Narzędzie wspomagające proceduralną generację poziomów dla gier typu roguelike w systemie Unity

(Supporting tool for procedural generation of levels for games of roguelike genre in Unity engine)

Vladyslav Yablonskyi

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

24 lutego 2023

Streszczenie

W dzisiejszych czasach gry komputerowe to jedna z najpopularniejszych rozrywek. Prawdopodobnie każdy, kto kiedykolwiek miał styczność z komputerem, miał też styczność z grą. Jednym z najpopularniejszych gatunków gier komputerowych wśród twórców niezależnych jest gatunek roguelike. Kluczowe cechy gatunku to generowanie proceduralnych poziomów gry, a także wysoki poziom trudności i permanentna śmierć. Te cechy sprawiają, że gry tego gatunku są niezwykle grywalne. W tej pracy zostanie opisany proces tworzenia narzędzie do generacji proceduralnej w Unity, wspomagającego potencjalnym deweloperom gier stworzenie własnej produkcji.

Nowadays computer games are one of the most popular kind of entertainment. Probably everyone who had anything to do with computers, stumbled upon a video games some day. One of the most popular game genre among indie developers is roguelike. The key points of roguelikes are procedural level generation, high difficulty, and permanent death. These features ensure high replayability for such games. This thesis contains a description of the process of creating a tool for the procedural level generation of Unity Engine, which will help potential game developers to make their own products.

Spis treści

1. Wstęp 6
    1.1. Motywacja 6
    1.2. Wybór silnika 7
    1.3. PCG w grach 7
    1.4. Cel 8
2. Analiza rynku 10
3. Teoria 12
    3.1. Proceduralne generowania zawartości 12
    3.2. Wady generacji proceduralnej 13
4. Implementacja 15
    4.1. Algorytm A* znajdowania najkrótszej ścieżki 15
    4.2. Algorytm Kruskala 16
    4.3. Algorytm błądzenie losowego 18
    4.4. Interfejs 18
    4.5. Ogólny algorytm generowania poziomu typu Hub 21
5. Instrukcja użytkownika 24
    5.1. Redaktor pokoju 24
    5.2. Ustawienia generatora 25
    5.3. Generowanie poziomu 28
6. Instrukcja dla programisty 29
    6.1. Instalacja i konfiguracja 29
    6.2. Narzędzia 29
7. Wnioski 30

1. Wstęp

Tworzenie gier to już nie tylko zajęcie dla wielkich korporacji. Korzystanie z platform dostarczania gier, takich jak *Steam*[1] czy *Epic Games Store*[2], pozwala mniejszym, niezależnym deweloperom mieć większą szansę na to żeby znaleźć swoje miejsce w rozwijającym się rynku gier.

I chociaż takie gry mogą być ekscytujące i interesujące, nadal potrzebują znacznej ilości zawartości, aby zainteresować graczy i zmotywować ich do zakupu gry. Tworzenie zawartości gry zajmuje dużo czasu i wysiłku, zwłaszcza jeśli tworzeniem zajmuje się jedynie kilkuosobowy zespół.

Mając możliwość stworzenia przynajmniej części tej zawartości automatycznie pozwoli deweloperom skupić się na innych aspektach, utrzymując wysoki poziom zawartości gry. Ten zautomatyzowany proces nazywa się proceduralną generacją zawartości (PCG – Procedural Content Generation).

Jednym z pierwszych przykładów wykorzystania generacji była gra *Rogue*[3] w której wykorzystano stochastyczne algorytmy do tworzenia labiryntu złożonego z połączonych korytarzami pomieszczeń, po których poruszała się postać gracza. Proceduralna generacja umożliwiała stworzenie praktycznie nieograniczonej liczby unikalnych poziomów oraz rozmieszczanie w nich przedmiotów i przeciwników. (stąd nazwa gier zbliżonych do niej tematycznie -– roguelike). Teraz PCG można spotkać nie tylko w grach gatunku roguelike. Na przykład FPS (First Person Shooter) [4] używa generacji proceduralnej do generowania broni, a w grach survivalowych (*Minecraft*[5], *Terraria*[6], *Don't starve*[7]) generowane są całe światy gry.

1.1. Motywacja

Tworzenie zawartości gry: wrogów, przedmiotów, poziomów, zajmuje czas i jest kosztowne. W porównaniu z komputerem ludzie pracują dość wolno. Jeśli jakaś część gry może zostać wygenerowana algorytmicznie, zaoszczędzi to czas i pieniądze developera. W przypadku niezależnych studiów taki algorytm pozwoli konkurować z grami AAA[8] pod względem ilości zawartości.

Ponadto gry z generacją online[Rozdział 3.1.] pozwalają znaleźć coś ciekawego nawet po wielu godzinach rozgrywki i mają potencjalnie nieskończoną zawartość. W przypadku gier typu roguelike PCG jest jedną z kluczowych cech.

[1] https://store.steampowered.com

[2] https://store.epicgames.com/

[3] https://store.steampowered.com/app/1443430/Rogue/

[4] *Borderlands 2* https://store.steampowered.com/app/49520/Borderlands_2/

[5] https://www.minecraft.net/pl-pl

[6] https://store.steampowered.com/app/105600/Terraria/

[7] https://store.steampowered.com/app/219740/Dont_Starve/

[8] https://pl.wikipedia.org/wiki/AAA_(gry_komputerowe)

1.2. Wybór silnika

Wraz ze wzrostem popularności gier komputerowych rosła dostępność narzędzi do ich tworzenia. Tworzenie gier ma wiele powtarzalnych zadań, które są obecne w większości projektów, takich jak renderowanie, odtwarzanie dźwięku, obsługa danych wejściowych, symulacja fizyki, itp. Aby ułatwić tworzenie gier, powstały silniki gier, które mają zaimplementowane rozwiązania do tych zadań. Silnik gry to w pełni funkcjonalna aplikacja lub platforma do tworzenia gier. Same silniki gier dzielą się na dwa typy.

Ogólnego przeznaczenia, takie jak *Unity*[9] i *UnrealEngine*[10], posiadają szeroki zestaw narzędzi, który pozwala na tworzenie gier różnego typu i gatunku, ale zazwyczaj są trudne do opanowania. Wyspecjalizowane, dla określonego typu gier lub jednej konkretnej gry, takie jak *RPGMaker*[11] dla tworzenia gier gatunku JRPG (*ang.* Japanese role-playing game) lub *RenPy*[12] dla tworzenia powieści wizualnych.

Do tej pracy dyplomowej wybrałem silnik *Unity*, który działa poprzez proste skrypty zachowań napisanych w języku C$^\sharp$. Wybrałem go ze względu na jego popularność, zwłaszcza wśród niezależnych twórców (stanem na 2020 rok *Unity* miął dwa miliardy aktywnych użytkowników końcowych miesięcznie[13]), wieloplatformowość i łatwość użytkowania.

1.3. PCG w grach

W dzisiejszych czasach proceduralna generacja zawartości przestała być domeną gier typu roguelike. Zaczęła być powszechna w różnych produkcjach w branży. W tym podrozdziale przedstawione zostaną przykłady jej zastosowań w grach różnych gatunków.

Dobrym przykładem wykorzystania PCG do tworzenia świata jest *Minecraft*[Rysunek 1]. Generator wykorzystuje spójne szumy, takie jak simpleks szum i szum Perlina, do tworzenia świata gry[14]. Takie funkcje są ciągłe, różniczkowalne i mogą być obliczone dla dowolnej liczby. Wynik takiej funkcji należy do przedziału [-1,1]. Ważne jest to, że te funkcje zależą od początkowego ziarna, więc żeby wygenerować ten sam świat, wystarczy tylko podać odpowiednie ziarna generacji.

