



Politechnika
Śląska

PROJEKT INŻYNIERSKI

Implementacja proceduralnej generacji modelu terenu zawierającego
latające wyspy w silniku Unity

Michał POKRZYWA

Nr albumu: 295732

Kierunek: Informatyka

Specjalność: Grafika i oprogramowanie

PROWADZĄCY PRACĘ

dr hab. inż. Agnieszka Szczęsna, prof. PŚ.

KATEDRA Grafiki, Wizji Komputerowej i Systemów Cyfrowych
Wydział Automatyki, Elektroniki i Informatyki

Gliwice 2024

Implementacja proceduralnej generacji modelu terenu zawierającego latające wyspy w silniku Unity

Streszczenie

Celem niniejszej pracy dyplomowej było zaimplementowanie proceduralnej generacji modelu terenu zawierającego latające wyspy w silniku Unity. Implementacja obejmuje tworzenie terenu, wybieranie modelu terenu do wygenerowania, proceduralne poszerzanie terenu, a także umożliwia zapisywanie i wczytywanie wygenerowanych terenów. Ostateczny rezultat projektu został przedstawiony w postaci programu na platformę Windows, który demonstruje możliwości generowania terenu oraz modyfikowania parametrów.

Słowa kluczowe

Unity, Teren, Generacja, Proceduralność, Latające Wyspy

Implementation of procedural generation of a terrain model containing flying islands in the Unity engine

Abstract

The purpose of this thesis was to implement the procedural generation of a terrain model containing flying islands in the Unity engine. The implementation includes terrain creation, selection of a terrain model to generate, procedural terrain expansion, and allows saving and loading of generated terrains. The final result of the project is presented in the form of a Windows platform program that demonstrates terrain generation and parameter modification capabilities.

Key words

Unity, Terrain, Generation, Procedurality, Flying Islands

Spis treści

1	Wstęp	1
2	Analiza temat	3
2.1	Istniejące gry wykorzystujące proceduralne generowanie terenu	3
2.1.1	Gra Minecraft	4
2.1.2	Gra Astroneer	4
2.1.3	Gra No Man's Sky	4
2.2	Istniejące algorytmy wykorzystywane do tworzenia terenu	5
2.2.1	Szum Perlina	5
2.2.2	Ułamkowy ruch Browna	6
2.2.3	Maszerujące sześciiany	6
2.3	Dostępne rozwiązania w silniku Unity	7
2.3.1	Unity Terrain	7
2.3.2	Tworzenie siatek powierzchni terenu	7
2.3.3	World Creator i Gaia	8
2.4	Latające wyspy w różnych mediach	8
2.4.1	Media Filmowe	8
2.4.2	Gry	8
2.5	Analiza algorytmu latających wysp	9
2.6	Ostateczne Wybory Rozwiązań	10
3	Wymagania i narzędzia	13
3.1	Wymagania funkcjonalne	13
3.2	Wymagania niefunkcjonalne	13
3.3	Przypadki użycia	14
3.4	Opis użytych narzędzi	14
3.4.1	Silnik Unity	14
3.4.2	Język C# 9.0	15
3.4.3	Text Mesh Pro	16
3.4.4	Standard Unity Assets	16
3.4.5	IJobParrarel	16

3.4.6	Visual Studio 2022	16
4	Specyfikacja zewnętrzna	17
4.1	Wymagania programu VoxelGenerator	17
4.2	VoxelGenerator	17
4.2.1	Tworzenie nowego terenu	19
4.2.2	Wczytywanie terenu	21
4.2.3	Poruszanie się po stworzonym terenie	21
5	Specyfikacja wewnętrzna	23
5.1	Główna idea implementacji programu	23
5.2	Tworzenie funkcji szumowych	26
5.2.1	Szum ułamkowego ruchu Browna	27
5.2.2	Szum ułamkowego ruchu Browna w 3D	28
5.3	Skład modelu terenu	28
5.3.1	Modyfikowanie pojedynczej warstwy	29
5.3.2	Modyfikowanie warstwy jaskiń	29
5.3.3	Zapisywanie ustawień z warstw	30
5.4	Generator terenu	30
5.4.1	Przygotowanie generacji	30
5.4.2	Rozpoczęcie generacji	31
5.4.3	Proceduralne poszerzanie generowanego terenu	33
5.4.4	Tworzenie obiektu klasy ChunkBlock	34
5.4.5	Tworzenie obiektu klasy Block	35
5.4.6	Przypisanie rodzaju bloku	35
5.5	Zachowanie Stanu Terenu	36
5.5.1	Zapisywanie do pliku	36
5.5.2	Wczytanie terenu z pliku	36
5.6	Opis ważniejszych klas	37
5.6.1	WorldCreator	37
5.6.2	ChunkBlock	39
5.6.3	MeshUtils	39
5.6.4	Quad	40
5.6.5	Block	40
5.6.6	WorldSaver	41
5.6.7	CalculateBlockTypesJobs	42
5.6.8	WorldVisualization	43
5.6.9	PerlinGrapher	44
5.7	Diagram klas	46
5.8	Implementacja programu VoxelGenerator	46

5.8.1	Opis Sceny	46
5.8.2	Ustawienia Unity	47
6	Weryfikacja i walidacja	49
6.1	Testowanie w środowisku Unity	49
6.2	Testowanie stworzonej aplikacji	50
7	Podsumowanie i wnioski	51
7.1	Dalszy rozwój implementacji	52
	Bibliografia	54
	Spis skrótów i symboli	57
	Lista dodatkowych plików, uzupełniających tekst pracy	59
	Spis rysunków	61

Rozdział 1

Wstęp

Człowiek na przełomie XV i XVI wieku zaczął interesować się odkrywaniem nowych miejsc oraz zakątek naszej planety. Pobudzona została ciekawość, podróżowanie oraz poszerzona została wiedza geograficzna. Dzięki temu, w tym okresie zostało odkryte wiele części świata, które znamy do dziś. W obecnym wieku znamy niemalże każdy skrawek naszej planety, dlatego szukamy i próbujemy tworzyć nasze własne światy, czy również przemierzać stworzone przez innych uniwersa. Stworzone przez ludzką wyobraźnię światy znajdują się w różnych mediach jak książki, filmy czy gry komputerowe. W książkach uniwersum oraz teren są opisywane słownie, a ich wyobrażenie widzimy jedynie we własnym umyśle. W przypadku filmów czy gier komputerowych można zobaczyć choćby na ekranie świat stworzony przez ich twórców. W przypadku filmów jest on tworzony ręcznie, dopieszczony do drobnych szczegółów. W grach również można zobaczyć światy stworzone ręcznie, ale także wygenerowane przez algorytmy. W przypadku tworzenia automatycznego modelu terenu jest generowany proceduralnie przy pomocy różnych technik i algorytmów. Dzięki temu można stworzyć różne wyglądy terenu. Może to być teren na podstawie naszej planety, ale również pod postacią samotnych wysp na gigantycznym oceanie albo latających wysp unoszących się w powietrzu.

Celem projektu jest stworzenie proceduralnej implementacji generacji modelu terenu, w skład której zawarta jest generacja latających wysp w silniku Unity. Stworzona aplikacja ma za zadanie tworzyć teren przy pomocy wybranego generatora, którego parametry można modyfikować, gdzie jednym z dostępnych generatorów jest algorytm do tworzenia latających wysp. Efektem końcowym programu jest możliwość przemierzania wygenerowanego świata, który jest tworzony proceduralnie.

Celem pracy dyplomowej jest zaimplementowanie proceduralnej implementacji generacji modelu terenu zawierającego latające wyspy w silniku Unity, który odpowiada za:

- Graficzne przedstawienie stworzonego terenu.
- Możliwość graficzna zmiany parametrów w algorytmach.
- Zapis i wczytanie już wygenerowanych światów.
- Przemierzanie stworzonego świata.

Kolejne elementy pracy są następujące. W rozdziale 2 znajduje się analiza tematu, problemy, sposoby rozwiązania oraz które rozwiązania zostały wybrane do końcowej pracy. W 3 rozdziale zostały opisane wszystkie narzędzia, jakie użyłem w projekcie. W 4 rozdziale znajduje się opisana specyfikacja zewnętrzna, gdzie opiszę podstawowe działanie aplikacji, jak można zobaczyć wizualizację algorytmu oraz zapisu i odczytu wygenerowanego terenu do pliku. W 5 rozdziale jest opisana specyfikacja wewnętrzna, czyli logika generowania terenu, różnice w algorytmach generowania oraz opis najważniejszych metod i klas. Działanie i validacja programu znajduje się w rozdziale 6. Podsumowanie oraz końcowe wnioski pracy dyplomowej są zapisane w rozdziale 7.

Rozdział 2

Analiza temat

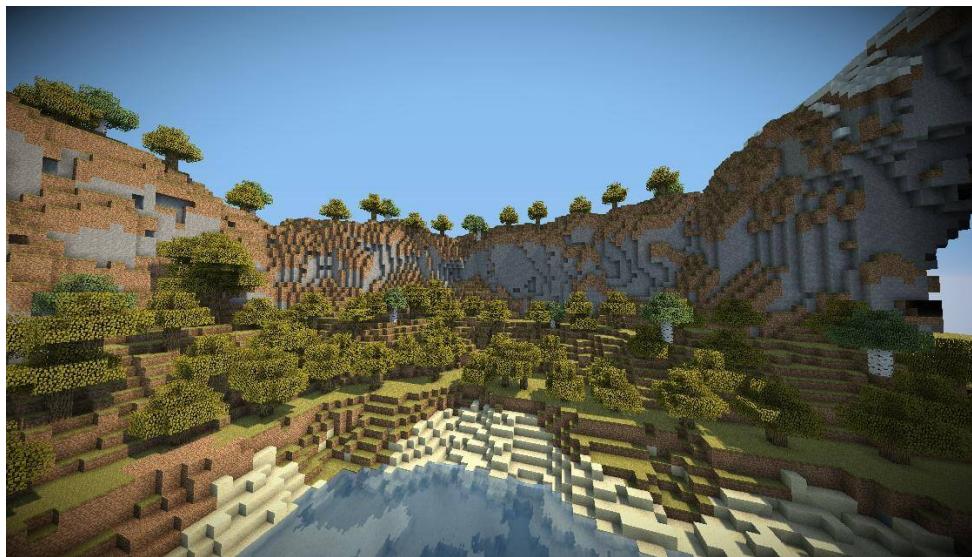
Celem pracy dyplomowej jest implementacja proceduralnej generacji modelu terenu zawierającego latające wyspy w silniku Unity. Główną częścią pracy jest stworzenie generatora, który będzie miał możliwość implementacji tworzenia wielu modeli teren, gdzie jeden z modeli są latające wyspy. Sama implementacja powinna mieć możliwość modyfikacji parametrów, aby generowane modele były zróżnicowane. Ważną częścią programu jest to, że powinien posiadać możliwość zapisu oraz wczytania stworzonych terenów. Ponadto powinna być możliwość przemierzania i poszerzania stworzonego terenu. W dodatku program powinien korzystać z wielowątkowości jak to tylko możliwe, aby program działał płynnie podczas jego używania.

2.1 Istniejące gry wykorzystujące proceduralne generowanie terenu

Gry, które wykorzystują proceduralne generowanie terenu, używają go jako część rozgrywki albo mechanik. Gra może wykorzystywać teren jako piaskownice, która pozwala graczowi przekształcanie tego placu zabaw w ramach jego wizji albo jako świat, w którym odkrywamy jego faunę i historię.

2.1.1 Gra Minecraft

Jednym z przykładów takiej gry jest *Minecraft* od firmy Mojang, który na początku przygody pozwalał wygenerować teren, na którym miała odbywać się rozgrywka. Teren w grze jest zbudowany z wokseli [14], które połączone ze sobą tworzą teren (rys. 2.1). Początkowy ekran posiadał kilka pól do modyfikacji, które pozwalały na możliwość zmiany generowanego terenu. W obecnej wersji gry (1.20.4 [8]) jest możliwość stworzenia modelu świata zaproponowanego przez twórców, ale również istnieje możliwość stworzenia własnego modelu świata [5]. Warto wspomnieć, że we wcześniejszych wersjach gry, była dostępna większa różnorodność modeli terenu takich jak latające wyspy czy jeden biom [6].



Rysunek 2.1: Przykład stworzonego terenu [20]

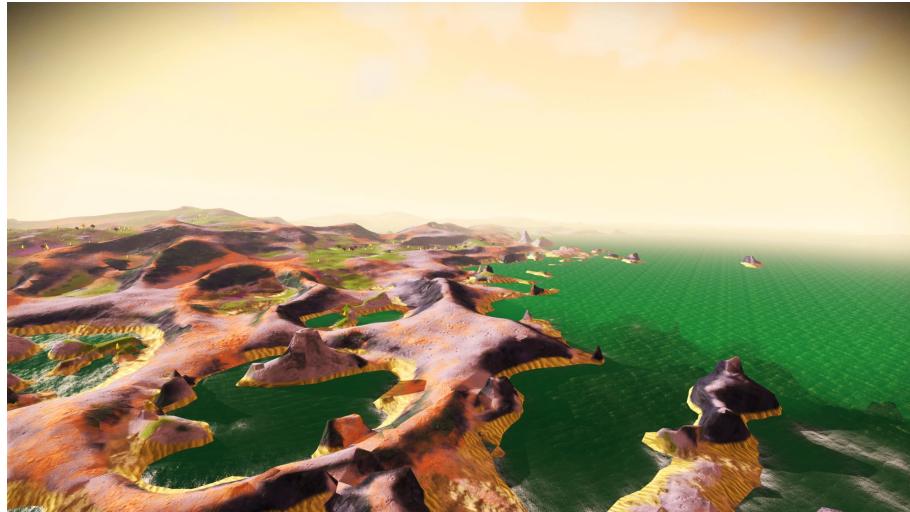
2.1.2 Gra Astroneer

Kolejnym przykładem takiej gry jest gra *Astroneer* od studia System Era Softworks, gdzie gracz modyfikuje teren, zbiera zasoby i buduje swoją kolonię na obcej planecie. Wygenerowany teren jest tworzony z małych wokseli, dzięki którym teren posiada bardziej naturalny kształt. Główną mechaniką w grze jest narzędzie do modyfikowania terenu, które pozwala nam teraformować powierzchnię oraz do zbierania surowców. W przeciwieństwie do gry *Minecraft* gracz nie ma możliwości modyfikowania parametrów generowania terenu.

2.1.3 Gra No Man's Sky

Ostatnim popularnym przykładem gry, która zawiera proceduralne generowanie terenu, jest gra *No Man's Sky* od studia Hello Games. Gra posiada poza proceduralnym

terenem, również posiada proceduralne planety oraz galaktyki. Tak jak w grze *Astro-neer*, gra nie ma możliwości modyfikowania parametrów tworzenia, jednak przy prawie nieskończonej ilości planet jest bardzo duża różnorodność wyglądu i rodzaju terenu, pokazywanego przez grę, co można zobaczyć na rys. 2.2.



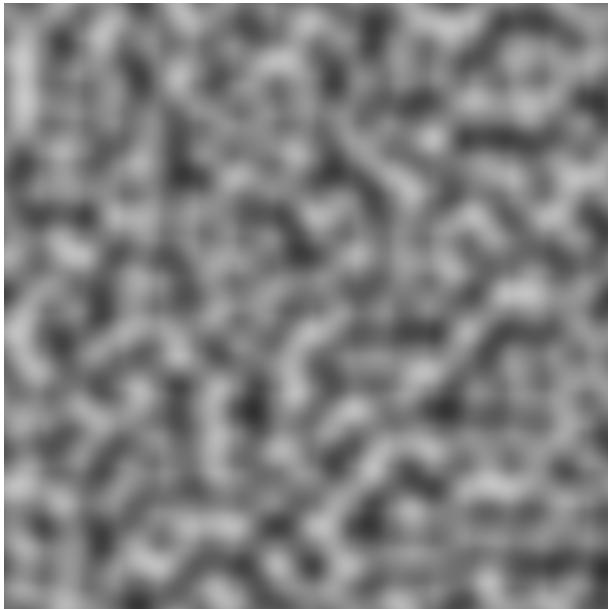
Rysunek 2.2: Przykład terenu na planecie [7]

2.2 Istniejące algorytmy wykorzystywane do tworzenia terenu

Podczas tworzenia terenu możemy zastosować różne algorytmy, aby teren wyglądał jak najbardziej naturalnie podobny do naszej plany albo abstrakcyjnie niepodobny dla naszego umysłu.

