STRESZCZENIE

Niniejsza praca dyplomowa zawiera opis procesu projektowania, implementacji i testowania gry komputerowej z gatunku *RPG*/*Survival*, stworzonej w środowisku Unity. Projekt realizowany jest na katedrze Inteligentnych Systemów Interaktywnych wydziału Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej. Promotorem jest dr inż. Mariusz Szwoch. Gra nosi nazwę The Mighty Marian i zawiera elementy charakterystyczne dla gatunku *RPG*, takie jak zdobywanie doświadczenia czy drzewo umiejętności, jak również *survival*, czyli toksyczne środowisko i walka o przetrwanie oraz elementy zręcznościowe. Szczególną uwagę poświęcono zagadnieniu proceduralnej generacji terenu, które jest cechą wyróżniającą projektu. W finalnej wersji projektu udało się osiągnąć zamierzony cel, końcowy produkt gwarantuje ciekawą i dynamiczną rozgrywkę, dlatego zaproponowano dalsze opcje rozwoju.

Praca, pomijając rozdział wstępny, składa się następujących części:

Rozdział 2 – Wprowadzenie – krótko tłumaczy, dlaczego stosowana jest proceduralna generacja. Zawiera opis wykorzystywanych technologii i narzędzi oraz wprowadzenie do dziedziny automatów komórkowych. W podpunkcie 2.4 Proces tworzenia pracy dyplomowej przedstawiono harmonogram wykonanych prac i podział pracy w zespole.

Rozdział 3 – Projekt systemu – wyjaśnia założenia i cele projektu oraz opisuje planowany sposób rozpowszechnienia aplikacji. Definiuje ograniczenia dotyczące projektu i przedstawia specyfikację wymagań użytkowych, takich jak środowisko pracy systemu czy zakres funkcjonalny. W podpunkcie 3.3 opisane są elementy projektu, które pomimo braku ścisłego planu, zostały zdefiniowane przed rozpoczęciem implementacji. Autorem rozdziału jest Krzysztof Jasiak. Autorem podpunktu „Wymagania dla generatora map” jest Dominika Sokołowska.

Rozdział 4 – Implementacja – opisuje zaimplementowane algorytmy, jak generator map oraz sztuczną inteligencję wrogów i prezentuje ich praktyczne działanie. Zawiera opis i prezentację zaimplementowanych umiejętności bohatera, działanie systemu ekwipunku i innych elementów gry. Autorem punktów 1-4 , 6.2-7 oraz 8.2 - 8.4 jest Dominika Sokołowska. Punkty 5 i 12 stworzył Krzysztof Jasiak. Podpunkty 6.1, 8.1 oraz punkty 9-11 są autorstwa Tobiasza Biernackiego.

Rozdział 5 – Testowanie, walidacja i weryfikacja – omawia proces testowania zastosowany w projekcie i weryfikuje założenia użytkowe aplikacji. Autorem punktu pierwszego jest Tobiasz Biernacki, a drugiego Krzysztof Jasiak.

Rozdział 6 – Podsumowanie – zawiera opis osobistych osiągnięć każdego członka zespołu i opisuje możliwości rozwoju. Przedstawia również wnioski i podsumowanie przeprowadzonych prac.

Słowa kluczowe: gra, tworzenie gier, unity, roguelike, rpg, survival, proceduralna generacja
Dziedzina nauki i techniki: Nauki przyrodnicze, nauki o komputerach i informatyka.

ABSTRACT

Hereby thesis contains description of designing, implementation and testing process of RPG/Survival video game created in Unity. The project is being realized at the Department of Intelligent Interactive Systems at the Faculty of Electronics, Telecommunications and Informatics of Gdańsk University of Technology. PhD MEng Mariusz Szwoch is project promoter. The game is called The Mighty Marian and it contains typical RPG elements, such as gaining experience, or skill tree, as well as survival elements, such as toxic environment, fighting for survival and dexterity-requiring elements. Particular attention was given to procedural terrain generation, which is a distinguishing feature of the project. The final version of the project succeeds to provide an interesting and dynamic gameplay, hence the proposal for further development options.

Work, apart from an introductory chapter consists of the following parts:

Chapter 2 - Introduction - briefly explains why procedural generation is used. Contains a description of the technology and tools, and introduction to the field of cellular automata. In subsection 2.4 Process of creating the thesis presents a schedule of work performed and the division of labor in the team.

Chapter 3 - System design - explains the aims and objectives of the project and describe the plan to publish the application. It also defines restrictions on the project and shows the specifications of user requirements, such as the environment of the system or the functionality. In subsection 3.3 there is also description of the design elements that despite the lack of a strict plan were defined before implementation. The author of the chapter is Krzysztof Jasiak. The author of sub-paragraph "Requirements for map generator" is Dominika Sokołowska.

Chapter 4 - Implementation - describes the algorithms implemented, such as map generator and the artificial intelligence of enemies and presents their practical effect. It includes description and presentation of implemented hero’s skills, the equipment system, and other elements of the game. The author of paragraphs 1-4, 6.2-7 and 8.2 - 8.4 is Dominika Sokołowska. Sections 5 and 12 were created by Krzysztof Jasiak. Subsections 6.1, 8.1 and sections 9-11 are made by Tobiasz Biernacki.

Chapter 5 - Testing, validation and verification - discusses the testing process used in the project and verify the utility assumptions of the application. The author of the first paragraph is Tobiasz Biernacki, and the second Krzysztof Jasiak.

Chapter 6 - Summary - contains a description of the personal achievements of each team member, and describes further development opportunities. It also presents conclusions and a summary of the carried out work.

Keywords: game, game development, unity, rogulelike, rpg, survival, procedural generation

Field of science and technology: Natural sciences, computer and information sciences.

SPIS TREŚCI

1 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. Realizacja pracy dyplomowej inżynierskiej 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. Unreal Development Kit, stosunkowo łatwo jest opanować jego podstawy, głównie dzięki mnogości dostępnych materiałów szkoleniowych.

Definicja gatunku gier *RPG (ang. 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 *Rogue RPG*, którego charakterystyczną cechą jest losowość świata, a większość elementów świata przedstawionego (lochy, potwory, skarby, miasta a nawet imiona spotykanych postaci) generowana jest losowo przy każdej rozgrywce oraz brak opcji zapisu gry, 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 w The Mighty Marian, 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 prastarej katedry. 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.

1.1 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 Roguelike 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. [15]

Potencjalni odbiorcy: Gra skierowana jest raczej do starszych odbiorców, powyżej szesnastego roku życia, ze względu na wysoki poziom trudności i bardzo dynamiczną rozgrywkę. 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 również jako *Indie (ang. Independent games)*. Jedną z pierwszych 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 postać gracza przetrwała w nieprzyjaznym jej świecie. O sukcesie tego pomysłu z całą pewnością świadczy fakt, iż firma Mojang, wraz z grą, została zakupiona przez Microsoft, a gra rozeszła się w milionach kopii.

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, rozeszła się w milionach kopii. 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 produktów wzrasta z dnia na dzień. [14]

Cechy wyróżniające produkt

- mapy generowane proceduralnie
- brak zapisu gry (ang. permadeath),
- 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 poszczególnych poziomów,
- mechanizm nagłej śmierci poganiający gracza,
- brak zapisu gry, śmierć oznacza konieczność zaczęcia od początku.

2 WPROWADZENIE

2.1 Gry survival RPG

Wraz z rozwojem możliwości komputerów domowych, papierowe gry RPG przeniosły się stopniowo w świat elektronicznej rozrywki. Gry komputerowe to stosunkowo nowy, szybko rozwijający się rynek. Mnogość dostępnych platform, od telefonów, po konsole czy komputery stacjonarne, w połączeniu z ogromną ilością gatunków gier powoduje, że branża ma naprawdę dużą grupę docelową. Duża dostępność materiałów szkoleniowych i zasobów dostępnych za darmo w Internecie powoduje, że tworzeniem gier komputerowych zajmują nie tylko się ogromne międzynarodowe studia czy małe studia niezależne, ale nawet pojedynczy programiści. Popularność tytułów takich jak *Minecraft*, *Dwarf Fortress*, *Diablo* czy *Don't Starve* świadczy o tym, że do odniesienia sukcesu w branży nie potrzeba ogromnego zespołu doświadczonych ludzi. Wspominane tytuły łączy pewna cecha wspólna, którą zdecydowaliśmy się zaimplementować również w naszym projekcie – proceduralna generacja.

Generacja proceduralna to tworzenie zasobów, np. układu poziomu czy grafiki, przez program, w takcie jego działania. W starszych produkcjach decydowano się na takie rozwiązanie ze względu na niewielką ilość dostępnej pamięci. Dostarczanie razem z grą zapisanych wielu predefiniowanych poziomów z reguły wymaga więcej pamięci na nośniku niż dostarczenie algorytmu, który takie poziomy wygeneruje w razie potrzeby. Proceduralna generacja daje również możliwość tworzenia gier o nieskończonym czasie rozgrywki, kiedy gracz przejdzie pewien poziom, po prostu generowany jest kolejny.

Kiedy wydaję się grę z terenem predefiniowanym, każdy jego szczegół musi być dopracowany przez grafików i specjalistów. To zadanie bardzo czasochłonne, jak również kosztowne. Dla porównania zaprojektowanie kilku uniwersalnych elementów, a później składanie z nich skomplikowanych światów, raz zaimplementowanym algorytmem wydaje się być bardziej osiągalne dla twórców o niewielkich zasobach.

Na korzyść generowania proceduralnego przemawia również to, że gracz nie może nauczyć się na pamięć poziomu, czy znudzić się światem, ponieważ za każdym razem, kiedy rozpoczyna rozgrywkę, świat ten jest inny. To przedłuża żywotność produkcji.

2.2 Użyte technologie, narzędzia i inne aplikacje

2.2.1 Unity oraz środowisko MonoDevelop

Przy tak małym zespole projektowym i rozmiarze projektu, oczywistym wydaje się użycie gotowego silnika do tworzenia gier. Silnik 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 się 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.

2.2.2 System kontroli wersji

Na początku realizacji projektu używaliśmy systemu kontroli wersji SVN z darmowym repozytorium na serwerach code.google.com. 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.

2.2.3 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 [20] oraz IrfanView [21].

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.

2.2.4 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.

2.3 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 2.1 zaprezentowano te podstawowe rodzaje dla sąsiedztwa na promienia *r=1* i *r=2*.

Rys. 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*

Parametrem mającym kluczowy pływ na działanie automatu jest liczba jej żywych sąsiadek determinująca stan komórki w kolejnej iteracji. W klasycznej grze w życie, Conway's Game of Life, te wartości to 23/3, co oznacza, że 2 i 3 żywe sąsiadki warunkują przeżycie komórki, a dla trzech żywych powstaje nowa komórka dla sąsiedztwa Moora o *r=1*. Do opisania automatu komórkowego notację $X_1X_2\ldots/Y_1Y_2..$ gdzie X_n oznaczają kolejne wartości ilości żywych sąsiadów warunkujące przeżycie, a Y_n kolejne wartości warunkujące pojawienie się nowej, żywej komórki.

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.