Left 4 Dead, strzelanka kooperacyjna, wykorzystuje generację proceduralną, aby urozmaicić rozgrywkę. *The AI Director*[16] system który ciągle analizujący zachowanie

[9] https://unity.com
[10] https://www.unrealengine.com/en-US
[11] https://www.rpgmakerweb.com
[12] https://www.renpy.org
[13] https://www.theverge.com/2020/8/24/21399611/unity-ipo-game-engine\
-unreal-competitor-epic-app-store-revenue-profit
[14] https://minecraft.fandom.com/wiki/Seed_(level_generation)
[15] https://minecraft.fandom.com/wiki/Windswept_Hills
[16] https://developer.valvesoftware.com/wiki/Info_director

Rysunek 1: Przykład gór generowanych proceduralnie w *Minecraft*[15]

graczy i w zależności od aktualnej lokalizacji, liczby punktów zdrowia, posiadanych przedmiotów oraz innych parametrów określających stan gracza, może umieścić na mapie dodatkowych przeciwników, amunicję i inne obiekty, w celu dynamicznego dostosowania tempa i poziomu trudności rozgrywki. Oprócz zmian poziomu, *The AI Director* zadaje nastrój za pomocą efektów wizualnych i dynamicznej muzyki.

Innym znanym przykładem jest gra *Middle-earth: Shadow of War*[17]. Gra używa system *Nemesis*[18], który tworzy unikatowe relacje z każdym wrogiem i poplecznikiem. Ten system sprawia, że przeciwnicy pamiętają wcześniejsze interakcje z postacią gracza, i nadaje wrogom różne charakterystyczne cechy. Dzięki temu zwykły, ork napotkany przez gracza na początku gry może z czasem stać się realnym zagrożeniem, nękającą protagonistę na każdym kroku. Taki system tworze ciekawych przeciwników które dynamicznie reagują na działania gracza.

1.4. Cel

Celem tej pracy jest stworzenie narzędzia do generowania poziomów w 3D dla gier z gatunku roguelike dla Unity. Narzędzie powinno działać z zestawem płytek modułowych (*ang.* modular tileset), pozwalać na stworzenie własnych predefiniowanych pomieszczeń, które można wykorzystać do wygenerowania poziomu. Interfejs narzędzia musi być prosty i intuicyjny (pasować do stylu interfejsu w Unity, wtedy użytkownikowi będzie łatwiej przyzwyczaić się niego). Ważna jest również szybkość generowania poziomów. Algorytm generowania musi działać wystarczająco szybko, aby móc generować poziom nie tylko offline, ale także online [Rozdział 3.1.] czyli w trakcie działania gry.

[17]https://store.steampowered.com/app/356190/Middleearth_Shadow_of_War/
[18]https://patents.google.com/patent/US20160279522A1/en

Narzędzie będzie służyć tylko do generowania poziomów, inne elementy gry takie jak wrogowie czy przedmioty nie będą na nim umieszczane (w dalszym rozwoju narzędzia warto dodać możliwość umieszczania wrogów i nagród). Wygenerowany poziom nie może mieć niewykorzystanych, zbędnych elementów, takich jak niedostępne pomieszczenia czy korytarze ze ślepym zaułkiem.

2. Analiza rynku

Na rynku istnieje wiele narzędzi do proceduralnego generowania zawartości. Jednym z najbardziej znanych narzędzi do generowania scen jest *Gaia Pro – Rapid World Creator*[19] używana do generowania terenu. Główne zalety *Gaia Pro* to szybkie generowanie terenu, łatwość obsługi, skalowalność, wysoka wydajność oraz integracja z *Unity* 2019.3+. Główne wady to – wysoka cena oraz to, że użytkownik nie może dodawać własnych prefabrykatów. Z tego powodu wszystkie generowane tereny są podobne.

Rysunek 2: Przykład terenu wygenerowanego za pomocą *Gaia Pro*[20]

.

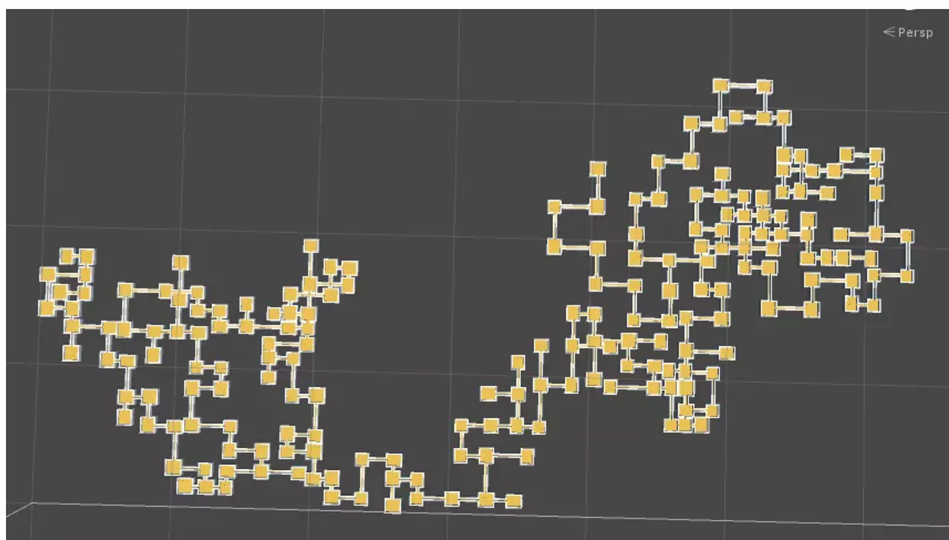
Jednym z najpopularniejszych narzędzi do generowania poziomów lochów w *Unity Asset Store* jest *Dungeonizer – Easy Random Dungeon Generator*[21]. Ten generator poziomów współpracuje z Unity 2019.3.1 i nowszymi. *Dungeonizer* umożliwia generowanie pomieszczeń zarówno w 3D, jak i 2D. Posiada prosty i przejrzysty interfejs wykonany w oknie inspektora *Unity*. Pozwala na użycie własnych prefabrykatów, ale dla każdego typu klatki można ustawić tylko jeden prefabrykat, co nieco ogranicza możliwości generatora. Również ma inne ograniczenia: wszystkie pomieszczenia są prostokątne i przez to wyglądają monotonnie, posiada tylko jeden rodzaj generacji poziomów oraz brakuje parametrów generacji, przez co wszystkie wygenerowane poziomy są do siebie podobne.

[19] https://www.procedural-worlds.com/products/professional/gaia-pro

[20] https://www.procedural-worlds.com/products/professional/gaia-pro

[21] https://assetstore.unity.com/packages/templates/systems/dungeonizer-easy-random-dungeon-generator-22162

[22] https://assetstore.unity.com/packages/templates/systems/dungeonizer-easy-random-dungeon-generator-22162



Rysunek 3: Przykład poziomu ze strony produktu Dungeonizer[22]

3. Teoria

3.1. Proceduralne generowania zawartości

W tej sekcji bardziej szczegółowo omówiono proceduralne generowanie zawartości.

Online and offline generation

Pierwszą kluczową rzeczą jest to, że generowanie proceduralne można podzielić na dwa typy. Generowanie online, które generuje zawartość podczas ładowania (przy starcie lub na ekranie ładowania) lub w sposób ciągły podczas rozgrywki. Oraz generowanie offline, gdy deweloperzy używają PCG dla generowania zawartość na etapie tworzenia gry.

Rysunek 4: Przykład proceduralnie generowanej broni z *Borderlands 2*[23]

Generowanie online ma surowe wymagania dotyczące wydajności algorytmu. Musi być szybki i zapewniać przewidywalne rezultaty. We współczesnych grach generowanie online służy zwykle do generowania wrogów (zwykle dostosowywane są tylko parametry wroga), nagród i animacji. Na przykład w grze *Borderlands 2* generator łączy różne predefiniowane części broni, aby tworzyć różne typy broni z

[23]https://borderlands.fandom.com/wiki/Borderlands_2_Weapons

wieloma odmianami szybkostrzelności, szybkości przeładowania, rodzaju obrażeń itp [Rysunek 4].

