wypadku gdy postać stoi w miejscu, porusza się pod górkę, lub odwrotnie. Za opadanie odpowiada napisana przez nas grawitacja, która po prostu zmniejsza prędkość postaci w górę, gdy ta nie dotyka podłoża.

Dość istotnym elementem jest też fakt, że w locie sterowność jest znacznie ograniczona i nie da się zmienić kierunku poruszania się tak łatwo jak podczas chodzenia po ziemi. Mimo to korzystanie z techniki zwanej „strafe-jump” opisanej powyżej początkowo powodowało dość radykalne zmiany kierunku poruszania się w chwili kontaktu z ziemią, w związku z czym algorytm sterujący poruszaniem został wzbogacony o kolejne warunki ograniczające sterowność w sytuacji, gdy lądujemy na ziemi tylko po to, by szybko znów podskoczyć. W efekcie możliwe jest rozwijanie coraz większych prędkości w stabilnym kierunku, natomiast jeśli nie to jest naszym celem również nic nie przeszkadza nam w doborze dowolnego toru ruchu.

Niektóre z opisanych powyżej funkcjonalności dałoby się osiągnąć za pomocą gotowych mechanizmów fizycznych takich jak grawitacja, czy tarcie. Jednak takie rozwiązanie ogranicza wolność w modyfikowaniu. W grze ponad realizm są istotne wrażenia z gry, zatem nie wszystko można zapisać używając praw fizyki imitujących rzeczywistość.

#### Rocket jumping

Kolejną techniką nietypowego poruszania się we wspomnianej grze, która była inspiracją do stworzenia opisywanej w tym punkcie mini gry jest tak zwany „rocket jumping”. Istotą tej techniki jest wykorzystanie siły odrzutu eksplozji rakiety do zyskania dodatkowego przyśpieszenia. Zwykle w tym celu strzela się rakietą pod własne nogi, aby dzięki temu móc skoczyć wyżej i dalej.

W naszej grze główną różnicą jest fakt, że zamiast rakiety jest wybuchowa kula ognia, która jednak w przeciwieństwie do rakiety nie zadaje obrażeń głównemu bohaterowi. Efekt działania jest bardzo podobny, czyli magiczna kula ognia po wystrzeleniu porusza się ze stałą prędkością w linii prostej, a na kontakcie z obiektami eksploduje powodując odrzut. Aby jednak uniknąć problemów z niezbyt dokładnym obliczaniem kolizji szybko poruszających się obiektów w momencie wystrzelenia ognistego pocisku jest symulowany promień. Jeśli przetnie on jakiś obiekt zostanie zapisana pozycja tego przecięcia, oraz czas w którym magiczna kula znajdzie się w tym miejscu. Czas ten jest obliczany na podstawie prędkości, aktualnego czasu, oraz odległości między punktem przecięcia, a źródłem promienia, czyli miejscem z którego kula ognia zaczyna swoją drogę. Następnie w momencie wyznaczonym obliczeniami opisanymi powyżej Marianowi zostanie nadana prędkość w wyniku odrzutu, jeśli był on wystarczająco blisko eksplozji. Kierunek tego impulsowego przyśpieszenia też oczywiście jest konkretnie obliczany na podstawie pozycji bohatera, oraz eksplozji.

Przedstawiony model fizyki nadaje atrakcyjność mini grze i z pewnością jest przyjemnym elementem dla bardziej wprawionych graczy, natomiast w wypadku mniej doświadczonych osób może sprawiać wrażenie trudnego do nauczenia, w związku z czym mini gra jest traktowana tylko jako startowy bonus i dobre wyniki w niej nie są konieczne do czerpania radości z całej gry.

