Spis treści

[Streszczenie 3](#_Toc405398127)

[Abstract 4](#_Toc405398128)

[1 Wprowadzenie 5](#_Toc405398129)

[1.1 Wprowadzenie do dziedziny pracy dyplomowej 5](#_Toc405398130)

[1.2 Technologie, algorytmy, narzędzia i inne aplikacje 5](#_Toc405398131)

[1.3 Proces tworzenia pracy dyplomowej 5](#_Toc405398132)

[2 Projekt sytemu 5](#_Toc405398133)

[2.1 Cel i przeznaczenie systemu 5](#_Toc405398134)

[2.1.1 Założenia i cel tworzenia aplikacji 5](#_Toc405398135)

[2.1.2 Planowany sposób rozpowszechnienia aplikacji 5](#_Toc405398136)

[2.1.3 Ograniczenia dotyczące projektu aplikacji 6](#_Toc405398137)

[2.2 Specyfikacja wymagań użytkowych 6](#_Toc405398138)

[2.2.1 Ogólna charakterystyka systemu 6](#_Toc405398139)

[2.2.2 Zbiór wymagań użytkowych 6](#_Toc405398140)

[2.2.3 Zakres funkcjonalny (wymagania funkcjonalne) 6](#_Toc405398141)

[2.2.4 Środowisko pracy systemu 6](#_Toc405398142)

[2.2.5 Wymagania jakościowe 6](#_Toc405398143)

[2.2.6 Wymagania projektowo-wdrożeniowe 6](#_Toc405398144)

[2.2.7 Kryteria akceptacji 6](#_Toc405398145)

[2.3 Projekt systemu 6](#_Toc405398146)

[2.3.1 Projekt architektury systemu 6](#_Toc405398147)

[2.3.2 Przypadki użycia 6](#_Toc405398148)

[2.3.3 Diagram klas 6](#_Toc405398149)

[2.3.4 Projekt interfejsu użytkownika 6](#_Toc405398150)

[2.3.5 Wybór środowiska implementacji 6](#_Toc405398151)

[2.3.6 Projekt danych 6](#_Toc405398152)

[2.3.7 Rozszerzalność aplikacji 6](#_Toc405398153)

[2.3.8 Zastosowane algorytmy 6](#_Toc405398154)

[3 Implementacja, testowanie, walidacja i weryfikacja 6](#_Toc405398155)

[3.1 Implementacja systemu 6](#_Toc405398156)

[3.2 Testowanie 6](#_Toc405398157)

[3.3 Walidacja 6](#_Toc405398158)

[3.4 Weryfikacja założeń użytkowych aplikacji 6](#_Toc405398159)

[3.5 Podsumowanie i wnioski 6](#_Toc405398160)

[3.6 Możliwość dalszego rozwoju 7](#_Toc405398161)

[4 Generowanie proceduralne grach 7](#_Toc405398162)

[4.1 Geneza 7](#_Toc405398163)

[4.2 Najczęściej używane algorytmy 7](#_Toc405398164)

[4.2.1 Labirynty 7](#_Toc405398165)

[4.2.2 Gra w życie 7](#_Toc405398166)

[4.2.3 Drzewa BFS 7](#_Toc405398167)

[4.3 Przykładowe zastosowania 7](#_Toc405398168)

[5 Generowanie map w grze The Mighty Marian 7](#_Toc405398169)

[5.1 Pożądany efekt, poprawność mapy 7](#_Toc405398170)

[5.2 Automat komórkowy - gra w życie 8](#_Toc405398171)

[5.3 Etapy procesu generowania mapy 9](#_Toc405398172)

[5.3.1 Etap labiryntu 9](#_Toc405398173)

[5.3.2 Położenie przejść między pokojami 11](#_Toc405398174)

[5.3.3 Etap pokoju 12](#_Toc405398175)

[5.3.4 Etap łączenia i wygładzania 13](#_Toc405398176)

[5.3.5 Etap erozji 14](#_Toc405398177)

[5.3.6 Efekt końcowy 15](#_Toc405398178)

[5.4 Rozmieszczenie gracza i wrogów 16](#_Toc405398179)

[5.4.1 Wyznaczenie początku i końca poziomu 16](#_Toc405398180)

[5.4.2 Pozycje początkowe wrogów 16](#_Toc405398181)

[Bibliografia 18](#_Toc405398182)

[Wykaz rysunków 18](#_Toc405398183)

[Wykaz tabel 18](#_Toc405398184)

# Streszczenie

# Abstract

# Wprowadzenie

## Wprowadzenie do dziedziny pracy dyplomowej

## Użyte technologie, algorytmy, narzędzia i inne aplikacje

### Unity oraz środowisko MonoDevelop

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.