2.2.1 Szum Perlina

Szum Perlina (ang. Perlin noise) to rodzaj matematycznego narzędzia używanego w grafice komputerowej do generowania pseudolosowych i naturalnie wyglądających wzorców [1]. Dlatego jest szeroko stosowany w różnych dziedzinach, w tym w generacji modelów terenu w grach komputerowych. W skrócie szum Perlina generuje gładkie, ciągłe zmiany wartości w przestrzeni trójwymiarowej. W kontekście generacji terenu może być używany do tworzenia urozmaiconych krajobrazów, takich jak pagórki, doliny i góry, nadając im bardziej naturalny wygląd. Algorytm ten polega na dodawaniu różnych warstw "szumów" o różnych częstotliwościach i amplitudach, co prowadzi do uzyskania kompleksowego i realistycznego efektu. Rysunek 2.3 jest jednym z przykładów jego dwuwymiarowej formy, która może być wykorzystywana jako podstawowy element generowania terenu.



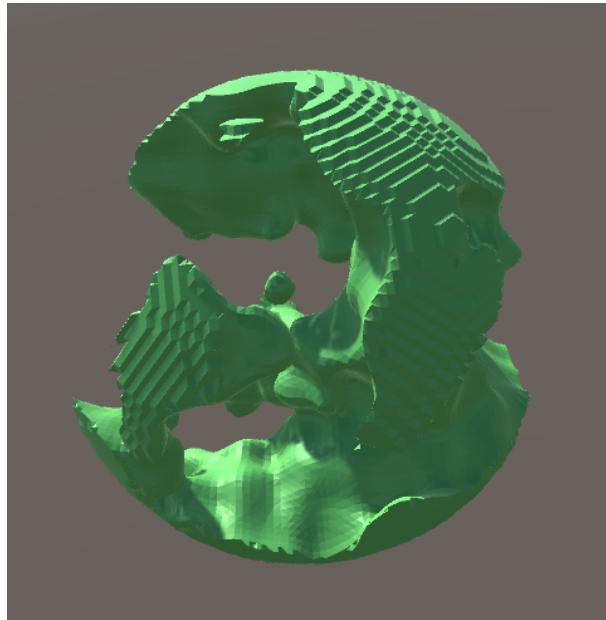
Rysunek 2.3: Wygląd tekstury 2D szumu Perlina reprezentowany w formie mapy wysokości [22]

2.2.2 Ułamkowy ruch Browna

Ułamkowy ruch Browna (ang. Fractional Brownian Motion (fBm)) to matematyczna koncepcja wykorzystywana w generowaniu terenu w grafice komputerowej. Jest to rodzaj szumu, który tworzy bardziej złożone i naturalne wzorce, podobnie jak w przypadku realistycznego terenu. Implementacja tego algorytmu polega na generowaniu kolejnych warstw szumu Perlina lub innej funkcji szumu, a następnie dodawaniu ich z różnymi wagami [13]. To pozwala na kontrolowanie skomplikowania i detaliczacji generowanego terenu. W praktyce ułamkowy ruch Browna często jest stosowany w połączeniu z innymi algorytmami generacji terenu, aby uzyskać bardziej zróżnicowane i realistyczne krajobrazy. Ten algorytm jest głównie wykorzystywany w grze *Minecraft* (2.1.1), gdzie do tworzenia każdej warstwy jest wykorzystywany ułamkowy ruch Browna w wersji 2D i 3D [23].

2.2.3 Maszerujące sześciany

Maszerujące sześciany (ang. Marching Cubes) to algorytm używany do generowania trójwymiarowych powierzchni na podstawie danych przestrzennych. Algorytm konwertuje trójwymiarowe dane skalarnie (np. wartości terenu) na siatkę trójkątów, co umożliwia wizualizację trójwymiarowych struktur, takich jak teren. Ten algorytm jest często wykorzystywany do wizualizacji medycznych obrazów rezonansu magnetycznego, ale również znalazł zastosowanie w grach komputerowych do generowania realistycznych terenów. Algorytm oprócz samego generowania powierzchni terenu, tworzy naturalnie wyglądające jaskinie lub obszary do akwarium [9]. Jednak użycie tego w komercyjnych rozwiązańach jest rzadkie, ponieważ użycie tego algorytmu jest dostępne od 2005 [19].



Rysunek 2.4: Algorytm Marching Cubes w użyciu; Źródło: <https://imgur.com/IsbIv8i>

2.3 Dostępne rozwiązania w silniku Unity

W silniku Unity znajdują się gotowe rozwiązania do tworzenia terenów, jak również narzędzia zewnętrzne przystosowane do samego silnika.

2.3.1 Unity Terrain

Popularnym gotowym rozwiązaniem w silniku Unity jest Unity Terrain. Jest to prosta płaszczyzna, która posiada przypięty do niego obiekt Terrain. Posiada możliwość ręcznego modyfikowania powierzchni przy pomocy różnorodnych pędzli, dodawanie drzew oraz dodawanie detali, takich jak trawa. Ponieważ wytworzony teren nie jest nieskończony, silnik pozwala na łączenie dodatkowo wytworzonych terenów razem z sobą [11]. Poza ręczną modyfikacją terenu z poziomu edytora silnika również jest opcja modyfikacji terenu z poziomu kodu. Dzięki temu rozwiązaniu można stworzyć teren proceduralnie przy wykorzystaniu algorytmów takich jak szum Perlin (2.2.1) czy przemieszczanie punktu środkowego (ang. MidPoint Displacement) [10].

2.3.2 Tworzenie siatek powierzchni terenu

W przypadku tworzenia siatek powierzchni terenu w silniku Unity istnieje również opcja ręcznego definiowania punktów kontrolnych, tworzenia trójkątów oraz przypisywania odpowiednich materiałów. Ten proces umożliwia pełną kontrolę nad kształtem terenu i dostosowanie detali do konkretnych wymagań projektu. Jest to bardziej zaawansowana technika, poprawnie zaimplementowana może przynieść satysfakcjonujące rezultaty.

2.3.3 World Creator i Gaia

Warto również wspomnieć, że do silnika Unity istnieją już gotowe narzędzia do tworzenia proceduralnych terenów. Z najbardziej popularnych narzędzi są World Creator i Gaia, które pozwalają na stworzenie artystycznego terenu w kilka minut. Jednak każde z tych narzędzi jest odpłatne przy wykorzystaniu.

2.4 Latające wyspy w różnych mediach

Różne media spróbowały swojej interpretacji latających wysp, takich jak gry czy filmy, jak widać na zbiorze rysunków numer 2.5. Aby zrozumieć, warto zobaczyć jak tam został wykorzystany model terenu.

2.4.1 Media Filmowe

Medium filmowe jeśli porusza temat latających wysp, to większość przypadków mówi o latających zamkach czy miastach w chmurach [21]. Najbardziej popularnym filmem, który poruszył temat naturalnych latających wysp, jest film *Avatar*, gdzie bohaterowie odwiedzali region nazwany "Góry Alleluja" (rys.2.5c). Są to latające góry, które cyrkulują w prądach magnetycznych [4]. Formacją tych wysp są góry, które mają rozmaite rozmiary i kształty. Wyglądem mogą przypominać pozbawione podstawy skalne iglice.

2.4.2 Gry

Medium gier częściej porusza temat latających wysp, ponieważ twórcy wykorzystują takie formacje w ramach elementów rozgrywki. Przykładem gry, która wykorzystuje latające wyspy jako element rozgrywki, jest *A story about my uncle*, która przy pomocy haku do chwytania porusza się między wyspami, które służą jako platformy dla gracza. Latające wyspy w tej grze przypominają latające stalaktyty, gdzie górna część jest pokryta zielenią, kiedy im niżej tym kamienna część jest węższa (rys.2.5d). Kolejnym przykładem gry, która wykorzystuje latające wyspy, jest gra *AER: Memories of old*, która wykorzystuje je w celach artystycznych, jako miejsca opowiadania historii gry. Wyspy przypominają bardziej naturalną formację, ponieważ mają bardziej zniekształconą dolną część, jak widać na rysunku 2.5a. W przypadku poprzednich gier wyspy są stworzone ręcznie przez artystów. Przypadkiem gry, która może tworzyć latające wyspy proceduralnie, jest gra *Minecraft* (2.1.1), gdzie jest możliwość wygenerowania modelu terenu latających wysp. Można zobaczyć na rysunku, że wyspy przypominają bardziej naturalny teren, gdzie jest podziurawiony przez jaskinie (rys.2.5b).



(a) Wygląd wysp w grze "AER: Memories of old"; Źródło: <https://www.ign.com/wikis/aer-memories-of-old/>



(b) Wygląd wysp w grze *Minecraft*; Źródło: <https://shorturl.at/syBSU>



(c) Wygląd wysp w filmie "Avatar"; Źródło: <https://static.wikia.nocookie.net/jamescameronavatar/images/b/b4/Samson.jpg/revision/latest?cb=20100201143445>



(d) Wygląd wysp w grze "A story about my uncle"; Źródło: <https://www.polygon.com/2014/5/12/5711578/a-story-about-my-uncle-steam-launch-may-28>

Rysunek 2.5: Przykłady wyglądu latających wysp w różnych mediach

2.5 Analiza algorytmu latających wysp

Latające wyspy składają się z dwóch części:

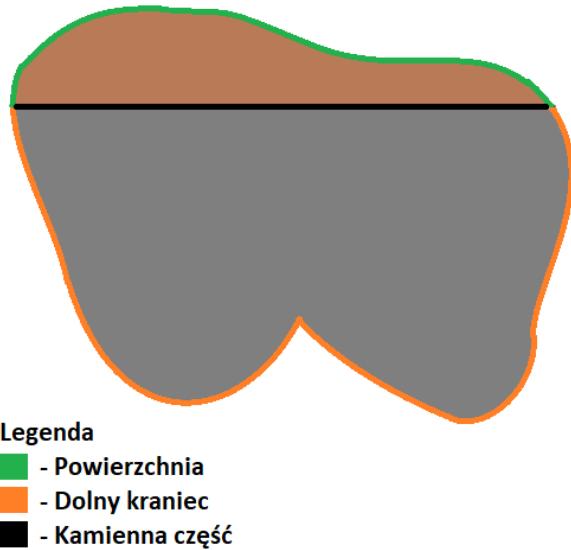
- Powierzchni, po której gracz może się poruszać. Mogą na niej znajdować się obiekty naturalne (drzewa, krzaki itp.) oraz obiekty stworzone przez gracza (budynki, farmy, pojazdy itp.).
- Części kamiennej "podtrzymującą" powierzchnię wyspy, która głównie jest złożona z kamienia. Zawartość tej części zmniejsza się wraz z wysokością.

Podczas tworzenia algorytmu, powinniśmy wyodrębnić przynajmniej 3 warstwy, które pozwolą nam stworzyć przybliżony wygląd latających wysp:

- Powierzchnia, która na wierzchu pokazuje elementy zielone i jest możliwa do prześwietlania przez gracza. Pod nią może znajdować się część ziemi, która nie jest widoczna i nie musi się zmniejszać wraz z wysokością.
- Część kamienna, która znajduje się pod powierzchnią, która powinna się zmniejszać wraz z wysokością albo zanikać.

- Dolny kraniec wyspy, który określi nam, gdzie kończy się dolna część latającej wyspy.

Latające wyspy mogą nie mieć dużej odległości między sobą albo być z sobą połączone (2.5b). Również wyspy mogą być oddalone od siebie o duże odległości. W takim przypadku istnieje algorytm próbkowania tarczowego Poissona (ang. Poisson Disk Samplink) [2], który pozwoli na rozrzucenie wysp w taki sposób, aby nie znajdowały się koło siebie.



Rysunek 2.6: Zobrazowanie warst latającej wyspy

2.6 Ostateczne Wybory Rozwiązań

Podsumowując analizę i dostępne rozwiązania, najlepszym rozwiązaniem jest stworzenie własnej implementacji figur w Unity, ponieważ podstawowy teren nie spełni wszystkich wymagań potrzebnych do stworzenia uniwersalnego generatora. Następnie główne algorytmy, które zostaną wykorzystane to szum Perlina (2.2.1) i ułamkowy ruch Browna (2.2.2), ponieważ spełniają wymaganie przy tworzeniu naturalnego terenu. Za pomocą algorytmu maszerujących sześciianów (2.2.3), może nie być możliwe na implementację dowolnego terenu. Istotną częścią implementacji programu jest, aby każdy z modeli terenu odwoływał się do tych samych zmiennych w generatorze podczas jego działania. Warto również zadbać o to, aby skomplikowanie każdego z modeli nie stanowiło ograniczenia, umożliwiając swobodną modyfikację istniejących modeli lub łatwą integrację nowych. Elastyczność implementacji samych modeli w generatorze terenu jest równie kluczowa, umożliwiając dynamiczne dostosowywanie parametrów i zachowując jednocześnie spójność w procesie generacji terenu. Ważne jest, aby struktura programu była zaprojektowana z myślą o przyszłych rozszerzeniach, co ułatwi utrzymanie i rozwijanie generatora w dłuższej perspektywie czasowej, dlatego główną inspiracją jest gra *Minecraft* (2.1.1), która składa

się z prostych obiektów (Woksel [14]) oraz gra zawiera modyfikacje dowolnego modelu terenu w jednym generatorze.

Rozdział 3

Wymagania i narzędzia

W obecnym rozdziale został stworzony opis wymagań funkcjonalnych, niefunkcjonalnych, opis narzędzi oraz przypadki użycia.

3.1 Wymagania funkcjonalne

Dla tworzonej implementacji zostały zdefiniowane wymagania funkcjonalne:

- Generowanie terenu.
- Wybieranie metody do generowania modelu terenu.
- Zapisywanie i wczytywanie wygenerowanych terenów.
- Modyfikacja parametrów algorytmu wybranego modelu.
- Możliwość przemierzania stworzonego terenu.

3.2 Wymagania niefunkcjonalne

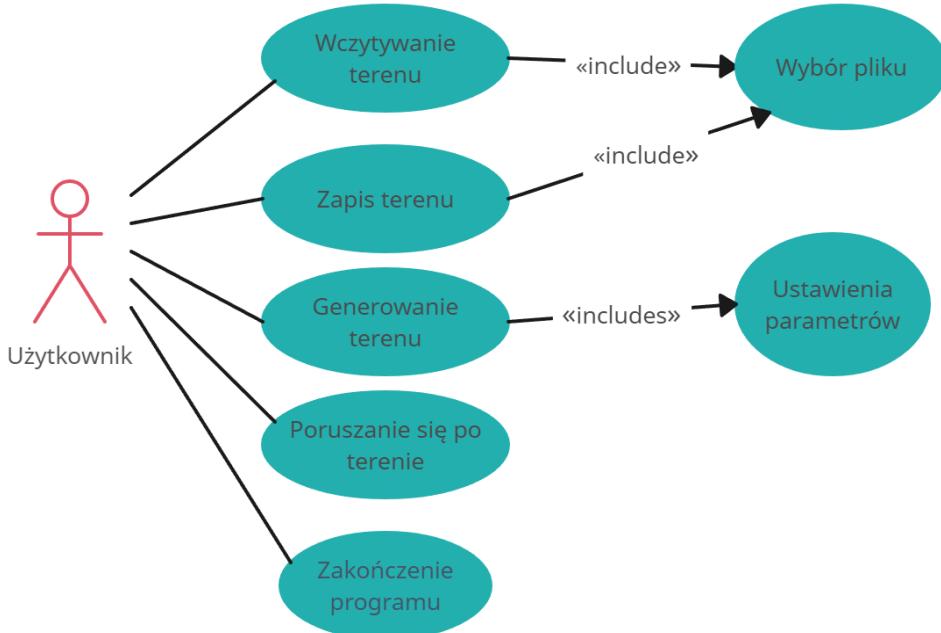
Dla tworzonej implementacji zostały zdefiniowane wymagania niefunkcjonalne:

- Program aktywnie wykorzystuje mechanizmy wielowątkowości, co umożliwia równoczesne przetwarzanie różnych aspektów generacji terenu. Dzięki temu, użytkownik doświadczy płynniejszego działania programu, co przyczyni się do bardziej satysfakcjonującego użytkowania.
- Program został zaprojektowany do współpracy z silnikiem Unity, co umożliwia skorzystanie z zalet i funkcji dostarczanych przez to środowisko.

