

**Uniwersytet Jagielloński w Krakowie**  
Wydział Fizyki, Astronomii i Informatyki Stosowanej

**Artur Chrząstowski**

Nr albumu: 1187801

# **Porównanie wydajności generowania terenu 3D tworzonego za pomocą algorytmu Marching Cubes na CPU i GPU**

Praca magisterska  
na kierunku Informatyka gier komputerowych

Praca wykonana pod kierunkiem  
dr Wojciecha Palacza  
z Instytutu Informatyki Stosowanej

Kraków 2023

## **Oświadczenie autora pracy**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....  
Kraków, dnia

.....  
Podpis autora pracy

## **Oświadczenie kierującego pracą**

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....  
Kraków, dnia

.....  
Podpis kierującego pracą

## **Abstract**

Marching Cubes algorithm is commonly used for 3D terrain generation in video games and other commercial products. This algorithm might be significantly accelerated by running some or all of the required computations on a graphic processor; however, research papers researching this possibility often focus their analysis only on a single aspect of performance, for example, generation time. The analysis performed as part of this research paper expands the performance analysis by implementing 5 different terrain generation methods using the Unity engine, each using the CPU and GPU resources to a different extent and in different ways, while also taking into account different factors impacting the practicality of different approaches, such as the number of frames drawn per second after finishing the generation process, GPU memory cache usage and the maximum achievable terrain size. Collected data helps to determine when and to what extent moving the computation to the GPU is beneficial and how that affects other aspects of the application's operation.

## **Abstrakt**

Algorytm Marching Cubes jest popularnie wykorzystywany do generowania terenu 3D w grach komputerowych i innych komercyjnych produktach. Ten algorytm może zostać znacznie przyspieszony poprzez wykorzystanie procesora graficznego do wykonywania części lub całości obliczeń, jednak prace naukowe eksplorujące tę możliwość często ograniczają zakres analizy do pojedynczego aspektu wydajności, przykładowo do samego czasu generacji. Analiza przeprowadzona w ramach tej pracy poszerza zakres badania wydajności, poprzez implementację 5 różnych metod generowania terenu w silniku Unity, w różnym stopniu i w różny sposób wykorzystujących zasoby CPU i GPU oraz biorąc pod uwagę dodatkowe czynniki wpływające na praktyczność zastosowania poszczególnych metod, takie jak liczba klatek rysowanych na sekundę po zakończeniu generacji, ilość wykorzystanej pamięci podręcznej GPU i maksymalny możliwy do uzyskania rozmiar terenu. Przeprowadzone pomiary pozwalają stwierdzić, w jakim stopniu i w jakich okolicznościach przeniesienie obliczeń na GPU pozwala na zwiększenie wydajności generowania terenu za pomocą algorytmu Marching Cubes oraz jakie są tego konsekwencje w innych aspektach działania aplikacji.

# Spis treści

<b>Wstęp</b>	<b>5</b>
<b>1 Teren w grach komputerowych</b>	<b>6</b>
1.1 Metody reprezentacji terenu w grach komputerowych . . . . .	6
1.2 Metody proceduralnej generacji terenu . . . . .	7
1.2.1 Metody generowania wartości terenu . . . . .	7
1.2.2 Metody generowania powierzchni terenu . . . . .	8
1.3 Wbudowanie narzędzia do tworzenia terenu dostępnego w silnikach gier . . . . .	11
<b>2 Algorytm Marching Cubes</b>	<b>12</b>
2.1 Historia algorytmu . . . . .	12
2.2 Działanie algorytmu . . . . .	14
2.3 Wykorzystanie algorytmu w grach komputerowych . . . . .	16
<b>3 Projekt badawczy</b>	<b>17</b>
3.1 Ogólny proces generowania terenu . . . . .	17
3.1.1 Generowanie pola skalarnego . . . . .	17
3.1.2 Wykonanie algorytmu Marching Cubes . . . . .	21
3.1.3 Wygładzanie powierzchni . . . . .	22
3.1.4 Budowanie siatki . . . . .	23
3.1.5 Teksturowanie . . . . .	24
3.2 Implementacje generowania terenu . . . . .	26
3.2.1 Metody CPU . . . . .	26
3.2.2 Metoda hybrydowa . . . . .	28
3.2.3 Metody GPU . . . . .	30
3.2.4 Modyfikacje metod . . . . .	33
3.3 Metody pomiaru wydajności . . . . .	34
3.3.1 Automatyczny test wydajności . . . . .	35
3.4 Finalny produkt . . . . .	36
<b>4 Wyniki pomiarów</b>	<b>37</b>
4.1 Parametry sprzętowe urządzeń . . . . .	37
4.2 Porównanie wydajności metod generowania terenu . . . . .	38
4.2.1 Porównanie ogólne . . . . .	38
4.2.2 Uproszczony teren . . . . .	42
4.2.3 Renderowanie pośrednie . . . . .	45
4.2.4 Przycinanie rozmiaru generycznych siatek i buforów . . . . .	46
4.2.5 Unikanie buforów typu <i>append</i> . . . . .	48
4.2.6 Zastosowanie innych interfejsów graficznych . . . . .	49
4.3 Podsumowanie wyników pomiarów . . . . .	50
<b>Podsumowanie</b>	<b>52</b>

# Wstęp

Teren stanowi integralną część wielu programów komputerowych. Jest to szczególnie widoczne w przypadku gier, gdzie teren pełni rolę wirtualnego środowiska, w którym często osadzona jest cała rozgrywka. Na przestrzeni lat powstało wiele technik tworzenia i reprezentacji terenu w grach komputerowych i innych produktach gdzie jest on wykorzystywany. Celem tej pracy jest zbadanie wydajności procesu proceduralnej generacji terenu 3D tworzonego za pomocą algorytmu Marching Cubes na CPU i GPU. Zagadnienie wydajności jest szczególnie istotne w kontekście proceduralnej generacji terenu w grach komputerowych, gdzie celem jest osiągnięcie maksymalnie płynnej rozgrywki, przy jednoczesnym zachowaniu jak najwyższej jakości wyświetlanego obrazu. Celem przeprowadzonej analizy jest umożliwienie wyboru metody robiącej to najlepiej.

Algorytm Marching Cubes jest obiektem zainteresowania tej pracy, ponieważ jest on jednym z najpopularniejszych algorytmów służących do generowania trójwymiarowego terenu. Dowodem popularności tego algorytmu jest wysoka liczba artykułów naukowych opublikowanych na jego temat oraz stworzone na jego podstawie liczne warianty, takie jak Marching Tetrahedra, Extended Marching Cubes, Adaptive Marching Cubes, Dual Marching Cubes oraz Surface Nets. Ponieważ wszystkie te algorytmy działają na podobnej zasadzie, większość wniosków wyciągniętych na temat wydajności implementacji algorytmu Marching Cubes można uogólnić na nie wszystkie.

W celu przeprowadzenia analizy wydajności stworzono projekt badawczy w silniku Unity. Wybór silnika Unity wynika z jego dużej popularności i dostępu do wielu narzędzi pozwalających wykorzystać w szerokim zakresie zarówno CPU, jak i GPU do generacji terenu. Stworzona aplikacja pozwala wykonywać pomiary w postaci czasu generacji, średniej liczby klatek rysowanych na sekundę oraz innych wielkości związanych z wydajnością, takich jak ilość wykorzystanej pamięci podręcznej GPU. Oprócz tego aplikacja udostępnia szereg parametrów pozwalających modyfikować działanie metod generujących teren oraz jego właściwości, a same pomiary wykonane zostały na dwóch różnych urządzeniach.

Szeroki zakres przeprowadzonej analizy, uwzględniający wiele czynników mających znaczenie w komercyjnym zastosowaniu, takim jak produkcja gier komputerowych, jest elementem wyróżniającym tę pracę na tle innych przeprowadzonych badań. Prace takie jak [1] oraz [2] również prezentują wydajne implementacje algorytmu Marching Cubes na procesory graficzne, jednak nie uwzględniają wielu aspektów wydajnościowych mających znaczenie w kontekście komercyjnym, takich jak liczba klatek rysowanych na sekundę po zakończeniu generacji terenu, oraz stopień wykorzystania pamięci podręcznej GPU. Z tego powodu trudno jest na podstawie tych badań wywnioskować, czy stosowanie przedstawionych tam implementacji jest opłacalne w kontekście tworzenia gier komputerowych lub innych produktów, podobnie wymagających pod kątem wydajności.

Treść pracy jest podzielona na 4 rozdziały. Rozdziały 1 i 2 są wprowadzeniem teoretycznym, odpowiednio do zagadnienia generacji terenu w kontekście gier komputerowych i do samego algorytmu Marching Cubes. Rozdział 3 skupia się na implementacji projektu badawczego, opisując kolejne kroki generowania terenu, implementację metod przeprowadzających je oraz sposób wykonywania pomiarów wydajności. W rozdziale 4 omówione są wyniki przeprowadzonych pomiarów oraz wyciągnięte na ich podstawie wnioski.

# 1. Teren w grach komputerowych

W kontekście gier komputerowych termin „teren” odnosi się do wirtualnego środowiska, w którym toczy się rozgrywka. Za teren można uznać wszelkiego rodzaju powierzchnie, roślinność, budynki i inne obiekty składające się na świat gry. Teren może być zarówno dwuwymiarowy, jak i trójwymiarowy, zależnie od potrzeb gry.

Teren w grach może być tworzony na różne sposoby. Jednym z podziałów, jakie można zastosować, jest podział na teren tworzony ręcznie przez artystę przed rozpoczęciem rozgrywki, a teren generowany proceduralnie w trakcie trwania rozgrywki. Oba podejścia, mają szerokie zastosowanie, w zależności od potrzeb końcowego produktu. Możliwe jest również podejście hybrydowe, gdzie teren przygotowywany jest przez artystę wykorzystującego narzędzia proceduralnej generacji.

Przedmiotem tej pracy jest realistyczny teren trójwymiarowy generowany proceduralnie, rozumiany wyłącznie jako powierzchnia tworząca podstawę świata gry, generowana w całości w trakcie działania programu na podstawie wcześniej zdefiniowanych parametrów. Pozostałe elementy składające się na szerzej rozumiany teren mogą zostać później dodane do świata gry za pomocą innych algorytmów, co wykracza poza zakres tej pracy.

## 1.1 Metody reprezentacji terenu w grach komputerowych

Jednym z kluczowych aspektów implementacji terenu w grach komputerowych jest jego reprezentacja w pamięci komputera. Od zastosowanej reprezentacji zależą właściwości i ograniczenia uzyskanego terenu. Na podstawie popularnych rozwiązań zastosowanych w istniejących grach komputerowych można wyróżnić następujące metody:

**Modele 3D i obrazy 2D** W wielu grach teren zbudowany jest z gotowych fragmentów, stworzonych i rozmieszczonych wcześniej w silniku gry lub zewnętrznym programie. Takie fragmenty przechowywane są w pamięci przed uruchomieniem programu w postaci modeli 3D lub obrazów 2D i są jedynie odpowiednio wczytywane w czasie działania programu. Zdarza się, że gotowe fragmenty rozmieszczone są proceduralnie, przykładowo na dwuwymiarowej siatce, jednak tego typu działanie nie jest obiektem zainteresowania tej pracy.

**Mapa wysokości** W grach z proceduralnie generowanym terenem częstym rozwiązaniem jest mapa wysokości. Jest to teren reprezentowany za pomocą dwuwymiarowej tablicy wartości, gdzie każda wartość reprezentuje wysokość jednego punktu terenu. Taka reprezentacja jest ograniczona i nie pozwala na generowanie takich struktur jak jaskinie i nawisy, jednak jest wystarczająca dla potrzeb wielu gier.

**Pola skalarne** Pole skalarne to funkcja przyporządkowująca do każdego punktu przestrzeni pojedynczą wartość liczbową. W kontekście reprezentacji terenu pole skalarne ma zwykle postać trójwymiarowej siatki równie oddalonych od siebie punktów. Takie pole można wizualizować generując jego izopowierzchnię. Izopowierzchnia pola skalarnego to powierzchnia oddzielająca punkty pola o wartościach niższych i wyższych niż pewna zadana wartość liczbowa. Takie wartości można interpretować jako znajdujące się wewnątrz i na zewnątrz terenu. Teren generowany w tej sposób pozwala tworzyć dowolne struktury w zależności od zadanej gęstości siatki.

Możliwe są także podejścia hybrydowe, łączące wiele powyższych metod, przykładowo teren wczytywany jako model 3D, a następnie konwertowany do reprezentacji pola skalarnego, w celu umożliwienia interaktywnej modyfikacji przez gracza. Oprócz tego każdy z powyższych rodzajów terenu może być dodatkowo wzbogacony, poprzez proceduralne algorytmy modyfikujące i dodające elementy, takie jak roślinność, formacje skalne, budynki i tym podobne.

## 1.2 Metody proceduralnej generacji terenu

Niezależnie od zastosowanego podejścia, proceduralne generowanie terenu można podzielić na dwie zasadnicze fazy; generowanie zestawu wartości charakteryzujących teren i wykorzystanie tych wartości w celu uzyskania reprezentacji terenu możliwej do wyświetlenia na ekranie przez silnik graficzny gry. W różnych fazach wykorzystywane są różne algorytmy.

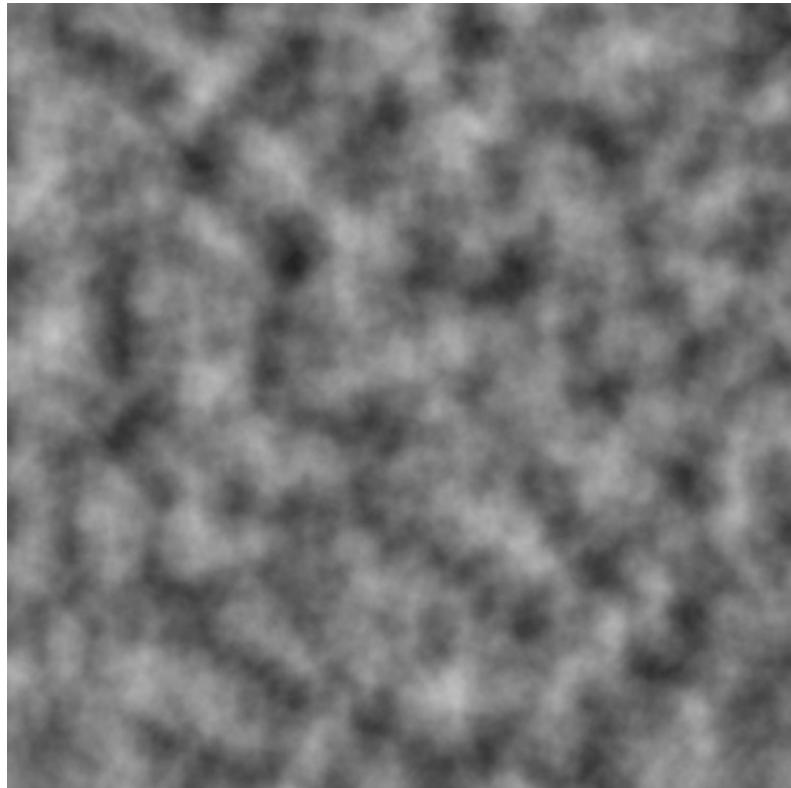
### 1.2.1 Metody generowania wartości terenu

Generowanie terenu w grach komputerowych jest trudne z powodu wysokiej złożoności struktur i powierzchni istniejących w przyrodzie, które próbujemy odwzorować za pomocą prymitywnych kształtów takich jak trójkąty i kwadraty. Teren jest zazwyczaj charakteryzowany jako fraktal, z powodu swojej nieskończonej złożoności i samopodobieństwa, rozumianego jako możliwość podziału terenu na mniejsze części, każda podobna do oryginału [3].