2.4 Proces tworzenia pracy dyplomowej

2.4.1 Harmonogram prac

Tabela 2.1 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 Implementacja ekwipunku Pojawiają się czary Zmiana repozytorium na GitHub
Trzeci tydzień listopada 15-21	Implementacja walki mieczem Dodanie nowych wrogów – pająków Implementacja kolejnych zaklęć i rozwój istniejących 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 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 2.1 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 nie zostało wylistowanych.

2.4.2 Podział prac w zespole

Tabela 2.2 Podział prac w zespole

Tobiasz Biernacki	Kamera podążająca za bohaterem Sztuczna inteligencja wrogów Poruszanie się bohatera Implementacja mini-gry Implementacja oraz efekty czarów, zdolności i ataków Dodanie dynamicznej muzyki i dźwięków
Krzysztof Jasiak	System ekwipunku Początkowa animacja i menu Balans i grywalność Muzyka w menu głównym
Dominika Sokołowska	Algorytmy i implementacja generacji terenu Rozmieszczenie wrogów na mapie Implementacja przejścia pomiędzy poziomami Animacja wrogów i bohatera

W tabeli 2.2 wypisano ogólnikowo podział pracy w zespole. Projekt realizowano w grupie trzyosobowej, w atmosferze ścisłej, przyjacielskiej współpracy. Ogólna zasada panująca w projekcie: za jakość i poprawność poszczególnych modułów odpowiada osoba, która je zaimplementowała. Role w projekcie zostały przyjęte naturalnie, zgodnie z umiejętnościami i zainteresowaniami danej osoby.

3 PROJEKT SYTEMU

3.1 Cel i przeznaczenie systemu

3.1.1 Założenia i cel tworzenia aplikacji

Celem naszego projektu jest stworzenie gry przy użyciu silnika Unity, która łączyć będzie elementy *Survival* 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 to osoby w wieku 16-22 lata.

3.1.2 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 udostępnia również generator, który produkuje gotową stronę HTML i odpowiedni kod JavaScript z osadzonym 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 dystrybuowany będzie głównie lokalnie, dla osób zainteresowanych, choć będzie również dostępny do pobrania w Internecie.

3.1.3 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. Dodatkowo by ukazać zależności kodu zamieścimy diagram zależności między klasami.

3.2 Specyfikacja wymagań użytkowych

3.2.1 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*.

3.2.2 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.

3.2.3 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.

3.2.4 Środowisko pracy systemu

Dzięki zastosowanemu silnikowi gra jest wieloplatformowa 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*.

3.2.5 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 ewentualnych błędów.

3.2.6 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.

3.3 Projekt systemu

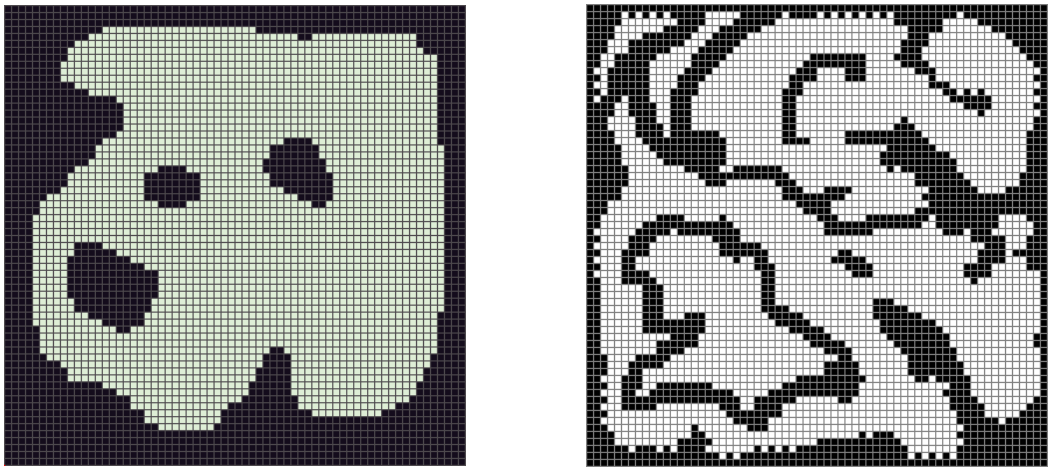
3.3.1 Projekt generatora map

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, za pomocą prostego algorytmu *flood fill (ang. wylewanie farby)* , to określenie stopnia skomplikowania i zawiłości korytarzy jest już zadaniem, którego w ramach tego projektu nie podjęliśmy się zrealizować. Na rysunku 3.1 przedstawiono mapy spójną i niespójną, ale interesująco skomplikowaną. 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 opisany jest szerzej w dalszej części dokumentu.

a) b)



Rys. 3.1. Prezentacja map spójnej i niespójnej

a) spójna mapa o niewielkim poziomie skomplikowania. Otwarty pokój, w którym gracz nie będzie miał czego odkrywać b) mapa niespójna, ale ciekawa pod względem grywalności

3.3.2 System RPG w grze The Mighty Marian - mechanika postaci

3.3.2.1 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.

3.3.2.2 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*. Ilość doświadczenia konieczna do zdobycia kolejnego poziomu została przedstawiona w tabeli 3.1.

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

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.

3.3.2.3 Zdobywanie statystyk

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

- każdy punkt siły zwiększa obrażenia zadawane bronią białą, zwiększa punkty 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 pulę many, przywraca część obecnej many oraz przyspiesza jej regeneracje. Dodatkowo zwiększa punkty obrażeń zadawane przez czary.

3.3.2.4 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. Wykaz umiejętności przedstawiono w tabeli 3.2.

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

- 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.

3.3.2.5 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.

3.3.2.6 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.

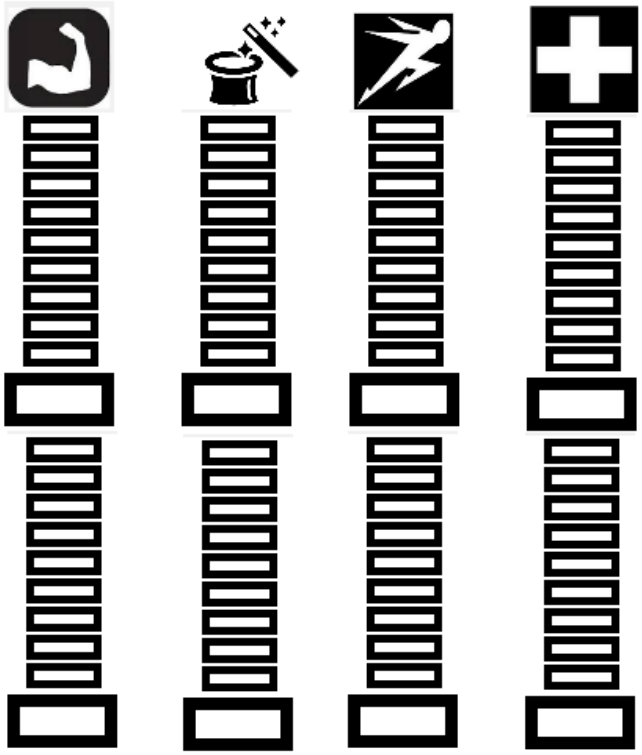
3.3.3 Ekwipunek

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.

3.3.4 Interfejs

Pierwsze inspiracje interfejsu wzięły się z gier typu MOBA, umiejętności miały być przypisane do klawiszy 1-5, Dodatkowo zaplanowaliśmy wstępny wygląd drzewka umiejętności co można zobaczyć na rysunku 3.2.



Rys. 3.2. Wygląd prototypu drzewka umiejętności

4 IMPLEMENTACJA

4.1 Postać gracza

Tytułowy Marian jest w grze przedstawiany jako gladiator w hełmie z czerwonym pióropuszem. Grafikę przedstawiono na rysunku 4.1.

Rys 4.1. Grafika głównego bohatera pochodzi ze źródła [6] i udostępniona jest nieodpłatnie do użytku niekomercyjnego

4.2 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.

4.3 Implementacja generatora map

W naszym projekcie każda rozgrywka polega na przejściu kilkunastu 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.

4.3.1 Etapy procesu generowania mapy

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

4.3.1.1 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:

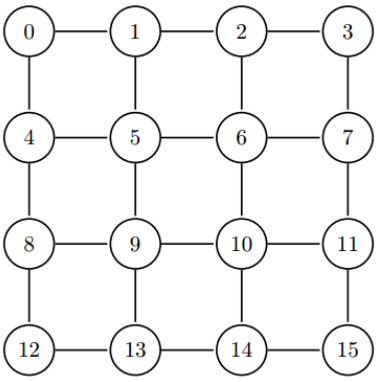
G(V,E) - graf nieskierowany

V - zbiór wierzchołków - pojedynczy wierzchołek reprezentuje jeden pokój

|*V*| = 4 * 4

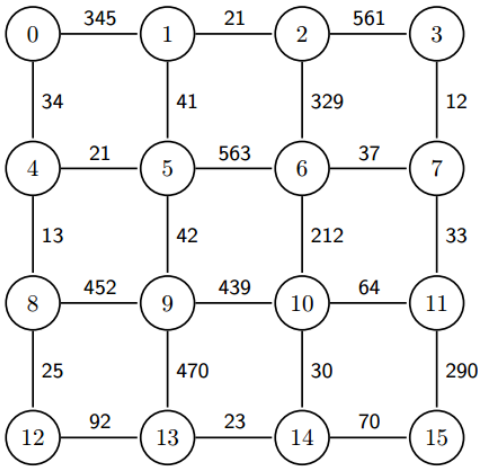
E - zbiór krawędzi - krawędź reprezentuje przejście między pokojami

Program generuje początkowy graf przejść między pokojami. Inicjalnie wszystkie możliwe przejścia między sąsiednimi pokojami istnieją, co zaprezentowano na rysunku 4.2.



Rys 4.2. 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.



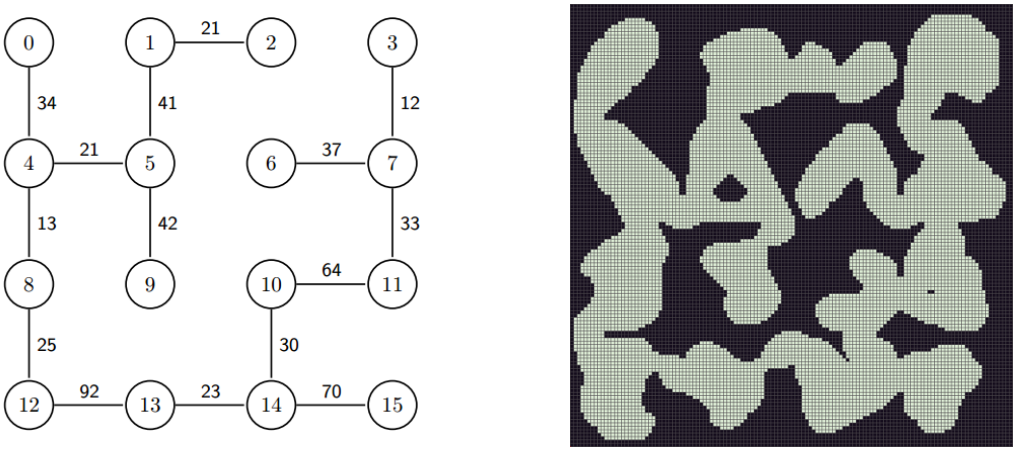
Rys 4.3. Graf przejść z przyporządkowanymi losowymi wagami na krawędziach

Krawędziom grafu G przyporządkowane zostają losowe wagi z zakresu (3,660). Wierzchołek startowy dla algorytmu również jest wybierany losowo, tak jak na rysunku 4.3.