### Standardowy poziom labiryntu

Generowany za pomocą opisanego algorytmu zestaw korytarzy przedstawiany jest w trójwymiarowej postaci. W tej przestrzeni umieszczani są wrogowie i bohater. Zadanie gracza polega na znalezieniu drabiny do kolejnego poziomu i rozwinięciu postaci.

|  |
| --- |
|  |
| Rysunek 5.7: Zrzut ekranu ze standardowego tryb rozgrywki. Marian ma tu już odblokowaną jedną umiejętność. |

#### Mechanizm nagłej śmierci

Mechanizm ten ma na celu wymuszenie tempa rozgrywki. Po przeminięciu określonego czasu obliczonego na podstawie długości drogi ze startu danego poziomu do jego końca nagła śmierć zostaje uaktywniona, co powoduje powolne zagęszczanie się trującego, zielonego dymu na poziomie. Dym ten zadaje coraz większe obrażenia, przez co gracz musi szybko znaleźć wyjście z poziomu, ponieważ w przeciwnym razie przegra.

|  |
| --- |
|  |
| Rysunek 5.8 Efekt nagłej śmierci |

### Poziom walki z bossem

Pierwsze spotkanie z bossem to szósty poziom lochów z perspektywy Gracza. Mapa na tym poziomie składa się z tylko jednego, dużego pokoju, w którym znajduje się jeden wróg – potężny boss. Jeśli Marian nie zdecyduje się z nim walczyć, zawsze może użyć drabiny do przejścia na kolejny poziom, ale straci wtedy możliwość zdobycia dużych ilości doświadczenia.

|  |
| --- |
|  |
| Rysunek 5.8 Zrzut ekranu z poziomu z bossem. Na tej mapie jest więcej otwartej przestrzeni. |

### Poziom finałowy

Dotarcie to poziomu finałowego oznacza połowę sukcesu. Bohater wreszcie może ujrzeć światło dzienne, ale czeka go ostateczna bitwa z bossem. To jedyna walka w całej grze, przed którą Marian nie może uciec. Wygląd poziomu zaprezentowano na rysunku 5.8.

|  |
| --- |
|  |
| Rysunek 5.8 Zrzut ekranu z poziomu finałowego. Pierwszy moment rozgrywki, kiedy Mariana nie otaczają ciemności. |

## Dźwięk

### Muzyka

Muzyka jest zaprogramowana w taki sposób, aby dostosowywała się do akcji. W miarę zadawania obrażeń, oraz skupiania na Marianie uwagi wrogów muzyka staje się coraz żywsza. Gdy akcja się skończy muzyka powoli cichnie i staje się spokojniejsza. Aby było to możliwe znaleźliśmy specjalnie przygotowaną do tego celu muzykę, na którą składa się kilka fragmentów pasujących do różnych sytuacji w grze. Fragmenty te trwają dokładnie wielokrotność tej samej ilości czasu, oraz dobrze nadają się do zapętlenia. Ponad to mają bardzo podobny motyw przewodni bębnów, dzięki czemu świetnie się ze sobą łączą i przenikają. Aby przejść z jednego fragmentu do drugiego pierwszy z nich jest stopniowo wyciszany, podczas gdy drugi staje się głośniejszy. Takie przejście jest płynne i niezauważalnie, a całość brzmi jak z profesjonalnej, komercyjnej produkcji.

### Efekty dźwiękowe

Zastosowane w grze efekty dźwiękowe w większości zostały stworzone przy użyciu programu o nazwie „sfxr”. Jest to darmowy program na licencji MIT, umożliwiający tworzenie dźwięków przypominających te z klasycznych 8-bitowych gier. Pozostałe zostały znalezione w Unity Asset Store, a więc również mogą być legalnie i darmowo użyte w grze.

## Ataki i umiejętności

### Mechanika

W zależności od rodzaju ataku, czy umiejętności mechaniki działania są różne. W wypadku ataków składających się z pocisków zawierają one „collidery”, dzięki czemu wykrywają zderzenie i zdają adekwatne obrażenia. Szczególnym przypadkiem takiego ataku jest kula ognia, która prócz wspomnianego wyżej działania eksploduje w kontakcie zarówno z wrogami jak i podłożem, czy ścianami zadając obszarowe obrażenia wokół. Erupcja równiesz zadaje obszarowe obrażenia, a ponad to robi to w sposób ciągły. Umiejętność ta jest używana na danym obszarze i na tylko na nim ma swoje działanie. Wiązka błyskawic również zadaje obrażenia w sposób ciągły, jednak zupełnie różni się implementacją. W tym wypadku generowany jest promień, a obrażenia zadawane jeśli promień ten napotka wroga w zasięgu ataku. Podobnie zachowuje się umiejętność nazwana przebijającym strzałem, ale działa tylko pojedynczym impulsem, oraz jest w stanie trafić wielu wrogów na raz. Pozostałe umiejętności są trywialne pod względem realizacji ich działania.