Najczęściej stosowanymi algorytmami do generacji wartości terenu są funkcje generujące pseudolosowy szum. Takie funkcje często przyjmują jako argument współrzędne punktu w przestrzeni dwu- lub trójwymiarowej i zwracają liczbę zmiennoprzecinkową w przedziale 0 do 1, lub -1 do 1. Aby teren był ciągły, ważne jest, aby do punktów leżących blisko siebie przypisane były zbliżone wartości. Pomimo tego, że taki szum nie jest w swojej naturze fraktalny, możliwe jest uzyskanie efektu imitującego to zjawisko, poprzez nałożenie na siebie wiele kolejnych oktafów szumu, z rosnącą częstotliwością i malejącą amplitudą. To podejście pozwala osiągnąć dowolnie wysoki poziom szczegółów, w zakresie, na jaki pozwala nam precyzja obliczeniowa komputera.

Do najpopularniejszych algorytmów generujących szum należą: Diamond-square [4], Value noise [5], Perlin noise [6], Simplex noise [7] i Worley noise [8]. Te algorytmy różnią się zarówno jakością generowanego szumu, jak i wydajnością obliczeniową i pamięciową, ale wszystkie umożliwiają generowanie wiarygodnego terenu. W zależności od potrzeb można wykorzystywać szum w różny sposób, na przykład bezpośrednio jako mapa wysokości (Rys. 1), lub pośrednio jako modyfikator wartości pola skalarnego.

Do innych metod proceduralnego generowania terenu należą przykładowo algorytmy ewolucyjne oraz symulacje zjawisk naturalnych, takie jak erozja hydrauliczna lub ruch płyt tektonicznych. Te metody są jednak zazwyczaj trudniejsze do wykorzystania niż zwykły szum. Algorytmy ewolucyjne trudno jest kontrolować za pomocą parametrów, ponieważ generowany przez nie teren zmienia się nieprzewidzialnie w trakcie procesu ewolucji. Z kolei algorytmy symulujące zjawiska naturalne zazwyczaj wymagają początkowego zestawu wartości, na którym mogą operować, wymuszając łączenie je z innymi metodami [3].

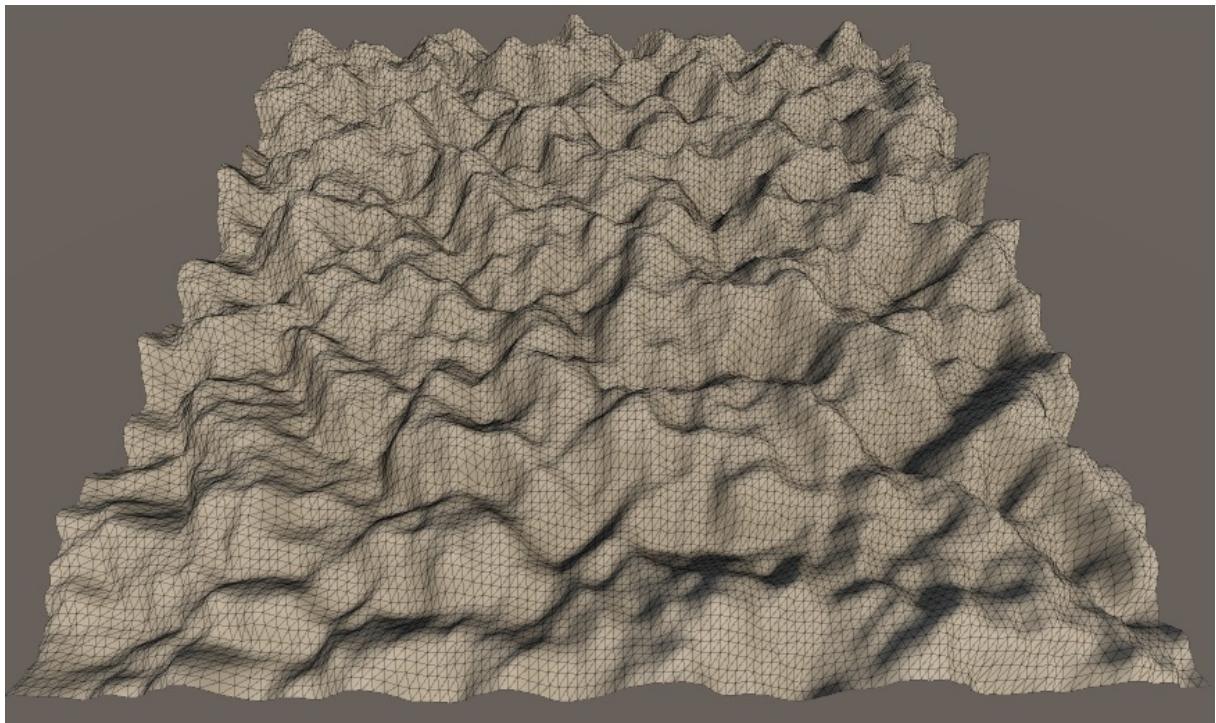


Rys. 1. Mapa wysokości wygenerowana jako tekstura w silniku Unity ze złożenia dziewięciu oktaw szumu Perlina.

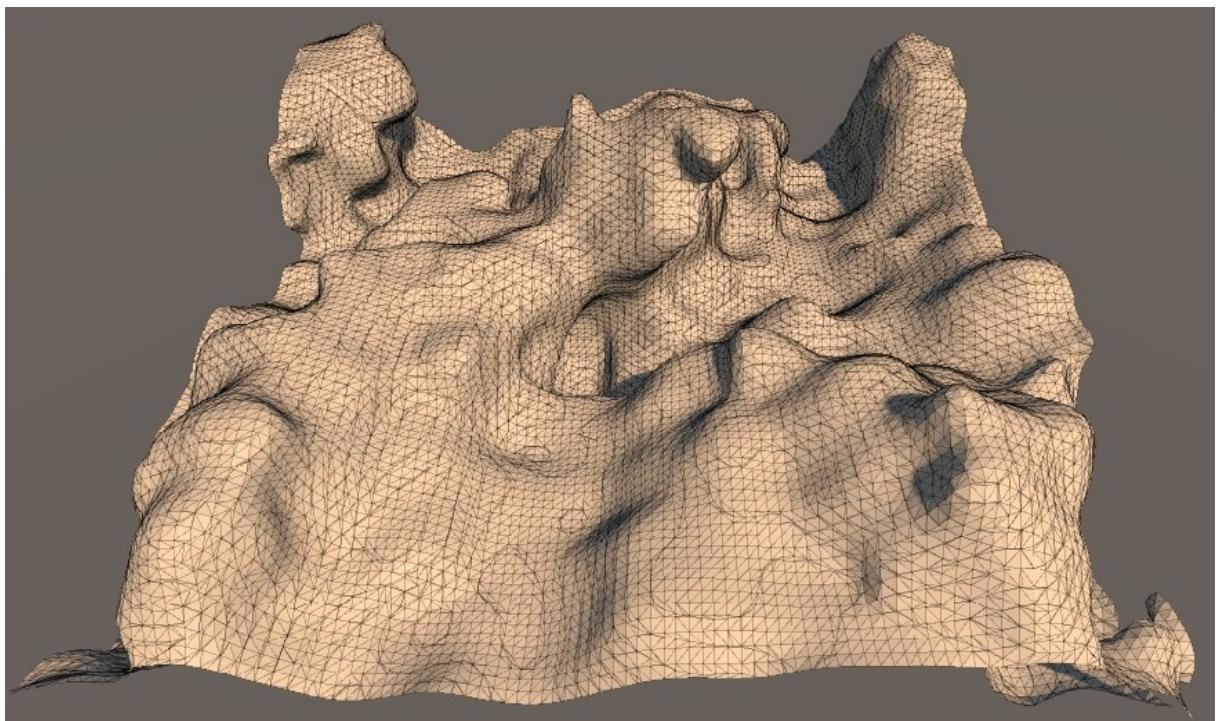
### 1.2.2 Metody generowania powierzchni terenu

Po wygenerowaniu wartości reprezentujących właściwości terenu można przystąpić do generowania jego powierzchni, najczęściej reprezentowanej w postaci listy wierzchołków i trójkątów. Jeżeli wykorzystanym formatem wartości jest mapa wysokości, stworzenie powierzchni ją reprezentującą jest trywialne. Taką powierzchnią jest płaszczyzna podzielona na kwadraty w taki sposób, że do każdego wierzchołka można przydzielić jedną wartość z mapy wysokości (Rys. 2). W przypadku zastosowania pola skalarnego należy wykorzystać jeden z algorytmów zdolnych zwizualizować jego powierzchnię, przykładowo algorytm Marching Cubes [9] (Rys. 3).

Algorytmy generujące powierzchnię na podstawie pól skalarnych rozważają te pola jako zbiór prostych brył, takich jak sześciiany lub czworościany w celu podziału danych na mniejsze fragmenty. Ta abstrakcja ułatwia tworzenie i intuicyjne zrozumienie algorytmów [10]. Zdefiniowane w ten sposób bryły są przeszukiwane, aby znaleźć izopowierzchnię pola skalarnego wewnątrz każdej z nich. Algorytmy różnią się między sobą pod wieloma względami. Do najbardziej znaczących różnic między nimi należą kształt wykorzystanych brył, metody przeszukiwania ich oraz szczegóły w procesie generowania izopowierzchni. Z tych powodów teren generowany przez różne algorytmy często ma inną jakość i właściwości, przydatne do różnych zastosowań. Algorytmy mogą również różnić się złożonością pamięciową i obliczeniową.



Rys. 2. Przykład powierzchni terenu wygenerowanej na podstawie mapy wysokości w silniku Unity.



Rys. 3. Przykład powierzchni terenu wygenerowanej na podstawie wartości pola skalarnego z wykorzystaniem algorytmu Marching Cubes w silniku Unity.

## Przykłady algorytmów generujących powierzchnię na podstawie pól skalarnych

**Marching Cubes** Marching Cubes jest jednym z najpopularniejszych i najlepiej zbadanych algorytmów generujących powierzchnię na podstawie pól skalarnych. Dzieli on przestrzeń na sześciany i określa wynikową powierzchnię dla każdego z nich na podstawie wartości pola skalarnego w jego wierzchołkach [9]. Na przestrzeni lat powstał szereg wariantów i pochodnych algorytmów adresujących niektóre z zaobserwowanych problemów Marching Cubes, takich jak niepożądane otwory w wygenerowanych powierzchniach i trójkąty o zbyt małej powierzchni, krótkich krawędziach i małych kątach.

**Extended Marching Cubes** Extended Marching Cubes adresuje problemy w standarodowej wersji algorytmu, powstające, kiedy wartości skalarne w wierzchołkach sześcianów są zbyt bliskie wartości granicznej zdefiniowanej dla algorytmu. Dzięki temu otrzymywana powierzchnia ma mniej niepoprawnych i zbyt małych trójkątów [11].

**Dual Marching Cubes** Dual Marching Cubes jest kolejnym wariantem standardowego algorytmu Marching Cubes, który stara się poprawić jakość generowanych powierzchni. Ten wariant algorytmu skupia się na generowaniu powierzchni zbudowanej z czworokątów, co pozwala uniknąć problemów związanych z niepoprawnie ukształtowanymi trójkątami. Dodatkową zaletą tego algorytmu są precyzyjnie wyznaczone wektory normalne, pozwalające lepiej wygładzić generowaną powierzchnię [12].

**Marching Tetrahedra** Marching Tetrahedra to wariant algorytmu Marching Cubes wykorzystujący czworościany zamiast sześcianów w celu poprawy topologicznych problemów poprzednika. Niestety zastosowane metody zwiększą koszt obliczeniowy algorytmu i liczbę trójkątów niezbędnych do reprezentacji powierzchni, jeśli nie zastosowane zostaną żadne dodatkowe techniki. Z tego powodu ten algorytm często jest dodatkowo poprawiany z wykorzystaniem algorytmów grupujących trójkąty, co poskutkowało powstaniem podwariantu algorytmu o nazwie Regularised Marching Tetrahedra [13].

**Surface Nets** Surface Nets jest algorytmem znaczaco różniacym się zasadą działania od algorytmu Marching Cubes i jest często określany jako łatwiejszy w implementacji. Oprócz tego generowane przez niego powierzchnie zwykle zbudowane są z mniejszej liczby trójkątów. Algorytm Surface Nets jest najskuteczniejszy przy generowaniu powierzchni gładkich lub o bardzo wysokiej rozdzielcości [14], co może ograniczyć jego przydatność przy tworzeniu terenu w grach komputerowych.

**Dual Contouring** Dual Contouring jest algorytmem inspirowanym zarówno algorytmem Marching cubes, jak i Surface Nets. Jego podstawową zaletą jest możliwość łatwego uzyskania ostrych krawędzi powierzchni, co jest trudne lub niemożliwe przy użyciu innych algorytmów obecnych na tej liście. Ta właściwość jest szczególnie przydatna w odwzorowaniu powierzchni nieorganicznych, takich jak budynki i maszyny [15]. W przypadku terenu, który nie ma zazwyczaj tylu ostrych krawędzi, przydatność tego algorytmu może być ograniczona.

## 1.3 Wbudowanie narzędzia do tworzenia terenu dostępne w silnikach gier

W wielu silnikach gier istnieją wbudowane narzędzia do tworzenia i edycji terenu. Przykładowo Unreal Engine wewnątrz swojego edytora pozwala tworzyć rozległe połacie terenu, kształtować je za pomocą szeregu narzędzi, tworzyć dla nich tekstury, optymalizować je, dodawać roślinność oraz proceduralnie wzbogacać. Te narzędzia są bardzo przydatne, jeżeli tworzona jest gra o świecie z terenem, który może zostać przygotowany przez artystę przed jej uruchomieniem. Dla terenów generowanych proceduralnie narzędzia są ograniczone i zazwyczaj wymagane jest tworzenia własnych dedykowanych skryptów i narzędzi.