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

- rozpoczynamy od grafu składającego się jedynie z wierzchołka startowego,
- krawędzie incydentne do wierzchołka umieszczamy na posortowanej wg. wag liście,
- zdejmujemy z listy krawędź o najmniejszej wadze i sprawdzamy, czy łączy wierzchołek wybrany z niewybranym. Jeśli tak, to znalezioną krawędź dodajemy do drzewa spinającego,
- dodajemy krawędzie incydentne z nowo wybranym wierzchołkiem do posortowanej listy,
- powtarzamy kroki 2 - 4 dopóki lista krawędzi nie będzie pusta. Osiągamy efekt z rysunku 4.4

a) b)



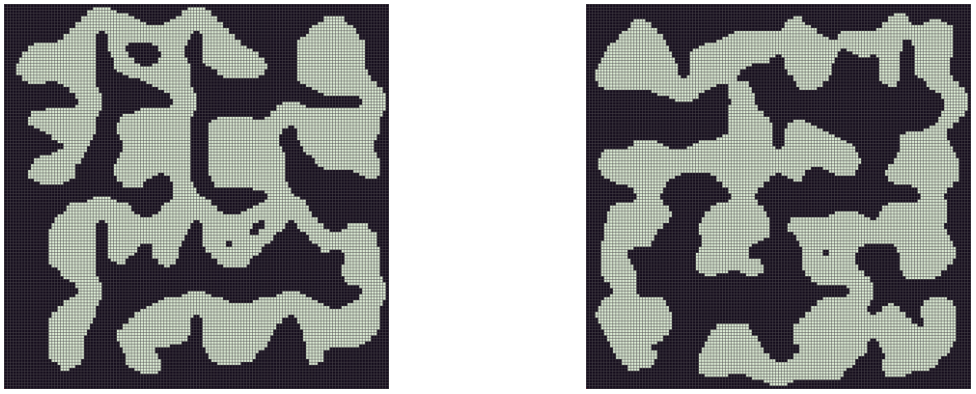
Rys 4.4. Labirynt i powstała z niego mapa

a) labirynt uzyskany po zastosowaniu algorytmu Prima b) przykładowa mapa wygenerowana dla uzyskanego labiryntu

4.3.1.2 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, złożony z komórek podłogi. 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 rysunku 4.5 porównano przykładową mapę z przejściami w tych samych miejscach w pokoju oraz mapę z czterostopniową różnorodnością w położeniu przejścia.

a) b)



Rys 4.5. Prezentacja efektów losowego rozłożenia przejść między pokojami

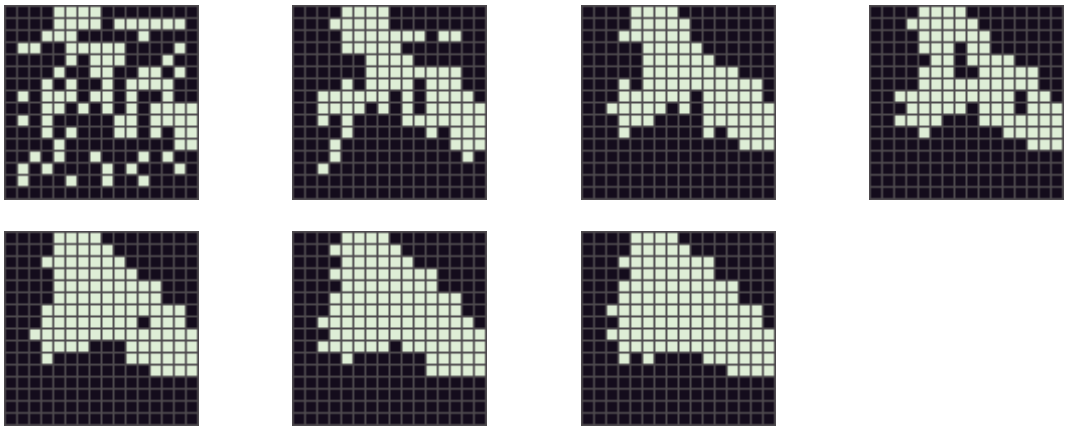
a) przykładowa mapa, w której przejścia między pokojami wygenerowano na szerokości oznaczonej przez wartość 4, b) mapa, gdzie przejścia między pokojami występują losowo na szerokościach oznaczanych przez wartości {1,4}

Już cztery stopnie możliwego położenia 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.

4.3.1.3 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 *r=2* liczba sąsiadów wynosi mniej niż 3 to komórka jest żywa. Jest również żywa, jeśli w sąsiedztwie Moora o *r=1* liczba sąsiadów należy do zbioru {5,6,7,8}. W przeciwnym razie komórka zostaje podłogą. Po czterech iteracjach parametry automatu zmieniają się na 5678/5678 z sąsiedztwem Moora o *r=1*. Zatem pierwsze trzy iteracje sprzyjają powstawaniu nowych komórek w miejscach opustoszałych i zapobiegają powstawaniu map nieciekawych i pustych. Pozostałe iteracje pełnią rolę wygładzającą. Proces jest zilustrowany na rysunku 4.6



Rys 4.6. 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

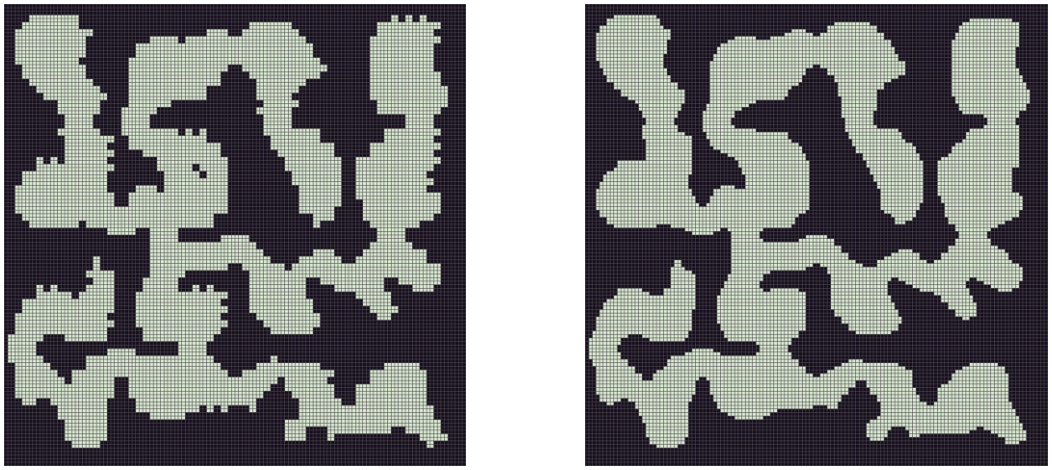
Po tym następuje sprawdzenie, czy uzyskany układ komórek jest spójny, to znaczy czy wszystkie komórki podłogi są połączone. Sprawdzane jest również, czy komórki podłogi stanowią co najmniej 30% całkowitej powierzchni pokoju. Jeżeli dane ułożenie komórek nie spełnia któregokolwiek z tych warunków rozwiązanie 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.

4.3.1.4 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)



Rys 4.7. Porównanie mapy po przeskalowaniu przed i po wygładzeniu realizowanym w metodzie CellularSmooth() klasy Map.

obraz przed wygładzeniem, b) obraz po wygładzeniu za pomocą automatu komórkowego

Jak można zaobserwować na rysunku 4.7, 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.

4.3.1.5 Etap erozji

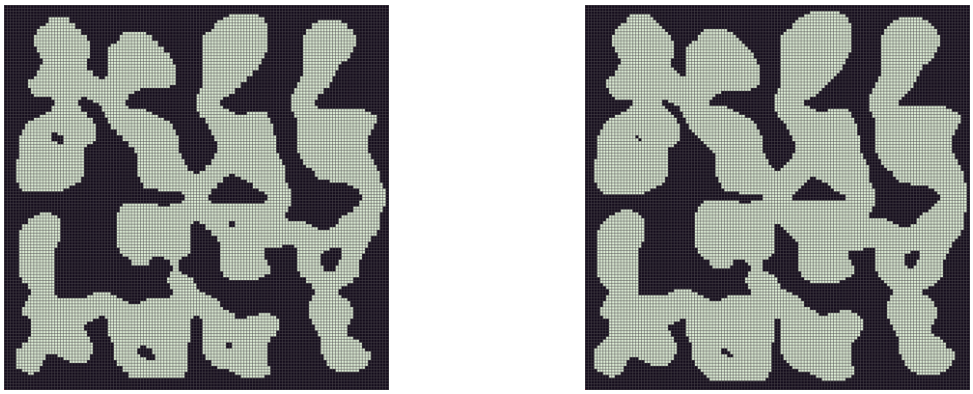
Ponieważ szerokość bohatera jest większa niż szerokość jednej komórki i koliduje on ze ścianami kołem o promieniu 1,5 komórki, spójność mapy nie gwarantuje, że będzie ona grywalna. Wszystkie przejścia muszą być co najmniej szerokości trzech komórek, aby gracz mógł się przez nie przecisnąć.

Nie znaleźliśmy stosunkowo łatwego w implementacji sposobu, aby upewnić się, że wszystkie przesmyki na mapie mają szerokość co najmniej trzech komórek więc zdecydowaliśmy się w końcowym etapie poszerzyć prewencyjnie wszystkie korytarze. 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 4.8 a i b możemy zaobserwować, że zastosowanie filtra erozyjnego ma swoje wady. Mapa na rysunku b 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.

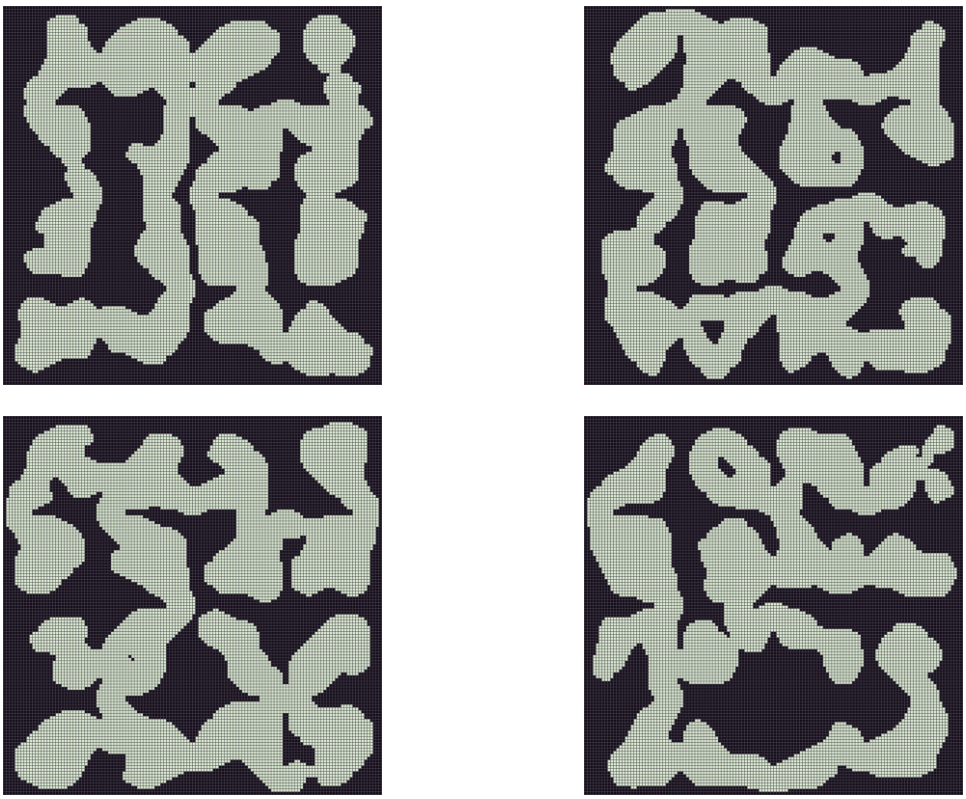
a) b)



Rys 4.8. a) mapa przed zastosowaniem filtra erozyjnego, można zaobserwować przejście o szerokości dwóch komórek, przez które bohater się nie przeciśnie, b) mapa z rysunku a) po zastosowaniu filtru erozyjnego, problematycznie wąskie przejście zwiększyło swoją szerokość do czterech komórek.