### Efekty wizualne

Efekty wizualne ataków i umiejętności są wykonane za pomocą systemu cząsteczkowego Shuriken. Tworząc efekty pobraliśmy duże ilości efektów cząsteczkowych z Unity Asset Store. Część z nich została wykorzystana, z czego większość została bardzo mocno zmodyfikowana, tak aby pasowały do gry. Wykorzystanie gotowych efektów zapewniło nam tekstury cząsteczek, dodatkowe pomysły i na przykładach pozwoliło dość dogłębnie zaznajomić się z działaniem tego systemu. Nabyte doświadczenie zostało też wykorzystane do zrobienia niektórych efektów od zera (jedynie z wykorzystaniem gotowych tekstur), jak było w wypadku na przykład przebijającego strzału inspirowanego bronią o nazwie „railgun” z gier Quake 2 i 3.

## Grafika

### Shadery i materiały

Do naszego projektu wykorzystywaliśmy głównie shadery dostępne domyślnie w Unity, jednak z dwoma wyjątkami.

Brakowało nam shadera, który zastosowany na prostokącie umożliwiałby oświetlenie go z obu stron światłem świecącym tylko z jednej strony. Ponad to shader ten musiał umożliwiać animacje poprzez zmianę offsetów w teksturze, ponieważ miał zostać zastosowany dla przeciwników oraz Mariana. Przy użyciu dostępnych shaderów postacie oświetlone z jednej strony były zupełnie czarne z drugiej strony, lub (w wypadku shadera do liści) nie działała w nich animacja. W celu rozwiązania tego problemu zmodyfikowaliśmy kod jednego z shaderów tak, aby poprzez modyfikacje wektora normalnego jednakowo reagował na światło z obu stron.

Drugi shader o którym mowa w tym punkcie został stworzony z potrzeby użycia dwóch map normalnych na raz, ponieważ posiadaliśmy jedną odpowiadającą zwykłej teksturze, oraz drugą, która została bardziej rozciągnięta i symuluje większe nierówności terenu dając tym samym ciekawszy wygląd poziomu. Do stworzenia tego shadera użyliśmy dostępnego na Unity Asset Store rozszerzenia edytora umożliwiającego tworzenie shaderów poprzez grafy, podobnie jak można było mieć okazję nauczyć się je robić na laboratoriach z przedmiotu Projektowanie Gier Komputerowych w UDK. Mapa normalnych w tym shaderze powstała poprzez obliczenie średniej wartości z dwóch wyżej wspomnianych map. Niestety shader ten wymagał jeszcze ręcznej ingerencji w kod, ponieważ narzędzie to zapisując kod shadera tworzyło za każdym razem błąd, który jednak udało się naprawić.

Materiały tworzyliśmy przeważnie sami dobierając odpowiednie shadery, tekstury oraz parametry, dla uzyskania jak najbardziej atrakcyjnego wyglądu. Część materiałów również pochodzi z Unity Asset Store i w większości uległy drobnym modyfikacjom.

### Modele i tekstury

Modele i tekstury w przeważającej mierze pobraliśmy z Unity Asset Store. Część stworzyliśmy sami, a część pobraliśmy z innych stron internetowych jeśli licencja na to pozwalała.