- Program dostarcza intuicyjny interfejs umożliwiający łatwe manipulowanie zmiennymi wpływającymi na generację modelu terenu. Użytkownik może swobodnie dostosowywać parametry generacji, co umożliwia eksperymentowanie i szybkie uzyskanie pożądanych efektów

3.3 Przypadki użycia

Diagram przedstawiony na rysunku 3.1 pokazuje przypadki użycia, które mogą zostać wykonane przez użytkownika



Rysunek 3.1: Diagram przypadków użycia programu

3.4 Opis użytych narzędzi

3.4.1 Silnik Unity

Silnik Unity[17] to kompleksowe środowisko programistyczne i silnik do tworzenia różnorodnych projektów interaktywnych w 2D i 3D. Wykorzystany w projekcie została wersja silnika 2021.3.23f1. Kluczowymi elementami Unity to:

- Scena, która jest przestrzenią, na której umieszczane są obiekty, postacie, światła, kamery i inne elementy składające się na interaktywne środowisko. Scena ma strukturę drzewa, gdzie obiekty są rodzicami, a dziećmi są obiekty znajdujące się pod rodzicami. Sceny mogą reprezentować poziomy w grach, lokacje w wirtualnych środowiskach czy interaktywne interfejsy.

- **GameObject**, który jest podstawowym obiektem w Unity, reprezentujący elementy w scenie. Mogą to być postacie, obiekty, światła, kamery itp. Komponenty **GameObject** definiują zachowanie i właściwości obiektu.
- Komponenty to moduły, które dołączają się do **GameObject**, aby nadawać mu funkcjonalności. Komponenty mogą dotyczyć kolizji, fizyki, dźwięku, animacji. Również komponenty mogą być skryptami stworzonymi przez programistę. Stworzony **GameObject** ma zawsze przyłączony skrypt typu **Transform**, który pozwala na ustalenie pozycji, rotacji oraz skali obiektu.
- Język skryptowy, który pozwala na dostosowywanie i kontrolowanie interakcji w grze przez programistę. Główną klasą, która pozwala na dołączanie stworzonych skryptów do obiektów na scenie jest klasa **MonoBehaviour**, gdzie stworzone skrypty muszą dziedziczyć po niej. Podstawowymi funkcjami, które powstają podczas tworzenia skryptu jest metoda **Start** i **Update**. **Start** jest wywoływany na starcie aplikacji, gdzie **Update** jest wywoywane co klatkę działania aplikacji. Aktualny językiem skryptowym jest C# w wersji 9.0.
- **Korutyny** (ang. Coroutines) stanowią kluczowy element asynchronicznego programowania w Unity, umożliwiając bardziej efektywne zarządzanie czasem w trakcie wykonywania operacji oczekujących na zakończenie. Zastosowanie korutyn pozwala na płynniejszą obsługę zadań takich jak ladowanie zasobów, animacje czy komunikacja z siecią, nie blokując jednocześnie głównego wątku gry.
- **Prefab** to przygotowany do wielokrotnego użycia szablon obiektu lub grupy obiektów w grze. Ułatwia to wielokrotne używanie tych samych elementów w różnych miejscach.
- Materiały, które kontrolują wygląd obiektów, oraz shadery, które definiują, jak obiekty są renderowane. Unity oferuje narzędzia do tworzenia i manipulowania materiałami i shaderami.

3.4.2 Język C# 9.0

Ponieważ w pracy dyplomowej jest wykorzystywany silnik Unity, głównym językiem programistycznym projektu jest C#. W wybranej wersji Silnika Unity, język skryptowy C# jest tworzony w wersji 9.0 i jest językiem obiektowym i silnie typowanym. Kod programu jest kompilowany do Common Intermediate Language (CIL), który następnie jest wykonywany w środowisku uruchomieniowym, takim jak .NET Core. C# pozwala na tworzenie delegatów i zdarzeń, służących do zarządzania stanami oraz wydarzeniami powiązanymi w aplikacji.

3.4.3 Text Mesh Pro

TextMeshPro[16] to zaawansowany system renderowania tekstu w silniku Unity. W przeciwieństwie do standardowego komponentu **Text**, **TextMeshPro** oferuje znacznie większą kontrolę nad wyglądem i formatowaniem tekstu. Dzięki obszernej gamie funkcji, takich jak obsługa stylów, efektów czy renderowania tekstu w trójwymiarowej przestrzeni, **TextMeshPro** jest powszechnie używany do tworzenia wysokiej jakości interfejsów użytkownika, dialogów, napisów czy innych elementów tekstowych w projektach opartych na Unity.

3.4.4 Standard Unity Assets

Standard Unity Assets[18], jest to paczka assetów od Unity Technologies, która zawiera podstawowe implementacje obiektów, które można wykorzystać do testowania w własnych projektach. W pracy dyplomowej została wykorzystana część paczki odpowiadająca za poruszanie się postacią z pierwszej osoby (ang. First Person Controller), która pozwoliła na testowanie poszerzania świata w czasie działania programu.

3.4.5 IJobParrarel

IJobParallelFor[15] to interfejs w Unity Job System, który umożliwia programistom równoległe przetwarzanie pętli w sposób efektywny. Działa to poprzez wykonywanie iteracji danej pętli równolegle na wielu wątkach, co przyspiesza przetwarzanie danych w przypadku operacji niezależnych. **IJobParallelFor** jest szczególnie użyteczny, gdzie istnieje potrzeba efektywnego rozproszenia obliczeń, takich jak manipulacje nad danymi w grafice komputerowej czy fizyka gry. Dzięki temu interfejsowi, można zoptymalizować wydajność projektów, wykorzystując potencjał wielu rdzeni procesora.

3.4.6 Visual Studio 2022

Visual Studio 2022[12] to zintegrowane środowisko programistyczne (IDE), które jest powszechnie używane do tworzenia aplikacji na platformę Unity. Dzięki zaawansowanym narzędziom do debugowania, intuicyjnej obsłudze, i pełnej integracji z silnikiem Unity, Visual Studio umożliwia programistom skuteczne pisanie, debugowanie i optymalizację kodu. Dodatkowo można rozszerzać funkcjonalność IDE poprzez instalacje dodatkowych pakietów.

Rozdział 4

Specyfikacja zewnętrzna

Implementacja proceduralnej generacji została zapisana w postaci programu o rozszerzeniu .exe o nazwie *VoxelGenerator*, która pozwala przedstawić działanie stworzonej implementacji na własnym komputerze, bez wymogu instalowania silnika Unity.

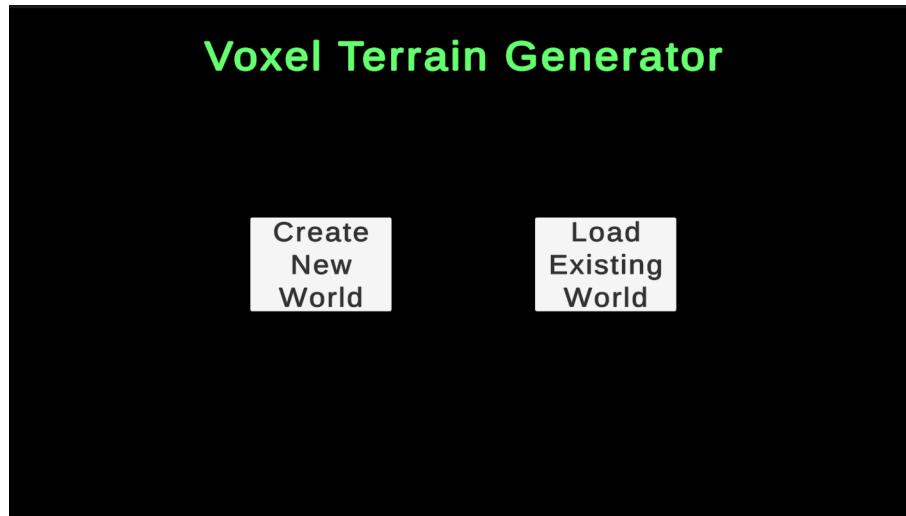
4.1 Wymagania programu VoxelGenerator

Program *VoxelGenerator* wymaga komputera z systemem Windows 10 lub wyższym. Poniżej znajdują się zalecane wymagania sprzętowe do uruchomienia programu:

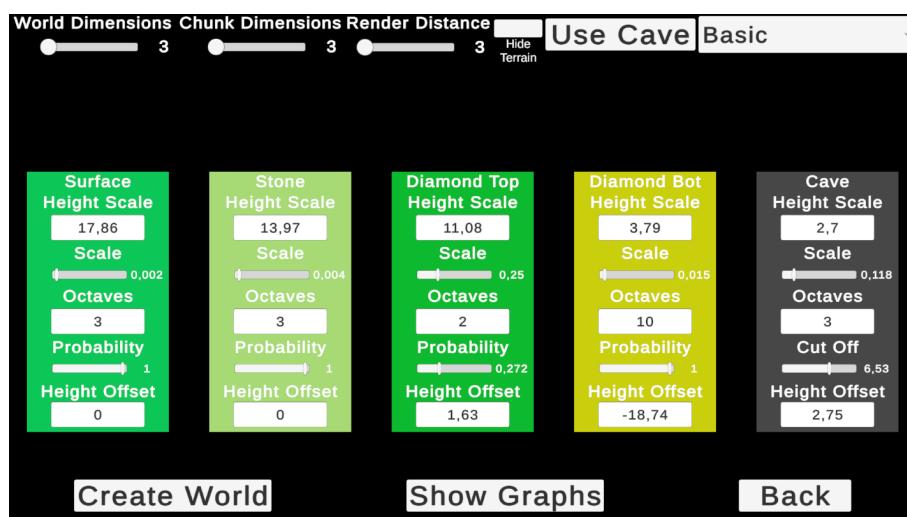
- Procesor AMD Ryzen 7 4800H lub szybszy.
- 16 GB pamięci RAM.
- 150 MB miejsca na dysku.
- NVIDIA GeForce RTX 2060 albo odpowiednik.

4.2 VoxelGenerator

Po uruchomieniu programu pojawia się menu, które pozwala wybrać stworzenie nowego terenu albo wczytanie terenu z pliku (rys. 4.1).



Rysunek 4.1: Menu główne programu *VoxelGenerator*



Rysunek 4.2: Menu tworzenia nowego terenu

4.2.1 Tworzenie nowego terenu

Po wybraniu opcji stworzenia nowego terenu pojawia się widok menu z rysunku 4.2. Całe menu można podzielić na 3 części. Pierwsza część menu znajdująca się na górze ekranu, to ustawienia ogólne, które są przeznaczone do każdego modelu terenu.

- **World Dimensions** (pl. Wymiar świata), to wartość, która ustala wielkość podstawowego terenu, który jest generowany na początku. Początkowy teren jest tworzony w postaci kwadratu (wartość X wartość). Przedział wartości jest od 3 do 30.
- **Chunk Dimensions** (pl. Wymiar fragmentu), to wartość, która pozwala ustalić wielkość pojedynczego tworzonego kawałka terenu. Przedział tej wartości jest od 3 do 10.
- **Render Distance** (pl. Zasięg Renderowania), to wartość, która określa, na jaką odległość jest renderowany stworzony teren. Przedział wartości jest od 3 do 10.
- **Hide Terrain** (pl. Chowanie Terenu), to wartość, która określa czy program powinien chować teren jeśli wyjdziemy poza wartość renderowania.
- **Use Cave** (pl. Użyj Jaskiń), to wartość, która określa czy program powinien tworzyć dziury oraz jaskinie w terenie.
- **Dropdown Model**, to menu wybierania modelu terenu do generacji. W programie znajdują się 3 zaimplementowane modele terenu:
 - **Basic** (pl. Podstawowy), który przypomina naturalny teren.
 - **Floating Islands** (pl. Latające Wyspy), który tworzy latające wyspy.
 - **Flat** (pl. Płaski), który bazowo tworzy płaski teren.

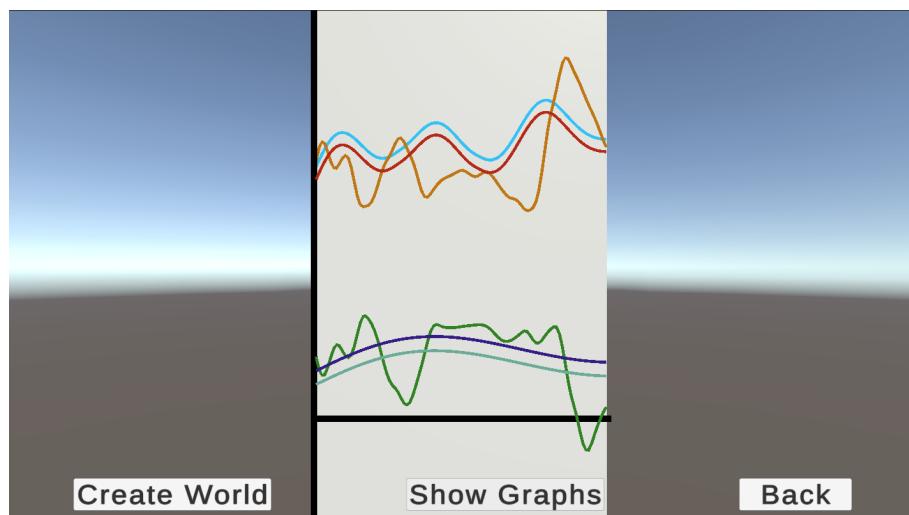
W drugiej części menu znajdującej się na środku ekranu, to ustawienia dotyczące każdej warstwy zawartej w danym modelu. Dla każdego modelu można modyfikować wartości względem własnych preferencji. Ponieważ głównym algorytmem w programie jest ułamkowy ruch Browna (2.2.2), to każda wartość w ustawieniu odpowiada wpisywanej wartości do algorytmu, gdzie:

- **Height Scale** (pl. Skala wysokości) to współczynnik skali wysokości, kontrolujący amplitudę i wpływ na wysokość generowanego terenu. Wartości bliżej zera, spowodują, że warstwa będzie bardziej płaska, a wartości wyższe od zera, spowodują bardziej zróżnicowaną wysokość warstwę.
- **Scale** (pl. Skala) odpowiada, określa rozmiar i zakres warstwy. Wartości bliżej 0 i 1 będą tworzyć bardziej płynne przejście między wartościami, kiedy wartości bliżej 0,5, spowodują że wartości warstwy będą bliżej siebie.

- **Octaves** (pl. Oktawy) to wartość, która mówi ile razy będzie nakładana ta sama warstwa algorytmu na siebie. Przykładowo, jeśli wartość Octaves wynosi 3, to oznacza że ten sam algorytm będzie nałożony 3 razy, aby uzyskać końcowy wynik.
- **Probability** (pl. Prawdopodobieństwo), to wartość, która ustala czy pojedyncze elementy tej warstwy będą się pojawiać. Jeśli wartość probability wynosi 1, to warstwa w całości się pojawi.
- **Height Offset** (pl. Przesunięcie wysokości) to przesunięcie wysokości, dodatkowy parametr pozwalający dostosować bazową wysokość generowanej warstwy.
- **Cut Off** (pl. Odcięcie), to wartość znajdująca się tylko przy modyfikowaniu wartości jaskiń (ang. Cave). Jest to prawdopodobieństwo pojawienia się pojedyńczych bloków w terenie, aby tworzyć dziury i jaskinie.