Unity w darmowej wersji udostępnia okrojony zestaw funkcjonalności

### System kontroli wersji

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

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

### Narzędzia graficzne

### Komunikatory internetowe

## Proces tworzenia pracy dyplomowej

# Projekt sytemu

## Cel i przeznaczenie systemu

### Założenia i cel tworzenia aplikacji

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

Docelowa grupa wiekowa

### Planowany sposób rozpowszechnienia aplikacji

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

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

### Ograniczenia dotyczące projektu aplikacji

## Specyfikacja wymagań użytkowych

### Ogólna charakterystyka systemu

### Zbiór wymagań użytkowych

### Zakres funkcjonalny (wymagania funkcjonalne)

### Środowisko pracy systemu

### Wymagania jakościowe

### Wymagania projektowo-wdrożeniowe

### Kryteria akceptacji

## Projekt systemu

### Projekt architektury systemu

### Przypadki użycia

### Diagram klas

### Projekt interfejsu użytkownika

### Wybór środowiska implementacji

### Projekt danych

### Rozszerzalność aplikacji

### Zastosowane algorytmy

# Implementacja, testowanie, walidacja i weryfikacja

## Implementacja systemu

## Testowanie

## Walidacja

## Weryfikacja założeń użytkowych aplikacji

## Podsumowanie i wnioski

## Możliwość dalszego rozwoju

# Generowanie proceduralne grach

## Geneza

## Najczęściej używane algorytmy

### Labirynty

### Gra w życie

Metoda zastosowana w projekcie The Mighty Marian jest połączeniem gry w życie oraz labiryntów.

### Drzewa BFS

## Przykładowe zastosowania

# Generowanie map w grze The Mighty Marian

W naszym projekcie każda rozgrywka polega na przejściu kilku, dziesięciu do piętnastu, poziomów jaskiń. Każda z jaskiń generowana jest niezależnie, przy użyciu algorytmu stworzonego na potrzeby projektu. Generowany obszar, pomimo tego, że w grze wizualnie reprezentowany jest w trzech wymiarach, na etapie generacji traktujemy, jako 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.

## Pożądany efekt, poprawność mapy

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

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

## 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.

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.

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

## Etapy procesu generowania mapy

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

### Etap labiryntu

Proces generowania mapy rozpoczyna się od stworzenia labiryntu, który określi czy pomiędzy wybranymi pokojami występuje połączenie. Parametrami istotnymi w tej fazie jest liczba pokojów, jakie chcemy uzyskać. W poniższych przykładach wybrana liczba to 4 w wymiarze X i 4 w wymiarze Y, więc razem 16.

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|= RoomsX \* RoomsY*

E - zbiór krawędzi - krawędź reprezenuje przejście między pokojami

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

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

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

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

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

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

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

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

### Położenie przejść między pokojami

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

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

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

### Etap pokoju

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

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

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% powierzchni pokoju. Jeżeli dane ułożenie komórek nie spełnia któregoś 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 i obrysowany prostokątem, ale również pokojów o kształcie wąskich korytarzy, sterując wspomnianym prawdopodobieństwem początkowym, bez obawy o zbyt długi czas oczekiwania na rozwiązanie.

### Etap łączenia i wygładzania

Kiedy wszystkie pokoje są już 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 repezentowana jest przez cztery komórki w tabeli map. Na tak przygotowanej planszy wykonywana jest jedna iteracja automatu komórkowego o parametrach 5678/5678 w celu wygładzenia powiększonej mapy.

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

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

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

### Etap erozji

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

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

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

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

### Efekt końcowy

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

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

## Rozmieszczenie gracza i wrogów

### Wyznaczenie początku i końca poziomu

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

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

### Pozycje początkowe wrogów

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

# Wykaz Literatury

1. Kensler A.: *Correlated Multi-Jittered Sampling*, http://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf, (data dostępu 26.11.2014 r.).
2. Hely, T.: How to use BSP trees to generate game maps, <http://gamedevelopment.tutsplus.com/tutorials/how-to-use-bsp-trees-to-generate-game-maps--gamedev-12268>, (data dostępu 24.11.2014 r.).

# Wykaz rysunków

# Wykaz tabel