4.3.2 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 4.9 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.



Rys 4.9. Cztery przykładowe mapy wygenerowane przez program

4.4 Rozmieszczenie gracza i wrogów

4.4.1 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 n^3 i istnieją lepsze pod tym względem rozwiązania tego problemu, ponieważ graf przejść ma w rzeczywistej implementacji maksymalnie 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.

4.4.2 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 w przestrzeni 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 opisanym dobrze w źródle [1], tylko występuje w mniejszej skali.

4.5 Implementacja umiejętności

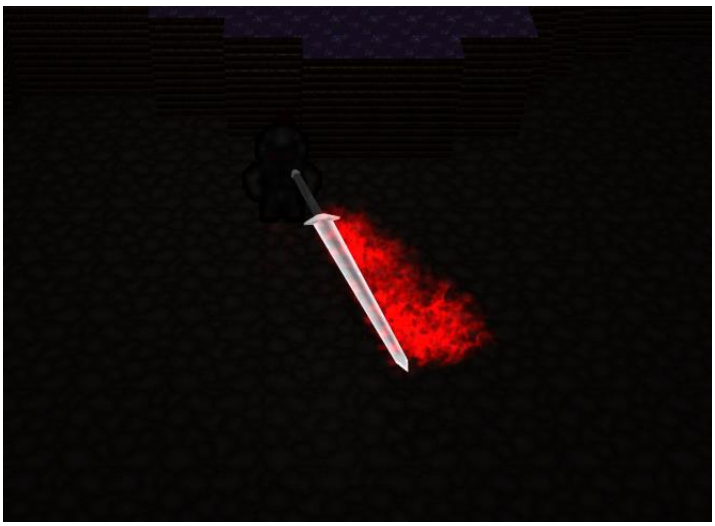
4.5.1 Drzewko umiejętności

Ostateczny wygląd drzewka umiejętności widać na rysunku 4.10.

Rys 4.10 Drzewko umiejętności

4.5.2 Wzmocnienie

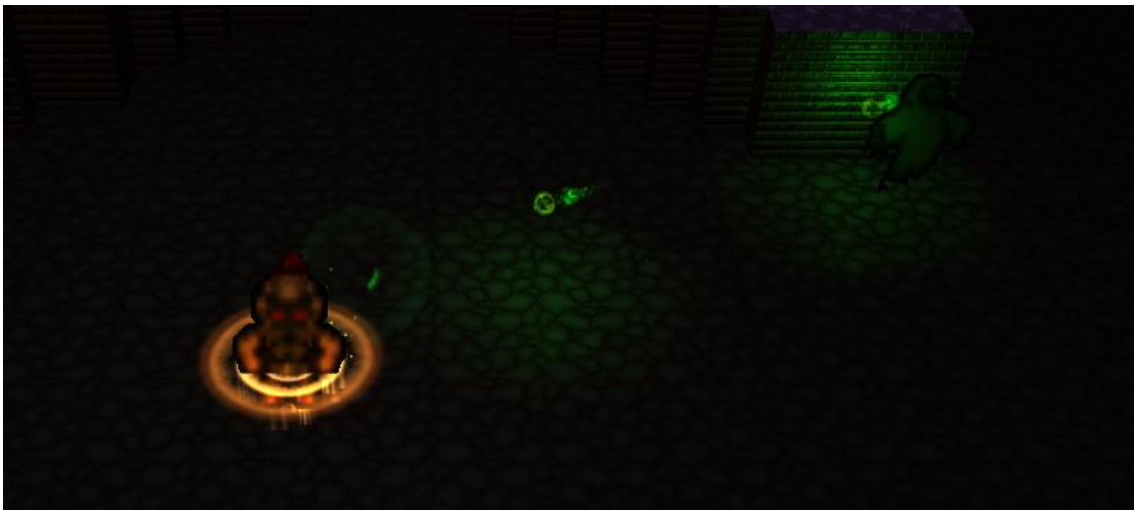
Wzmocnienie na dwie sekundy podwaja obrażenia zadawane przez bohatera bronią białą. Efekt graficzny widać na rysunku 4.11.



Rys 4.11 Wzmocnienie

4.5.3 Szał

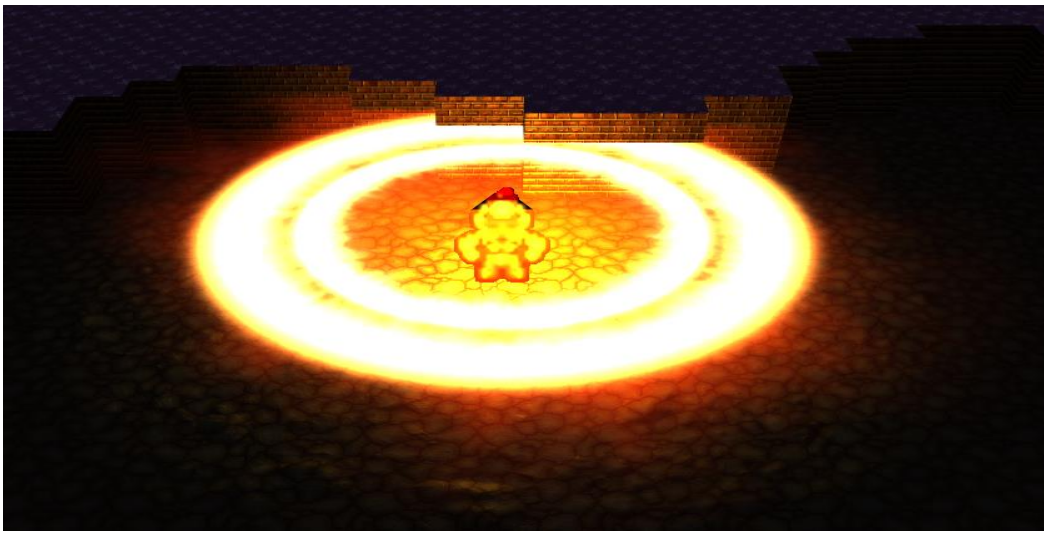
Szał (rysunek 4.11) sprawia, że ataki bronią białą kradną przeciwnikom punkty życia.



Rys 4.12 Szał

4.5.4 Ogłuszający krzyk

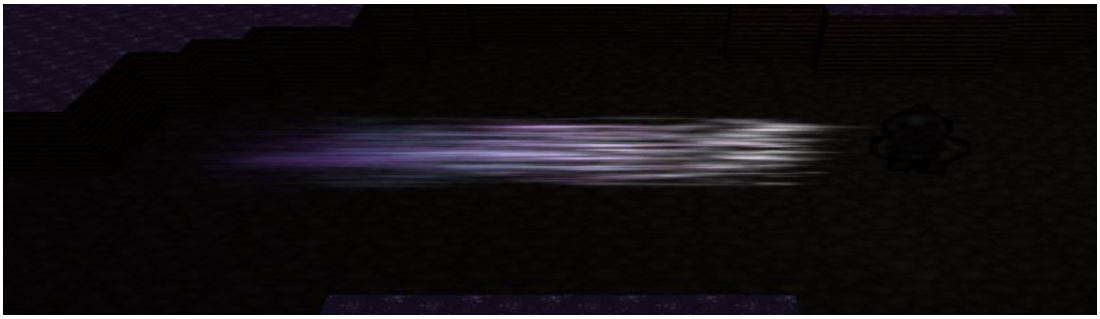
Aktywowanie tej umiejętności ogłusza na 5 sekund wszystkich przeciwników wokół bohatera, przez co są oni niezdatni do wykonania jakiejkolwiek akcji. Wygląd w grze można zobaczyć na rysunku 4.13.



Rys 4.13 Ogłuszający krzyk

4.5.5 Przyspieszenie

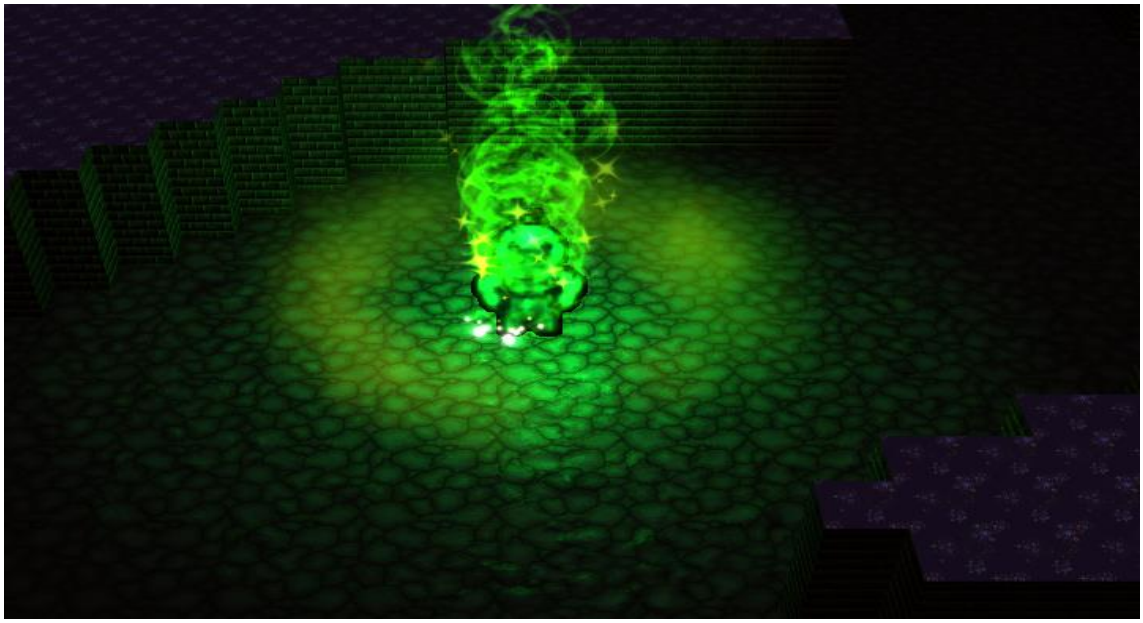
Ta umiejętność, zaprezentowana na rysunku 4.14, znacznie zwiększa prędkość postaci, i pozostawia za nią smugę światła.