W Ostatniej części menu znajdującej się w dolnej części ekranu znajdują się 3 przyciski:

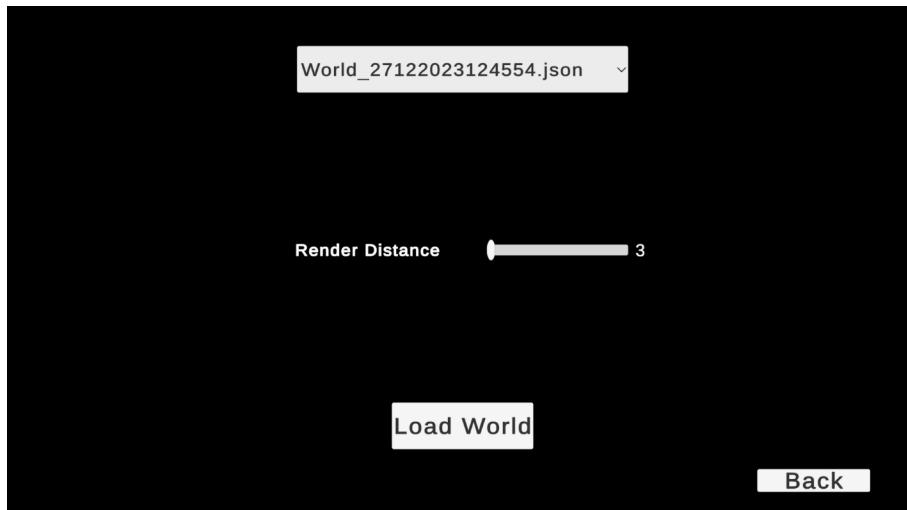
- **Create World** (pl. Stwórz świat), który zapisuje wybrane ustawienia w poprzednich dwóch częściach menu i zaczyna tworzenie terenu.
- **Show Graphs** (pl. Pokaż Wykresy), który pozwala na prostą wizualizację 2D ustawień wybranego modelu terenu. Przykład wyglądu przykładowych ustawień można zobaczyć na rysunku 4.3. Poszczególna linia odpowiada konkretnej warstwie poprzez kolor. Jest możliwość przesuwania kamery w pionowej orientacji przy pomocy strzałek albo klawiszy W i S. Dodatkowo jest możliwość zobaczenia ustawień warstwy jaskiń przy pomocy przycisku R.
- **Back** (pl. Wróć), odpowiada za powrót do poprzedniego menu.



Rysunek 4.3: Wizualizacja wyglądu warstw 2D

4.2.2 Wczytywanie terenu

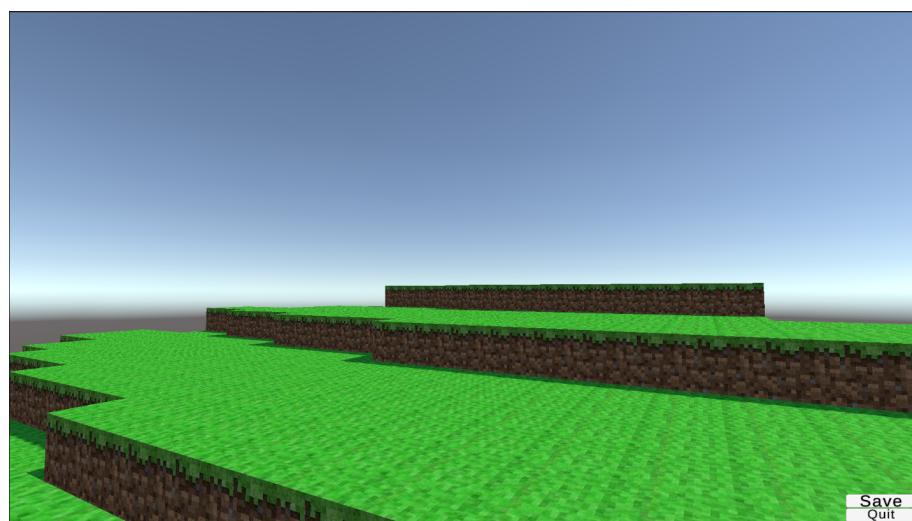
Po wybraniu opcji wczytywanie stworzonego terenu z pliku pokazuje się rozwijane menu (rys. 4.4) ze stworzonymi światami. Jeśli jednak w plikach nie znajduje się żaden stworzony świat, rozwijane menu poinformuje użytkownika o braku istniejących światów na danym urządzeniu i wyłączy opcję wczytania świata. Dodatkowo jest opcja zmiany zasięgu renderowania (ang. Render Distance), do wczytanego świata. Na dole ekranu znajduje się przycisk do wczytania wybranego świata oraz przycisk Back (pl. Wróć), aby powrócić do menu głównego.



Rysunek 4.4: Wygląd menu wczytywania nowego terenu

4.2.3 Poruszanie się po stworzonym terenie

Po wybraniu stworzenia nowego świata albo wczytaniu już stworzonego pojawia się ekran ładowania z paskiem postępu, który informuje o postępie wczytywania. Po skończeniu użytkownik posiada możliwość przejścia się po stworzonym terenie przy pomocy klawiszu WSAD, gdzie przycisk W służy do ruchu do przodu, S do ruchu do tyłu, A do ruchu w lewo i D do ruchu w prawo. Również jest opcja skakania oraz biegania pod klawiszami spacji, oraz lewego shift odpowiednio. Poruszanie się po stworzonym terenie, również powoduje tworzenie nowych kawałków terenu, czyli powiększa teren do przemieszczania się. Użytkownik również posiada dwa dostępne przyciski znajdujące się w prawym dolnym rogu ekranu (rys. 4.5). Przycisk Save (pl. Zapisz), który powoduje zapisanie świata do pliku oraz Quit (pl. Wyjdźcie), aby wyjść z programu i zakończyć jego działanie.



Rysunek 4.5: Wygląd programu po wejściu na stworzony teren

Rozdział 5

Specyfikacja wewnętrzna

W rozdziale została przedstawiona idea, logika i architektura implementacji oraz przedstawiono najważniejsze struktury, algorytmy i działanie wybranych klas i metod.

5.1 Główna idea implementacji programu

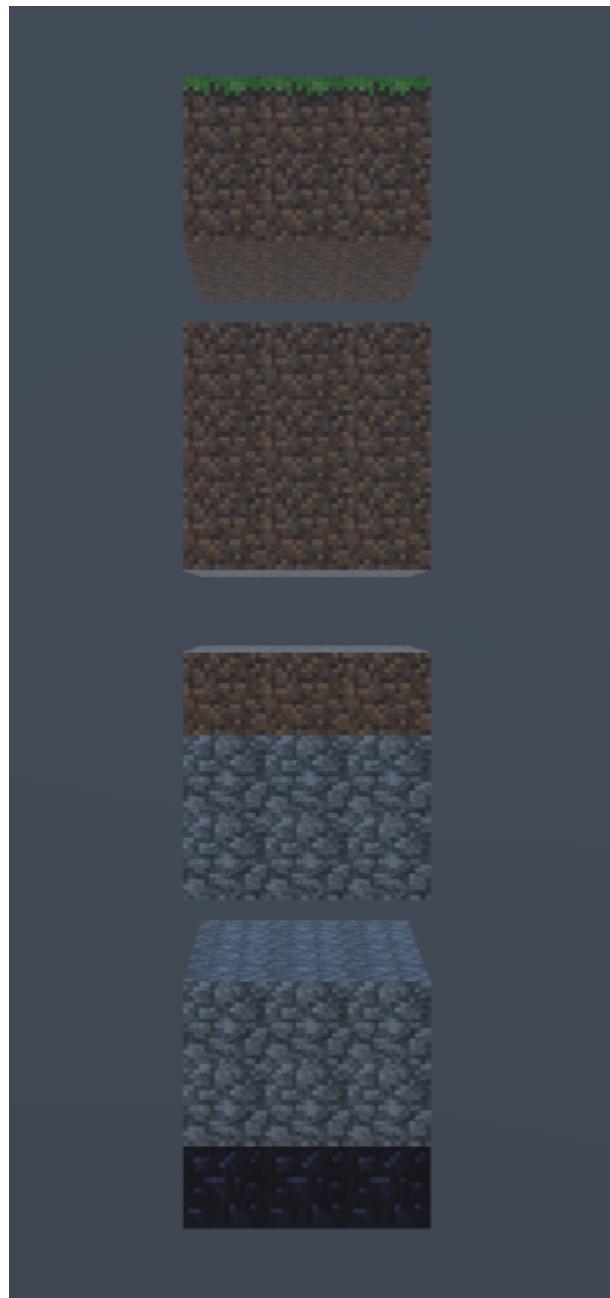
Centralną ideą implementacji jest stworzenie elastycznego generatora terenu dla każdego modelu terenu. Każdy z modeli terenu musi spełniać ustalone odgórnie zasady, aby główna klasa generatora przyjęła taki model do generacji. Główne zasady, które model musi spełniać to:

- Model musi posiadać wyszczególnione warstwy, aby była możliwość różnorodności warstw.
- Model musi posiadać własny odpowiednik podstawowego algorytmu, który wykorzystuje zmienne w generatorze terenu, dopasowaną do swoich warstw.
- Model posiada określoną liczbę warstw założoną z góry, aby odwołania w samym algorytmie, były proste.

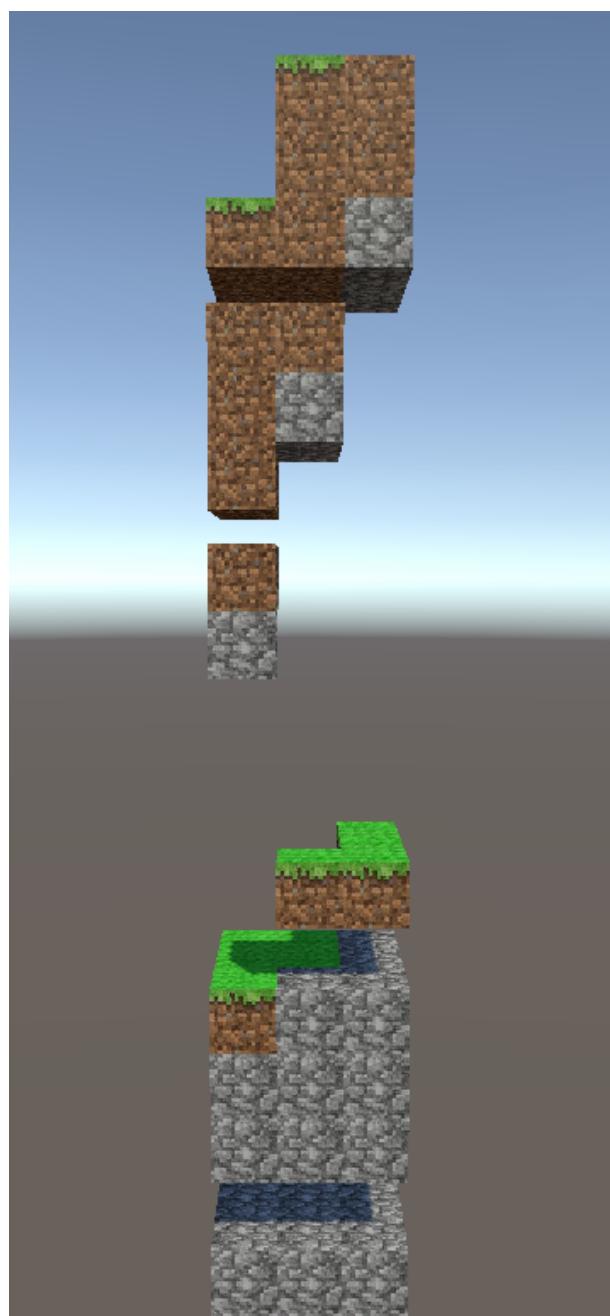
Każdy warstwa modelu posiada przypisany do niej algorytm (np. szum Perlina (2.2.1)), dzięki któremu powstaje odpowiedni kształt terenu. Dlatego każda warstwa powinna mieć publiczny dostęp do wartości algorytmu, które użytkownik może modyfikować względem własnych potrzeb. Dodatkowo dla użytkownika znajduje się opcja wizualizacji algorytmu, aby miał wyobrażenie, jak wygenerowany teren może wyglądać.

Generator terenu musi składać swój całokształt z pojedynczych części, które tworzy. W implementacji został wykorzystany na podstawie *Minecraft'a* (2.1.1) obiekt typu Chunk [3]. Chunk przechowuje informacje o tym jak wygląda, gdzie znajduje się w przestrzeni oraz czy ma być widoczny dla użytkownika. Dodatkowo, aby umożliwić dostosowanie generatora do różnych modeli terenu, teren pod danym położeniem wzduż osi X i Z (rys. 5.3) jest generowany jako tabela wielu kawałków terenu o zróżnicowanych wysokościach.

To podejście umożliwia tworzenie różnorodnych form terenu, takich jak jaskinie, dziury czy latające wyspy. Przykład kolumny kawałków terenu bez przerw w terenie jest przedstawiony na rysunku 5.1. Natomiast rysunek 5.2 ilustruje kolumnę kawałków terenu z uwzględnieniem przerw, gdzie generowane są obszary bez widocznych elementów terenu. Chunk musi składać się z pewnych elementów, aby był widoczny dla użytkownika jako element terenu. Dlatego każdy kawałek terenu składa się z pojedynczych wokseli [14], ponieważ jest prostym pojedynczym obiektem 3D łatwym do wizualizacji.



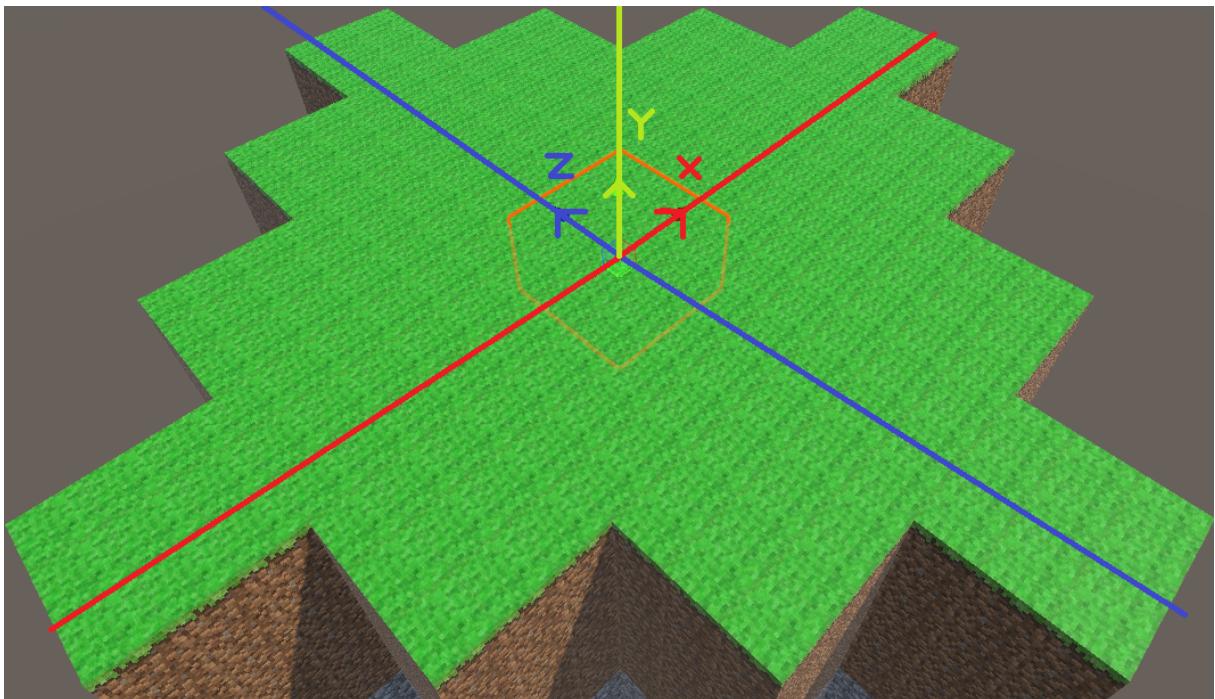
Rysunek 5.1: Wygląd podziału kolumny kawałków terenu



Rysunek 5.2: Wygląd podziału kolumny kawałków terenu z wytworzonymi przerwami

Pojedynczy woksel składa się z 6 ścian kwadratowych, które po połączeniu tworzą sześcienną kostkę. Ponieważ każdy model posiada różne warstwy, które mogą różnić się wyglądem, każda ściana będzie musiała posiadać różne tekstury odpowiadające warstwie. Przy niektórych warstwach takich jak ziemia, ściana posiada inną teksturę na górnej części ściany woksela, niż na dolnej. Dodatkowo jeśli pojedyńczy woksel ma niemożliwe do zobaczenia ściany dla użytkownika na przykład znajdującej się pod terenem, to chunk wymusi na wokselu, aby nie generował tekstury na tej ścianie.