Generowanie offline jest często używane jako narzędzie programistyczne, które pomaga zaoszczędzić wysiłek i czas. Po wygenerowaniu poziomu, wroga, broni lub innej zawartości za pomocą takiego algorytmu, projektant edytuje ją i dopiero po tym zawartość jest dodawana do gry. Na przykład artyści 3D ze studia *4A Games* przy tworzeniu potworów do gry *Metro 2033* używali narzędzie, które generuje trójwymiarową sylwetkę potwora, która następnie była edytowana.

Jak wspomniano powyżej, tworzony w ramach tej pracy algorytm generowania musi działać zarówno w trybie offline, jak i online.

Zawartość niezbędna i opcjonalna

Zawartość poziomu można podzielić na obowiązkową i opcjonalną. Obowiązkowa zawartość to taka, którą gracz musi zobaczyć/pokonać, aby ukończyć grę. Przeciwnicy, których trzeba pokonać lub przedmioty, które gracz musi zebrać, aby ukończyć poziom. Zawartość opcjonalna to zawartość, którą gracz może zignorować, nie jest konieczna do pokonania poziomu. Na przykład dodatkowa broń lub sekretny pokój. Zawartość obowiązkowa to podstawa gry, dlatego ona zawsze musi być poprawnie generowana. Na przykład, jeśli nie ma sposobu, aby dostać się do ostatniego pokoju, to ten poziom nie ma sensu.

Parametry generatora

Najprostsze generatory generują treść wyłącznie na podstawie ziarna generacji. Takie generatory nie są bardzo konfigurowalne. Innym sposobem jest przesłanie do generatora wektora parametrów, z których każdy odpowiada za określone właściwości generowanej zawartości.

Ponieważ celem tej pracy jest stworzenie elastycznego narzędzia, tworzony algorytm będzie akceptował plik konfiguracyjny, z którego będzie odczytywał parametry.

3.2. Wady generacji proceduralnej

Generacja proceduralna w grach daje graczowi poczucie, że doświadcza czegoś nowego. Dobrze zrobiona generacja proceduralna jest dobrym sposobem na zapewnienie, że gracze stale napotykają nowe sytuacje, w których można zastosować wiedzę o mechanikach gry. W przeciwnym razie gracz może się znudzić lub zdezorientować wynikiem generowania.

Bez zestawu reguł generowanie proceduralne może stworzyć prawdziwy bałagan. Tworzone poziomy mogą być trudne lub zbyt łatwe lub po prostu nudne. Wygenero-

wany poziom może być po prostu zbiorem losowych elementów, które w żaden sposób nie zaangażują gracza. Dobrym pomysłem jest stworzenie wielu prefabrykatów, które generator może używać i kombinować.

Generowanie proceduralne zawartości to świetne narzędzie dla deweloperów, które jednak wymaga dużo pracy, aby uzyskać dobre wyniki.

4. Implementacja

4.1. Algorytm A* znajdowania najkrótszej ścieżki

```
1 reference
public static Tuple<List<GridTile>, List<GridTile>> AStar(LevelGrid grid, GridTile start, GridTile end,
    bool avoidIntersection=false, bool straightPath=true)
{
    for (int y = 0; y < grid.gridRealSize.y; y++)
    {
        for (int x = 0; x < grid.gridRealSize.x; x++)
        {
            GridTile tile = grid.GetCorridorMapTile(x, y);
            tile.g = tile.initG;
            tile.f = Int32.MaxValue;
        }
    }

    start.f = Heuristic(start, end);

    List<GridTile> closedList = new List<GridTile>();
    List<GridTile> openList = new List<GridTile>() { start };
    Dictionary<Vector2Int, Vector2Int> parents = new Dictionary<Vector2Int, Vector2Int>();
    parents.Add(start.gridCoordinates, new Vector2Int(-1, -1));

    Func<GridTile, bool> Filter = t => (t.tag.map != TileMap.room || t.gridCoordinates == grid.FromCorridorToRoomMap(end).gridCoordinates);

    while (openList.Count != 0)
    {
        int r;
        if (straightPath)
            r = 0;
        else
            r = UnityEngine.Random.Range(0, Math.Min(openList.Count, 5));

        GridTile currentNode = openList[r];

        openList.RemoveAt(r);
        closedList.Add(currentNode);

        if (currentNode.gridCoordinates == end.gridCoordinates)
        {
            return Backpropagate(grid, start, end, parents);
        }

        List<GridTile> adjacent = grid.GetAdjacentTiles(currentNode);
        for (int i = 0; i < adjacent.Count; i++)
        {
            if (!closedList.Contains(adjacent[i]))
            {
                if (avoidIntersection && adjacent[i].tag.avoideInPathFinding)
                    continue;
                if (grid.CanConvertCorridorToRoomMap(adjacent[i]) && !Filter(grid.FromCorridorToRoomMap(adjacent[i])))
                    continue;
                if (!openList.Contains(adjacent[i]))
                {
                    int newG = currentNode.g + (int)Vector2Int.Distance(currentNode.gridCoordinates, adjacent[i].gridCoordinates);
                    adjacent[i].g = newG;
                    adjacent[i].f = newG + Heuristic(end, adjacent[i]);
                    parents[adjacent[i].gridCoordinates] = currentNode.gridCoordinates;
                    openList.Add(adjacent[i]);
                    openList.OrderBy(x => x.f).ToList();
                }
            }
        }
    }
    throw new OperationCanceledException("Cant find path from " + start.gridCoordinates + " to " + end.gridCoordinates);
}
```

Rysunek 5: Implementacja algorytmu A^*.

Aby stworzyć korytarze, mój algorytm musi najpierw znaleźć ścieżkę z jednego pokoju do drugiego, na której może zbudować korytarz. Do znalezienia takich ścieżek użyto algorytmu A^*.

A^* to algorytm heurystyczny odnajdujący najkrótszą ścieżkę pomiędzy dwoma wierzchołkami w grafie ważonym. On jest często wykorzystywany w sztucznej inteligencji oraz w grach komputerowych do imitowania inteligentnego zachowania. W mojej realizacji funkcją heurystyczną jest metryka odległości Manhattanu.

4.2. Algorytm Kruskala

W projekcie do reprezentacji grafu wybrano reprezentację w postaci macierzy. Mając macierz G, jeżeli istnieje krawędź pomiędzy pokojami i, j, to wartość

$$G_{ij} \text{ jest równa odległości między środkami pokoi } i, j.$$

W przeciwnym razie wartość

$$G_{ij} \leq 0.$$

Chcemy mieć pewność, że do każdego pomieszczenia można dotrzeć. Czyli chcemy, aby nie było niepołączonych pokoi. W tym celu w pewnym momencie generacji, korzystając z algorytmu Kruskala, budujemy minimalne drzewo rozpinające. Używamy tego drzewa do określenia, które pokoi będą połączone korytarzami.