W silniku Unity, za którego pomocą został stworzony projekt wykorzystany w tej pracy, również istnieją narzędzia do tworzenia i edycji terenu. Niestety te narzędzia nie są przydatne przy generowaniu terenu metodami proceduralnymi i w tym przypadku konieczne jest korzystanie z bardziej podstawowych funkcjonalności silnika. Do najważniejszych takich funkcjonalności należą:

- Tworzenie siatki modelu na podstawie zadanych tablic wierzchołków i trójkątów.
- Narzędzia ułatwiające wykonywanie obliczeń przy pomocy karty graficznej.
- Wbudowane funkcje matematyczne, metody generujące liczby pseudolosowe oraz szum Perlina.
- Materiały pozwalające na renderowanie, teksturowanie i oświetlanie stworzonych modeli oraz pozwalające na definiowanie niestandardowych shaderów, czyli programów opisujących właściwości pikseli i wierzchołków modeli wykonywanych na procesorze graficznym.
- System scen, kamer i oświetlenia, pozwalający zwizualizować teren w przestrzeni.

Wszystkie wymienione funkcjonalności sprawiają, że tworzenie gry z proceduralnie generowanym terenem w silniku takim jak Unity jest znacznie prostsze niż bez użycia silnika, lub z wykorzystaniem silnika bardziej prymitywnego. Niestety wiąże się to również z pewnymi ograniczeniami. Każda funkcjonalność jest zaimplementowana w konkretny sposób przez autorów silnika, bez szczegółowej wiedzy o tym, jak zostanie wykorzystana przez użytkowników. Sprawia to, że niektóre z nich mogą działać nieoptimalnie, jeśli zostaną użyte do proceduralnej generacji terenu. Zoptymalizowanie tych funkcjonalności jest często trudne lub nawet niemożliwe z powodu ograniczonego dostępu do kodu źródłowego i konieczności utrzymania kompatybilności pomiędzy różnymi systemami silnika. Pomimo tego na przestrzeni lat powstało wiele udanych gier wykorzystujących popularne silniki do proceduralnej generacji terenu, co pokazuje, że korzystanie z nich wciąż jest opłacalne. Przykładami takich gier są wydane w 2020 roku „Deep Rock Galactic”, stworzone w silniku Unreal Engine [16] oraz wydane w 2021 we wczesnym dostępie „Valheim”, stworzone w silniku Unity [17].

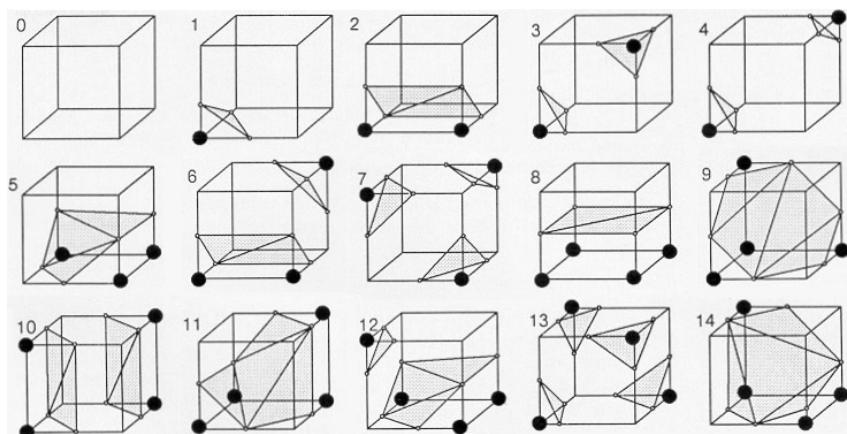
## 2. Algorytm Marching Cubes

Algorytm Marching Cubes należy do rodziny algorytmów wizualizujących obiekty trójwymiarowe, reprezentowane przez pewien zbiór danych punktowych w przestrzeni. Najczęściej wykorzystywana reprezentacją danych są trójwymiarowe pola skalarne, zdefiniowane jako zbiory regularnie rozłożonych punktów w przestrzeni 3D i wyznaczające wierzchołki sześciianów, gdzie każdy punkt posiada przyporządkowaną sobie wartość skalarną. Wizualizacja takich zbiorów ma zastosowanie w medycynie, grafice komputerowej i innych szerokich dziedzinach, co generuje duże zainteresowanie zarówno z perspektywy badań naukowych, jak i zastosowań komercyjnych.

### 2.1 Historia algorytmu

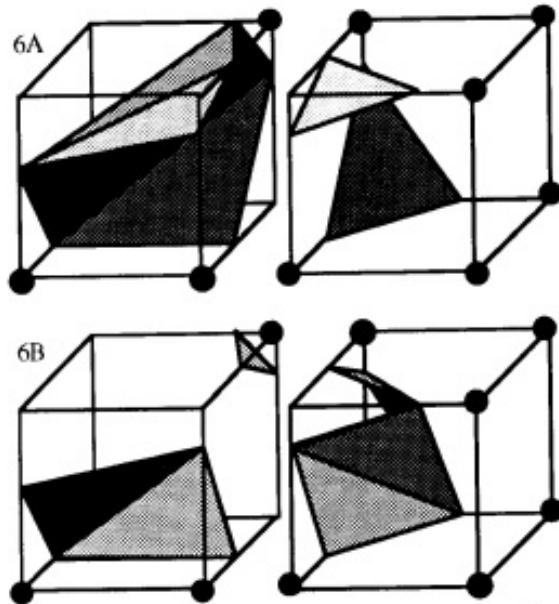
Algorytm Marching Cubes został pierwszy raz opublikowany w roku 1987, a następnie zaprezentowany na konferencji SIGGRAPH (Special Interest Group on Computer Graphics and Interactive Techniques) przez Williama E. Lorensena oraz Harveya E. Cline'a [9]. Algorytm powstał w wyniku badań prowadzonych z ramienia amerykańskiego konglomeratu General Electric, w celu wydajnej wizualizacji wyników badań przeprowadzanych za pomocą urządzeń CT (computed tomography) oraz MRI (magnetic resonance imaging) [18].

Ideą algorytmu jest podział wejściowego pola skalarnego na skończoną liczbę sześciianów, tak aby wierzchołki każdego z nich należały do pola. Następnie każdy sześciian jest badany w celu sprawdzenia, czy nie znajduje się w nim fragment powierzchni oddzielającej punkty, których wartości skalarne zdefiniowane są jako wnętrze obiektu od wartości zdefiniowanych jako powietrze. To rozróżnienie odbywa się na podstawie prostego porównania do ustalonej wcześniej wartości granicznej. Do każdego możliwego ułożenia wartości punktów w wierzchołkach sześciianu przypisany jest konkretny kształt powierzchni. Po złożeniu powierzchni powstały w każdym sześciianie uzyskujemy wyjściową powierzchnię modelu. W oryginalnie opublikowanym artykule przedstawiono 15 unikatowych wariantów powierzchni możliwych do uzyskania wewnętrz sześciianu (Rys. 4).



Rys. 4. 15 unikatowych wariantów powierzchni w oryginalnie opublikowanym algorytmie Marching Cubes.

W związku z popularnością i szerokim zastosowaniem algorytmu Marching Cubes, na przestrzeni lat stworzono wiele wariantów i poprawek mających na celu jego udoskonalenie. Jednym z najczęściej poruszanych problemów oryginalnego algorytmu są błędy topologiczne w generowanych przez niego powierzchniach. W roku 1988 Martin Dürst [19] zauważył, że podane w oryginalnej wersji algorytmu warianty powierzchni nie są wystarczające, ponieważ niektóre konfiguracje wartości umożliwiają wybór wielu różnych wariantów. Ta nieścisłość może spowodować niedokładności i otwory w generowanej powierzchni niebędące poprawną reprezentacją zadanego pola skalarnego. W celu zaadresowania tego problemu w 1991 powstał algorytm Asymptotic Decider, autorstwa Gregory'ego M. Nielsona i Bernda Hamanna [20]. To rozwiązanie jest ulepszeniem oryginalnego algorytmu Marching Cubes pozwalającym na rozstrzygnięcie niejednoznacznych przypadków wyboru powierzchni poprzez wprowadzenie nowych wariantów i metod ich doboru. Na rysunku 5 pokazano przykłady dwóch takich konfliktów, zaadresowanych przez algorytm.



Rys. 5. Przykłady dwóch niejednoznacznych wariantów wyboru powierzchni zaadresowanych przez algorytm Asymptotic Decider.

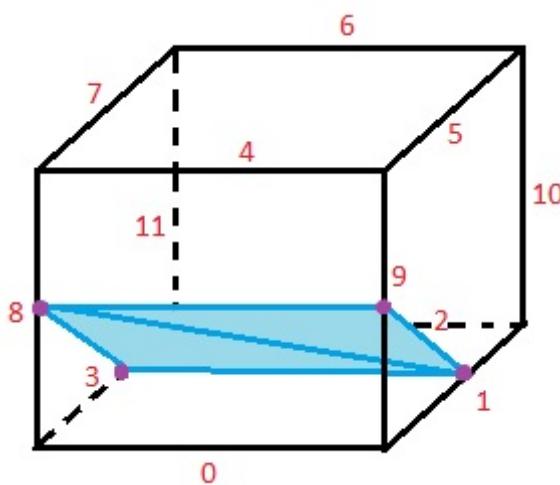
W kolejnych latach powstawały kolejne poprawki do oryginalnego algorytmu Marching Cubes, a także nowe algorytmy opierające się na nim. Jednymi z najpopularniejszych algorytmów w tej kategorii są: Extended Marching Cubes [11], Dual Marching Cubes [12] oraz Marching Tetrahedra [13]. Każdy z nich oferuje pewną poprawę w jakości powierzchni generowanej przez poprzednika, jednak zwykle kosztem zwiększenia złożoności obliczeniowej lub liczby wygenerowanych trójkątów. Zaproponowanych zostało również wiele metod poprawy wygenerowanych powierzchni po zastosowaniu dowolnego z algorytmów, jednak takie metody nie są obiektem zainteresowania tej pracy.

## 2.2 Działanie algorytmu

Standardowa implementacja algorytmu Marching Cubes korzysta z wcześniej przygotowanej tzw. tablicy triangulacji. Ta tablica określa jakie trójkąty należy wygenerować dla każdej z 256 możliwych konfiguracji ułożenia wartości w wierzchołkach sześcianu ( $2^8 = 256$ ). Na tym etapie rozważane jest jedynie czy wartość jest mniejsza, czy większa od wartości granicznej, będącej daną wejściową algorytmu. Każda konfiguracja to zbiór zawierający maksymalnie 5 trójkątów, a każdy trójkąt jest reprezentowany za pomocą trzech indeksów wyznaczających krawędzie sześcianu, na której leżą jego wierzchołki. Struktura takiej tablicy może wyglądać następująco:

```
1 int [256 * 16] TriangulationTable =
2 {
3     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
4     0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
5     0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
6     1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
7     1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
8     0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
9     9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
10    2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1, -1,
11    3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
12    ...
13 }
```

W powyższym przykładzie użyto tablicy jednowymiarowej. Dlatego, aby łatwo oddzielić od siebie poszczególne konfiguracje, każda z nich zawiera 15 indeksów (5 trójkątów \* 3 indeksy krawędzi) oraz jedną wartość oddzielającą od kolejnego zestawu. Nieistniejące trójkąty oraz wartości oddzielające zapisane są za pomocą indeksu -1. Przykładowo zestaw znajdujący się w linii oznaczonej powyżej numerem 6 wyznacza dwa trójkąty. Wierzchołki pierwszego trójkąta leżą na krawędziach sześcianu oznaczonych liczbami 1, 8 i 3, a wierzchołki drugiego trójkąta leżą na krawędziach oznaczonych liczbami 9, 8 i 1. Ten przypadek został przedstawiony na rysunku 6.



Rys. 6. Przykładowa triangulacja sześcianu. Kolorem czerwonym oznaczone są numery przypisane do krawędzi sześcianu, kolorem fioletowym oznaczone są wierzchołki trójkątów, a kolorem niebieskim same trójkąty.

Aby otrzymać poprawny zestaw wartości z tablicy triangulacji, należy wyznaczyć indeks tablicy odpowiadającej konfiguracji wartości w wierzchołkach danego sześcianu. Najczęściej stosowanym rozwiązaniem jest stworzenie indeksu jako liczba 8-bitowa, gdzie każdy bit odpowiada jednemu wierzchołkowi sześcianu i ma wartość 1, jeśli wartość w wierzchołku jest mniejsza niż zdefiniowana wartość graniczna algorytmu lub 0, jeśli wartość jest większa. Przykładowa implementacja tego kroku wygląda następująco:

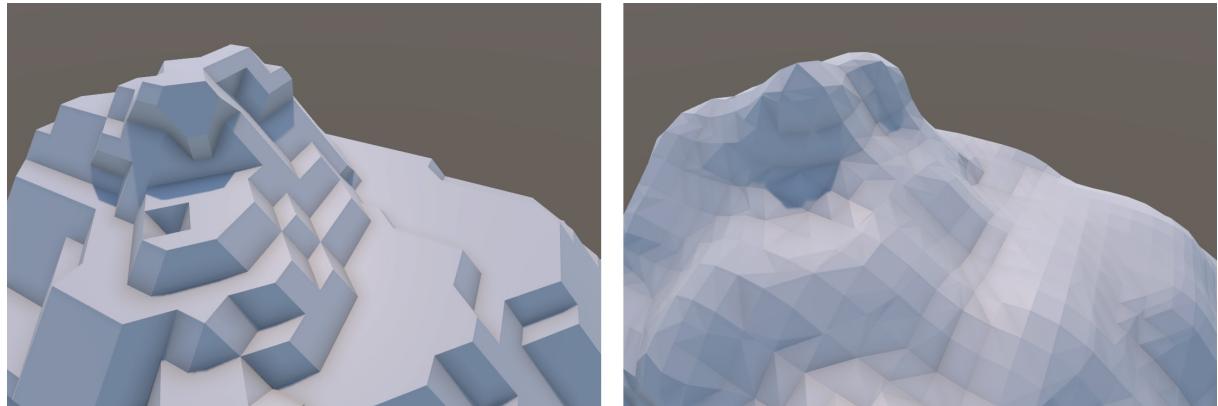
```

1 triangulationIndex = 0;
2 if (cornerValues[0] <= thresholdValue) triangulationIndex |= 1;
3 if (cornerValues[1] <= thresholdValue) triangulationIndex |= 2;
4 if (cornerValues[2] <= thresholdValue) triangulationIndex |= 4;
5 if (cornerValues[3] <= thresholdValue) triangulationIndex |= 8;
6 if (cornerValues[4] <= thresholdValue) triangulationIndex |= 16;
7 if (cornerValues[5] <= thresholdValue) triangulationIndex |= 32;
8 if (cornerValues[6] <= thresholdValue) triangulationIndex |= 64;
9 if (cornerValues[7] <= thresholdValue) triangulationIndex |= 128;
```

Ostatnim krokiem algorytmu jest wyznaczanie współrzędnych wierzchołków otrzymanych trójkątów. Te współrzędne można wyznaczyć przykładowo jako średnia arytmetyczna współrzędnych wierzchołków sześcianu wyznaczających krawędź, na której znajduje się dany wierzchołek trójkąta. Preferowaną metodą jest jednak interpolacja liniowa, czyli w tym przypadku wyznaczenie współrzędnych wierzchołków trójkąta w miejscu na przypisanej mu krawędzi sześcianu, w zależności od stosunku wartości na jej wierzchołkach. Jeśli wierzchołki sześcianu wyznaczające daną krawędź zostaną oznaczone jako  $P_1$  i  $P_2$ , a wartości pola skalarnego w tych wierzchołkach jako  $V_1$  i  $V_2$  to współrzędne wierzchołka trójkąta  $P$ , leżącego na tej krawędzi, można wyznaczyć następującym wzorem:

$$P = P_1 + (\text{thresholdValue} - V_1)(P_2 - P_1)/(V_2 - V_1)$$

Interpolacja liniowa jest metodą preferowaną, ponieważ oferuje większą dokładność i gładkość powierzchni, co jest pokazane na rysunku 7.