Ważną funkcjonalnością dla programu jest możliwość zapisania oraz wczytania stworzonych terenów przez użytkownika. Każdy z terenów posiada informacje, jak wygląda, ile kawałków terenu zostało stworzonych, jaką mają swoją lokalizację, jakiego modelu terenu i algorytmu korzystają oraz ustawienia dotyczące chowania terenu. Do zapisu powyższych zmiennych potrzebna jest struktura, która zapisze wszystkie dane do pliku typu JSON, aby również była możliwość podejrzenia zapisanych danych poza programem.



Rysunek 5.3: Układ współrzędnych dla kawałków terenu

5.2 Tworzenie funkcji szumowych

Klasa **MeshUtils**, ma zaimplementowane dwie funkcje szumowe, które umożliwiają tworzenie bardziej naturalnego wyglądu.

5.2.1 Szum ułamkowego ruchu Browna

Pierwszą funkcją szumową jest funkcja **fBM**, która opiera się na koncepcji ułamkowego ruchu Browna. W skrócie, jest to proces generowania szumu, w którym kilka warstw szumu Perlin'a jest ze sobą kombinowanych. Każda warstwa (oktawa) ma różne częstotliwości i amplitudy, co prowadzi do uzyskania bardziej złożonego i naturalnego efektu.

Wykorzystana funkcja widoczna na rysunku 5.4, jako parametry przyjmuje współrzędne punktu w przestrzeni (X i Y), liczbę oktaw (**octaves**), która określa liczbę warstw szumu, współczynnik skali (**scale**), określający rozmiar i zakres generowanego szumu, współczynnik skali wysokości (**heightScale**), kontrolujący, jak bardzo wpływa na wysokość generowanego terenu oraz przesunięcie wysokości (**heightOffset**), dodatkowy parametr pozwalający dostosować bazową wysokość. Dodatkowo w funkcji znajduje się pomocnicza zmienna **frequency**, która oblicza częstotliwość kolejnych oktaw szumu Perlina. Co każdą iterację zostaje wyliczona wartość szumu Perlina, która jest dodawana do całkowitej wartości w tym punkcie, gdzie wartość częstotliwości zostaje podwojona na końcu pętli. Na końcu funkcja zwraca wartość wyliczonego punktu wraz z przesunięciem wysokości.

```

1 public static float fBM( float x, float z, int octaves, float
2   scale, float heightScale, float heightOffset )
3 {
4     float total = 0;
5     float frequency = 1;
6
7     for (int i = 0; i < octaves; i++)
8     {
9         total += Mathf.PerlinNoise(x * scale * frequency, z *
10          scale * frequency) * heightScale;
11         frequency *= 2;
12     }
13
14     return total + heightOffset;
15 }
```

Rysunek 5.4: Funkcja **fBM** w klasie **MeshUtils**

5.2.2 Szum ułamkowego ruchu Browna w 3D

Druga funkcją szumową jest funkcja **fBM3D**, która również opiera się na koncepcji ułamkowego ruchu Browna, jednak jest przystosowana do trójwymiarowego wymiaru. Funkcja **fBM3D** wykorzystuje kombinację wartości generowanych przez funkcję **fBM** na różnych płaszczyznach przestrzeni trójwymiarowej. Funkcja również przyjmuje w większości te same wartości co funkcja **fBM**. Dodatkowo przyjmuje wartość Y z układu współrzędnych. Wartości szumu są generowane na różnych płaszczyznach przestrzeni trójwymiarowej (XY, XZ, YZ, YX, ZX, ZY) dla podanych współrzędnych (X, Y, Z). Każda z tych wartości reprezentuje wpływ szumu na daną płaszczyznę. Ostateczny rezultat jest uzyskiwany poprzez uśrednienie wartości szumu generowanego na różnych płaszczyznach. Działa to jak filtr, eliminując ewentualne skrajne wartości i tworząc gładką powierzchnię terenu. Zaimplementowana funkcja jest widoczna na rysunku 5.5

```
1 public static float fBM3D(float x, float y, float z, int octaves,
                           float scale, float heightScale, float heightOffset)
2 {
3     float XZ = fBM(x, z, octaves, scale, heightScale,
                  heightOffset);
4     float XY = fBM(x, y, octaves, scale, heightScale,
                  heightOffset);
5     float YZ = fBM(y, z, octaves, scale, heightScale,
                  heightOffset);
6     float YX = fBM(y, x, octaves, scale, heightScale,
                  heightOffset);
7     float ZX = fBM(z, x, octaves, scale, heightScale,
                  heightOffset);
8     float ZY = fBM(z, y, octaves, scale, heightScale,
                  heightOffset);
9
10    return (XZ + XY + YZ + YX + ZX + ZY) / 6.0f;
11 }
```

Rysunek 5.5: Funkcja **fBM3D** w klasie **MeshUtils**

5.3 Skład modelu terenu

Główną klasą modelu terenu jest **WorldVisualization**, która zawiera wszystkie zmienne i elementy potrzebne do wygenerowania terenu. Klasa na początku programu przypisuje sobie listę warstw, które znajdują się na pod nią, jak można zobaczyć na rysunku 5.6 oraz przypisuje sobie klasę pochodną od klasy **CalculateBlockTypesJobs**, aby przy generacji terenu odwołał się do poprawnej części funkcji, gdzie poprawnie wykorzysta zapisane warstwy.



Rysunek 5.6: Hierarchia obiektów dla modelu bazowego terenu

5.3.1 Modyfikowanie pojedynczej warstwy

Klasa odpowiadająca za parametry warstwy, **PerlinGrapher**, posiada parametry potrzebne do wykorzystania funkcji **fBM** znajdującej się w klasie **MeshUtils**. Parametry znajdujące się w klasie to:

- **heightScale**, to współczynnik skali wysokości, kontrolujący amplitudę i wpływ na wysokość generowanej warstwy. Wartość ta mnożona jest przez każdą wartość szumu Perlin'a, regulując, jak bardzo wpływa na ostateczną wysokość warstwy.
- **scale**, określa rozmiar i zakres generowanego szumu Perlin'a.
- **octaves** to liczba oktaw, określająca, iloma warstwami szumu Perlin'a zostanie nałożonych na siebie, wpływając na poziom detali w generowanym szumie.
- **probability**, to wartość, która ustala czy pojedyncze elementy tej warstwy będą się pojawiać.
- **heightOffset** to przesunięcie wysokości, dodatkowy parametr pozwalający dostosować bazową wysokość generowanej warstwy. Dodaje stałą wartość do całej warstwy, co umożliwia dostosowanie ogólnej wysokości warstwy.

Klasa jest połączona jako komponent do obiektu warstwy, co można zobaczyć na rysunku 5.8. Dodatkowo obiekt posiada możliwość wizualizacji 2D przy pomocy komponentu **LineRenderer**, który posiada tablice punktów, które możemy modyfikować. Klasa posiada funkcje **Graph**, która aktualizuje wygląd wykresu **LineRenderer**, aby odpowiadał zmodyfikowanym wartością. Przy każdej modyfikacji wartości w edytorze Unity, klasa posiada funkcję **OnValidate**, która pozwala na automatyczne zaktualizowanie wartości oraz komponentu **LineRenderer**. Klasa na początku programu tworzy losowy kolor materiału, który przypisuje do **LineRenderer** oraz do tła ustawienia warstwy w menu tworzenia nowego terenu.

5.3.2 Modyfikowanie warstwy jaskiń

Klasa **PerlinGrapher3D** została zaprojektowana do tworzenia jaskiń oraz dziur w trójwymiarowej przestrzeni. Posiada ona parametry analogiczne do klasy **PerlinGrapher**.

pher, z niewielką różnicą. Wartość **probability** z klasy **PerlinGrapher** została zamieniona na zmienną **drawCutOff**. Wartość **drawCutOff** została dostosowana do obszaru trójwymiarowego. Proces wizualizacji opiera się na tworzeniu prostych kostek jako podstawowych obiektów Unity. Następnie kostki są aktualizowane zgodnie z funkcją **fBM3D**, a te, których wartość przekracza **drawCutOff**, są usuwane, co umożliwia generowanie jaskiń oraz dziur w terenie.

5.3.3 Zapisywanie ustawień z warstw

Przed rozpoczęciem generacji, klasa **WorldVisualization** wywołuje funkcję **CreateSettings**, która tworzy listę struktur **perlinSettings**, która przenosi wszystkie parametry z klasy **PerlinGrapher** do wyżej wymienionej struktury. Dzięki temu rozwiązaniu możemy przy tworzeniu terenu wykorzystywać strukturę, która korzysta z interfejsu **IJobParallelFor**, aby zrównoleglić działanie programu.

5.4 Generator terenu

Główna klasa odpowiedzialna za proceduralne generowanie terenu jest klasa **WorldCreator**. Posiada parametry za wielkość świata, wielkość pojedynczych kawałków terenu, wybrany model terenu oraz zapisane pozycje obiektów klasy **ChunkBlock**. Dzięki tej klasie jest możliwość tworzenia podstawowego terenu, poszerzania terenu proceduralnie oraz chowanie kawałków terenu jeśli znajdą się poza zasięgiem renderowania.

5.4.1 Przygotowanie generacji

Funkcja **StartWorld** rozpoczyna przygotowanie do generowania nowego terenu. Funkcja przyjmuje parametry:

- **WorldVisualization**, który posiada wybrany model terenu do generacji i zostaje przypisana do zmiennej **worldVisualization**. Po przypisaniu zmiennej następnym krokiem jest wywołanie metody **CreateSettings**, który przygotowuje listę **PerlinSettings** z wszystkich warstw, które posiada wybrany model.
- **Vector3Int**, który:
 - Pod wartością X przechowuje wielkość podstawowego terenu (wartości X i Z), która zapisuje do zmiennej **Vector3Int worldDimensions**.
 - Pod wartością Y posiada wymiary pojedynczego obiektu klasy **ChunkBlock** (wartości X, Y i Z), które zapisuje do zmiennej **Vector3Int chunkDimensions**, która posiada .

- Pod wartością **Z** znajduje się wartość zasięgu renderowania, którą zapisuje do zmiennej **drawRender**.
- **useCavesChoose**, który określa czy przy generowaniu ma używać warstwy jaskiń.
- **isHideTerrain**, który określa czy generator ma chować stworzony teren znajdujący się poza zasięgiem renderowania.

Po Przypisaniu wszystkich wartości, generator przechodzi do metody **StartBuilding**, która przy podaniu wartości false rozpoczyna korutynę **BuildWorld**

5.4.2 Rozpoczęcie generacji

Po uruchomieniu korutyny **BuildWorld**, proces przekształcania terenu rozpoczyna iterację po wartościach **X** i **Z** zmiennej **worldDimensions**. Celem jest stworzenie kolumny obiektów klasy **ChunkBlock** o wysokości zdefiniowanej przez wektor **worldDimensions** (wartość **Y**). Innymi słowy, jeśli wartość **Y** zmiennej **worldDimensions** wynosi na przykład 3, generator terenu utworzy trzy obiekty klasy **ChunkBlock** ustawione na kolejnych poziomach wysokości. Przykład takiego terenu jest zobrazowany na rysunku 5.7, gdzie każdy kolor reprezentuje osobną kolumnę obiektów.

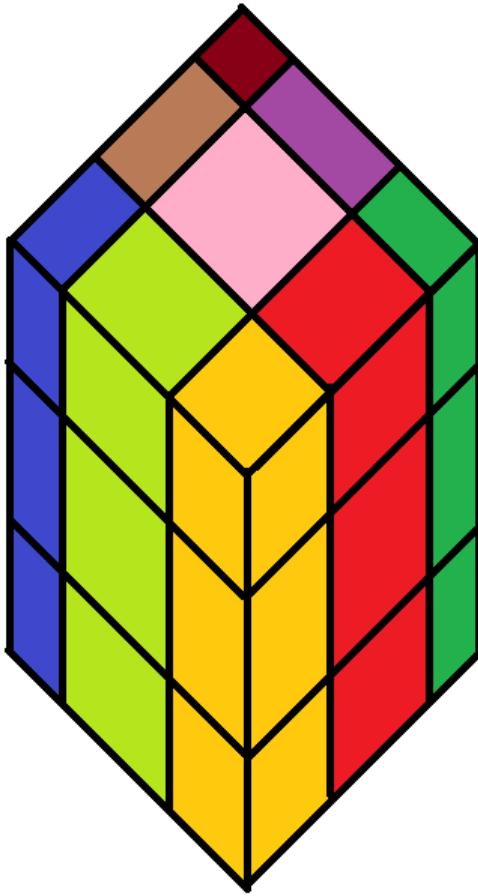
Iteracje po wartości **Y worldDimensions** wykonuje korutyna **BuildChunkColumn**, która przyjmuje wartości **X** i **Z**, określające położenie kolumny w 2D. Na początku tworzonej jest zmienna **position** typu **Vector3Int**, aby zapisać na jakiej pozycji w generowaniu pracuje generator. Każda z wartości w **BuildChunkColumn** z wektora **worldDimensions** jest przed operacjami tworzenia dodatkowo pomnożona przez odpowiedniki wartości wektora **chunkDimensions**, aby pozycje były prawidłowo dopasowane w stworzonym terenie. Po stworzeniu zmiennej **position**, sprawdzamy, czy taka pozycja istnieje w tablicy pozycji generatora. Tablica pozycji (w klasie **WorldCreator** zmienna o nazwie **chunkChecker**) służy nam do sprawdzania, czy generator już stworzył obiekt klasy **ChunkBlock** na tej pozycji, aby nie tworzyć ponownie istniejących już kawałków terenu. Dlatego w generatorze jest wykorzystywany typ tablicy **HashSet**, który przechowuje unikalne wartości. Jeśli jednak takiej pozycji nie posiadamy, zaczynamy procedurę generowania nowego obiektu klasy **ChunkBlock**. Dokładne zachowanie tworzenia obiektu klasy **ChunkBlock** znajduje się w rozdziale 5.4.4.

Po stworzeniu obiektu klasy **ChunkBlock**, generator zapisuje do tablicy pozycji położenie nowego kawałku terenu oraz zapisuje do słownika kawałków terenu (która w klasie **WorldCreator** jest to zmienna **chunks**) stworzony obiekt klasy **ChunkBlock** oraz jego pozycję. Słownik obiektów klasy **ChunkBlock** (ang. Chunk Dictionary) pozwala nam przypisać do pozycji stworzony obiekt klasy **ChunkBlock**, aby odwoływanie się do obiektów klasy **ChunkBlock** na konkretnych pozycjach było łatwiejsze, na przykład przy chowaniu obiektów klasy **ChunkBlock** znajdujących się poza zasięgiem renderowania.

Następną częścią po sprawdzeniu czy obiekt klasy **ChunkBlock** istnieje na takiej pozycji, jest włącznie **GameObject** oraz **meshRenderer**, aby istniejący **ChunkBlock** pojawił się na scenie i był widoczny dla użytkownika. Po skończeniu tworzenia kolumny obiektów klasy **ChunkBlock** na pozycji X i Z w terenie ta pozycja zostaje zapisna do tablicy (która w klasie **WorldCreator** jest typem **HashSet** o nazwie **chunkColumn**). Dzięki takiemu rozwiązaniu możemy bezpośrednio odwołać się do pozycji kolumny, w której znajdują się wszystkie stworzone chunki.