Dodatkowo zaimplementowana jest funkcja, która wyznacza czy skonstruowany graf jest cykliczny. [Rysunek 7]

```
private static float[][] BuildMinimalSpaningTree(int roomsNumber, List<Tuple<int, int, float>> edges)
{
    edges = edges.OrderBy(x => x.Item3).ToList();
    int edgesCount = 0;
    float[][] graph = new float[roomsNumber][];
    for (int i = 0; i < roomsNumber; i++)
    {
        graph[i] = new float[roomsNumber];
        for (int j = 0; j < roomsNumber; j++)
            graph[i][j] = -1;
    }
    while (edgesCount != roomsNumber - 1)
    {
        if (edges.Count == 0) { throw new Exception("all edges has been visited"); }
        Tuple<int, int, float> newEdge = edges[0];
        edges.RemoveAt(0);


        graph[newEdge.Item1][newEdge.Item2] = newEdge.Item3;
        graph[newEdge.Item2][newEdge.Item1] = newEdge.Item3;

        if (CycleSearch(graph))
        {
            graph[newEdge.Item1][newEdge.Item2] = -1;
            graph[newEdge.Item2][newEdge.Item1] = -1;
        }
        else
            edgesCount++;
    }

    return graph;
}
```

Rysunek 6: Metoda budowania drzewa rozpinającego.

```
2 references
private static bool CycleSearch(float[][] graph)
{
    List<int> visited = new List<int>();
    List<int> finished = new List<int>();
    for (int i = 0; i < graph.Length; i++)
    {
        if (HaveCycle(graph, i, -1, ref visited, ref finished))
            return true;
    }

    return false;
}

2 references
private static bool HaveCycle(float[][] graph, int node, int prevNode, ref List<int> visited, ref List<int> finished)
{
    if (finished.Contains(node))
        return false;
    if (visited.Contains(node))
        return true;
    visited.Add(node);
    List<int> children = Neigbours(graph, node, prevNode);
    for (int i = 0; i < children.Count; i++)
    {
        if (HaveCycle(graph, children[i], node, ref visited, ref finished))
            return true;
    }
    visited.Remove(node);
    finished.Add(node);
    return false;
}
```

Rysunek 7: Metoda sprawdzania czy graf jest cykliczny.

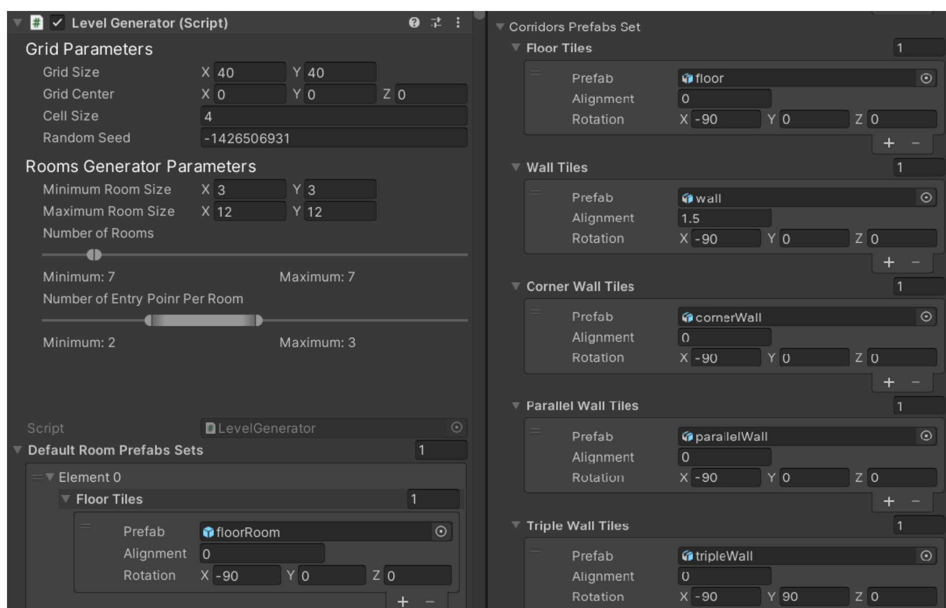
4.3. Algorytm błądzenie losowego

Algorytm błądzenie losowego używany w generatorze *Main path*[24] do wyboru pozycji pokoi. Algorytm ten jest używany, aby rozmieszczenie pomieszczeń było spójne i wyglądało organicznie. Agent rozpoczyna ruch na losowo wybranej komórce siatki. W każdym kroku losowo wybiera kardynalny kierunek ruchu i umieszcza w tej komórce pokój (jeśli on nie ma kolizji z innymi już umieszczonymi pokojami). Agent powtarza tą akcję, dopóki wszystkie pokoje nie zostaną umieszczone. W ten sposób umieszczone pokoje można łatwo łączyć tak, aby tworzyły ścieżkę z pomieszczenia początkowego do końcowego.

4.4. Interfejs

Pojawiły się pewne trudności z implementacją interfejsu. *Unity* ma dwie standardowe opcje implementacji interfejsu. Pierwszą z nich jest modyfikacja standardowego okna inspektora[25]. To rozwiązanie dobrze pasuje do skryptów, które nie mają wielu parametrów i są ściśle związane z GameObject[26].

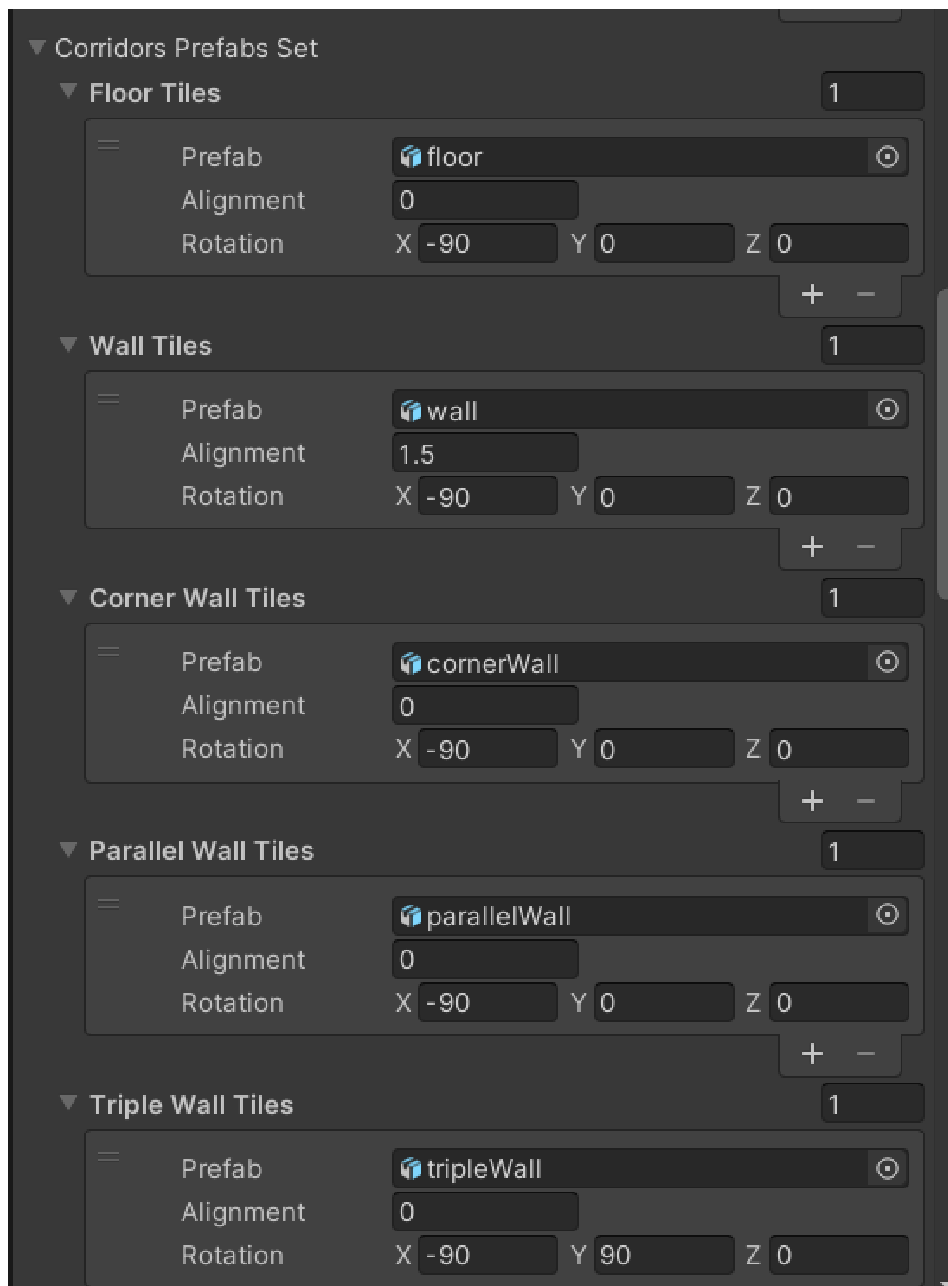
Drugą opcją jest stworzenie własnego okna edytora. Ten sposób jest bardziej elastyczny, pozwala tworzyć okna dowolnego typu i nie wiąże okna z konkretnym obiektem.