Rys. 7. Przykłady powierzchni terenu wygenerowanych przez algorytm Marching Cubes nie wykorzystując interpolacji liniowej (po lewej) i wykorzystując interpolację liniową (po prawej).

W całości, algorytm Marching Cubes można przedstawić w następujących krokach:

1. Podziel wejściowe pole skalarnie na sześciany. Każdy sześciyan to zbiór ośmiu wierzchołków o przypisanych współrzędnych i wartościach skalarnych.
2. Dla każdego sześciyanu:
  - (a) Wyznacz indeks triangulacji.
  - (b) Odczytaj dane o trójkątach z tablicy triangulacji.
  - (c) Wyznacz współrzędne wierzchołków otrzymanych trójkątów.
3. Zwróć listę wszystkich wygenerowanych trójkątów.

## 2.3 Wykorzystanie algorytmu w grach komputerowych

Wiele gier komputerowych wykorzystuje algorytm Marching Cubes lub jeden z jego wariantów do zaimplementowania proceduralnie generowanego terenu. Przykładem takiej gry jest wspominane wcześniej „Deep Rock Galactic” [16], gdzie wykorzystany algorytm pozwala na proceduralne generowanie ogromnych systemów jaskiń, w których toczy się rozgrywka. Wygenerowany w ten sposób teren może być ponadto niszczony i kształtowany przez graczy (Rys. 8). Ta funkcjonalność jest szczególnie prosta do osiągnięcia dzięki wykorzystanym przez algorytm polom skalarnym. Intuicyjna zmiana kształtu generowanej powierzchni może zostać uzyskana poprzez prostą modyfikację wartości tych pól, przykładowo podwyższenie wartości pola wewnętrz sfery o zadanym środku i promieniu.

Silniki do gier takie jak Unity i Unreal Engine nie posiadają wbudowanej implementacji algorytmu Marching Cubes, jednak w ich sklepach internetowych dostępne są narzędzia, które wykorzystują ten algorytm lub udostępniają jego implementację.



Rys. 8. Fragment terenu wygenerowanego w grze „Deep Rock Galactic”, niszczony przez gracza. Obrazek pochodzi z materiałów marketingowych udostępnionych przez wydawcę gry na platformie Steam.

# 3. Projekt badawczy

Na potrzeby badań prowadzonych w ramach pracy został stworzony projekt w silniku Unity (wersja edytora 2021.3.10f1). Wykorzystanie tego popularnego silnika pozwala na realistyczną ocenę możliwości zastosowanych metod w praktycznych zastosowaniach. Z tego samego powodu platformą docelową aplikacji są komputery z systemem operacyjnym Windows. Zastosowane języki programowania to C# (domyślny język Unity) i HLSL (język używany w shaderach).

Projekt umożliwia proceduralne generowanie fotorealistycznego terenu za pomocą algorytmu Marching Cubes z możliwością wyboru zastosowanych technik i parametrów przez użytkownika. Aplikacja pozwala również na pomiar i zapis czasów wykonywania poszczególnych części programu.

## 3.1 Ogólny proces generowania terenu

Czynności wykonywane w ramach generowania terenu wewnątrz aplikacji można podzielić na 5 zasadniczych faz:

1. Generowanie pola skalarnego.
2. Wykonanie algorytmu Marching Cubes.
3. Opcjonalne wygładzanie powierzchni.
4. Budowanie siatki.
5. Teksturowanie.

Poszczególne fazy mogą być wykonywane zarówno na CPU, jak i na GPU, w zależności od wybranej przez użytkownika techniki, co zostało opisane w późniejszej części pracy.

### 3.1.1 Generowanie pola skalarnego

Pole skalарne może zostać wygenerowane na wiele różnych sposobów. Zastosowaną w projekcie metodą jest proceduralne generowanie wartości pola z wykorzystaniem trójwymiarowego szumu Perlina i tzw. funkcji gęstości. Ta metoda jest inspirowana technikami opublikowanymi w 2007 roku przez R. Geissa [21].

Funkcja gęstości (ang. density function) to funkcja przyjmująca jako parametr pozycję punktu w trójwymiarowej przestrzeni, a zwracająca pojedynczą wartość skalarną. Oznacza to, że funkcja gęstości definiuje pewne pole skalarnie. Pojęcie gęstości można interpretować jako gęstość materii. Niska wartość gęstości w danym punkcie przestrzeni może reprezentować powietrze, a wysoka gęstość skałę. W praktyce możemy wyznaczyć konkretną wartość gęstości jako wartość graniczną pomiędzy powietrzem a terenem, a następnie interpretować każdy punkt o gęstości większej od tej wartości jako leżący wewnątrz terenu, a każdy punkt o gęstości mniejszej jako powietrze.

Pole skalarnie jest reprezentowane w programie jako trójwymiarowa tablica wartości w punktach przestrzeni rozlokowanych w węzłach regularnej siatki. Wartości pola skalarnego w tych punktach są wyznaczanie poprzez próbkowanie funkcji gęstości. Otrzymana tablica w uproszczony sposób reprezentuje pewien skończony wycinek pola skalarnego, w celu umożliwienia przeprowadzenia na nim algorytmu Marching Cubes.

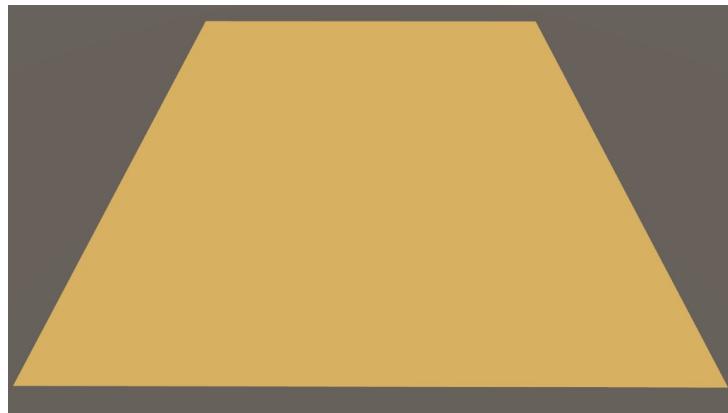
## Budowa funkcji gęstości

Osiągnięcie ciekawego, realistycznego terenu wymaga stworzenia ciekawej funkcji gęstości. W tym celu można wykorzystać różne wbudowane operacje matematyczne albo zdefiniować własne.

W projekcie jako podstawę generowanego terenu wykorzystano płaszczyznę. Jeśli dane wejściowe funkcji (pozycja punktu) zostaną oznaczone symbolami  $x, y, z$  (gdzie  $y$  to pozycja w osi pionowej), funkcja gęstości  $\rho$  wyznaczająca płaszczyznę wygląda następująco:

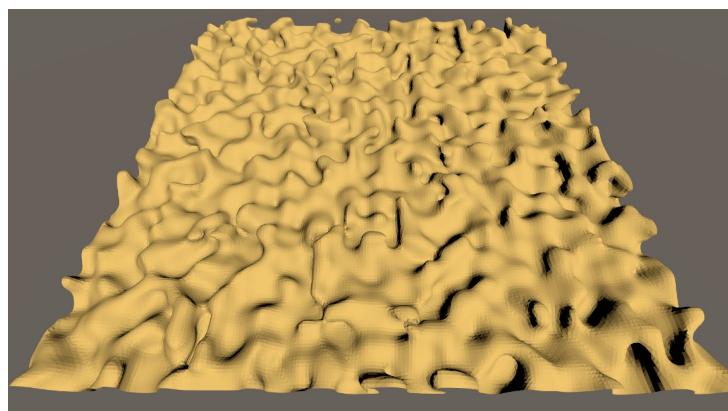
$$\rho(x, y, z) = -y$$

Powyzsza funkcja dzieli świat na dwa obszary, ten poniżej płaszczyzny  $y = 0$  (wartości dodatnie), który można interpretować jako „ziemia” i ten powyżej płaszczyzny (wartości ujemne), który można interpretować jako „powietrze”. Na rysunku 9 przedstawiono otrzymany w ten sposób teren.



Rys. 9. Płaski teren wygenerowany z wykorzystaniem podstawowej funkcji gęstości.

W celu urozmaicenia terenu, do stworzonej funkcji gęstości dodawane są wartości generowane za pomocą szumu Perlina. Szum Perlina został wykorzystany ze względu na swoje właściwości. W przeciwnieństwie do wartości losowych, wartości generowane za pomocą szumu Perlina zmieniają się płynnie w przestrzeni, tzn. punkty leżące w swoim bezpośrednim sąsiedztwie mają bardzo zbliżone do siebie wartości. Zastosowanie szumu Perlina pozwala uzyskać efekt organicznego „falowania” terenu (Rys. 10).

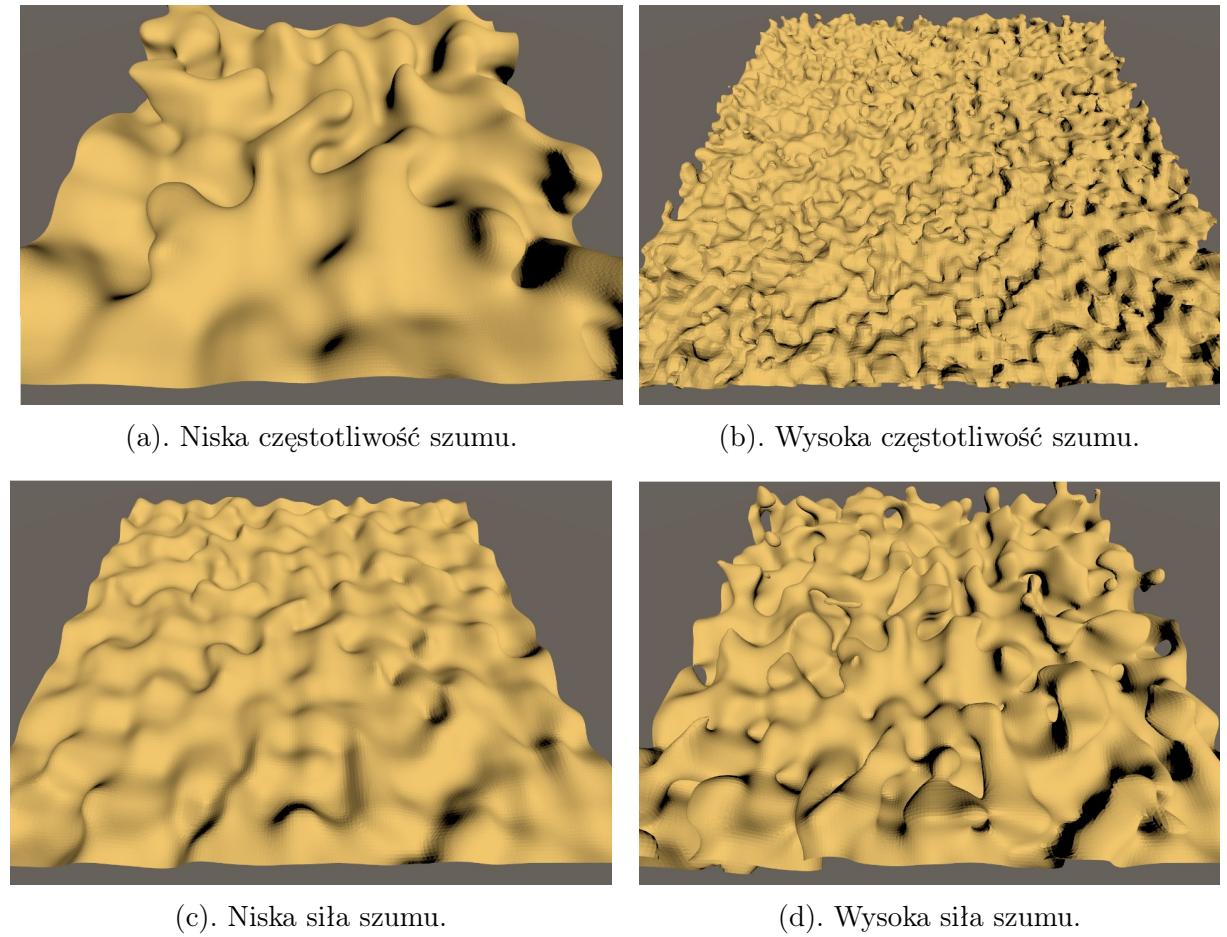


Rys. 10. Teren wygenerowany po dodaniu do funkcji gęstości szumu Perlina.

Warto zauważyć, że wartości generowane za pomocą szumu Perlin nie są losowe. W zależności od zastosowanej implementacji konkretne dane wejściowe (pozycja punktu), wygenerują taką samą wartość szumu. Aby uzyskać efekt losowości w projekcie, do wejściowej pozycji punktu dodawany jest losowy wektor (ten sam dla każdego punktu). Dzięki zastosowaniu tej techniki możliwe jest wygenerowanie różnych kształtów terenu o identycznych parametrach, poprzez modyfikację parametru ziarna (ang. seed) w programie.

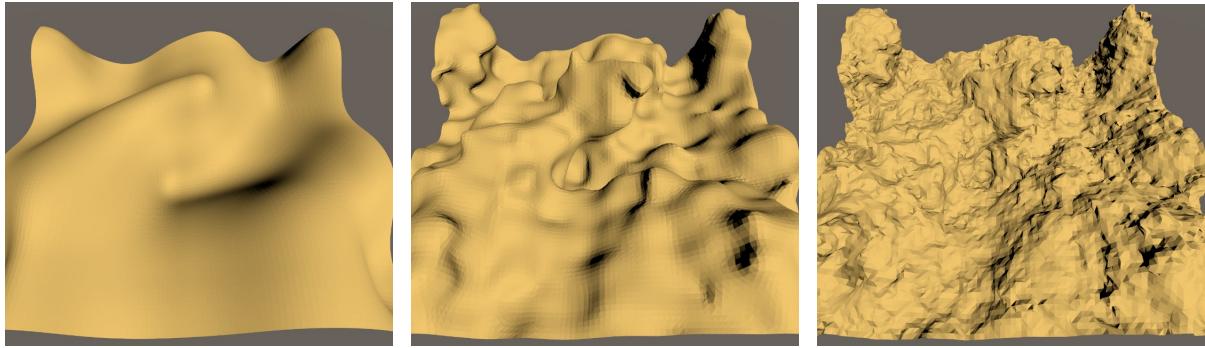
Wykorzystywane wartości szumu można lepiej kontrolować, dając dwa parametry: częstotliwość (ang. frequency) i siłę (ang. strength). Częstotliwość szumu to wielkość określająca, jak szybko wartości szumu zmieniają się w przestrzeni. Wysoka częstotliwość będzie skutkowała bardziej pofalowanym terenem, a niska częstotliwość terenem bardziej płaskim. Częstotliwość można kontrolować poprzez mnożenie wejściowego parametru szumu (pozycji punktu) przez parametr. Siła szumu to wartość określająca jak wysoka jest wartość dodawanego szumu. Można ją kontrolować, mnożąc otrzymaną wartość szumu przez parametr.