Po skończeniu tworzenia podstawowej części terenu generator przygotowuje dla użytkownika postać, aby pojawiła się na środku stworzonego terenu oraz przygotowuje generator do aktualizowania terenu, podczas działania aplikacji. Wtedy zaczyna wywoływanie korutyn:

- **BuildCoordinator**, która sprawdza, czy istnieją zadania budowy w kolejce (ang. Queue). Jeśli takie zadania znajdują się w kolejce, rozpoczyna wykonywanie ich w kolejności ich przyjścia.
- **UpdateWorld**, która aktualizuje teren i generuje nowe elementy terenu.
- **BuildExtraWorld**, która na stworzonych krańcach stworzonego terenu, generuje kolejne elementy terenu, gdy użytkownik może już poruszać się po podstawowym terenie.



Rysunek 5.7: Wyznaczenie budowy kolumny obiektów klasy **ChunkBlock**

5.4.3 Proceduralne poszerzanie generowanego terenu

Po rozpoczęciu korutyny **UpdateWorld**, generator pozostaje w tej pętli aż do zakończenia programu. W trakcie jej działania sprawdza, czy odległość między ostatnią znaną pozycją budowy a aktualną pozycją gracza przekroczyła ustaloną wartość. Jeśli tak, oznacza to, że gracz przesunął się na tyle daleko od ostatnio zbudowanej części świata, co wymaga aktualizacji. W przypadku spełnienia tego warunku, funkcja aktualizuje pozycję ostatnio zbudowanej części na pozycję gracza, a następnie oblicza nowe współrzędne startowe dla procesu budowy. Następnie dodaje zadanie budowy (w klasie **WorldCreator** jest to korutyna **BuildRecursiveWorld**) do kolejki budowy, aby zainicjować proces aktualizacji świata na nowym obszarze. Jeśli opcja **hideTerrain** jest aktywowana, dodatkowo do kolejki dodawane jest zadanie (korutyna **HideColumns**), które ukrywa kolumny terenu na obszarze poza zasięgiem renderowania. Cała korutyna jest zatrzymywana na krótki okres (na przykład 0.1 sekundy), co pozwala na regularną aktualizację terenu, minimalizując jednocześnie obciążenie procesora.

Korutyna, która dobudowuje teren (**BuildRecursiveWorld**) jest odpowiedzialna za rekurencyjną konstrukcję kolumn terenu, przyjmując współrzędne X i Y oraz promień za-

sięgu renderowania. Na początku dekrementuje wartość promienia zasięgu renderowania o 1 i sprawdza, czy jest ona mniejsza lub równa zeru. Jeśli tak, funkcja jest przerywana, co kończy proces rekurencji na danym poziomie. Następnie, dla każdego z czterech kierunków (góra, dół, prawo, lewo), funkcja inicjuje budowę kolumny terenu (korutyna **BuildChunkColumn**) i dodaje zadanie rekurencyjne dla nowych współrzędnych do kolejki budowy. To sprawia, że proces budowy kolumny terenu będzie kontynuowany we wszystkich kierunkach. Każda sekwencja budowy kolumny i dodawania zadań rekurencyjnych jest przerwana na jedną klatkę, co pozwala na płynność procesu budowy i minimalizuje obciążenie procesora.

5.4.4 Tworzenie obiektu klasy ChunkBlock

Klasa odpowiedzialna za pojedynczy kawałek terenu (w implementacji jako klasa **ChunkBlock**) rozpoczyna swoje działanie od funkcji tworzenia obiektu (**CreateChunk**), która przyjmuje parametry dotyczące wymiarów kawałka terenu, jego położenia w terenie oraz informację, czy ma zostać przebudowany pod względem struktury rodzajów bloków. Na początku wszystkie zmienne są przypisywane do obiektu klasy, a także dodawane są komponenty **MeshFilter** oraz **MeshRenderer**, umożliwiające późniejsze modyfikacje wyglądu. Jeśli określone jest, że obiekt ma być przebudowany, uruchamiana jest funkcja odpowiedzialna za zbudowanie kawałka terenu z bloków (**BuildChunk**).

Proces budowania kawałka terenu rozpoczyna się od obliczenia łącznej liczby bloków terenu (**blockCount**) na podstawie szerokości, głębokości i wysokości kawałka terenu. Następnie przygotowywana jest tablica przechowująca typy bloków, aby funkcja tworząca kawałek terenu miała informacje o rodzajach bloków. Dodatkowo przypisywany jest model terenu, który informuje funkcję, jakie parametry oraz funkcje zastosować podczas generacji bloków z wybranego modelu terenu. Kolejnym krokiem jest uruchomienie wielowątkowego procesu generacji poszczególnych bloków w oparciu o parametry wybranego modelu terenu, a szczegóły tego procesu są dokładnie opisane w rozdziale 5.4.6. Po zakończeniu przeliczeń i wypełnieniu tablicy typów bloków, funkcja kończy swoje działanie i powraca do funkcji tworzenia kawałka terenu (**CreateChunk**).

W ramach procesu tworzenia obiektu klasy **ChunkBlock**, zagnieżdżone pętle iterują przez wszystkie bloki terenu w trzech wymiarach. Dla każdego pojedyńczego elementu kawałka terenu tworzony jest obiekt klasy **Block**, reprezentujący pojedynczy blok terenu, który jest następnie dodawany do tablicy bloków. W przypadku, gdy dany blok posiada siatkę (ang. mesh), informacje dotyczące tej siatki są zbierane do listy, a także zapisywane są pozycje początkowe w tablicach wierzchołków (ang. Vertex) i trójkątów. Te dane są przekazywane do struktury, która będzie odpowiedzialna za równoległe przetwarzanie danych siatek bloków (**ProcessMeshDataJob** w implementacji). Na podstawie zebranych danych jest tworzony nowy obiekt siatki, a następnie konfigurowane są jego parametry, ta-

kie jak wierzchołki, trójkąty oraz inne atrybuty. Ten proces ma na celu utworzenie jednej wspólnej siatki dla całego kawałka terenu. Następnie nowo utworzona siatka zostaje przypisana do komponentu **MeshFilter**, który znajduje się w obiekcie klasy **ChunkBlock**. To umożliwia aktualizację wyglądu kawałka terenu na podstawie zebranych danych. Na końcu dodawany jest komponent **MeshCollider**, aby umożliwić detekcję kolizji z użytkownikiem.

5.4.5 Tworzenie obiektu klasy Block

Klasa **Block** odpowiada za stworzenie pojedynczego sześcianu, który znajduje się w kawałku terenu. Wywołanie konstruktora klasy **Block** inicjalizuje parametry klasy, takie jak lokalna pozycja w obiekcie klasy **ChunkBlock**, rodzaj bloku oraz referencja do rodzica (**parentChunk**). Następnie sprawdzane są otaczające obiekty klasy **Block** wokół bieżącego obiektu, aby zdecydować, które ściany bloku będą widoczne. W zależności od tego generowane są odpowiednie elementy graficzne, reprezentujące poszczególne ściany sześcianu (**Quad**). W przypadku braku sąsiadującego bloku, odpowiednia ściana jest tworzona przy użyciu klasy **Quad** i odpowiednich tekstur zdefiniowanych w klasie **MeshUtils**. Jeśli blok posiada przynajmniej jedną widoczną ścianę, to tworzone są oddzielne siatki dla każdej z tych ścian, a następnie łączone w jedną siatkę, która jest optymalizowana i nadawana odpowiedniej nazwie. Powyższe operacje są przeprowadzane tylko dla bloków, które nie są typu powietrza (ang. air), co pozwala na pominięcie pustych przestrzeni, zwiększać wydajność generacji terenu.

5.4.6 Przypisanie rodzaju bloku

Struktura **CalculateBlockTypes** jest odpowiedzialna za przypisanie rodzaju bloku w zależności od wybranej funkcji generującej model terenu. W trakcie wykonywania zadania równoległego (**IJobParallelFor**), dla każdego bloku w obszarze obiektu klasy **ChunkBlock**, wybierana jest odpowiednia funkcja generująca na podstawie wartości zmiennej przypisanej w pochodnej klasie **CalculateBlockTypesJob (function)**. W stworzonej implementacji zostały przedstawione 3 różne modele terenu, które mają swoją własną funkcję w strukturze, ponieważ interfejs **IJobParallelFor** nie pozwala na zmienne, które są klasami i wywoływanie funkcji. Przykładami modelu terenu w implementacji są:

- **Base**, który przypomina teren zbliżony do terenu ziemi. Powierzchnia jest zielenią, pod którą znajduje się ziemia. Następnie jest warstwa kamienia, w której może znajdować się rudy, takie jak diamenty. Dodatkowo mogą pojawiać się jaskinie, które powodują przerwy i dziury w terenie.
- **Floating Islands**, gdzie teren składa się z dwóch poziomów latających wysp. Każdy

z poziomów składa się zieleni, ziemii oraz kamienia. Ustawienie jaskiń jest wykorzystywane do tworzenia przerw w terenie oraz dziur w poziomie kamienia.

- **Flat**, który tworzy płaski teren z warstwą zieleni, ziemii i kamienia. Również występuje opcja tworzenia jaskiń w tym modelu świata.

5.5 Zachowanie Stanu Terenu

5.5.1 Zapisywanie do pliku

Procedura zapisywania terenu do pliku typu JSON jest realizowana poprzez publiczną funkcję **SaveWorld** w klasie generatora terenu (**WorldCreator**). Funkcja ta korzysta z metody zapisu terenu w osobnej klasie zarządzającej tworzeniem i zapisem plików, o nazwie (w implementacji) **WorldSaver**. W tej klasie znajduje się deklaracja struktury (w implementacji nazwanej **WorldData**), która zawiera wszystkie niezbędne wartości z generatora i terenu, umożliwiające późniejsze odtworzenie terenu. Funkcja zapisu (metoda **Save** w klasie **WorldSaver**), jako parametr przyjmuje generator (obiekt klasy **WorldCreator**). Na początku tworzona jest nazwa pliku, która będzie zapisana w folderze aplikacji. Nazwa pliku składa się z nazwy World oraz ciągu liczb, które określają datę stworzenia zapisu. Następnie tworzony jest nowy obiekt struktury **WorldData**, do którego przekazywane są wszystkie parametry z generatora. Znajdują się wśród nich takie zmienne jak wielkość podstawowego świata, ustawienia wybranego modelu terenu, rozmiar pojedynczego obiektu klasy **ChunkBlock**, wszystkie stworzone kawałki terenu wraz z ich pozycjami i przypisaniem do kolumny. Po zakończeniu wypełniania struktury, jest ona zapisywana do formatu JSON jako ciąg znaków, który następnie jest zapisywany do pliku. Aby uniknąć konfliktów nazw w plikach, każda nazwa zapisanego świata jest unikalna.

5.5.2 Wczytywanie terenu z pliku

Funkcja, która pozwala na wczytywanie terenu z pliku (w implementacji, funkcja **Load** w klasie **WorldSaver**), przyjmuje jako parametr nazwę pliku. Jeśli plik nie istnieje, to funkcja zwraca wartość **null**. Jeśli plik istnieje, funkcja odczytuje wszystko co zostało zapisane w pliku JSON i zapisuje do tymczasowej zmiennej typu **string**. Następnie zostaje stworzona deserializacja z zmiennej tymczasowej do struktury przechowującej wartości dla generatora, która funkcja zwraca. W klasie generatora (**WorldCreator**), wszystkie zmienne z struktury, zostają przypisane i generator rozpoczyna odtwarzanie terenu z pliku. Procedura wczytywania terenu z pliku (w implementacji metoda **Load** w klasie **WorldSaver**) przyjmuje jako parametr nazwę pliku. Jeśli plik nie istnieje, funkcja zwraca wartość **null**. W przypadku istnienia pliku, funkcja odczytuje jego zawartość w formie JSON i zapisuje do tymczasowej zmiennej typu **string**. Następnie dokonuje deserializacji z tej

zmiennej do struktury przechowującej wartości dla generatora (**WorldSaver**). Ostatecznie funkcja zwraca tę strukturę. W klasie generatora (**WorldCreator**), wszystkie zmienne z deserializowanej struktury są przypisywane, aby generator rozpoczął odtwarzanie terenu z pliku.

5.6 Opis ważniejszych klas

5.6.1 WorldCreator

Jest to główna klasa generatora terenu. Zarządza tworzeniem nowych kawałków terenu, proceduralnym generowaniem terenu oraz chowaniem elementów terenu poza zasięgiem renderowania.

Pola Klasy

- **worldDimensions**: wymiary całego świata gry.
- **extraWorldDimensions**: dodatkowe wymiary dla obszaru generowanego poza głównym światem.
- **chunkDimensions**: wymiary pojedynczego kawałka terenu (bloku terenu).
- **chunkPrefab**: prefabrykat obiektu klasy **ChunkBlock**, który zostanie sklonowany podczas generacji terenu.
- **fpc, mCamera**: teferencje do obiektów kamery i użytkownika (FPC).
- **drawRadius**: promień widoczności generowanych obiektów klasy **ChunkBlock** wokół gracza.
- **worldVisualization**: instancja klasy **WorldVisualization** odpowiadającej za wizualizację generowanego świata.
- **chunkChecker**: zbiór pozycji obiektów klasy **ChunkBlock**, które zostały już zainicjowane.
- **chunkColumns**: zbiór kolumn obiektów klasy **ChunkBlock**, które zostały już zainicjowane.
- **chunks**: słownik przechowujący instancje obiektów klasy **ChunkBlock** na podstawie ich pozycji.
- **lastBuildPosition**: ostatnia pozycja, w której zainicjowano proces budowy obiektów klasy **ChunkBlock**.

- **load**: flaga określająca, czy generować świat od nowa czy wczytać go z pliku.
- **useCaves**: flaga określająca, czy używać generacji jaskiń (ustawiona przez użytkownika).
- **hideTerrain**: flaga określająca, czy ukrywać teren (ustawiona przez użytkownika).
- **buildQueue**: kolejka zadań do zbudowania, zarządzana w pętli koordynującej budowę.

Funkcje Klasy

- **Awake**: zapewnia, że istnieje tylko jedna instancja klasy **WorldCreator**.
- **StartWorld**: inicjuje generację świata na podstawie parametrów przekazanych przez użytkownika. Ustawia parametry generacji, takie jak styl wizualizacji, wymiary świata, itp. Przygotowuje się do generacji, zmieniając interfejs użytkownika na stan ładowania.
- **StartBuilding**: rozpoczyna proces budowy świata na podstawie parametrów z metody **StartWorld**. Określa, czy budować świat od nowa czy wczytać go z pliku.
- **BuildWorld**: iteracyjnie buduje kolejne chunki w obrębie określonego obszaru. Włącza gracza po zakończeniu generacji terenu.
- **BuildChunkColumn**: generuje pojedynczą kolumnę obiektów klasy **ChunkBlock** wzdłuż osi X i Z. Tworzy instancję **ChunkBlock** i dodaje go do kolekcji obiektów klasy **ChunkBlock**.
- **HideChunkColumn**: ukrywa kolumnę obiektów klasy **ChunkBlock**, ustawiając ich flagę widoczności na false.
- **HideColumns**: ukrywa kolumny obiektów klasy **ChunkBlock** spoza zadanego obszaru widoczności wokół gracza.
- **BuildExtraWorld**: generuje dodatkowy obszar świata poza głównym światem.
- **LoadWorldFromFile**: wczytuje świat z pliku i przywraca jego stan na podstawie danych z pliku.
- **RedrawChunk**: usuwa istniejące komponenty **ChunkBlock** i ponownie je generuje.
- **BuildRecursiveWorld**: rekurencyjnie buduje obszar wokół zadanego punktu.
- **UpdateWorld**: sprawdza, czy gracz przemieścił się wystarczająco daleko, aby rozpocząć proces ponownej generacji.