Rysunek 8: Przykład implementacji interfejsu w oknie inspektora

Początkowo interfejs został zaimplementowany w oknie inspektora, ponieważ ten spośob jest prostszy. Ale wraz z rozwojem aplikacji liczba parametrów rosła. Trudno

[24] generator ma kilka różnych metod generowania poziomu
[25] https://docs.unity3d.com/Manual/UsingTheInspector.html
[26] https://docs.unity3d.com/560/Documentation/Manual/class-GameObject.html



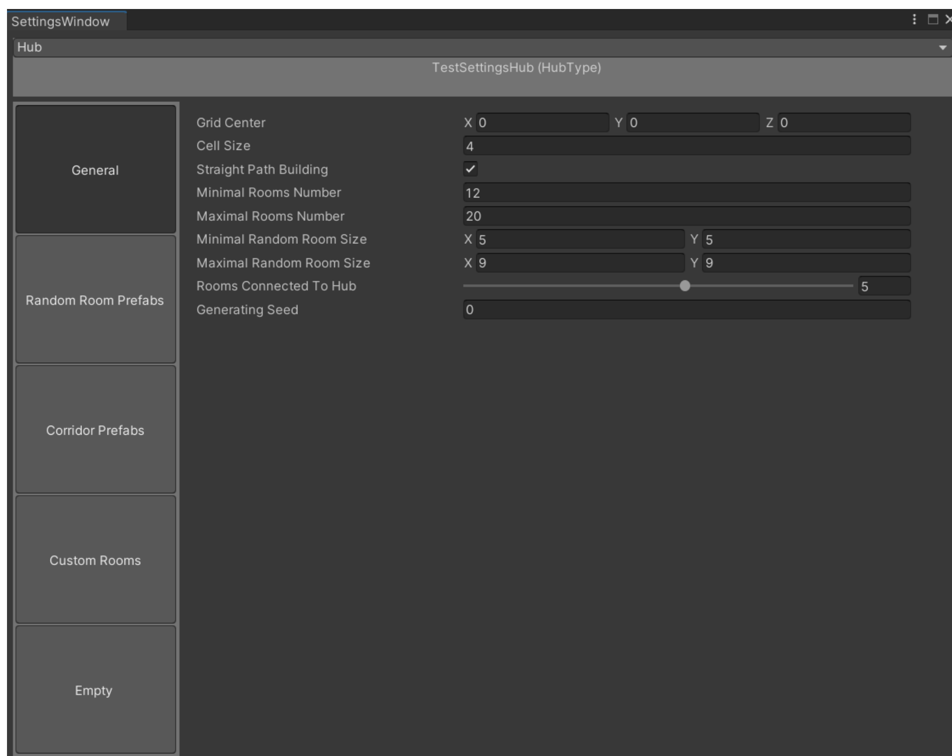
Rysunek 9: Przykład implementacji interfejsu w oknie inspektora

było coś odczytać lub zmodyfikować w inspektorze.

Postanowiłem przerobić interfejs. Stworzyłem osobną klasę dla ustawień GenerationSettings, która dziedziczy po ScriptableObject[27]. Takie rozwiązanie pozwala na zapisanie ustawień generatora jako osobnych plików, z których będą pobierane parametry generowania.

Teraz, mając plik konfiguracyjny, użytkownik musi mieć interfejs do pracy z nim. Klasa SettingsWindow implementuje taki interfejs. Wyświetla wszystkie para-

[27] https://docs.unity3d.com/Manual/class-ScriptableObject.html



Rysunek 10: Ostateczny interfejs. Zaimplementowany jako osobne okno

metry generatora i umożliwia ich modyfikację. Dodatkowo przeprowadza walidację, na przykład minimalna wielkość pomieszczenia nie może być większa niż jego wielkośc maksymalna.

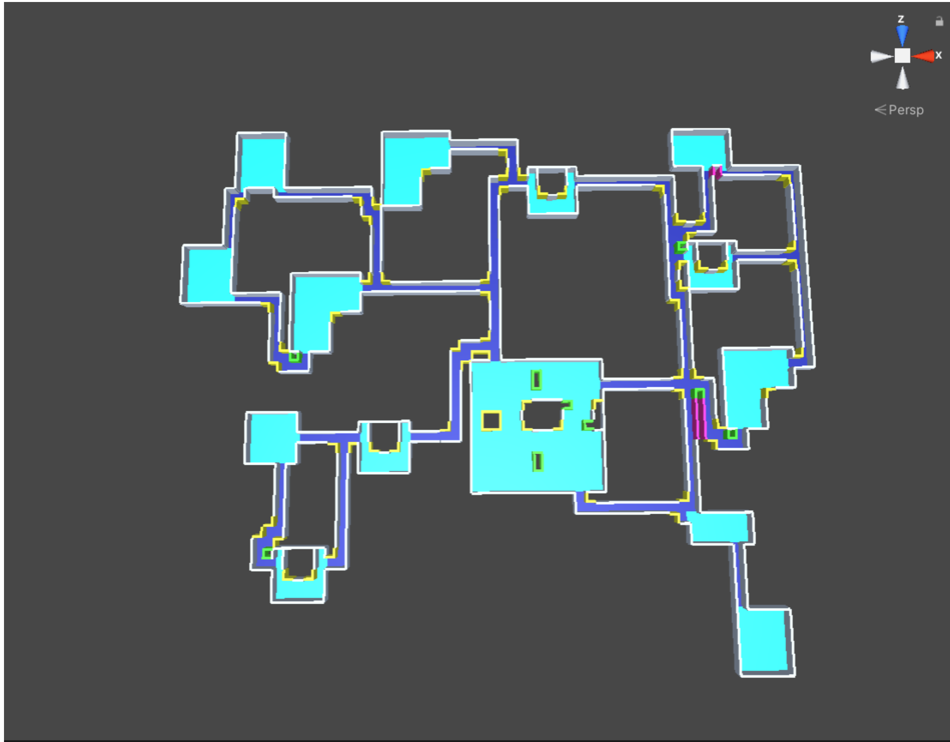
4.5. Ogólny algorytm generowania poziomu typu Hub

Ogólna koncepcja poziomu typu Hub jest następująca. Gracz rozpoczyna poziom w pokoju Hub, który ma kilka wyjść. Sam poziom podzielony jest na grupy połączonych pokoi, niektóre z grup mają połączenia między sobą.