Rys 4.14 Przyspieszenie

4.5.6 Leczenie

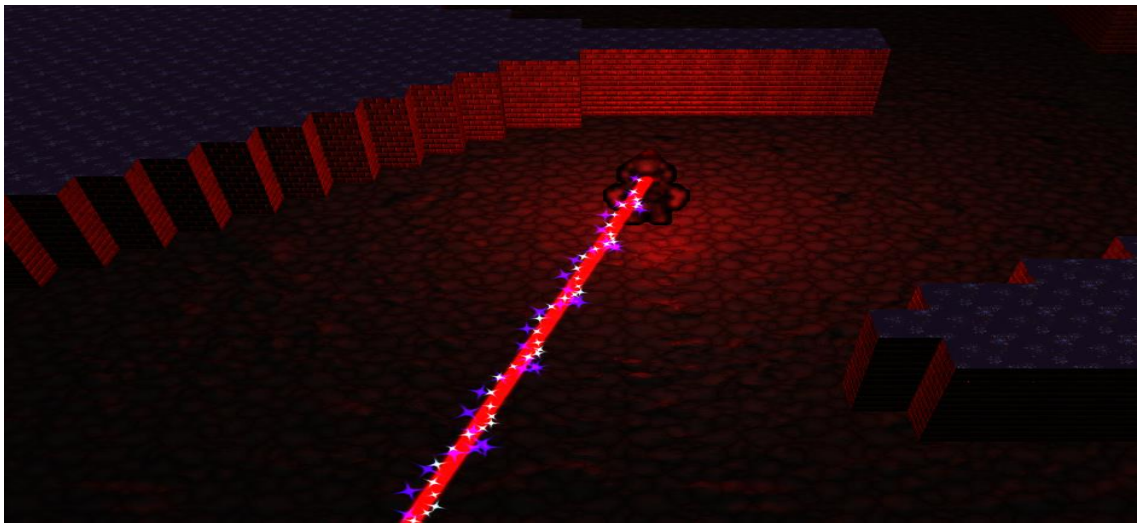
Postać zostaje otoczona zielona mgiełką (rysunek 4.15), która przywraca jej cześć utraconych punktów życia.



Rys 4.15. Leczenie

4.5.7 Przebijający strzał

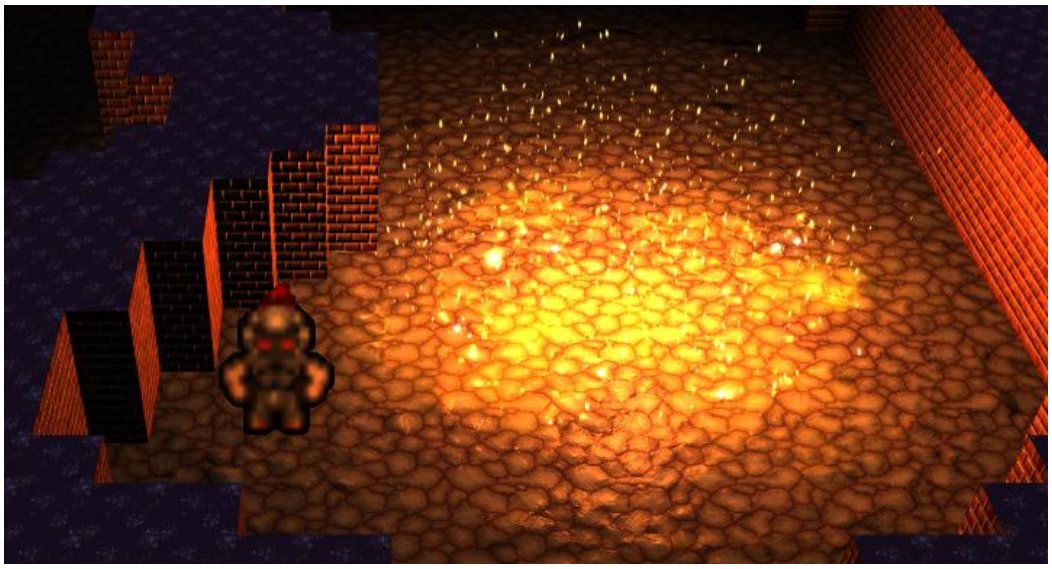
Na rysunku 4.16 widać promień który zadaje poważne obrażenia każdemu przeciwnikowi którego tafii.



Rys 4.16 Przebijający strzał

4.5.8 Erupcja

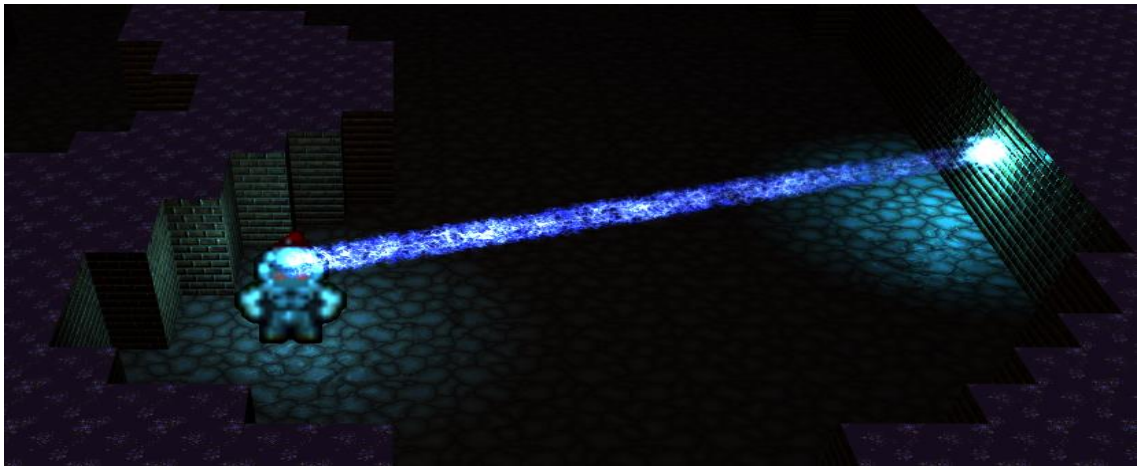
Każdy przeciwnik który stoi w obszarze płonącej ziemi, widocznej na rysunku 4.17, otrzymuje obrażenia.



Rys 4.17 Erupcja

4.5.9 Wiązka błyskawic

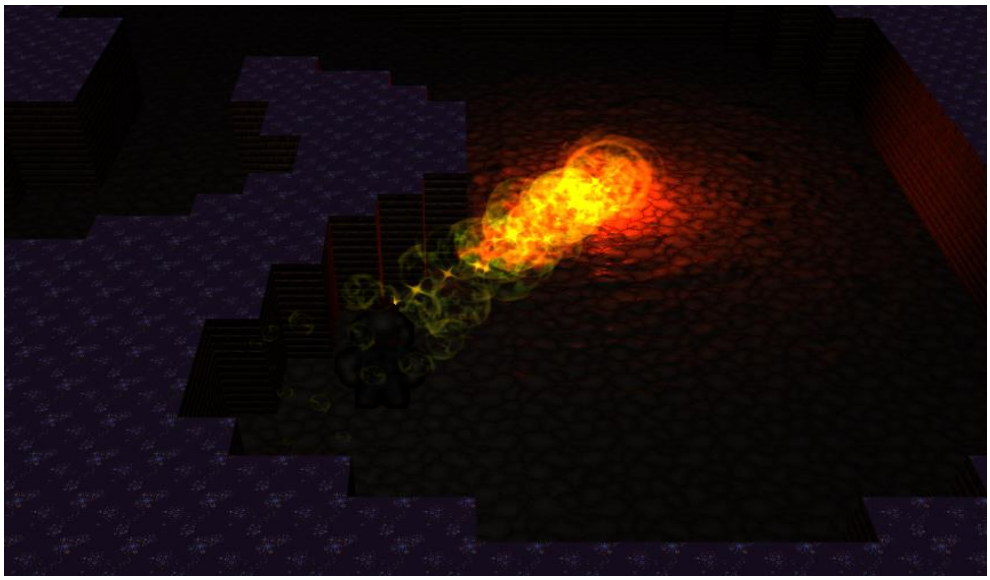
Rysunek 4.18 ukazuje naszego bohatera wystrzeliwującego wiązkę błyskawic.



Rys 4.18 Wiązka błyskawic

4.5.10 Kula ognia

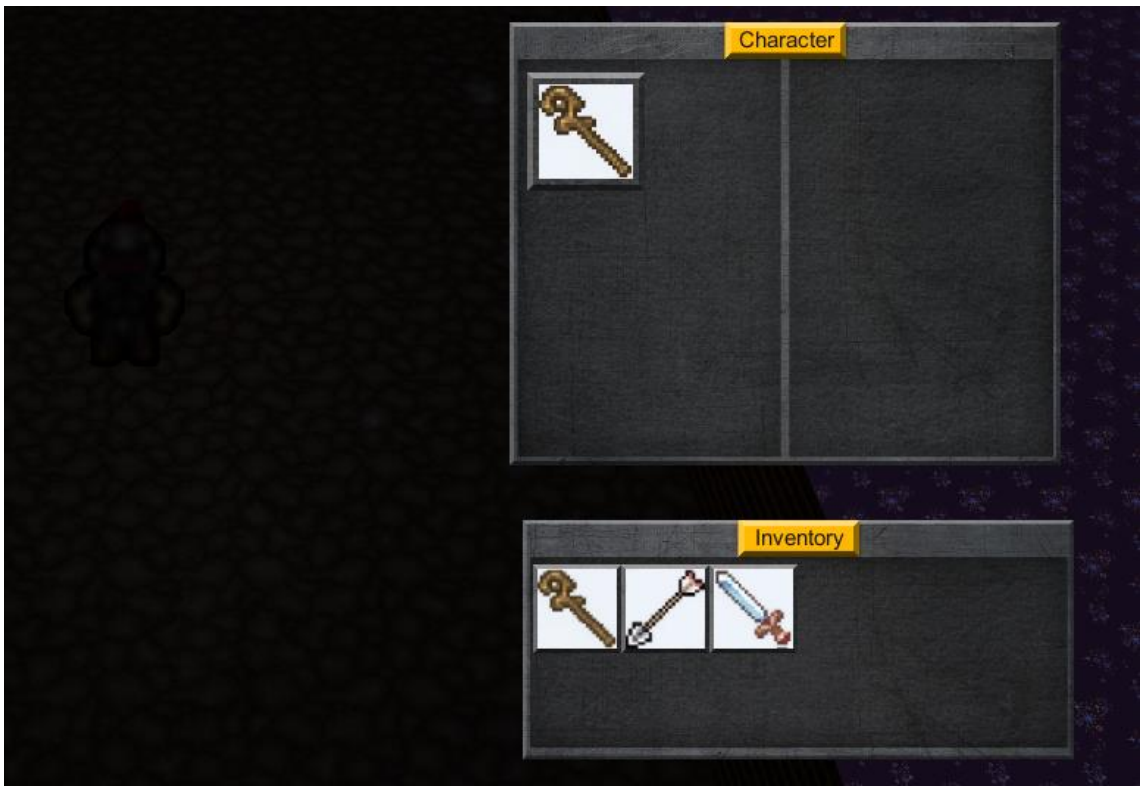
Na rysunku 4.19 widać efekt graficzny wystrzelania kuli ognia. Kula ognia zadaje obrażenia nie tylko przeciwnikowi, którego trafi, ale również innym przeciwnikom, jeśli znajdują się w niewielkim oddaleniu od miejsca wybuchu. Jest bardzo skuteczną bronią przeciwko dużym grupom wrogów.