Wprowadzenie powyższych parametrów pozwala na intuicyjne kontrolowanie kształtu terenu. Na obrazku 11 zaprezentowano wpływ zastosowania różnych wartości tych parametrów.



Rys. 11. Przykłady czterech fragmentów terenu wygenerowanych z wykorzystaniem różnych wartości parametrów kontrolujących szum.

Aby teren był bardziej szczegółowy, można zastosować wiele nałożonych na siebie warstw szumu Perlina z rosnącą częstotliwością i malejącą siłą (Rys. 12). Zazwyczaj siła kolejnej warstwy jest dwa razy mniejsza od poprzedniej, a jej częstotliwość jest mniej więcej dwa razy większa. Korzystne jest, aby częstotliwość kolejnej warstwy nie była dokładanie dwa razy większa od poprzedniej, bo ta różnica pozwala na wprowadzenie dodatkowego zakłócenia do powtarzalnej struktury szumu.



(a). 1 warstwa szumu. (b). 3 warstwy szumu. (c). 9 warstw szumu.

Rys. 12. Przykłady trzech identycznych fragmentów terenu wygenerowanych z wykorzystaniem różnej liczby warstw szumu.

### Implementacja algorytmu generowania pola skalarnego

W uproszczeniu algorytm generowania wartości pola skalarnego dla zadanego punktu (woksela) został zaimplementowany w następujący sposób:

```

1 Vector3 voxelDensityInput = voxelPosition / terrainSize;
2 float density = -voxelDensityInput.y;
3 float density += GetPerlinNoiseValue(voxelDensityInput * noiseFrequency,
                                         noiseLayers, seed) * noiseStrength;
```

Oznaczenia:

**voxelPosition** Pozycja zadanego woksela, względem środka terenu.

**terrainSize** Długość krawędzi terenu. Jeśli długości krawędzi terenu są różne, wykorzystana jest długość najdłuższej z nich.

**voxelDensityInput** Wartość wejściowa funkcji gęstości, będąca pozycją woksela względem środka terenu, sprowadzona do przedziału [-1,1]

**density** Wynikowa wartość funkcji gęstości

**GetPerlinNoiseValue** Funkcja zwracająca wartość szumu Perlina na podstawie pozycji woksela, liczby warstw szumu i wartości ziarna.

**noiseFrequency** Częstotliwość szumu z przedziału [0.01, 100]

**noiseLayers** Liczba warstw szumu z przedziału [1, 10]

**seed** Wartość ziarna z przedziału [0, 999999]

**noiseStrength** siła szumu z przedziału [0.01, 1]

## Implementacja algorytmu szumu

Zastosowana w projekcie implementacja szumu Perlina korzysta z biblioteki FastNoise Lite [22]. Jest to zewnętrzna biblioteka kompatybilna z silnikiem Unity, zawierająca implementację różnych algorytmów generujących dwu- i trójwymiarowy szum. Istotnym atutem tej biblioteki jest to, że implementuje te same algorytmy w różnych językach, co pozwala na uzyskanie identycznego terenu z wykorzystaniem zarówno języka C#, jak i HLSL. Biblioteka FastNoise Lite posiada również wbudowany mechanizm wprowadzania losowości do generowanego szumu, poprzez możliwość zdefiniowania dowolnego ziarna (ang. seed).

Wykorzystanie szumu Perlina zamiast innych dostępnych algorytmów generowania szumu jest uzasadnione tym, że jest on jednym z najpopularniejszych metod wykorzystywanych do generowania terenu. Szum Perlina nie jest uważany za najbardziej wydajny obliczeniowo, jednak nie ma to istotnego wpływu na wyniki uzyskane w ramach projektu, ponieważ każda z badanych metod wykorzystuje go w identycznym zakresie, a różnice w wydajności poszczególnych algorytmów generowania szumu są nieznaczne (według analizy zawartej w dokumentacji biblioteki FastNoise Lite [22]).

### 3.1.2 Wykonanie algorytmu Marching Cubes

Do generowania wierzchołków siatki terenu na podstawie wartości pola skalarnego został zastosowany podstawowy wariant algorytmu Marching Cubes, w identycznej postaci co w oryginalnej publikacji [9]. Chociaż różne warianty algorytmu różnią się między sobą wydajnością obliczeniową i jakością generowanego terenu, to zasada ich działania jest bardzo podobna. Wszystkie wykorzystują pole skalarne, które dzielą na sześciiany lub czworościany, a następnie analizują je w celu znalezienia izopowierzchni w każdym z nich. Z tego powodu wszystkie te algorytmy przejawiają podobny potencjał do przeniesienia części lub całości obliczeń na procesor graficzny. Wyniki implementacji podstawowego wariantu algorytmu są więc z dużym prawdopodobieństwem uniwersalne dla wszystkich wariantów, pod kątem badania wydajności zastosowanych technik względem siebie.

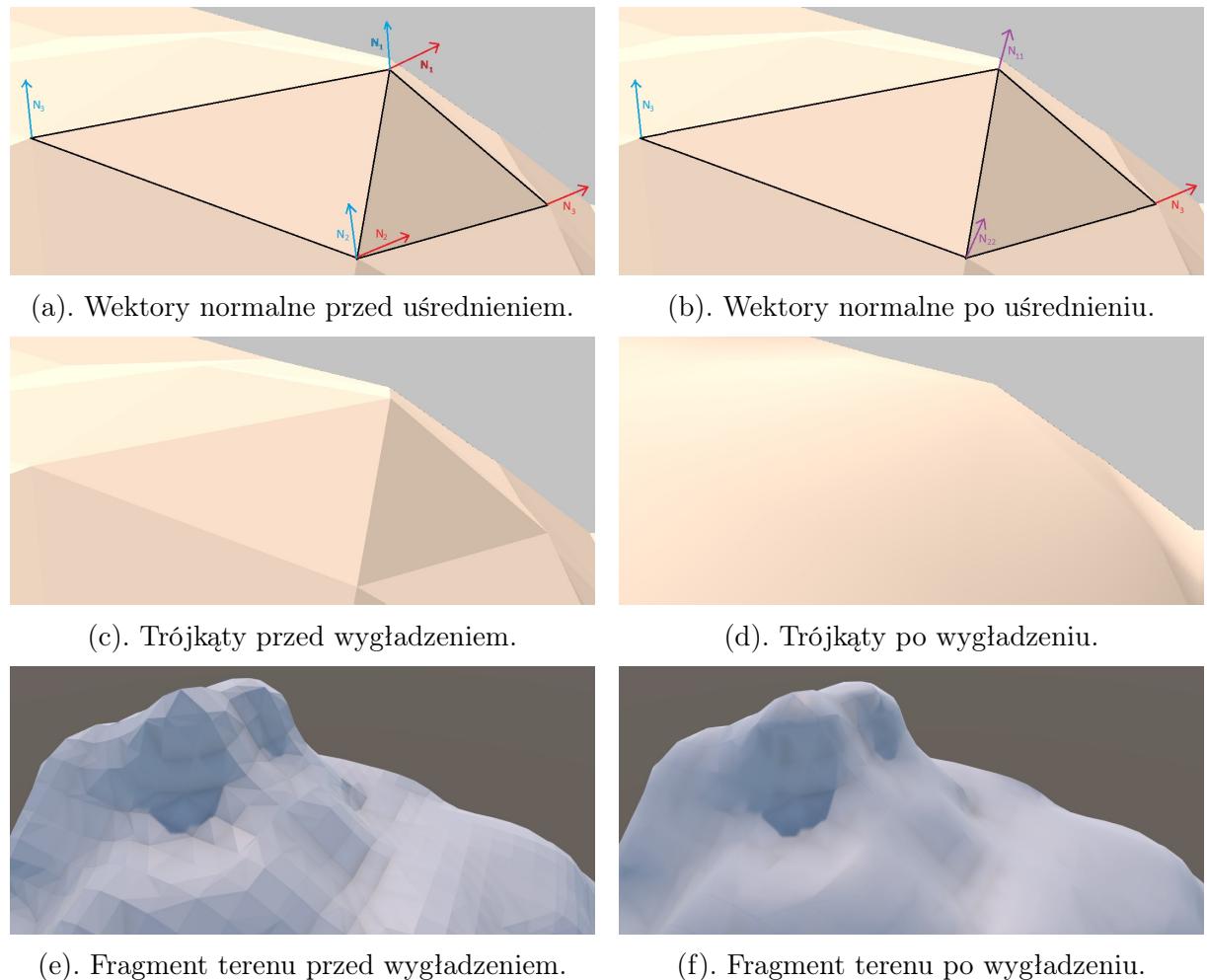
Aby wygenerować teren, niezbędne jest zdefiniowane długości krawędzi sześciianów wykorzystywanych przez algorytm Marching Cubes, a także ich liczbę. W stworzonej aplikacji te parametry są wyliczane na podstawie parametrów pośrednich, które są bardziej intuicyjne dla użytkownika. Te parametry to wielkość terenu w każdej z trzech osi układu współrzędnych oraz liczba sześciianów przypadająca na jedną jednostkę długości (w aplikacji zastosowano domyślną jednostkę Unity, niemającą odzwierciedlenia w prawdziwych jednostkach). Zwiększenie liczby sześciianów pozwala na osiągnięcie wyższego poziomu szczegółów, poprzez dokładniejsze odwzorowanie izopowierzchni pola skalarnego, kosztem zwiększenia czasu wykonywania poszczególnych fragmentów programu.

Wynikiem działania algorytmu Marching Cubes w stworzonej aplikacji jest jednowymiarowa tablica zawierające współrzędne wierzchołków wszystkich wygenerowanych trójkątów w odpowiedniej kolejności, tak aby wierzchołki tych samych trójkątów znajdowały się obok siebie. Na podstawie danych w tej tablicy generowana jest również tablica trójkątów oraz wektorów normalnych poszczególnych wierzchołków. Trójkąty są zdefiniowane jako zbiór indeksów trzech wierzchołków, z których są zbudowane. Trzy powstałe w ten sposób tablice mogą zostać wykorzystane do zbudowania siatki terenu.

### 3.1.3 Wygładzanie powierzchni

Wygładzanie jest opcjonalnym krokiem, który można wykonać po zastosowaniu algorytmu Marching Cubes. Sprowadza się ono do uśrednienia wektora normalnego każdego wierzchołka wygenerowanego terenu z wektorami normalnymi wszystkich jego instancji (czyli wierzchołkami o identycznych współrzędnych wchodzących w skład sąsiednich trójkątów). Wektory normalne są wykorzystywane przez silnik graficzny do wyliczania oświetlenia, dlatego uśrednienie ich pozwala uzyskać efekt wygładzenia powierzchni. Ten proces został w uproszczeniu zwizualizowany na rysunkach 13a i 13b, gdzie wektory normalne dwóch instancji tych samych wierzchołków (oznaczone czerwonymi i niebieskimi strzałkami) są uśredniane, tworząc wektory fioletowe. W wyniku działania tego procesu wygenerowany teren jest bardziej gładki, a poszczególne trójkąty są mniej widoczne (Rys. 13e i 13f).

Zazwyczaj proces wygładzania siatki odbywa się w silniku Unity automatycznie. Wyobraża to jednak tego, aby siatka była ciągła, tzn. żeby wierzchołki siatki faktycznie współdzieliły trójkąty. Teren generowany algorytmem Marching Cubes nie spełnia tego wymagania, ponieważ każdy trójkąt zbudowany jest w nim z osobnych wierzchołków, a wrażenie ciągłości jest jedynie wizualne. Budowanie siatki ciągłej uniemożliwiłoby równoległe przedstawiania sześciianów, dlatego wygładzanie siatki ręcznie jest metodą preferowaną.



Rys. 13. Uproszczona wizualizacja zasady działania i efektu wygładzania terenu przez uśrednianie wektorów normalnych wierzchołków.

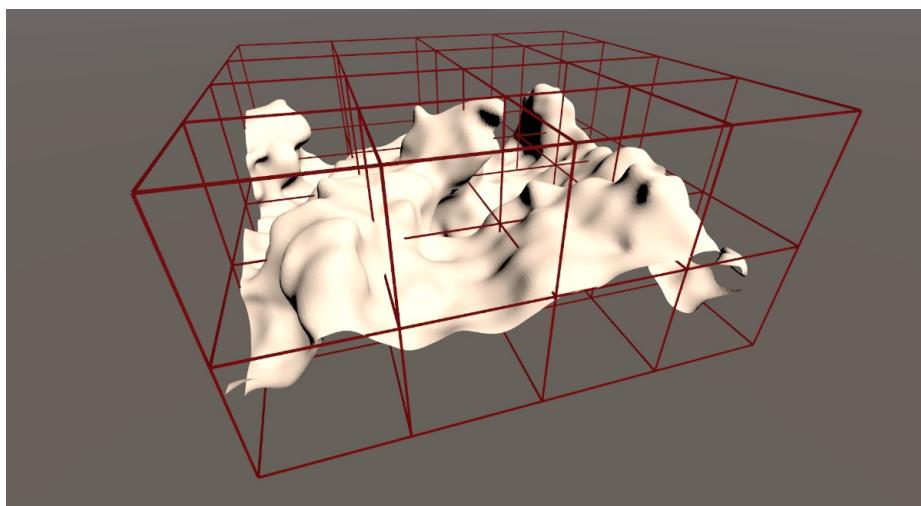
### 3.1.4 Budowanie siatki

Metody budowania siatki terenu są silnie zależne od wybranego przez użytkownika wariantu generacji i zostaną szczegółowo omówione w późniejszej części pracy, natomiast istnieją między nimi pewne cechy wspólne. Najważniejszą z nich jest limit liczby wierzchołków, z których może być zbudowana siatka dowolnego obiektu. W silniku Unity ten limit wynosi  $4\ 294\ 967\ 296$  ( $2^{32}$ ), ponieważ tablica wierzchołków indeksowana jest liczbą 32-bitową.

W celu wygenerowania terenu o większej liczbie wierzchołków niż limit narzucony przez silnik konieczne jest podzielenie go na fragmenty. Powszechnie w grach komputerowych wykorzystujących proceduralnie generowany teren, stosuje się podział wykorzystujący regularną dwu- lub trójwymiarową siatkę fragmentów (ang. chunks) w kształcie odpowiednio prostopadłościanów lub sześcianów. Pozwala to na intuicyjne rozmieszczenie gotowych fragmentów w przestrzeni tak, żeby do siebie pasowały oraz wygodne przechowywanie informacji o nich w odpowiedniego wymiaru tablicach. W aplikacji zastosowano podział na sześcienne fragmenty (Rys. 14), ponieważ to rozwiązanie jest bardziej uniwersalne (działa tak samo dobrze niezależnie od rozmiaru terenu w dowolnym kierunku układu współrzędnych).