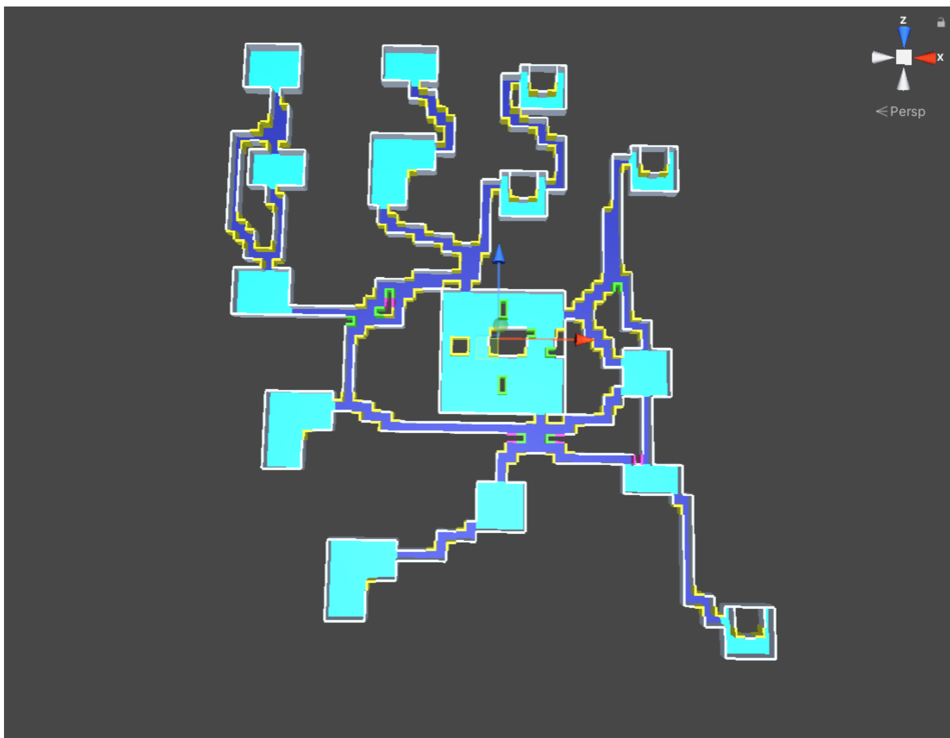
Algorytm działa w następujący sposób:

1. Pobieramy parametry z pliku konfiguracyjnego (w tym pomieszczenia, które mają być użyte do generacji)
2. Tworzymy wszystkie potrzebne pokoje i jeden z nich zaznaczamy jako hub (centralny pokój).
3. Umieszczamy wszystkie pokoje z wyjątkiem centralnego na wyimaginowanej siatce. Następnie umieszczamy hub na środku siatki, a wszystkie pokoje, z którymi on ma kolizje są usuwane z siatki i ponownie umieszczane w innych miejscach.
4. Dalej tworzymy graf odległości między pomieszczeniami w postaci macierzy. Wszystkie kolejne kroki używają tego grafu.
5. Tworzymy graf połączeń między pokojami. Łączymy kilka sąsiednich pokoi z hubem (w zależności od podanych parametrów generacji).
6. Dodajemy do tego grafu nowe krawędzi, aż otrzymamy drzewo rozpinające. W ten sposób mamy kilka grup pokoi połączonych między sobą a hubem.
7. Najmniejsze (pod względem liczby pokoi) grupy łączymy jednym lub kilkoma korytarzami.
8. W największych grupach tworzymy dodatkowe połączenia.
9. Korzystając ze skonstruowanego grafu tworzymy korytarze pomiędzy odpowiednimi pokojami.

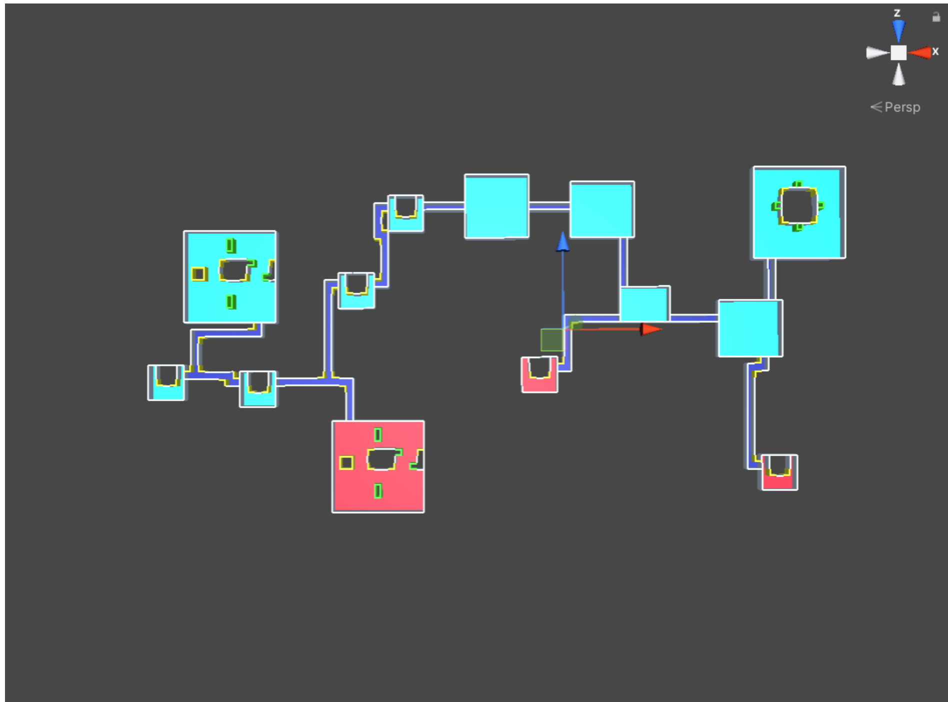
Poniżej znajdują się przykłady wygenerowanych poziomów. Dla jasności różne elementy są oznaczone różnymi kolorami. Podłoga w opcjonalnym pokoju oznaczona jest kolorem czerwonym, w zwykłym pokoju jest zaznaczona na jasnoniebieski, a podłoga korytarza na ciemnoniebieski. Innymi kolorami zaznaczone różne rodzaje ścian.



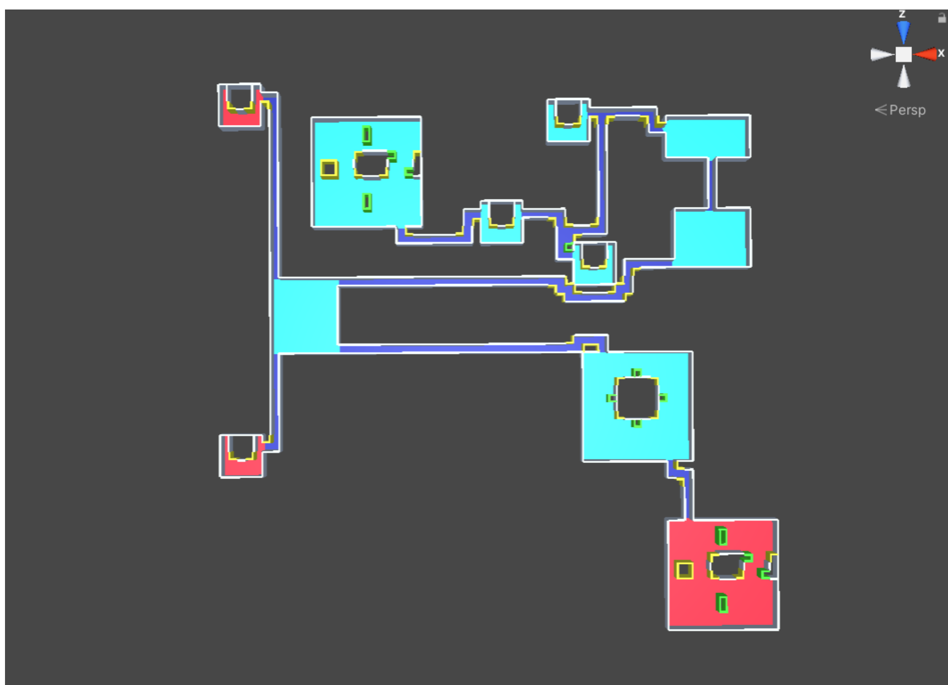
Rysunek 11: Przykład utworzonego poziomu typu Hub



Rysunek 12: Przykład utworzonego poziomu typu Hub z wyłączoną opcją 'straight path building'