Rys 4.19 Kula ognia

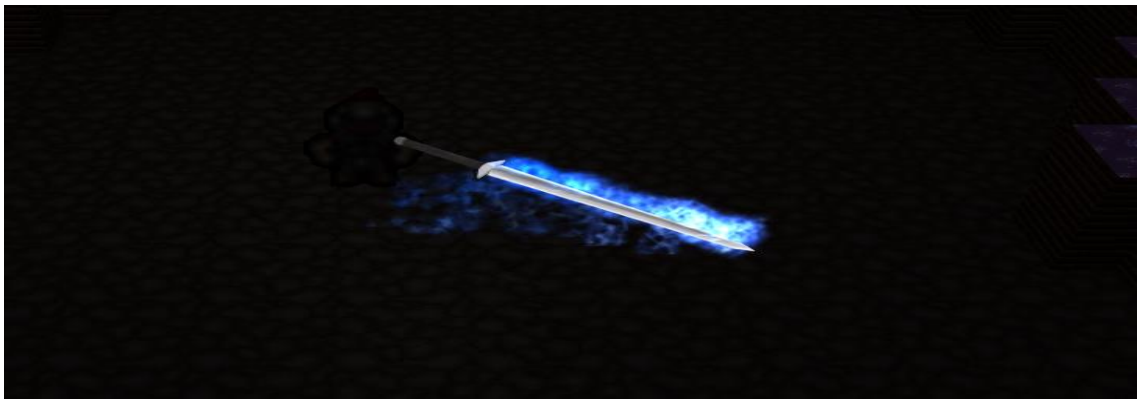
4.5.11 Implementacja Ekwipunku

W naszej produkcji zdecydowaliśmy się zastosować gotowy system ekwipunku. Na stronie Unity Asset Store [19] jest dostępny gotowy system, bardzo łatwy w implementacji. Inventory System [16] stworzony przez Brackeys 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. Niestety zastosowanie tego modułu przeszkodziło w implementacji niektórych funkcjonalności systemu ekwipunku, takich jak pokazywanie wartości obrażeń po najechaniu na przedmiot. Ostateczny wygląd widać na rysunku 4.20. Posiadamy trzy rodzaje broni, miecz (rys 4.21), strzały (rys 4.22) i różdżkę (rys 4.22).

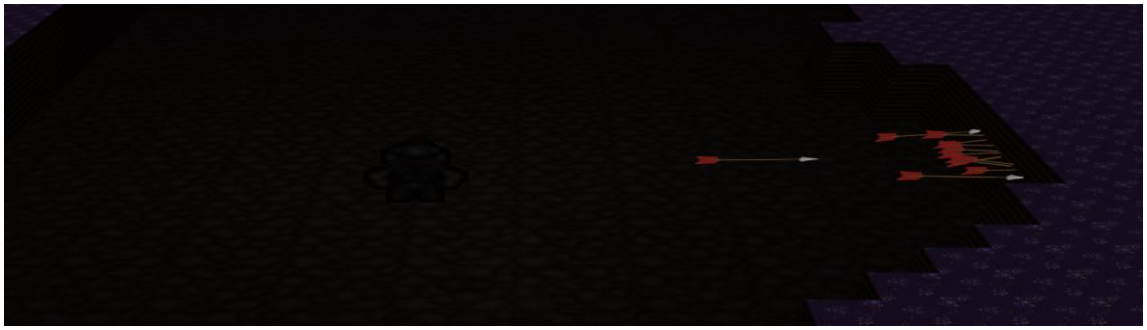


Rys 4.20 Ekwipunek

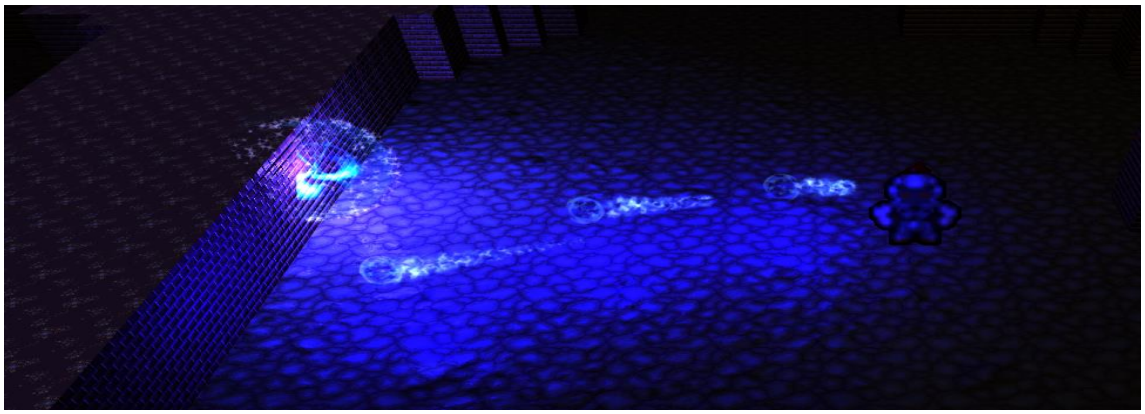
4.5.12 Implementacja rodzajów broni



Rys 4.21. Miecz do walki wręcz



Rys 4.22 Strzały do walki na dystans



Rys 4.23 Magiczne pociski do walki na dystans

4.6 Implementacja wrogów

4.6.1 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 dostępnego pod linkiem [17]. 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, ze trudno Quake 2 porównać do naszej produkcji, ponieważ gry różnią się pod wieloma wglądami, między innymi gatunkiem. 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

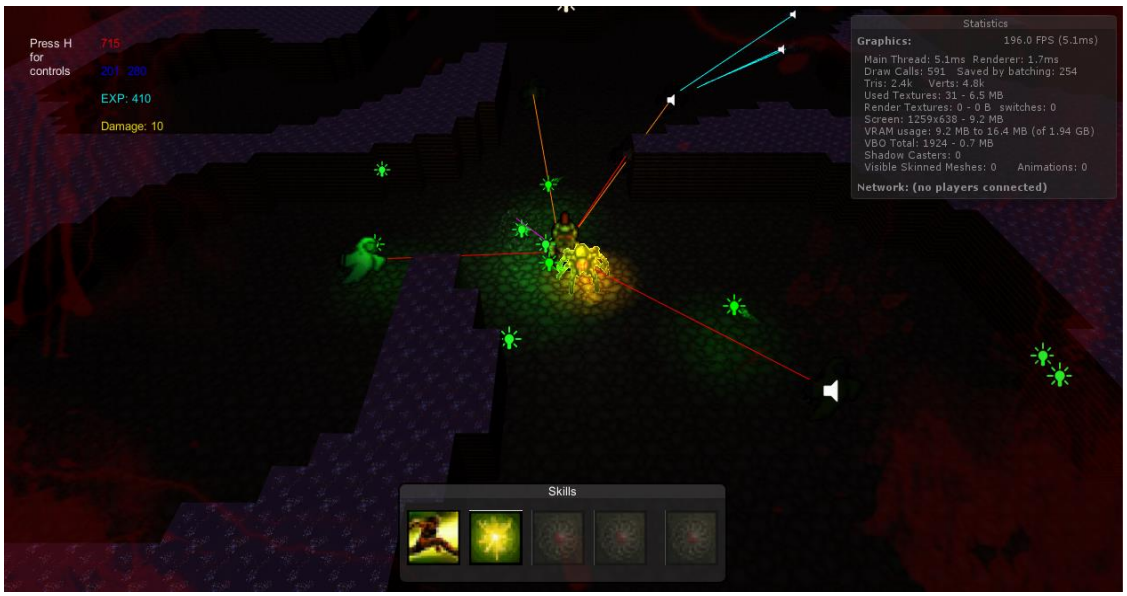
z otoczenia, od tak oczywistych jak ujrzenie głównego bohatera, po mniej oczywiste, jak na przykład zaobserwowanie innego zachowania sąsiednich sojuszników.

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 punktu, 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 bohatera 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.

Aby zaobserwować stan i cel wrogich jednostek podczas testowania zaimplementowane zostało rysowanie odpowiednich linii w edytorze, widocznych na rysunku 4.24. Czerwone oznaczają atakowanie Mariana, pomarańczowe gonienie go, natomiast niebieskie poruszanie się do innego typu celu, którym w tym wypadku jest wroga jednostka widząca bohatera.



Rys. 4.24. Linie informujące o zamiarach wrogów w edytorze.

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. Oprócz 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. Jednostki zachowują się dość naturalnie poprzez ragowanie na zdarzenia w grze i informowanie się nawzajem. Gdy gra jest na bardziej zaawansowanym etapie i zagęszczenie wrogów jest większe powoduje to ich grupowanie się. 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.

4.6.2 Rodzaje wrogów

4.6.2.1 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 4.25, które różnią się od siebie wytrzymałością na obrażenia. Duchy atakują dystansowo.

Rys. 4.25. Grafika duchów. Źródło: [2]

4.6.2.2 Pająki

Pająki występują z mniejszym prawdopodobieństwem niż duchy. Ich ataki są silne i bezpośrednie. Na rysunku 4.26 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.

Rys. 4.26. Animacja chodzącego pająka, grafikę udostępniono na licencji CC-sa-3.0, umożliwiającej wykorzystanie nawet do celów komercyjnych pod warunkiem podania źródła. Źródło [4]

4.6.2.3 Tanki

Ich rolą jest przyjmowanie dużej ilości obrażeń. Są najbardziej wytrzymałymi ze wszystkich wrogów. Atakują bezpośrednio. Występują ze stosunkowo niskim prawdopodobieństwem. Wygląd tanka przedstawiono na rysunku 4.27.

Rys. 4.27. Grafika tanka została stworzona specjalnie na potrzeby gry The Mighty Marian

4.6.2.4 Zielone kamikadze

Wróg o bardzo niewielkiej wytrzymałości, przedstawiony na rysunku 4.28, zainspirowany stworem *Baneling* z gry *Starcraft* 2. Kiedy zauważy bohatera wbiegają w niego z dużą prędkością. Przy zderzeniu z Marianem wybucha i zadaje obrażenia.

Rys. 4.28. Ten pozornie uroczy stworek potrafi zrobić dużą krzywdę

4.6.2.5 Boss

Boss jest większą i o wiele silniejszą wersją tanka. Oba te stwory mają zbliżoną grafikę, jednak boss ma bardziej jaskrawe barwy, rysunek 4.29. Jest jedynym przeciwnikiem spotykanym na etapie finałowym i wcześniejszych poziomach specjalnych. Jego rozmiar jest o wiele większy niż zwykłego wroga, żeby podkreślić jego siłę.