Maksymalna wielkość fragmentu terenu wewnętrz aplikacji nie jest ograniczona wyłącznie limitem liczby wierzchołków siatki. Ten sam podział jest wykorzystywany do prowadzenia obliczeń i wykonywania algorytmów, co często wymaga alokowania większych struktur niż tablica wierzchołków. Najbardziej ograniczającą strukturą danych są bufore GPU, mogące przechowywać jedynie  $2^{31}$  bitów informacji łącznie. Przetworzenie pojedynczego sześcianu przez algorytm Marching Cubes wymaga alokacji struktur zdolnych przechować informację o pięciu trójkątach (najbardziej złożony przypadek triangulacji). Każdy wierzchołek wymaga 192 bitów pamięci (6 32-bitowych wartości zmiennoprzecinkowych opisujących jego położenie i wektor normalny). Z tego powodu, maksymalna liczba przeanalizowanych sześcianów, w przypadku metod stosujących bufore GPU, to jedynie  $745\ 654$  ( $2^{31}/(6 * 32 * 3 * 5)$ ), co przekłada się na 11 184 810 wierzchołków (zakładając najbardziej złożony przypadek Marching Cubes w każdym sześcianie).

Wielkość fragmentów może być dodatkowo kontrolowana przez użytkownika aplikacji, w celu znalezienia optymalnego rozmiaru dla zastosowanych parametrów programu.



Rys. 14. Wizualizacja podziału wygenerowanego terenu na fragmenty. Granice poszczególnych fragmentów oznaczone są bordowymi liniami.

### 3.1.5 Teksturowanie

Nakładanie na obiekt tekstur jest istotnym elementem stworzonej aplikacji, ponieważ wydajność zaimplementowanych metod może zależeć od złożoności shaderów wykorzystywanych do renderowania terenu. Pominiecie tego procesu mogłoby doprowadzić do wypaczenia wyników na korzyść algorytmów wykorzystujących w większym stopniu procesor graficzny.

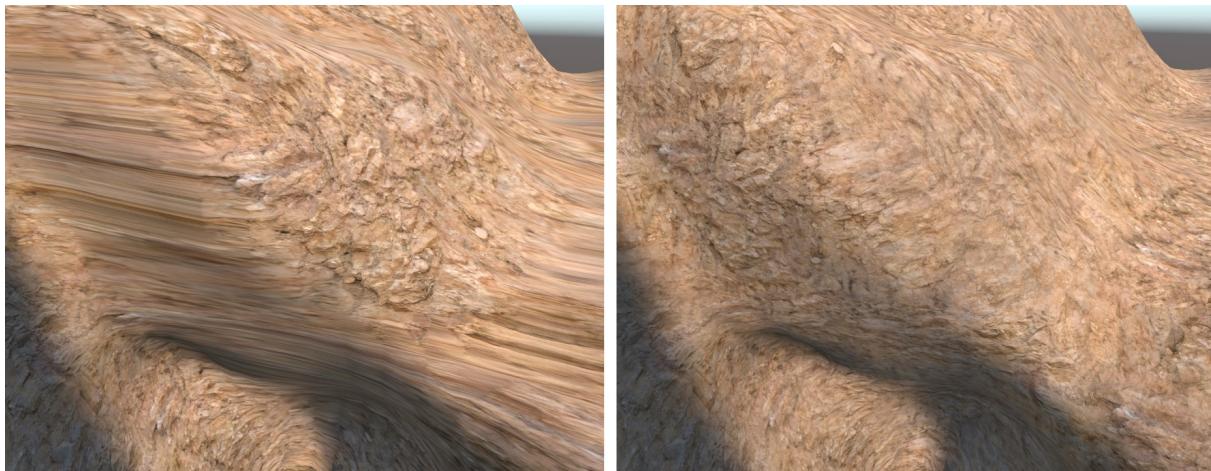
Zazwyczaj teksturowanie obiektów trójwymiarowych odbywa się przy pomocy procesu mapowania współrzędnych tektury (ang. UV mapping). Ten proces przyporządkowuje każdemu wierzchołkowi obiektu dwuwymiarowe współrzędne odpowiadającego mu punktu na teksturze. W przypadku terenu generowanego proceduralnie algorytmem Marching Cubes wyznaczenie takiego mapowania jest trudne, ze względu na arbitralność i nierównomierność położenia wierzchołków modelu względem siebie.

Problem mapowania współrzędnych tektury może zostać rozwiązany za pomocą techniki rzutowania planarnego (ang. planar projection). To podejście pozwala obejść problem arbitralności położenia wierzchołków, poprzez rzutowanie na nie tektury w zależności od ich globalnego położenia w przestrzeni, a nie położenia względem siebie. W przypadku teksturowania terenu rzutowanie można zrealizować poprzez ustawienie płaszczyzny tektury wzdłuż dwóch osi układu współrzędnych, a następnie mapowanie wierzchołków terenu na tekturę na podstawie ich współrzędnych na tych osiach. Na rysunku 15 została przedstawiona przykładowa tektura oraz fragment oteksturowanego nią terenu z wykorzystaniem techniki rzutowania planarnego.



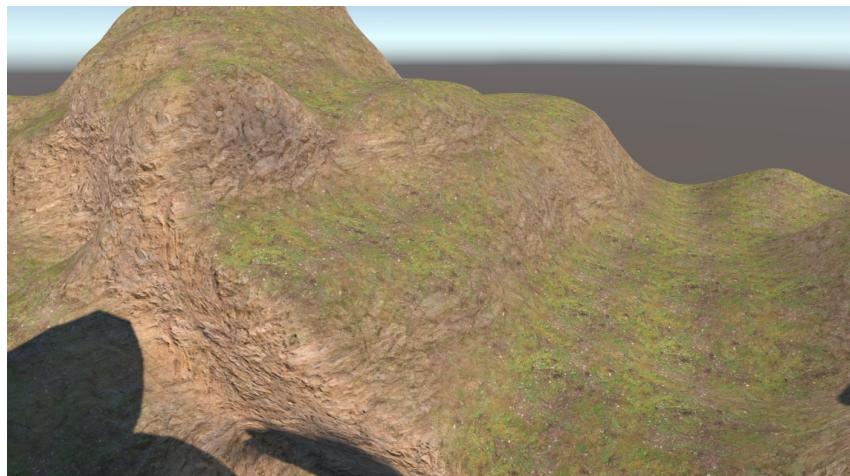
Rys. 15. Przykładowa tekstura skały (po lewej) i fragment oteksturowanego nią terenu (po prawej) z wykorzystaniem techniki rzutowania planarnego. Czarne strzałki oznaczają kierunek rzutowania.

Jak można zaobserwować na rysunku 15 wykorzystanie pojedynczego rzutowania planarnego nie jest wystarczające do poprawnego oteksturowania terenu, ponieważ rzutowanie daje pożądany efekt tylko z jednego kierunku, a ze wszystkich innych jest silnie zniekształcone. Ten problem może zostać rozwiązany przez zastosowanie rzutowania triplanarnego, polegającego na jednoczesnym zastosowaniu trzech rzutowań planarnych, jednego wzdłuż każdej osi układu współrzędnych, a następnie dobieranie siły oddziaływania poszczególnych rzutowań w zależności od kierunku wektora normalnego rozpatrywanego punktu terenu. Dzięki temu możliwe jest wybranie dla każdego punktu terenu takiego rzutowania, które spowoduje najmniejsze zniekształcenie oraz płynne przejście pomiędzy różnymi rzutowaniami (Rys. 16). Ta technika została szczegółowo opisana i spopularyzowana przez R. Geissa i M. Thompsona w 2007 roku [23].



Rys. 16. Fragment terenu z rysunku 15 oteksturowany techniką rzutowania planarnego (po lewej) i rzutowania triplanarnego (po prawej).

Dodatkową korzyścią wynikającą z zastosowania techniki rzutowania triplanarnego jest możliwość wyboru różnych tekstur dla każdego rzutowania składowego. W aplikacji wykorzystano teksturę trawy w rzutowaniu działającym w osi pionowej, a teksturę skały w rzutowaniach działających w osiach poziomych. Pozwala to odwzorować naturalną dla prawdziwego terenu właściwość preferencyjnego wyrastania roślinności na płaskich powierzchniach (Rys. 17).



Rys. 17. Fragment terenu z rysunku 15 oteksturowany techniką rzutowania triplanarnego, wykorzystując różne tekstury do poszczególnych rzutowań składowych.

### Tekstura wypukłości

W grafice komputerowej powszechnie stosuje się tekstury wypukłości (ang. normal maps) równolegle do zwykłych tekstur, w celu symulowania drobnych wypukłości powierzchni, zbyt małych by mogły być odwzorowane przez geometrię. W aplikacji również zastosowano tekstury wypukłości, stosując na nich tę samą technikę mapowania triplanarnego co w przypadku zwykłych tekstur. Proces mapowania przebiega podobnie w obu przypadkach, jedyną różnicą jest konieczność operowania w przestrzeni stycznej (ang. tangent space) w przypadku tekstur wypukłości, zamiast w przestrzeni obiektu (ang. object space), jak w przypadku zwykłych tekstur.

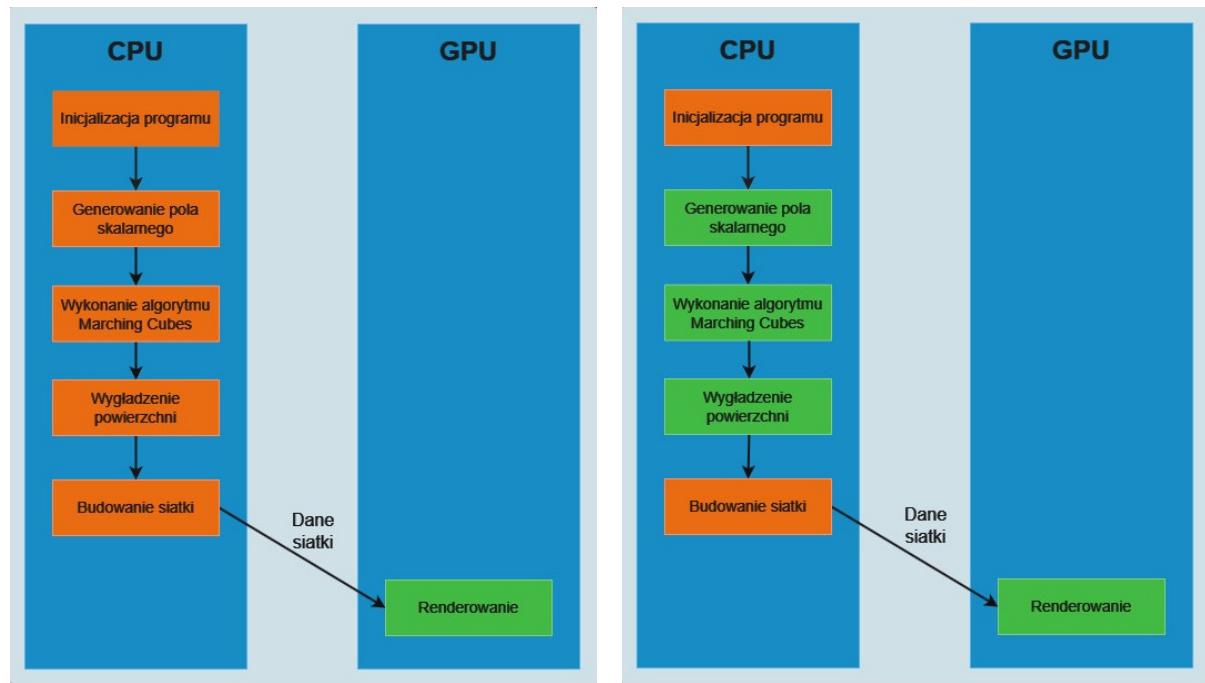
## 3.2 Implementacje generowania terenu

Przedstawione ogólne kroki generowania terenu można zaimplementować w silniku Unity na wiele sposobów. Ponieważ celem pracy jest porównanie wydajności implementacji na CPU i GPU, podstawowym rozróżnieniem zastosowanych metod jest to, w jaki sposób wykorzystują one te zasoby. Oczywiście każda metoda korzysta w jakimś stopniu z obu zasobów, więc są one rozróżniane ze względu na to, gdzie przeprowadzane są czasochłonne obliczenia i tworzenie obiektów terenu. Zaimplementowano łącznie 5 różnych metod, które można podzielić na 3 kategorie:

1. Metody wykorzystujące głównie CPU.
2. Metoda hybrydowa, korzystająca zarówno z CPU, jak i z GPU.
3. Metody wykorzystujące głównie GPU.

### 3.2.1 Metody CPU

Do kategorii metod wykorzystujące głównie CPU, należą dwie metody. Działają one na tej samej zasadzie, ale jedna z nich wykorzystuje wyłącznie główny wątek programu, a druga działa wielowątkowo tam, gdzie jest to możliwe. W dalszej części pracy te metody będą identyfikowane za pomocą skrótów *CPU* i *CPU\_MT* (MT od ang. multithreading). Zasadę działania metod przedstawiono na diagramach przepływu danych pomiędzy CPU i GPU (Rys. 18). Ta forma diagramu pozwala pokazać, z jakich zasobów korzystają poszczególne fragmenty programu i jak wygląda wymiana danych pomiędzy nimi, co jest szczególnie istotne podczas optymalizacji wydajności.



(a). Diagram przepływu metody *CPU*.

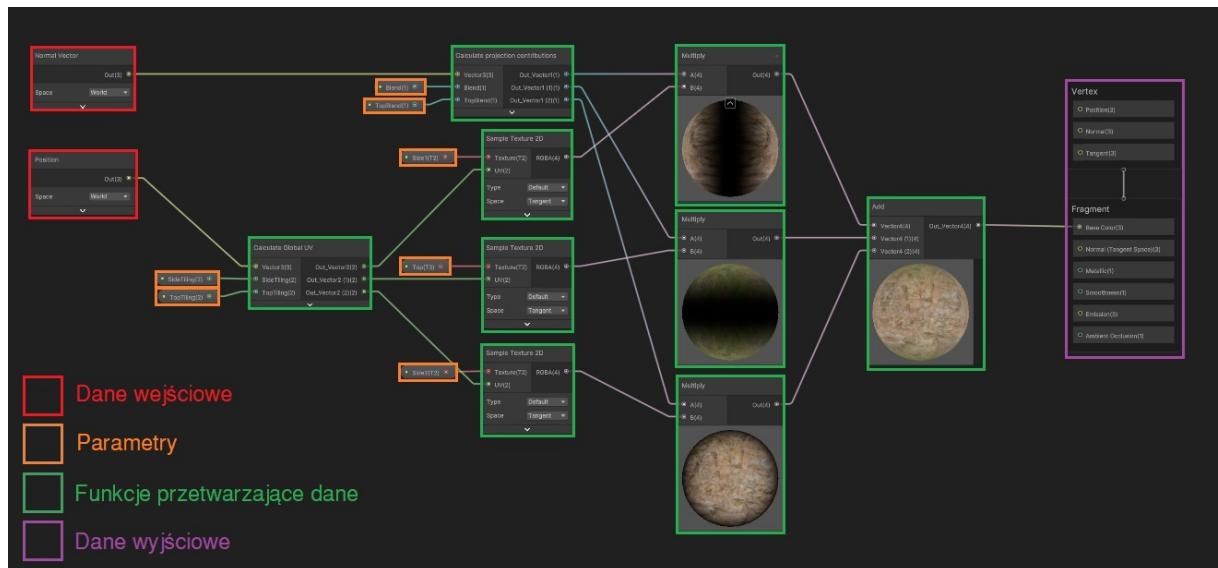
(b). Diagram przepływu metody *CPU\_MT*.