Rysunek 13: Przykład utworzonego poziomu typu Main Path

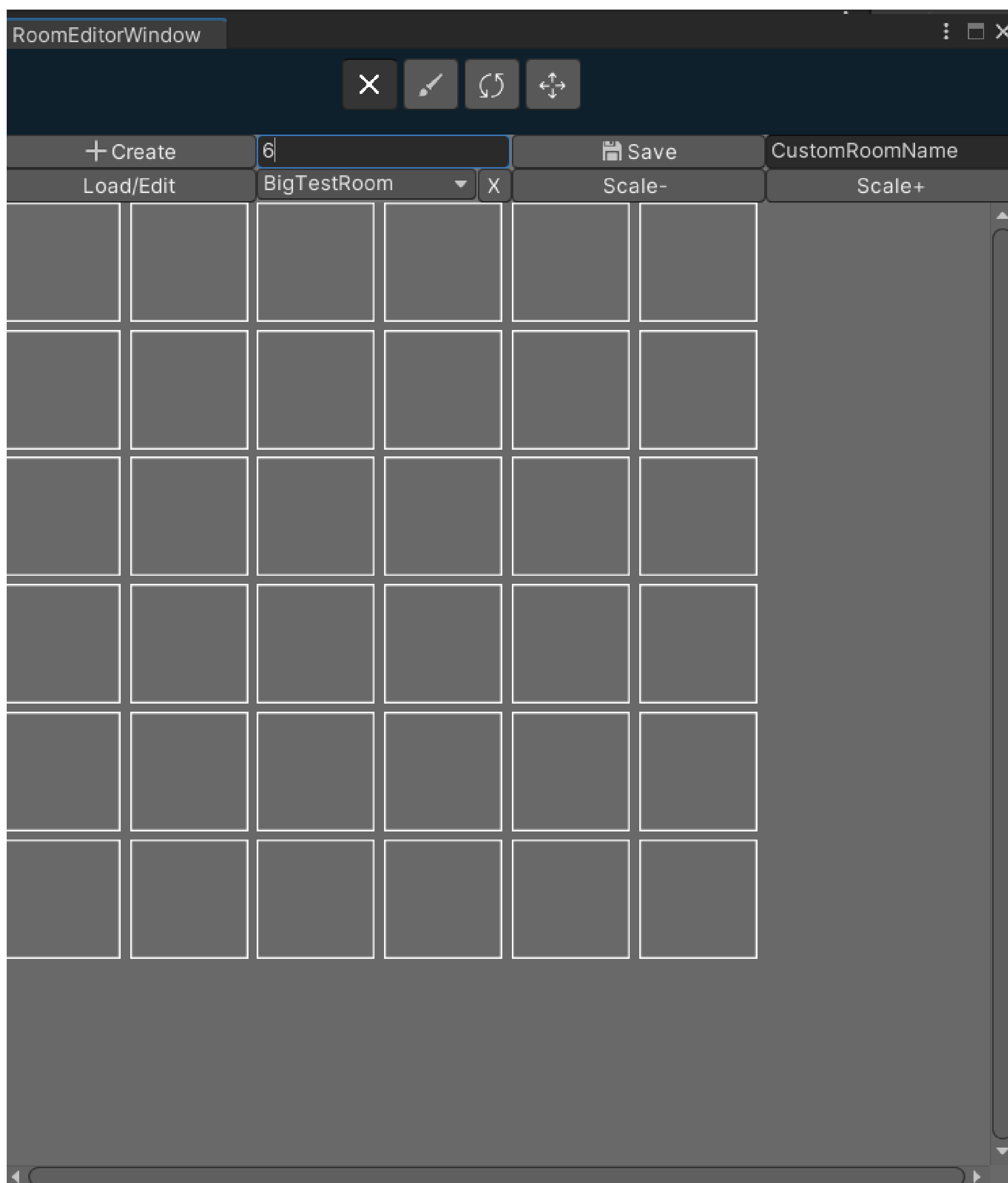


Rysunek 14: Przykład utworzonego poziomu typu Main Path

5. Instrukcja użytkownika

Generator jest bardzo łatwy w wykorzystaniu. Istnieją dwa główne komponenty, z którymi użytkownik będzie pracować. Pierwsza to redaktor pokoju i druga to konfiguracja generatora.

5.1. Redaktor pokoju



Rysunek 15: Okno redaktora.

Aby otworzyć okno redaktora pokoju użytkownik musi otworzyć w *Unity* Win-

dow > Room Editor.

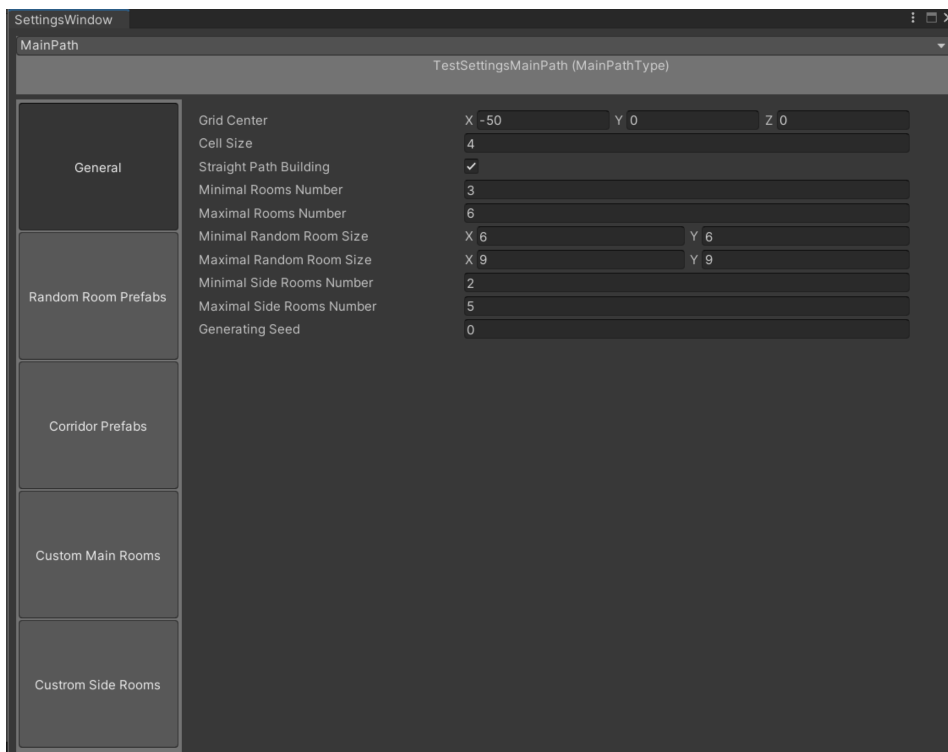
Aby stworzyć nowy pokój, trzeba zadać jego rozmiar (musi być większy od 2) w pole obok przycisku *Create* i kliknąć *Create*. Aby zapisać pokój trzeba wprowadzić nazwę nowego pokoju i nacisnąć *Save*. Można także wczytywać istniejące pokoje za pomocą *Load/Edit*. W pobliżu menu wyboru pokoju znajduje się również przycisk do usunięcia wybranego pokoju. Zmienić rozmiar widoku za pomocą *Scale-* i *Scale+*.

Na górze okno redaktora[Rysunek 15] można zobaczyć pasek narzędzi.

1. Delete — służy do oczyszczenia pola.
2. Paint — służy do zmiany typu pola.
3. Rotate — służy do obracania pola.
4. Align – służy do wyboru wyrównania pola.

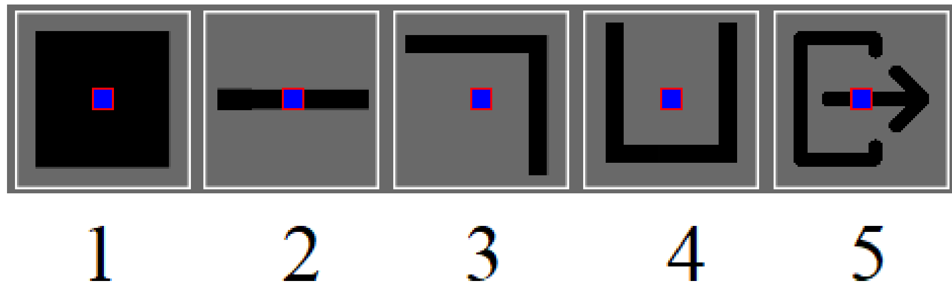
5.2. Ustawienia generatora

Żeby utworzyć nowy plik konfiguracyjny dla generatora trzeba otworzyć menu kontekstowe w zakładce projektu i wybrać "Create > Generation Settings". Po otwarciu utworzonego pliku należy wybrać rodzaj generacji. Od rodzaju generacji zależy zawartość okna konfiguracji.