Rys. 4.29. Grafika bossa pojawiającego się w grze. Podobnie jak grafika tanka również została stworzona na potrzeby projektu

4.7 Animacja

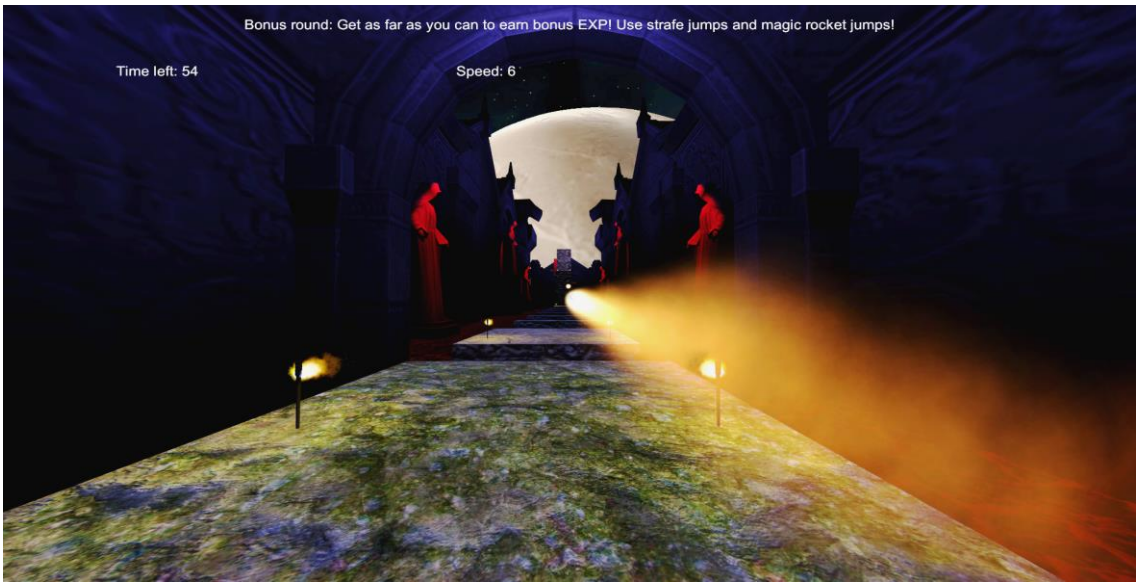
Do animacji wrogów użyto operacji odpowiedniego obracania powierzchni, na której rysowany jest sprite (zestaw obrazów przedstawiających kolejne klatki animacji postaci). Takie rozwiązanie daje iluzję ruchu nawet dla statycznych obrazów. W przypadku wrogów, których animacja ma więcej niż jedną klatkę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

4.8 Tryby rozgrywki

4.8.1 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 4.30 przedstawia zrzut ekranu podczas początkowych chwil rozgrywki. Po minucie gracz przenoszony jest do standardowego trybu gry.



Rys. 4.30. Zrzut ekranu z mini gry w The Mighty Marian. Obrazy są rozjaśnione i ze zmniejszonym kontrastem, żeby w wydrukowanej wersji pracy były czytelne.

4.8.1.1 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 *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.

4.8.1.2 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.

4.8.2 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. Wygląd gry widać na rysunku 4.31.

Rysunek 4.31: Zrzut ekranu ze standardowego tryb rozgrywki. Marian ma tu już odblokowaną jedną umiejętność.

4.8.2.1 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. Jest on widoczny na rysunku 4.32.

Rysunek 4.32 Efekt nagłej śmierci

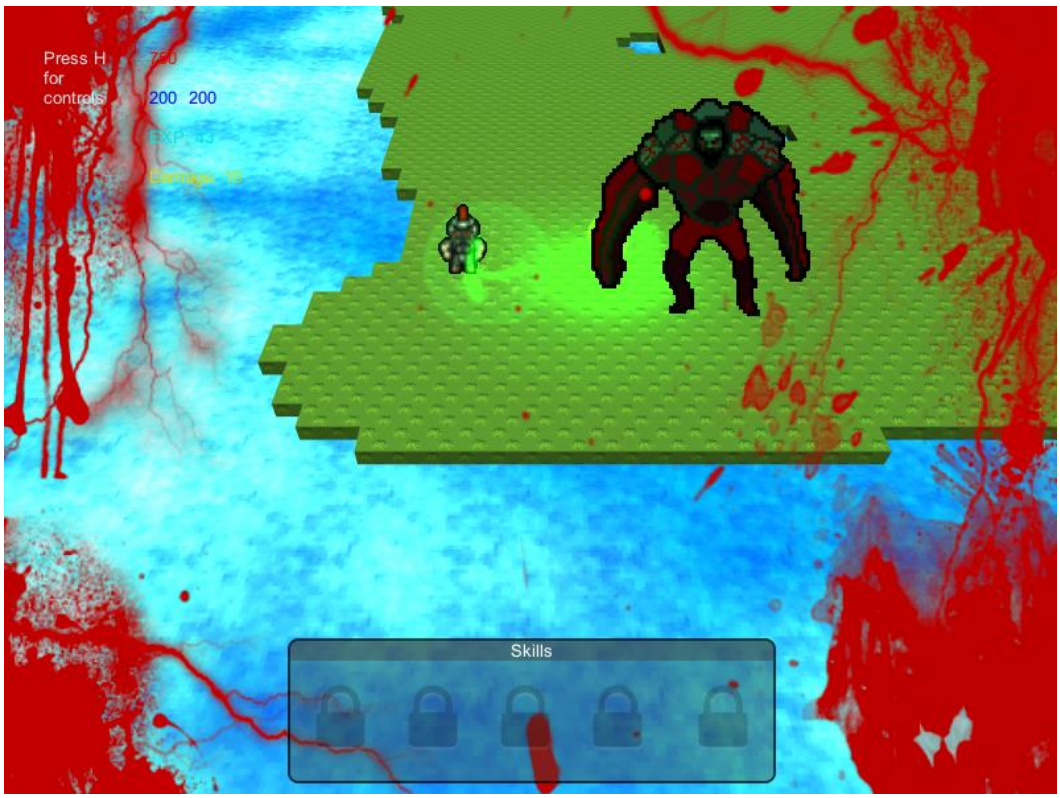
4.8.3 Poziom walki z bossem

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 gracz nie zdecyduje się z nim walczyć, zawsze może użyć drabiny do przejścia na kolejny poziom, ale straci wtedy możliwość zdobycia dużych ilości doświadczenia. Walka na poziomie z bossem została zaprezentowana na rysunku 4.33.

Rysunek 4.33 Zrzut ekranu z poziomu z bossem. Na tej mapie jest więcej otwartej przestrzeni.

4.8.4 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 4.34.



Rysunek 4.34 Zrzut ekranu z poziomu finałowego. Pierwszy moment rozgrywki, kiedy Mariana nie otaczają ciemności.

4.9 Dźwięk

4.9.1 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.

4.9.2 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 [19], a więc również mogą być legalnie i darmowo użyte w grze.

4.10 Ataki i umiejętności

4.10.1 Mechanika

W zależności od rodzaju ataku, czy umiejętności mechaniki działania są różne. W wypadku ataków składających się z pocisków zawierają one *collider*, 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 również 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.

4.10.2 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 [19]. 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. Do uzyskania krwawego obrazu po otrzymaniu obrażeń posłużyła grafika znaleziona w źródle [18].

4.11 Grafika

4.11.1 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 postaci Mariana, więc powinien działać prawidłowo z animacją. 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 [19] 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.

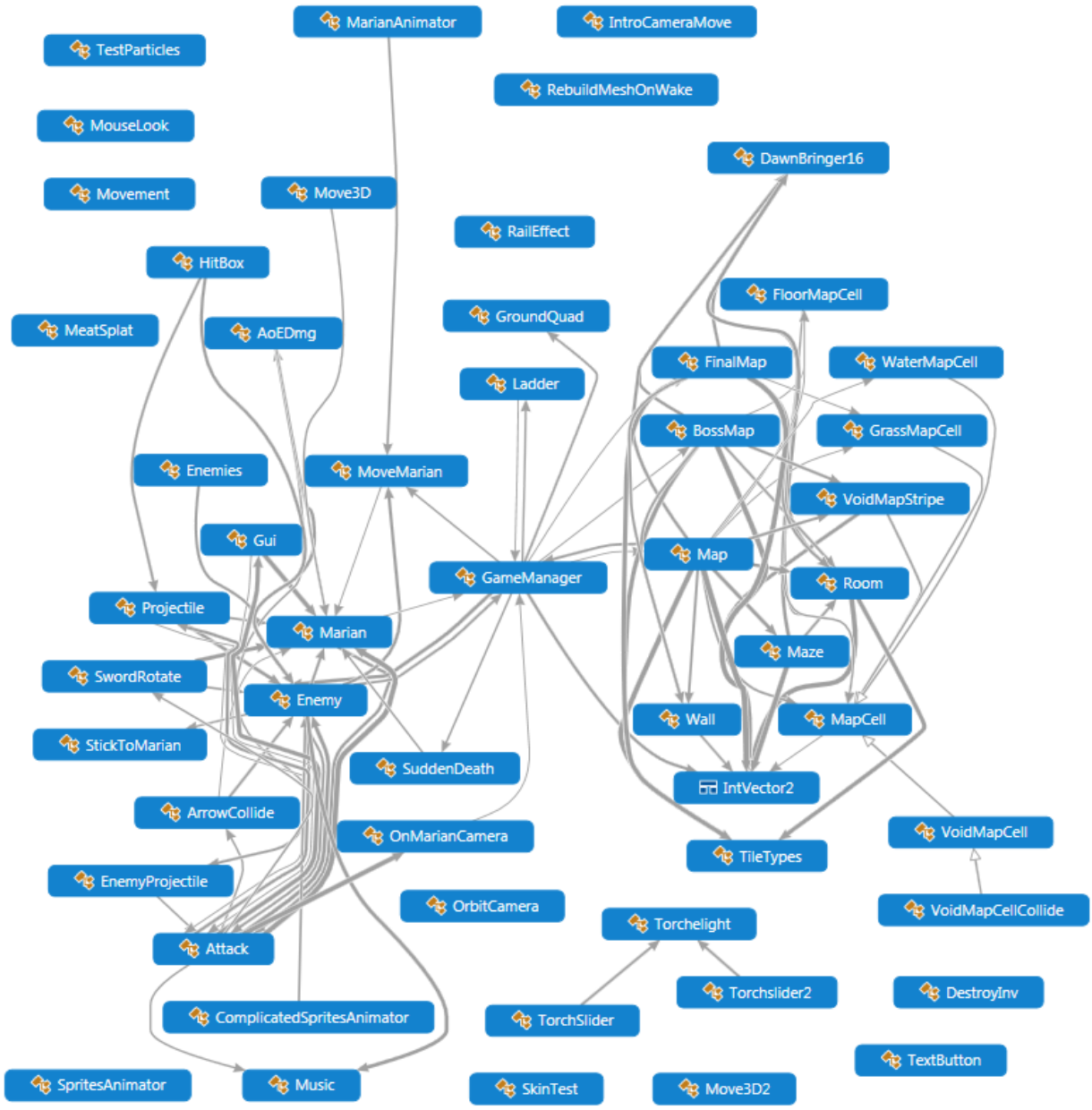
Materiały tworzyliśmy przeważnie sami dobierając odpowiednie shadery, tekstury oraz parametry, dla uzyskania jak najbardziej atrakcyjnego wyglądu. Część materiałów również pochodzi z Unity Asset Store [19] i w większości uległy drobnym modyfikacjom.