Rys. 18. Diagramy przepływu danych metod *CPU* i *CPU\_MT*. Kolorem zielonym oznaczono kroki wykonywane równolegle. Renderowanie zawiera wszystkie operacje związane z wyświetlaniem siatki na ekranie, włącznie z teksturowaniem.

## Zasada działania metod *CPU* i *CPU\_MT*

Zasada działania metody *CPU* jest bardzo prosta. Wykonuje ona wszystkie kroki opisane wcześniej w pracy, po kolej dla każdego fragmentu terenu. Niemal całość stworzonego kodu zawiera się w skryptach napisanych w języku C# i wykonywanych na CPU. Jedynie po zbudowaniu siatki wykonywane jest wbudowane w silniku Unity polecenie wyrenderowania jej. To polecenie automatycznie wysyła dane siatki do karty graficznej i renderuje go na ekranie z wykorzystaniem zadanego materiału. Zastosowany materiał wykorzystuje shader implementujący technikę rzutowania triplanarnego w celu oteksturowania terenu po stronie GPU.

Wykorzystany shader został stworzony za pomocą wbudowanego w Unity narzędzia Shader Graph, pozwalającego tworzyć shadery techniką programowania wizualnego. Na rysunku 19 pokazano uproszczoną strukturę zaimplementowanego shadera, obrazującą proces rzutowania triplanarnego. Shader zbudowany jest z bloczków połączonych liniami, oznaczającymi przepływ danych. Danymi wejściowymi tego shadera są pozycje wierzchołków i ich wektory normalne, a danymi wyjściowymi są odpowiadające im kolory. Każdy bloczek wykonujący obliczenia umożliwia wizualizację danych po ich przekształceniu.



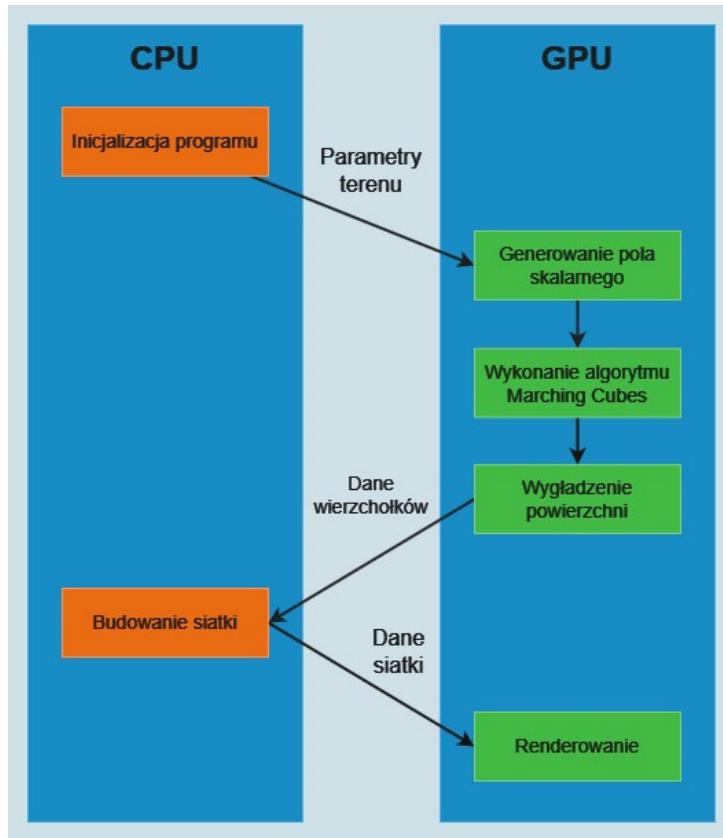
Rys. 19. Struktura shadera implementującego rzutowanie triplanarne w narzędziu Shader Graph. Bloczki pełniące różne funkcje zostały oznaczone różnymi kolorami.

Metoda *CPU\_MT* działa podobnie do metody *CPU*, ale wykorzystuje wiele wątków procesora. Silnik Unity nie wspiera przetwarzania wielowątkowego, więc można go stosować jedynie w przypadku procesów niekorzystających z obiektów silnika. Do tych procesów należą: generowanie pola skalarnego, wykonanie algorytmu Marching Cubes oraz wygładzenie powierzchni. Budowanie siatki wymaga skorzystania z metod Unity, dlatego nie może zostać przeprowadzone poza głównym wątkiem programu, co negatywnie odbija się na wydajności tego rozwiązania.

W celu podziału zadań na wątki wykorzystano istniejący już podział na fragmenty terenu. Każdy wątek przetwarza jeden fragment. Pozwala to na znaczne uproszczenie implementacji, ale sprawia, że wydajność obliczeniowa metody *CPU\_MT* może być zależna od liczby fragmentów. Ma to szczególne znaczenie w przypadku małego terenu, ponieważ nie ma wtedy wystarczającej liczby fragmentów, aby wykorzystać wszystkie wątki procesora. W przypadku większych terenów problem przestaje być odczuwalny.

### 3.2.2 Metoda hybrydowa

Metoda hybrydowa (zwana dalej *CPU\_GPU*) jest metodą wykorzystującą GPU w celu przeprowadzenia najbardziej czasochłonnych obliczeń równolegle, ale w dalszym ciągu korzystająca z CPU w celu budowania siatki terenu. Diagram przepływu tej metody został pokazany na rysunku 20.



Rys. 20. Diagram przepływu danych metody *CPU\_GPU*.

W celu przetwarzania danych na procesorze graficznym wykorzystano wbudowany w silnik Unity mechanizm shaderów obliczeniowych (ang. compute shader). Są to specjalne shadery, które pozwalają na wykonywanie dowolnych obliczeń na GPU, a nie tylko przetwarzanie tekstur i siatek obiektów, jak w przypadku shaderów graficznych. Dane, na których operują shadery obliczeniowe są przechowywane w specjalnych buforach, które pełnią funkcję tablic na GPU i mogą zostać zainicjalizowane przez CPU na tym samym etapie, na którym przekazywane są parametry terenu.

#### Struktury danych wykorzystywane przez GPU

Poszczególne etapy generowania terenu korzystają z różnych buforów, przechowujących różne struktury danych. Na etapie generowania pola skalarnego przetwarzane są struktury typu „Voxel” przechowujące pozycję i wartość poszczególnych punktów pola. Wykonanie algorytmu Marching Cubes tworzy struktury typu „TriangleTmp” przechowujące pozycje wierzchołków poszczególnych trójkątów i ich wspólny wektor normalny. Etap wygładzania tworzy finalne struktury typu „Triangle”, które również przechowują informacje o pozycji wierzchołków, ale każdy z nich posiada osobny wektor normalny.

Implementacja struktur danych wykorzystywanych przez GPU wygląda następująco:

```
1 struct Voxel
2 {
3     float3 position;           // pozycja punktu pola skalarnego
4     float value;              // wartość pola skalarnego
5 };
6
7 struct TriangleTmp
8 {
9     float3 v1, v2, v3;        // pozycje wierzchołków trojkata
10    float3 n;                // wektor normalny trojkata
11 };
12
13 struct Triangle
14 {
15     float3 v1, v2, v3;        // pozycje wierzchołków trojkata
16     float3 n1, n2, n3;       // wektory normalne wierzchołków
17 };
```

## Właściwości buforów wykorzystywanych przez GPU

Bufory wykorzystane w programie są inicjalizowane przez CPU z konkretną ilością przypisanej pamięci. Pozwalają one przechowywać dowolne struktury danych języka C#, dopóki ich typ posiada bezpośredni odpowiednik w języku HLSL (np. Vector3 = float3). W trakcie działania programu bufore przekazywane są w pamięci podręcznej GPU, aż nie zostaną ręcznie zwolnione odpowiednim poleceniem wykonanym na CPU. W projekcie wykorzystano dwa typy buforów oferowanych przez Unity:

**Default buffer** Domyślny bufor silnika Unity. Elementy indeksowane są analogicznie do zwykłych tablic wykorzystywanych przez programy działające na CPU.

**Append buffer** Bufor działający podobnie do bufora domyślnego, ale posiadający licznik elementów, które zostały do niego dodane. Umożliwia dodawanie elementów analogiczne do list wykorzystywanych przez programy działające na CPU, ale w przeciwieństwie do nich musi zostać zadeklarowany ze statycznym rozmiarem, tak samo, jak inne typy buforów.

## Zasada działania metody *CPU\_GPU*

Metoda *CPU\_GPU* wykonuje następujące kroki:

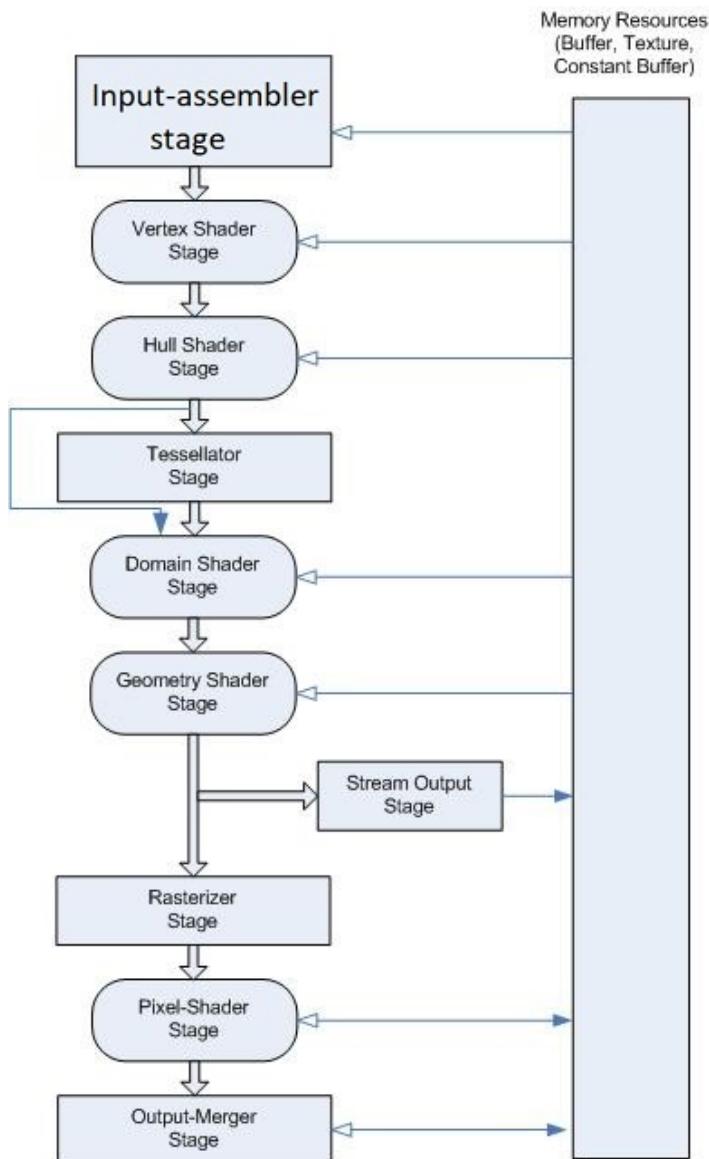
1. Na CPU zainicjalizuj program i stwórz niezbędne obiekty, a następnie prześlij do GPU parametry generowania terenu.
2. Prześlij do GPU szereg kolejnych instrukcji zlecających wykonanie kolejnych etapów generowania terenu oraz zarządzających wykorzystaniem buforów. Do obliczeń wykorzystywane są bufore typu *default*.
3. Oczekuj na wykonanie wszystkich instrukcji przez GPU, a następnie pobierz gotowe dane wierzchołków na CPU. Dane wierzchołków są zapisane w buforze typu *append*, ponieważ ich liczba nie jest stała, tylko zależna od generowanego terenu.
4. Z otrzymanych wierzchołków zbuduj siatkę terenu i wyrenderuj ją w sposób analogiczny do metod *CPU* i *CPU\_MT*.

### 3.2.3 Metody GPU

Metody GPU wykorzystują możliwości shaderów graficznych w szerszym zakresie niż metoda hybrydowa, w celu minimalizacji wymiany danych z CPU. Do zrozumienia zasady działania tych metod, konieczne jest zrozumienie zasady działania shaderów graficznych.

#### Zasada działania shaderów graficznych

Shadery graficzne są fragmentami szerszego procesu zwanego potokiem renderowania (ang. graphics pipeline). Podstawowym celem tego procesu jest przetransformowanie zadanej trójwymiarowej sceny w jej dwuwymiarową reprezentację na ekranie. Shadery graficzne realizują kolejne etapy tej transformacji. Na rysunku 21 pokazano jak wygląda przebieg takiej transformacji w wykorzystanym w projekcie potokiem renderowania *Direct3D 11*. Warto zauważyć, że cały ten proces jest przeprowadzany ponownie w każdej kolejnej klatce programu tzn. za każdym razem, gdy obraz na ekranie jest uaktualniany.



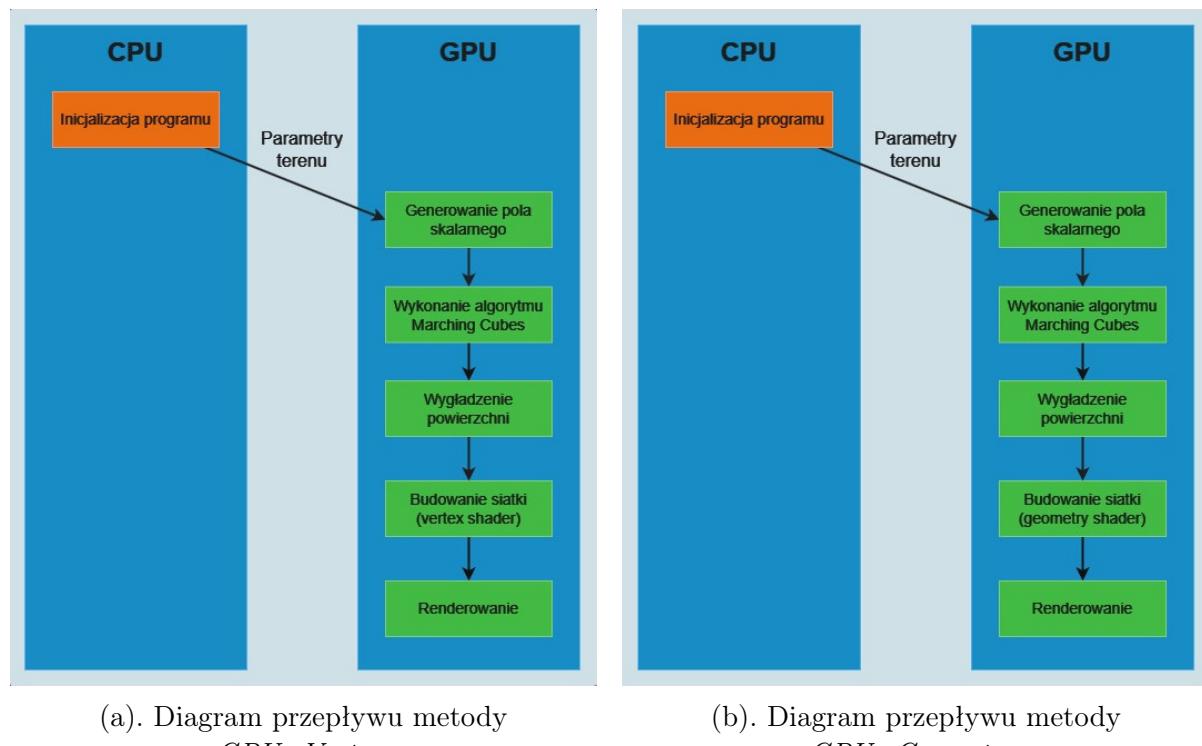
Rys. 21. Diagram przepływu danych w potoku renderowania *Direct3D 11*, przedstawiony w dokumentacji Microsoft [24].