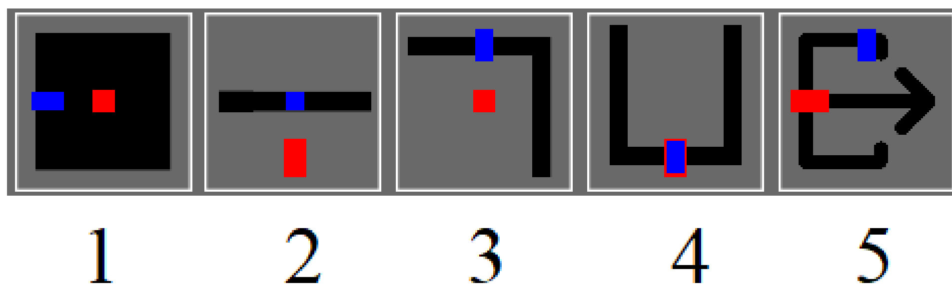


Rysunek 16: Okno konfiguracji.

Tiles



Rysunek 17: Płytki



Rysunek 18: Płytki z ustawionym obrotem i wyrównaniem.

Dostępnych jest pięć rodzajów płytek.

1. Floor tile
2. Wall tile
3. Corner tile
4. Triple wall tile
5. Entrance tile (określona klatka będzie służyć jako wejście/wyjście)

Na rysunku 18 są widoczne linie pomocnicze. Czerwone wskazują obrót obiektu (obrót odbywa się względem osi z), a niebieskie za jego wyrównanie (odległość wyrównania podawana jest podczas dodawania obiektu w pliku konfiguracyjnym).

Poniżej znajduje się opis poszczególnych zakładek okna konfiguracji.

General

- Grid center — współrzędne środka generowanego poziomu.
- Cell size — wielkość jednej komórki.

- Straight path building — jeśli ta opcja jest włączona, korytarze będą budowane wzdłuż najbardziej optymalnej ścieżki używając linii prostych.
- Minimal rooms number — minimalna liczba pokoi.
- Maximal rooms number — maksymalna liczba pokoi.
- Minimal random rooms size -– minimalna wielkość losowego pokoju.
- Maximal random rooms size — maksymalna wielkość pokoju losowego.
- Generation seed – seed generacji (ustawiony na zero oznacza, że na początku generacji wybieramy losowy seed).
- Rooms connected to hub — liczba pokoi podłączonych do hub'a.
- Minimal side rooms number -– minimalna liczba pokoi pobocznych.
- Maximal side rooms number – maksymalna liczba pokoi pobocznych.

Random room prefabs

Tutaj można skonfigurować prefabrykaty dla losowych pokoi. Do wygenerowania pokoju jest wybierany jeden z zestawów. W zestawie dla każdego rodzaju płytek definiujemy listę prefabrykatów. Dodatkowo dla każdego z prefabrykatów możemy skonfigurować jego obroty i wyrównanie.

Corridor prefabs

Te same opcje co w *Random room prefabs* z tym wyjątkiem, że dla korytarzy mamy tylko jeden możliwy zestaw.

Custom rooms

Ta zakładka pozwala dodać gotowe pokoje. Tak jak poprzednio możemy skonfigurować zestawy. Ponadto, każdy zestaw ma swój przepisany pokój i dodatkowe parametry, takie jak: szansa generowania, maksymalna liczba wystąpień oraz dodatkowe flagi *IsStartRoom* i *IsEndRoom*.

Custom side rooms (tylko w generatorze *Main Path*)

Ta zakładka pozwala dodać gotowe pokoje, które zostaną dodane do zbioru pokoi pobocznych. Ma taki sam interfejs co *Custom rooms* za wyjątkiem tego, że flagi *IsStartRoom* i *IsEndRoom* nie mają wpływy na generowanie poziomu.

5.3. Generowanie poziomu

Aby wygenerować poziom trzeba dodać do obiektu z hierarchii skrypt *LevelGenerator*. W pole *generation settings* dodać żądaną konfigurację i kliknąć *Generate*. Jako alternatywę, można stworzyć skrypt, który przekaże parametr do *generation settings* i wywoła funkcję *Generate* z *LevelGenerator*.

6. Instrukcja dla programisty

6.1. Instalacja i konfiguracja

Dla instalacji na urządzeniu musi być zainstalowane *Unity3D* w wersji przynajmniej 2020.3. Żeby uruchomić aplikację, trzeba wykonać następujące procedury:

1. Sklonować repozytorium z GitHub:

   `https://github.com/Vlad311010/DungeonGenerationTool-Rework-.git`

2. Skopiować foldery *Editor*, *Resources* i *Scripts* do plików *Assets* w hierarchii folderów projektu *Unity*. (Pozostałe pliki zawierają modeli i materiały których można użycz do testowania.)

3. Teraz *Unity* automatycznie skompiluje projekt.

4. W folderze "`c:/Users/<User_name>/AppData/LocalLow/<Unity_Project_Company>/<Project_name>`" stworzyć folder *CustomRoomsData*.

6.2. Narzędzia

- Jako IDE wybrałem *Visual Studio 2019 Community*. Jest to potężne środowisko, z wieloma możliwościami rozszerzenia, mnogością wtyczek i ułatwień programistycznych (IntelliSense, inteligentne wyszukiwanie, autoformatowanie) oraz wbudowaną integracją z systemem kontroli wersji *Git*.

- Do modelowania użyłem *Blender*'a.

- System kontroli wersji – GitHub.

7. Wnioski

Celem tej pracy było stworzenie narzędzia dla *Unity*, które pozwoli łatwo i szybko tworzyć poziomy do gier 3D z gatunku roguelike. Można powiedzieć, że główny cel pracy udało się osiągnąć. Narzędzie skutecznie generuje poziomy. Może pracować zarówno w trybie online, jak i offline, umożliwia konfigurowanie generatora i pozwała używać własnych prefabrykatów.

Niestety nie udało się zrealizować część pomysłów. Jednym z nich jest zmniejszenie liczby generowanych obiektów. Ponieważ generator tworzy obiekt dla każdej klatki. W obecnej realizacji zwykły pokój 10×10 będzie się składać z ponad stu obiektów. Za pomocą metody łączenia siatek wielokątów[28] (*ang.* mesh) można połączyć wszystkie obiekty w jedną siatkę, co znacznie zwiększa wydajność aplikacji.

Dodatkowo, aby ułatwić dystrybucję, narzędzie powinno zostać przekształcone w pakiet *Unity*. Ułatwi to nie tylko proces instalacji narzędzia, ale także umożliwi jego dystrybucję za pośrednictwem *Unity Asset Store*.

[28] `https://docs.unity3d.com/ScriptReference/Mesh.CombineMeshes.html`

Bibliografia

[1] Unity documentation. https://docs.unity3d.com/ScriptReference

[2] Noor Shaker, Julian Togelius, Mark J. Nelson: Procedural Content Generation in Games. 2016

[3] Unity manual. https://docs.unity3d.com/Manual/editor-CustomEditors.html

[4] Patrick Lester: A* Pathfinding for Beginners. http://csis.pace.edu/ benjamin/teaching/cs627/ webfiles/Astar.pdf. July 18, 2005

[5] Kleinberg, Jon; Éva Tardos (2006); https://ict.iitk.ac.in/wp-content/uploads/CS345-Algorithms-II-Algorithm-Design-by-Jon-Kleinberg-Eva-Tardos.pdf; ISBN 0-321-29535-8