## Diagram Zależności

|  |
| --- |
|  |
| Rysunek 5.8 Diagram zależności |

# testowanie, walidacja i weryfikacja

## Testowanie i walidacja

Testowanie było wykonywane na bieżąco. W miarę dodawania nowych funkcjonalności były one w sposób ciągły testowane i udoskonalane w szczególności przez osoby, które nad daną funkcjonalnością pracowały. Następnie zmiany były testowane przez pozostałych członków zespołu, co czasem pozwalało wykryć błędy przeoczone przez autora zmian. O swoich spostrzeżeniach informowaliśmy się również na bieżąco, co pozwalało na relatywnie szybkie wprowadzanie ustaleń i rozwiązywanie zaistniałych problemów. Dobra komunikacja jest bardzo ważnym elementem w procesie tworzenia gry, o czym mieliśmy okazję się przekonać. Poza tym prosiliśmy też o przetestowanie gry przez naszych znajomych, aby zasięgnąć opinii na temat niektórych decyzji.

## Weryfikacja założeń użytkowych aplikacji

Aplikacja jest trudna, trwa 20-25 minut, oraz da się ją opublikować w przeglądarce. Wszystkie wymagania użytkowe zostały spełnione, a sama gra została wykonana w terminie. Dzięki zastosowaniu metodyki zwinnej mogliśmy reagować na zmiany jakie zachodziły w koncepcji gry, w trakcie jej pisania. Dzięki dość ogólnym wymaganiom początkowym produkcja ewoluowała w trakcie tworzenia, co poprawiło jej jakość.

# Podsumowanie

## Podsumowanie i wnioski

Krzysiek:

Jestem bardzo zadowolony z wyniku naszej pracy. Gra jest wypadkową naszych pomysłów, dzięki temu jest oryginalna i ciekawa. W trakcie tworzenia pracy inżynierskiej zdobyłem doswiadczenie ze środowiskiem Unity, oraz udoskonaliłem swoją znajomośc języka C#.

## Możliwość dalszego rozwoju

# Wykaz Literatury

1. Kensler A.: *Correlated Multi-Jittered Sampling*, http://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf, (data dostępu 26.11.2014 r.).
2. Balmer:*Skeleton and Ghost spritesheets*, <http://opengameart.org/content/skeleton-and-ghost-spritesheets-ars-notoria>, (data dostępu 24.11.2014 r.).
3. Ante j.: *Animating Tiled texture*, <http://wiki.unity3d.com/index.php?title=Animating_Tiled_texture>, (data dostępu 25.11.2014 r.).
4. Fother: *The Mana World Pixel Art*, <https://wiki.themanaworld.org/index.php/User:Fother/Pixel_Art>, (data dostępu 23.10.2014 r.).
5. Kazemi D.: *Spelunky Generator Lessons,* [*http://tinysubversions.com/spelunkyGen/*](http://tinysubversions.com/spelunkyGen/)*,* (data dostępu 20.10.2014 r.).
6. TheInquisitor.: Gladiators Character Sets, <http://www.rpg-palace.com/visual-resources/character-sets-rm2k3/gladiators>, (data dostępu 21.11.2014 r.).
7. E.Adams: Projektowanie gier. Podstawy, Helion 2010.
8. P. Felicia: Getting Started with Unity, PACKT Publishing, 2013.
9. T. Norton: Learning C# by Developing Games with Unity 3D Beginner's Guide, PACKT Publishing, 2013.
10. R. H. Creighton: Unity 3D Game Development by Example Beginner's Guide, PACKT Publishing, 2010
11. J. Dean: Unity 4 Character Animation with Mecanim, PACKT Publishing, 2013
12. A. Stagner: Unity Multiplayer Games, PACKT Publishing, 2013
13. K. D’Aoust: Mastering Unity 4 Scripting, PACKT Publishing, 2013 (Video Tutorial)

# Wykaz rysunków

# Wykaz tabel

# Dodatek A: Plakat gry

|  |
| --- |
| C:\Users\Sharow\Pictures\Plakat.png |
| Rysunek A.1 Plakat w języku Polskim |

|  |
| --- |
| C:\Users\Sharow\Pictures\Poster.png |
| Rysunek A.2 Plakat w języku Polskim |