4.11.2 Modele i tekstury

Modele i tekstury w przeważającej mierze pobraliśmy z Unity Asset Store [19]. Część stworzyliśmy sami, a część pobraliśmy z innych stron internetowych jeśli licencja na to pozwalała.

4.12 Diagram Zależności

Na rysunku 4.35 widać wygenerowany przez Visual Studio 2013 diagram zależności.



Rysunek 4.35 Diagram zależności

5 TESTOWANIE, WALIDACJA I WERYFIKACJA

5.1 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.

5.2 Weryfikacja założeń użytkowych aplikacji

Gra ma wysoki poziom trudności, 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. Produkcja jest stabilna, dzięki skalowaniu zapewnionemu przez unity działa także na słabszych komputerach.

6 PODSUMOWANIE

6.1 Podsumowanie osobistych osiągnięć

6.1.1 Tobiasz Biernacki

Podczas realizacji projektu dyplomowego inżynierskiego zdobyłem sporo doświadczenia w środowisku Unity. Nauczyłem się korzystać z systemu cząsteczkowego Shuriken przygotowując liczne efekty cząsteczkowe do naszej gry. Zdobyłem również podstawową wiedzę na temat tworzenia shaderów i wykorzystałem ją w praktyce. Zająłem się zapewnieniem odpowiedniej oprawy graficznej oraz dźwiękowej, co mimo braku wcześniejszego doświadczenia wyszło moim zdaniem dość dobrze jak na taką drobną produkcje. Udało mi się zaimplementować złożone zachowanie wrogich jednostek zapewniające grywalność. Jestem też odpowiedzialny za działanie większości ataków i umiejętności. Jako sukces uważam także stworzenie mini gry, w której zaimplementowana została nietypowa fizyka.

6.1.2 Krzysztof Jasiak

Jestem bardzo zadowolony z wyniku naszej pracy. Gra jest wypadkową naszych pomysłów, dzięki temu jest oryginalna i ciekawa. W trakcie tworzenia pracy inżynierskiej zdobyłem doświadczenie ze środowiskiem Unity, oraz udoskonaliłem swoją znajomość języka C#. Bardzo ciekawym problemem było połączenie kodu napisanego w *JavaScripcie* z kodem napisanym w C#. Pracując nad tym połączeniem zdobyłem wiele przydatnych umiejętności, a samo powiązanie działa bardzo dobrze. Kolejnym osiągnięciem było przekonanie zespołu do porzucenia topornego systemu kontroli wersji Tortoise SVN i przeniesienie kodu na GitHub. Mimo początkowego oporu ze strony reszty zespołu, ostatecznie zaoszczędziło nam to bardzo dużo czasu. Wymyślenie i zaimplementowanie mechaniki było ciekawym wyzwaniem, ale udało się stworzyć stosunkowo prosty, a jednocześnie ciekawy system.

6.1.3 Dominika Sokołowska

Realizując projekt dyplomowy inżynierski poszerzyłam wiedzę związaną z algorytmami grafowymi. Poznałam szeroką gamę praktycznych rozwiązań stosowanych w nowoczesnych grach komputerowych w dziedzinie generacji terenu i zaimplementowałam na potrzeby projektu autorski pomysł, będący połączeniem generowania labiryntu oraz automatów komórkowych o różnych parametrach. Nie mając wcześniej do czynienia ze środowiskiem Unity, udało się stworzyć zaawansowany, łatwo parametryzowalny generator, który po stworzeniu kształtu mapy reprezentuje go w postaci trójwymiarowej i w odpowiednich proporcjach i miejscach na mapie umniejsza wrogów.

6.2 Podsumowanie i wnioski

Przez ostatnich kilka miesięcy udało nam się zdobyć sporo doświadczenia w środowisku Unity tworząc niniejszy projekt. Udało nam się spełnić wymagania i większość założeń, a wiele z naszych wizji przełożyć na kod i działającą grę. Nigdy wcześniej nie mieliśmy do czynienia z tworzeniem nietrywialnej gry w zespole, przez co mimo sukcesów podczas produkcji uczyliśmy się również na błędach, z których można wyciągnąć najwięcej wniosków. W naszym zespole atutem była dobra komunikacja, jednak mimo wspólnego ustalania założeń, podziału prac, oraz sposobów implementacji i łączenia produktów naszej pracy prawdopodobnie sporo czasu można by zaoszczędzić wprowadzając bardziej formalne metody opisu wymienionych powyżej aspektów. Nasza wiedza na temat organizacji pracy w zespole z pewnością się poprawiła. Nauczyliśmy się też grupowej pracy nad projektem w Unity, co początkowo sprawiało wiele problemów.

6.3 Możliwość dalszego rozwoju

Projekt w obecnym stadium to produkcja grywalna, zbalansowana i o wysokim poziomie trudności. Jesteśmy zadowoleni z efektów naszej pracy i po zakończeniu projektu dyplomowego inżynierskiego będziemy nadal rozwijać grę The Mighty Marian.

Planujemy w pierwszej kolejności zwiększyć różnorodność wrogów, szczególnie dodać co najmniej dwa nowe rodzaje bossów, aby w ciągu jednej rozgrywki dwa razy nie spotkać tego samego bossa. Obecnie główną przeszkodą jest brak odpowiednich grafik.

Obecnie zaimplementowana mini gra jest ciekawym urozmaiceniem rozgrywki. Być może dodanie większej ilości takich przerywników byłoby świetnym sposobem na zmniejszenie nieco napięcia powodowanego trudną rozgrywką.

W planie jest również dodanie do gry waluty i postaci handlarza, który będzie pojawiał się na niektórych poziomach, oferował zakup przedmiotów po bardzo wygórowanych cenach i skupował od gracza te przedmioty, które do tej pory zebrał po zabiciu wrogów i uważa, że nie będą mu potrzebne. To jest jednak plan dalekosiężny, ponieważ wspomniana funkcjonalność jest dość skomplikowana i wymaga głębokiego przemyślenia, aby jej implementacja nie zaburzyła balansu gry.

O istnieniu gry wiedzą obecnie jedynie nasi znajomi i rodzina. Dobrym pomysłem wydaje się promocja projektu w mediach społecznościowych, np. założenie strony fanowskiej dla gry i bloga, na których na bieżąco informować będziemy o postępach prac i umieszczać kolejne wersje gry. Chcielibyśmy też nagrać kilka filmów z przebiegu rozgrywki i umieścić je na YouTube.

Ciekawym pomysłem na zwiększenie poziomu trudności rozgrywki jest zapisywanie statystyk odnośnie śmierci bohatera na dysku gracza. Chcielibyśmy przechowywać takie wartości jak czas rozgrywki, osiągnięty poziom lochów, doświadczenie, zebrane przedmioty, oraz ilość otrzymanych obrażeń od poszczególnych wrogów i ilość zadanych obrażeń. Na tej

podstawie można profilować kolejne rozgrywki, np. tak aby wrogowie, którzy najczęściej są przyczyną śmierci Mariana pojawiają się z większym prawdopodobieństwem, aby gracz wpracował przeciwko nim skuteczną taktykę.

Planujemy również opracować sterowanie dla ekranów dotykowych i wybudować grę dla platformy Android. Dzięki temu, że Unity wspiera tworzenie gier na Androida, proces ten uda się przeprowadzić bez większych zmian w aplikacji. Po drobnym uproszczeniu gra nie będzie bardzo skomplikowana graficznie, więc powinna działać płynnie na nowszych urządzeniach.

WYKAZ LITERATURY

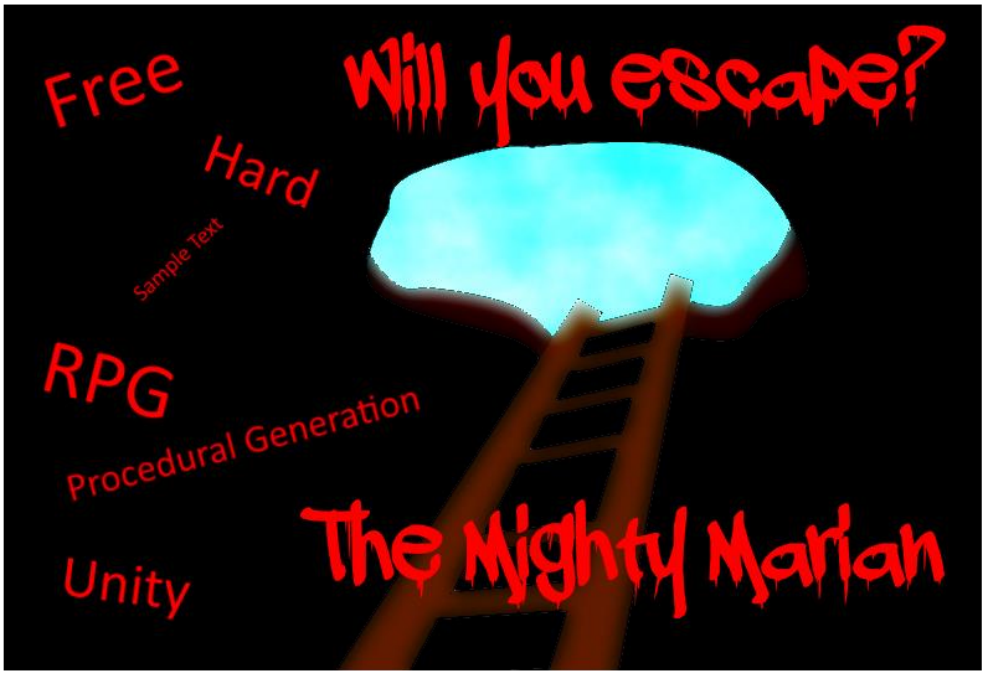
1. Kensler A.: *Correlated Multi-Jittered Sampling*, http://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf, (data dostępu 26.11.2014 r.).
2. Balmer: *Skeleton and Ghost spritesheets*, http://opengameart.org/content/skeleton-and-ghost-spritesheets-ars-notoria, (data dostępu 24.11.2014 r.).
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/*, (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.Adam.s: 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)
14. M. Mahardy: Roguelikes: The rebirth of the counterculture, IGN http://uk.ign.com/articles/2014/07/04/roguelikes-the-rebirth-of-the-counterculture (data dostępu 21.11.2014 r.).
15. Skrót licencji, Creative Commons https://creativecommons.org/licenses/by-nc-sa/4.0/ (data dostępu 21.11.2014 r.).
16. Brackeys, Inventory System https://www.assetstore.unity3d.com/en/#!/content/10384 (data dostępu 21.11.2014 r.).
17. J. Carmack.: Kod źródłowy Quake 2 https://github.com/id-Software/Quake-2, (data dostępu 4.12.2014 r.).
18. M. Benitez.: Advanced Zombie Bloody screen pack http://armedunity.com/gallery/image/92-blood2/ (data dostępu 16.11.2014 r.).
19. Unity Asset Store https://www.assetstore.unity3d.com (data dostępu wrzesień – grudzień 2014 r.).
20. Strona domowa programu Paint.Net, http://www.getpaint.net/, (data dostępu 2.11.2014 r.).
21. Strona domowa programu IrfanView, http://www.irfanview.com/, (data dostępu 2.11.2014 r.).

WYKAZ RYSUNKÓW

WYKAZ TABEL

7 DODATEK A: PLAKAT GRY

Rysunek A.1 Plakat w języku polskim



Rysunek A.2 Plakat w języku angielskim