5.6.2 ChunkBlock

Klasa **ChunkBlock** jest odpowiedzialna za reprezentację i generację pojedynczego kawałka terenu.

Pola klasy

- **atlas**: materiał używany do renderowania **ChunkBlock**.
- **width, height, depth**: wymiary **ChunkBlock**.
- **blocks**: tablica przechowująca informacje o blokach wewnętrz **ChunkBlock**.
- **cData**: tablica przechowująca typy bloków dla generacji meshu.
- **meshRenderer**: komponent renderujący siatkę **ChunkBlock**.
- **location**: pozycja **ChunkBlock** w świecie gry.
- **calculateBlockTypes**: wybrany model terenu,
- **calculateBlockTypesJobs**: struktura zawierająca funkcje generujące różne rodzaje terenu,
- **handle**: zarządza działaniem wielowątkowej funkcji.

Funkcje

- **BuildChunk**: generuje chunk, inicjując i obliczając typy bloków na podstawie wybranego modelu terenu.
- **CreateChunk**: tworzy nowy chunk na podstawie podanych wymiarów i pozycji. Opcjonalnie odbudowuje strukturę bloków w obiekcie klasy **ChunkBlock**. Następnie tworzy mesh na podstawie informacji o blokach i dodaje skonfigurowany do komponentu **MeshFilter**. Dodaje **MeshCollider** dla detekcji kolizji.

5.6.3 MeshUtils

MeshUtils to statyczna klasa narzędziowa, wykorzystywana w projekcie do efektywnej manipulacji siatkami 3D. Zawiera definicje bloków, tekstury, oraz funkcje do generowania terenu przy użyciu funkcji szumu Perlina.

Pola Klasy

- **BlockType:** enumeracja reprezentująca różne typy bloków używane w generacji terenu.
- **BlockSide:** enumeracja reprezentująca różne strony (ściany) bloku.
- **blockUVs:** dwuwymiarowa tablica przechowująca współrzędne UV dla różnych typów bloków i ich ścian.

Funkcje Klasy

- **fBM:** funkcja generująca wartość funkcji szumu Perlina 2D dla określonej pozycji (x, z). Składa się z określonej liczby oktaw, kontrolujących częstotliwość i amplitudę szumu.
- **fBM3D:** funkcja generująca wartość funkcji szumu Perlina 3D dla określonej pozycji (x, y, z). Wykorzystuje trzy niezależne generatory 2D dla różnych kombinacji współrzędnych.
- **MergeMeshes:** łączy wiele siatek (meshy) w jedną. Wartości punktów i trójkątów są przechowywane w słowniku i zbiorze, aby uniknąć duplikatów i optymalizować połączenie.
- **ExtractArrays:** ekstrahuje tablice punktów (vertices), normalnych i współrzędnych UV z danego słownika i przypisuje je do podanej siatki (mesh).

5.6.4 Quad

Quad to klasa reprezentująca siatkę kwadratu w trójwymiarowej przestrzeni, wykorzystywana do generowania geometrii siatki dla bloków w grze.

Elementy klasy

- **mesh:** obiekt reprezentujący geometrię siatki.
- **Quad:** konstruktor klasy, tworzący kwadrat (siatkę) w trójwymiarowej przestrzeni na podstawie podanych parametrów.

5.6.5 Block

Klasa **Block** w Unity reprezentuje blok w trójwymiarowym świecie gry, przechowując geometrię bloku w postaci siatki kwadratów (**Quad**), a także informacje o przynależności do konkretnego **ChunkBlock** oraz metody sprawdzającej obecność stałych sąsiadów w

przestrzeni obiektów klasy **ChunkBlock**. Konstruktor klasy tworzy geometrię bloku w zależności od typu bloku i otoczenia, co umożliwia dynamiczną generację terenu.

Klasa Block

- **mesh**: obiekt klasy **Mesh** przechowujący geometrię bloku.
- **parentChunk**: Obiekt klasy **ChunkBlock**, reprezentujący blok w przestrzeni obiektów klasy **ChunkBlock**.
- **Block**: konstruktor klasy **Block**, inicjalizujący blok na podstawie przesunięcia, typu bloku oraz przynależności do danego **ChunkBlock**.
- **HasSolidNeighbour**: metoda sprawdzająca, czy sąsiedni blok w podanych wspólnych rzędnych jest stały (nie jest typu AIR lub WATER).

5.6.6 WorldSaver

Klasa **WorldSaver** w implementacji pełni rolę odpowiedzialną za zapisywanie i wczytywanie terenu.

Struktura WorldData

Struktura **WorldData** reprezentuje kluczowe informacje o stanie terenu, zawierając m.in. pozycje obiektów klasy **ChunkBlock**, dane bloków, pozycję gracza (First Person Controller), wymiary świata, ustawienia dla ustawień warstw oraz modele terenu do obliczenia typów bloków.

- **chunkCheckerValues**: tablica przechowująca wartości pozycji obiektów klasy **ChunkBlock** jako trójkę liczb całkowitych (x, y, z).
- **chunkColumnsValues**: tablica przechowująca wartości pozycji kolumn obiektów klasy **ChunkBlock** jako parę liczb całkowitych (x, z).
- **chunkData**: tablica przechowująca dane bloków dla wszystkich obiektów klasy **ChunkBlock** w świecie.
- **chunkVisibility**: tablica przechowująca informacje o widoczności obiektów klasy **ChunkBlock**.
- **fpcX, fpcY, fpcZ**: pozycja gracza (First Person Controller) w trzech wymiarach.
- **worldDimensions**: tablica przechowująca wymiary świata w trzech wymiarach.
- **chunkDimensions**: tablica przechowująca wymiary pojedynczego chunka w trzech wymiarach.

- **perlinSettings**: tablica ustawień dla funkcji szumu Perlina używanych do generacji terenu.
- **calculateBlockTypes**: model terenu do obliczenia typów bloków.
- **hideTerrain**: flaga określająca, czy teren jest ukryty.
- **useCave**: flaga określająca, czy używane są jaskinie w generacji terenu.

Pola i metody klasy WorldSaver

- **worldData**: obiekt przechowujący dane świata do zapisu.
- **allFiles**: lista nazw plików zawierających zapisane światy.
- **CreateBuildFileName**: metoda tworząca nazwę pliku na podstawie aktualnej daty i godziny.
- **LoadBuildFileName**: metoda zwracająca nazwę pliku na podstawie indeksu w liście plików.
- **Save**: metoda statyczna zapisująca stan świata na podstawie obiektu **WorldCreator**.
- **Load**: metoda statyczna wczytująca dane świata z pliku o podanej lokalizacji.

5.6.7 CalculateBlockTypesJobs

Klasa CalculateBlockTypesJobs

Jest częścią systemu generacji terenu w grze, odpowiedzialną za obliczanie typów bloków w określonych obszarach trójwymiarowej przestrzeni. Jest to klasa bazowa, po której trzeba dziedziczyć, aby stworzyć nowe modele terenu. Każda klasa pochodna posiada tę samą strukturę **generationJob**, ponieważ struktura korzysta z interfejsu **IParallerJobFor**, która nie pozwala na wykorzystywanie klas wewnętrz siebie.

- **generationJob**: obiekt klasy **CalculateBlockTypes** odpowiedzialny za przetwarzanie generacji typów bloków.
- **AssignValues**: metoda przypisująca wartości do obiektu **generationJob** na podstawie przekazanych parametrów.

Struktura CalculateBlockTypes

CalculateBlockTypes to struktura używana do obliczeń związanych z generacją modelu terenu. Struktura ta implementuje interfejs **IJobParallelFor**, co pozwala na efektywne wykonywanie obliczeń równoległych na wielu wątkach, co pozwala na przyspieszenie generowania terenu. Wszystkie modele terenu zawierają swoją metodę, która generuje typy bloków odpowiednie wzgółdem swojego terenu.

- **chunkData:** **NativeArray** przechowująca typy bloków dla **ChunkBlock**.
- **width, height:** Wymiary **ChunkBlock**.
- **location:** pozycja chunka w świecie.
- **randoms:** **NativeArray** przechowująca generatory liczb losowych dla każdego punktu w obiekcie klasy **ChunkBlock**.
- **function:** zmienna określająca rodzaj generatora terenu do użycia .
- **Execute:** implementacja interfejsu **IJobParallelFor**, odpowiedzialna za generację typów bloków dla danego indeksu i.
- **BaseGenerator:** metoda generująca typy bloków dla generatora modelu terenu typu Base.
- **IslandGenerator:** metoda generująca typy bloków dla generatora modelu terenu typu Flying Island.
- **FlatGenerator:** metoda generująca typy bloków dla generatora modelu terenu typu Flat.

5.6.8 WorldVisualization

Klasa **WorldVisualization** zarządza ustawieniami funkcji szumu Perlina dla generacji świata. Zawiera obiekty **PerlinGrapher** reprezentujące wykresy funkcji szumu Perlina oraz **Perlin3DGrapher** dla trójwymiarowej wizualizacji. Ustawienia są przechowywane w strukturze **PerlinSettings**. Metoda **CreateSettings** tworzy ustawienia na podstawie obiektów **PerlinGrapher** i **Perlin3DGrapher**.

Klasa WorldVisualization

Klasa **WorldVisualization** pełni rolę kontrolera wizualizacji terenu. Odpowiada za zarządzanie obiektami reprezentującymi wykresy funkcji warstw oraz inicjalizację ustawień generacji terenu.

- **calculate**: obiekt klasy CalculateBlockTypesJobs odpowiedzialny za generację bloków w świecie.
- **perlinGraphers**: lista obiektów klasy PerlinGrapher reprezentujących wykresy funkcji szumu Perlina.
- **perlinGrapher3D**: obiekt klasy Perlin3DGrapher reprezentujący wykres funkcji szumu Perlina w trzech wymiarach.
- **perlinSettings**: lista ustawień funkcji szumu Perlina.
- **Start**: metoda uruchamiana podczas startu, inicjalizująca ustawienia, takie jak dodanie brakujących obiektów PerlinGrapher do listy.
- **CreateSettings**: metoda tworząca ustawienia funkcji szumu Perlina na podstawie obiektów PerlinGrapher i Perlin3DGrapher.

Struktura PerlinSettings

PerlinSettings to struktura przechowująca parametry konfiguracyjne funkcji szumu Perlina.

- **heightScale**: skala wysokości dla funkcji szumu Perlina.
- **scale**: skala dla funkcji szumu Perlina.
- **octaves**: liczba oktaw dla funkcji szumu Perlina.
- **heightOffset**: przesunięcie wysokości dla funkcji szumu Perlina.
- **probability**: prawdopodobieństwo dla funkcji szumu Perlina.
- **PerlinSettings**: konstruktor ustawień funkcji szumu Perlina z parametrami: skala wysokości, skala, liczba oktaw, przesunięcie wysokości, prawdopodobieństwo.

5.6.9 PerlinGrapher

PerlinGrapher to klasa, która implementuje graficzne przedstawienie funkcji szumu Perlina. Służy do wizualizacji wyników generowanych przez funkcję szumu Perlina na wykresie liniowym w środowisku Unity oraz w programie.

Pola klasy

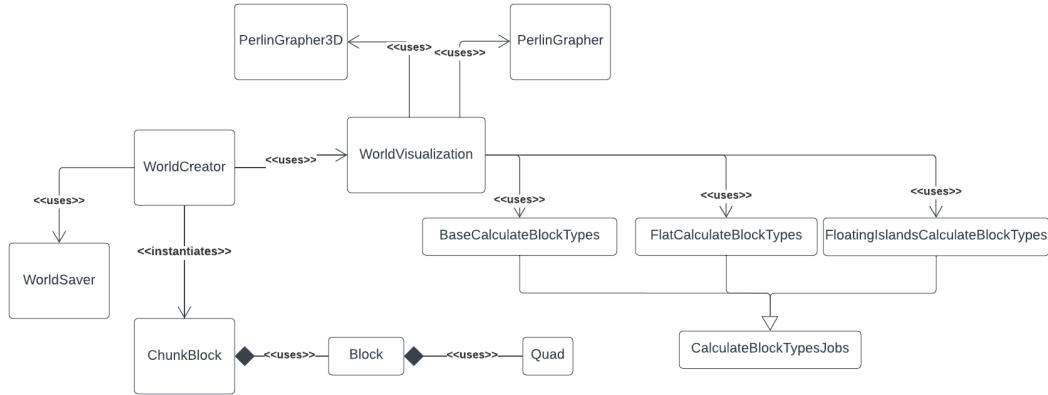
- **lr:** obiekt klasy LineRenderer do rysowania linii na podstawie funkcji szumu Perlina.
- **heightScale:** skala wysokości dla funkcji szumu Perlina.
- **scale:** skala dla funkcji szumu Perlina.
- **octaves:** liczba oktaw dla funkcji szumu Perlina.
- **heightOffset:** przesunięcie wysokości dla funkcji szumu Perlina.
- **probability:** prawdopodobieństwo dla funkcji szumu Perlina.
- **lineMaterial:** materiał używany do rysowania linii.
- **isInitComplete:** flaga wskazująca, czy inicjalizacja została już zakończona.

Metody klasy

- **Start:** metoda wywoływana podczas startu, inicjalizująca i rysująca funkcję szumu Perlina.
- **StartSetup:** metoda inicjalizująca obiekt klasy LineRenderer i materiał.
- **Graph:** metoda rysująca funkcję szumu Perlina i aktualizująca obiekt LineRenderer.
- **OnValidate:** metoda wywoływana podczas zmiany wartości w inspektorze Unity, aktualizująca funkcję szumu Perlina.

5.7 Diagram klas

Rysunek 5.8 przedstawia uproszczony diagram klas implementacji.



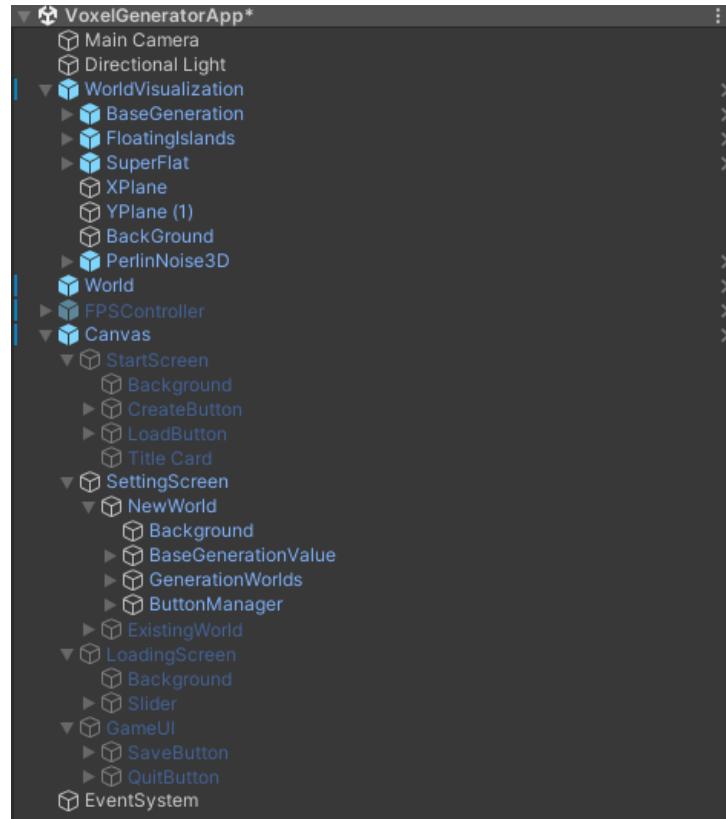
Rysunek 5.8: Uproszczony diagram klas implementacji

5.8 Implementacja programu VoxelGenerator

Program *VoxelGenerator* ma za zadanie pokazanie działania implementacji proceduralnego generatora modelu terenu. Program składa się z jednej sceny o nazwie *VoxelGeneratorApp*, która jest uruchamiana przy starcie działania aplikacji.

5.8.1 Opis Sceny

Na scenie znajdują się elementy UI, które pozwalają na sprawdzenie działania implementacji. Układ sceny znajduje się na rysunku 5.9. Obiekt **WorldVisualization** składa się z przykładowych modeli terenu (**BaseGeneration**, **FloatingIslands**, **SuperFlat**), pod którymi znajdują się poszczególne warstwy. Dodatkowo znajduje się obiekt **PerlinNoise3D**, który służy wizualizacji wastwy jaskiń. Każdy z modeli zawiera klasę **WorldVisualization**, do której również podłączony jest klasa pochodna **CalculateBlockTypeJobs**. Każda z warstw ma podłączoną klasę **PerlinGraph**, aby była możliwość tworzenia widocznych grafów do wizualizacji algorytmu. Obiekt **World** zawiera podłączoną klasę **WorldCreator**, która odpowiada za generowanie terenu. Obiekt **Canvas** zawiera w sobie elementy UI, które służą wywoływania i uruchamiania implementacji generowania terenu. Dodatkowo na scenie znajduje się **FPSController**, wykorzystany z paczki Standard Unity Assets (3.4.4), który służy do poruszania się po stworzonym terenie.



Rysunek 5.9: Okno Hierarchi na scenie VoxelGeneratorApp

5.8.2 Ustawienia Unity

Platforma docelowa została ustawiona na "Windows, Mac, Linux" z parametrem "Target Platform" ustawioną na "Windows" i parametrem "Architecture" ustawioną "Intel 64-bit".

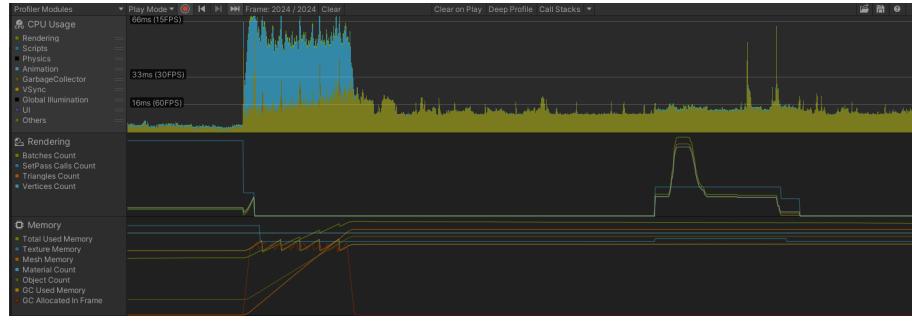
Rozdział 6

Weryfikacja i walidacja

6.1 Testowanie w środowisku Unity

W środowisku Unity istnieje możliwość dynamicznego testowania kodu dzięki funkcji komplikacji w locie i uruchamianiu w trybie Playmode bezpośrednio w edytorze. W trakcie tego procesu użytkownicy mają dostęp do konsoli Unity, która informuje o pojawiających się błędach związanych z kodem aplikacji lub konfliktami wewnętrz silnika. Dodatkowo, w trakcie działania trybu Playmode, istnieje możliwość zatrzymania aplikacji w konkretnej klatce lub ręcznego przemieszczania się między klatkami za pomocą funkcji Pause i Step. W trybie Playmode można również dokonywać zmian w widocznych parametrach komponentów oraz na scenie, takich jak zmiany położenia elementów. Ważne jest jednak pamiętanie, że wprowadzone w trybie Playmode modyfikacje nie zostaną zachowane po wyjściu z tego trybu i zostaną zresetowane. W oknie Inspector edytora istnieje dodatkowo opcja wejścia w tryb Debug, który umożliwia podgląd parametrów zazwyczaj ukrytych w standardowym widoku. Ten tryb jest aktywny również podczas trybu Playmode, co ułatwia debugowanie. Przy korzystaniu z narzędzia Visual Studio, istnieje opcja dołączenia go do środowiska Unity, co umożliwia ustawianie punktów przerwania w kodzie. Gdy aplikacja osiągnie punkt przerwania w trakcie trybu Playmode, nastąpi automatyczne zatrzymanie operacji w silniku Unity, co umożliwia przeglądanie wartości zmiennych w Visual Studio.

Do zoptymalizowania projektu w środowisku Unity wykorzystuje się narzędzie o nazwie Profiler. Profiler to narzędzie analityczne, które gromadzi informacje dotyczące danej klatki, uwzględniając wszystkie opóźnienia oraz czasy wykonania poszczególnych klas i metod. Dodatkowo Profiler umożliwia monitorowanie zajętości pamięci, co pozwala ocenić, czy pamięć jest właściwie zwalniana w trakcie działania aplikacji. Przykład działania tego narzędzia został zilustrowany na rysunku 6.1.



Rysunek 6.1: Działanie narzędzia Profiler podczas testowania aplikacji

6.2 Testowanie stworzonej aplikacji

Program był testowany na komputerze o następujących parametrach:

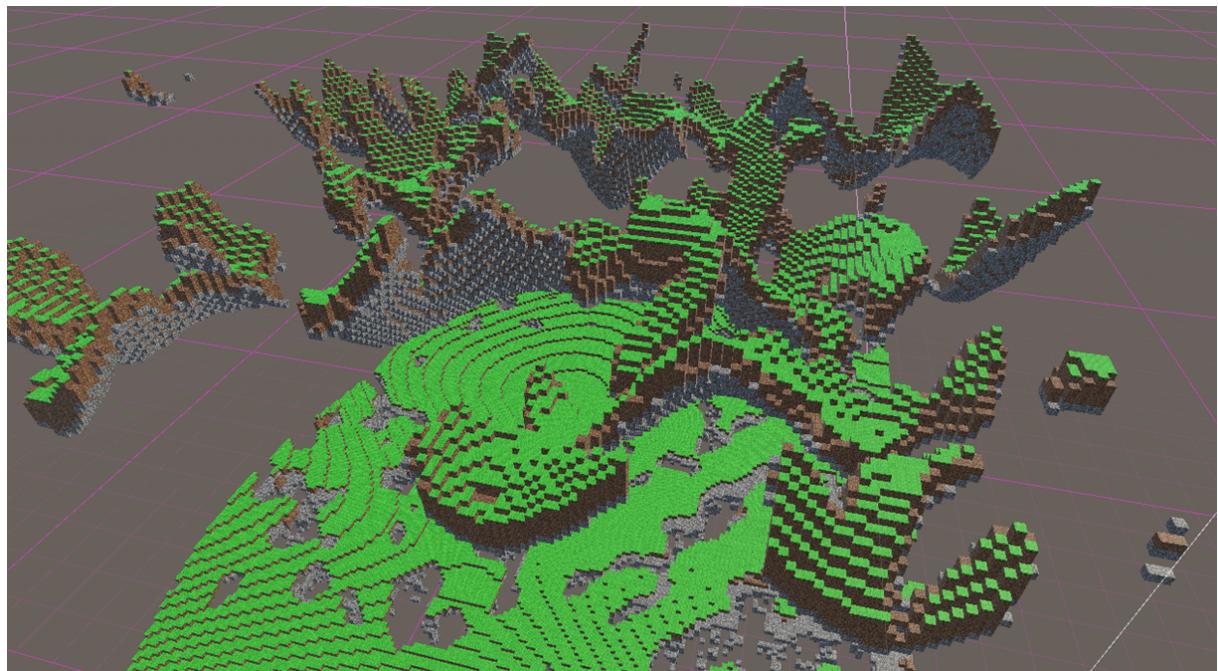
- Procesor AMD Ryzen 7 4800H,
- 32GB pamięci RAM,
- NVIDIA GeForce RTX 2060,
- Dysku Samsung 980 evo.

Podczas początkowego generowania terenu, program wykorzystywał 70 procent procesora przy taktowaniu około 3.5 GHz. Jednocześnie program działał w okolicach 15 klatek na sekundę (FPS). Po stworzeniu podstawowego terenu program wykorzystywał około 10 procent procesora przy takim samym taktowaniu, jednocześnie osiągał przedział 130 a 90 klatek na sekundę. Podczas poszerzania terenu, program nie zmieniał ilości klatek na sekundę, kiedy procesor ponownie wykorzystywał 30 procent procesora.

Rozdział 7

Podsumowanie i wnioski

Cel pracy dyplomowej został zrealizowany. Została stworzona implementacja proceduralnej generacji modelu terenu zawierającego latające wyspy w silniku Unity. Implementacja generuje teren za pomocą wybranej metody modelu terenu. Głównym zaimplementowanym modelem jest model latających wysp. Przykład wygenerowanych latających wysp jest widoczny na rysunku 7.1. Został stworzony program o nazwie *VoxelGenerator*, który pokazywał stworzoną implementację w akcji.



Rysunek 7.1: Przykład wygenerowanego modelu latających wysp

7.1 Dalszy rozwój implementacji

Implementację można rozbudowywać o kolejne modele terenu, jak również poszerzyć możliwości generatora, przykładowo o tworzenie terenów wodnych czy dodawanie drzew w terenie. W dalszym rozwoju można zoptymalizować pod względem tworzenia obiektów klasy **ChunkBlock**. Możliwa jest opcja tworzenia jednego dużego obiektu klasy **ChunkBlock**, który sięga do górnego poziomu stworzonego terenu, zamiast kolumny obiektów klasy **ChunkBlock**. Dodatkowo można poszerzyć i powiększyć możliwości algorytmowe, aby modele terenu można jeszcze bardziej modyfikować i tworzyć bardziej unikalne.

Bibliografia

- [1] R. Cook T. DeRose G. Drettakis D.S. Ebert J.P. Lewis K. Perlin M. Zwicker A. Lagaë S. Lefebvre. „State of the Art in Procedural Noise Functions”. W: *Eurographics 2010 - State of the Art Reports*. 2010, s. 1–19.
- [2] Robert Bridson. *Fast Poisson Disk Sampling in Arbitrary Dimensions*. 2007. URL: <https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf> (term. wiz. 25.12.2023).
- [3] Sabine Coquillart. „Extended Free-Form Deformation: Sculpturing Tool for 3D Geometric Modeling”. W: *Computer Graphics* 24.4 (1990), s. 187–196.
- [4] Avatar Fandom. *Hallelujah Mountains*. 2023. URL: https://james-camerons-avatar.fandom.com/wiki/Hallelujah_Mountains (term. wiz. 24.12.2023).
- [5] Minecraft Fandom. *Minecraft custom world generation*. 2023. URL: https://minecraft.fandom.com/wiki/Custom_world_generation#Feature (term. wiz. 23.12.2023).
- [6] Minecraft Fandom. *Minecraft World type*. 2023. URL: https://minecraft.fandom.com/wiki/World_type (term. wiz. 23.12.2023).
- [7] No Man's Sky Fandom. *Voxel*. 2022. URL: [https://nomanssky.fandom.com/wiki/Terrain_\(Atlas\)](https://nomanssky.fandom.com/wiki/Terrain_(Atlas)) (term. wiz. 23.12.2023).
- [8] Minecraft Feedback. *Minecraft: Java Edition - 1.20.4*. 2023. URL: <https://feedback.minecraft.net/hc/en-us/articles/22046600558221-Minecraft-Java-Edition-1-20-4> (term. wiz. 23.12.2023).
- [9] Sebastian Lague. *Coding Adventure: Marching Cubes*. 2019. URL: <https://www.youtube.com/watch?v=M3iI2l0ltbE> (term. wiz. 23.12.2023).
- [10] Steve Losh. *Terrain Generation with Midpoint Displacement*. 2016. URL: <https://stevelosh.com/blog/2016/02/midpoint-displacement/> (term. wiz. 24.12.2023).
- [11] Unity Manual. *Terrain: Create Neighbot Terrain*. 2022. URL: <https://docs.unity3d.com/Manual/terrain>CreateNeighborTerrains.html> (term. wiz. 23.12.2023).
- [12] Microsoft. *Visual Studio 2022 Documentation*. 2023. URL: <https://learn.microsoft.com/en-us/visualstudio/windows/?view=vs-2022&preserve-view=true> (term. wiz. 26.12.2023).

- [13] Jen Lowe Patricio Gonzalez Vivo. *The Book of Shaders*. 2015. URL: <https://thebookofshaders.com/13/> (term. wiz. 23.12.2023).
- [14] Cantero X. Salazar M. et Santamaría-Ibirika A. „Procedural approach to volumetric terrain generation”. W: *The Visual Computer* 30.9 (2013), s. 997–1007.
- [15] Unity Technologies. *IJobParallelFor*. 2023. URL: <https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJobParallelFor.html> (term. wiz. 26.12.2023).
- [16] Unity Technologies. *TextMeshPro*. 2023. URL: <https://docs.unity3d.com/Manual/com.unity.textmeshpro.html> (term. wiz. 26.12.2023).
- [17] Unity Technologies. *Unity user manual 2021.3 (lts)*. 2023. URL: <https://docs.unity3d.com/2021.3/Documentation/Manual/UnityManual.html> (term. wiz. 26.12.2023).
- [18] Unity Technoloies. *Old Unity Standard Assets*. 2020. URL: <https://github.com/jamschutz/Unity-Standard-Assets> (term. wiz. 27.12.2023).
- [19] Hong Yi Timothy S. Newman. „A survey of the marching cubes algorithm”. W: *Computers & Graphics* 30.5 (2006), s. 854–879.
- [20] Tlauncher. *Mountain Basin Custom Terrain*. 2020. URL: https://tlauncher.org/en/maps_24/mountain-basin-custom-terrain_9429.html (term. wiz. 23.12.2023).
- [21] Wikipedia. *Floating cities and islands in fiction*. 2023. URL: https://en.wikipedia.org/wiki/Floating_cities_and_islands_in_fiction#Fictional_examples (term. wiz. 24.12.2023).
- [22] Wikipedia. *Perlin Noise*. 2023. URL: https://en.wikipedia.org/wiki/Perlin_noise (term. wiz. 23.12.2023).
- [23] Alan Zucconi. *The World Generation of Minecraft*. 2022. URL: <https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/> (term. wiz. 23.12.2023).

Dodatki

Spis skrótów i symboli

fBm Fractal Brownian Motion

UI User Interface

fpc First Person Controller

FPS Frames Per Seconds

Lista dodatkowych plików, uzupełniających tekst pracy

W systemie do pracy dołączono dodatkowe pliki zawierające:

- zbudowany program dla systemu Windows 10 i Windows 11,
- repozytorium z kodem źródłowym,
- link do filmiku pokazujący działanie programu,
- dokumentację wygenerowaną w Doxygen

Spis rysunków

2.1	Przykład stworzonego terenu [20]	4
2.2	Przykład terenu na planecie [7]	5
2.3	Wygląd tekstury 2D szumu Perlina reprezentowany w formie mapy wysokości [22]	6
2.4	Algorytm Marching Cubes w użyciu; Źródło: https://imgur.com/IsbIv8i	7
2.5	Przykłady wyglądu latających wysp w różnych mediach	9
2.6	Zobrazowanie warst latającej wyspy	10
3.1	Diagram przypadków użycia programu	14
4.1	Menu główne programu <i>VoxelGenerator</i>	18
4.2	Menu tworzenia nowego terenu	18
4.3	Wizualizacja wyglądu warstw 2D	20
4.4	Wygląd menu wczytywania nowego terenu	21
4.5	Wygląd programu po wejściu na stworzony teren	22
5.1	Wygląd podziału kolumny kawałków terenu	24
5.2	Wygląd podziału kolumny kawałków terenu z wytworzonymi przerwami	25
5.3	Układ współrzędnych dla kawałków terenu	26
5.4	Funkcja fBM w klasie MeshUtils	27
5.5	Funkcja fBM3D w klasie MeshUtils	28
5.6	Hierarchia obiektów dla modelu bazowego terenu	29
5.7	Wyzualizacja budowy kolumny obiektów klasy ChunkBlock	33
5.8	Uproszczony diagram klas implementacji	46
5.9	Okno Hierarchi na scenie VoxelGeneratorApp	47
6.1	Działanie narzędzia Profiler podczas testowania aplikacji	50
7.1	Przykład wygenerowanego modelu latających wysp	51