Każdy z shaderów graficznych w potoku renderowania *Direct3D 11* można dostosować do potrzeb programu wykorzystując język HLSL. W projekcie zostały w ten sposób wykorzystane dwa shadery:

**Vertex shader** Shader wykonujący operacje na wierzchołkach obiektu.

**Geometry shader** Opcjonalny shader (może zostać pominięty) wykonujący operacje na prymitywnych składowych obiektu (np. na trójkątach). Umożliwia tworzenie nowych prymitywnych składowych oraz pomijanie istniejących.

Oba z wymienionych powyżej shaderów mogą zostać wykorzystane do zbudowania siatki terenu z wykorzystaniem GPU. W projekcie zostały zaimplementowane dwie takie metody, nazwane odpowiednio *GPU\_Vertex* i *GPU\_Geometry*. Na rysunku 22 pokazano ich diagramy przepływu danych.



Rys. 22. Diagramy przepływu danych metod *GPU\_Vertex* i *GPU\_Geometry*.

## Implementacja shaderów

Implementacja shaderów wykorzystywanych w metodach GPU jest wysoce utrudniona przez fakt, że muszą one bezpośrednio korzystać z arbitralnie zdefiniowanych buforów, w których znajdują się dane wierzchołków wygenerowanego terenu. Taka czynność nie jest wspierana ani przez narzędzie Shader Graph, ani przed istniejące shadery silnika Unity i wymaga stworzenia własnych shaderów. Aby zachować identyczny wygląd terenu z tym generowanym innymi metodami, zastosowano rozwijanie polegające na ekstrakcji i modyfikacji kodu źródłowego shaderów wykorzystanych w poprzednich metodach.

## Zasada działania metod *GPU\_Vertex* i *GPU\_Geometry*

W obu metodach GPU generowanie danych o wierzchołkach terenu wygląda tak samo, jak w metodzie *CPU\_GPU*. Jedynym krokiem, którym różnią się te trzy metody, jest proces budowania siatki. Zamiast pobierać dane wierzchołków na CPU i tam budować siatkę, metody GPU tworzą siatkę w odpowiednich shaderach, wykorzystując bezpośrednio bufor przechowujący dane wierzchołków, już istniejący na GPU.

Zarówno vertex, jak i geometry shader operują na wierzchołkach lub trójkątach pewnego wcześniej istniejącym modelu. Z tego powodu konieczne jest dostarczenie im takiego modelu, który może zostać przekształcony w oczekiwany teren, z wykorzystaniem informacji zawartych w wynikowym buforze wierzchołków. Te generyczne modele muszą zostać stworzone bez wiedzy o liczbie i pozycji wierzchołków finalnego terenu, dlatego są zbudowane z maksymalnej możliwej liczby wierzchołków potrzebnych do reprezentacji dowolnego terenu. Wielkość tą można wyznaczyć, znając liczbę sześciianów wykorzystanych do generowania terenu (a dokładniej do pojedynczego fragmentu) i wiedząc, że każdy sześciian może zawierać maksymalnie 5 trójkątów (jest to najbardziej złożony przypadek przewidziany przez algorytm Marching Cubes).

W metodzie *GPU\_Vertex* maksymalna liczba potrzebnych wierzchołków to po prostu liczba sześciianów wykorzystywanych przez każdy fragment terenu, pomnożona przez 15 (5 trójkątów po 3 wierzchołki). Każde trzy kolejne wierzchołki tworzą jeden trójkąt. Na tej podstawie tworzona jest po stronie CPU generyczna siatka, identyczna dla każdego fragmentu terenu i przekazywana do GPU w trakcie inicjalizacji programu. Po przeprowadzeniu operacji generowania wartości terenu vertex shader poszczególnych fragmentów terenu ustawia ich wierzchołki w odpowiednich pozycjach, odczytując ich dane z bufora.

W metodzie *GPU\_Geometry* trójkąty są tworzone dopiero w geometry shaderze. Każdy wierzchołek początkowej siatki jest zamieniany w jeden trójkąt, dlatego maksymalna liczba potrzebnych wierzchołków jest równa maksymalnej liczbie możliwych trójkątów. Na etapie vertex shadera odczytywane są parametry wierzchołków i przekazywane dalej. Następnie na ich podstawie geometry shader tworzy trójkąty finalnej siatki.

## Przycinanie rozmiaru generycznych siatek i buforów

Niepożądaną cechą algorytmów GPU jest fakt, że wykorzystane do nich wstępne siatki mają znacznie więcej wierzchołków, niż jest to niezbędne do reprezentowania terenu. Dzieje się tak, ponieważ sześciiany często są puste, a założony najgorszy możliwy przypadek pięciu trójkątów w jednym sześciianie zachodzi rzadko. Wykorzystanie większej liczby wierzchołków niż jest to niezbędne, ma negatywny wpływ na wydajność programu, ponieważ wszystkie wierzchołki są przetwarzane przez vertex i potencjalnie geometry shader w każdej klatce, niezależnie czy są wykorzystane w finalnej siatce terenu. Dodatkowo zwiększa się wielkość buforów przechowywanych w pamięci podrzecznej GPU. Z tego powodu zaimplementowano dwie metody ograniczania liczby wierzchołków generycznych siatek wejściowych:

1. Arbitralne ograniczenie liczby maksymalnej liczby wierzchołków siatek generycznych przez użytkownika. Wymaga ręcznego testowania, w celu znalezienia wartości odpowiedniej dla danego terenu.
2. Odczytywanie liczby wygenerowanych wierzchołków przez CPU, po ich wyznaczaniu przez GPU. Pozwala precyjnie określić potrzebny rozmiar siatki i buforów, ale spowalnia wykonanie programu.

## Zarządzanie pamięcią GPU

Bufory wykorzystywane przez GPU są przechowywane w dedykowanej pamięci podręcznej. W momencie, gdy dany bufor nie jest już potrzebny, może on zostać zwolniony poprzez wysłanie odpowiedniego polecenia z CPU. W przypadku metody *CPU\_GPU*, po pobraniu danych wierzchołków, wszystkie bufory mogą zostać zwolnione. W przypadku metod *GPU\_Vertex* i *GPU\_Geometry* nie jest to możliwe, ponieważ bufor przechowujący dane wierzchołków jest przez cały czas wykorzystywany (ponownie w każdej klatce programu). Im większy teren został wygenerowany, tym więcej pamięci GPU musi zostać przeznaczone na przechowywanie buforów. Wysokie zużycie pamięci może powodować znaczne obniżenie wydajności aplikacji, a w ekstremalnym przypadku jej zatrzymanie. Wykorzystanie pamięci GPU jest istotnym ograniczeniem podczas korzystania z metod *GPU\_Vertex* i *GPU\_Geometry*.

### 3.2.4 Modyfikacje metod

W aplikacji zaimplementowano trzy dodatkowe opcje pozwalające użytkownikowi zmienić niektóre aspekty działania metod generowania terenu. Te opcje to:

**Użyj uproszczonych shaderów** Ta opcja pozwala użyć prostszego wariantu shaderów dla każdej z metod. Zaimplementowany prosty shader wykonuje tylko jedną iterację (ang. pass) operacji na siatce modelu w przeciwieństwie do siedemnastu iteracji wykonywanych przez shader stworzony przy pomocy narzędzia Shader Graph. Wygenerowany w efekcie teren nie jest oteksturowany i posiada tylko podstawowe cieniowanie, ale za to oszczędza zasoby aplikacji, co może mieć znaczenie przy porównaniu wydajności poszczególnych metod.

**Wykorzystaj renderowanie pośrednie** Unity oferuje dwie różne metody służące do renderowania siatki o zadanych wierzchołkach. Metodą używaną domyślnie w aplikacji jest *Graphics.RenderMesh*, przyjmująca jako argument gotową siatkę obiektu. Alternatywna metoda renderowania to *Graphics.RenderPrimitivesIndirect*, która zamiast korzystać z gotowej siatki obiektu, buduje siatkę na GPU, wykorzystując proces tzw. *GPU instancing*. Największą zaletą tej metody jest fakt, że wielkość generowanej siatki jest odczytywana z GPU (na podstawie rozmiaru bufora z danymi), zamiast być podawana przez CPU, co potencjalnie może rozwiązać problem zbyt dużej liczby wierzchołków siatek generycznych, bez korzystania z operacji przycinania.

**Unikaj buforów typu *append*** Domyślnie aplikacja zapisuje wygenerowane na GPU wartości wierzchołków terenu w buforze typu *append*. Wykorzystanie tego typu bufora jest wygodne w implementacji, ale może ciągnąć ze sobą pewien dodatkowy koszt obliczeniowy, związany z koniecznością synchronizacji wątków procesora graficznego. Z tego powodu wykorzystując metody *GPU\_Vertex* i *GPU\_Geometry* istnieje możliwość zastosowania standardowego bufora zamiast bufora typu *append*. Ta opcja jest niedostępna dla metody *CPU\_GPU*, ponieważ korzysta ona z możliwości przeczytania liczby elementów w buforze typu *append* przy pobieraniu ich z GPU, co nie jest możliwe przy zastosowaniu bufora standardowego.

### 3.3 Metody pomiaru wydajności

W trakcie i po wygenerowaniu terenu aplikacja wykonuje pomiary wydajności. Mierzonymi wartościami są:

- czas wykonania poszczególnych etapów generowania terenu oraz całości procesu
- liczba klatek rysowanych na sekundę przez aplikację
- ilość pamięci GPU aktywnie wykorzystywanej przez bufory

#### Pomiar czasów wykonania

Czasy wykonania są mierzone z dokładnością do jednej milisekundy na podstawie czasu systemowego, jaki upłynął pomiędzy rozpoczęciem a zakończeniem poszczególnych faz generowania terenu z perspektywy CPU. Czas wykorzystany na ewentualne przycinanie rozmiaru generycznych siatek i buforów w metodach *GPU\_Vertex* i *GPU\_Geometry* jest również mierzony osobno.

Dużym utrudnieniem spowodowanym przez wykorzystanie GPU jest fakt, że CPU nie czeka na zakończenie obliczeń wykonywanych na GPU po zleceniu ich, a wymuszenie tego nie jest możliwe bez znacznego obniżenia wydajności. Z tego powodu część pomiarów nie odpowiada rzeczywistości, ponieważ określa ją jedynie, ile czasu zajęło CPU wysłanie instrukcji do GPU, a nie ile zajęło wykonanie tych instrukcji. W przypadku metod *GPU\_Vertex* i *GPU\_Geometry* sprawia to, że zmierzony przez CPU czas generowania terenu jest prawie zawsze równy 0 milisekund, chociaż mija znacznie więcej czasu, zanim teren zostanie wyświetlony na ekranie. Z tego powodu dodano pomiar tzw. czasu renderowania, który mierzy czas rysowania pewnej stałej liczby klatek po zakończeniu generacji terenu. Na podstawie testowych pomiarów wykonanych dla różnej liczby klatek, stwierdzono, że zmierzenie czasu rysowania 5 kolejnych klatek pozwala otrzymać adekwatny całkowity czas generowania terenu. Ten pomiar jest wykonywany dla każdej metody tak samo, aby zapobiec zniekształceniu wyników poszczególnych metod względem siebie.

Szczególnym traktowaniem jest również objęta metoda *CPU\_MT*. Ponieważ obliczenia są w niej wykonywane asynchronicznie w różnych wątkach, nie jest możliwy jawny pomiar czasu wykonania poszczególnych faz. Z tego powodu, każdy wątek mierzy czas wykonania poszczególnych operacji generowania terenu osobno, a następnie wyciągana jest z nich średnia arytmetyczna. Jedynie czas renderowania i całkowity czas generacji są mierzone w tym przypadku bezpośrednio.

#### Pomiar liczby klatek na sekundę

Średnia liczba klatek rysowanych na sekundę jest wyznaczana na podstawie liczby klatek narysowanych przez zadany interwał czasowy.

#### Pomiar wykorzystanej pamięci GPU

Wykorzystanie pamięci GPU jest sumą rozmiarów (w bajtach) wszystkich buforów GPU, które nie mogą zostać zwolnione od razu po zakończeniu generacji terenu, bo korzystają z nich shadery graficzne.

### 3.3.1 Automatyczny test wydajności

Aplikacja umożliwia wykonanie automatycznego testu wydajności generowania terenu i zapisanie wyników w formacie csv. Dla użytkownika dostępne są następujące parametry:

- limit sześciianów na jednostkę odległości
- granica krótkiego pomiaru (s)
- liczba krótkich pomiarów (krótszych niż *granica krótkiego pomiaru*)
- liczba długich pomiarów (dłuższych niż *granica krótkiego pomiaru*)
- limit czasu pomiaru (s)
- czas badania liczby klatek na sekundę (s)

Test wykonuje serie identycznych pomiarów i wyciąga z nich średnie arytmetyczne, w celu zminimalizowania zakłóceń powodowych działaniem urządzenia. Krótkie pomiary są bardziej podatne na zakłócenia i są powtarzane więcej razy. Przebieg automatycznego testu wygląda następująco:

1. Zapisz parametry generowania terenu i automatycznego testu.
2. Wygeneruj teren pierwszą dostępną metodą, ustawiając liczbę sześciianów na jednostkę odległości na 1 i zapisz czas generowania poszczególnych etapów.
3. Jeśli całkowity czas generowania jest mniejszy niż *granica krótkiego pomiaru*, wykonaj krok 2 tyle razy, ile wynosi *liczba krótkich pomiarów* i wylicz średnią arytmetyczną wyników. W przeciwnym razie wykonaj krok 2 tyle razy, ile wynosi *liczba długich pomiarów*. Przy wyznaczaniu średniej arytmetycznej wyników zignoruj wynik pierwszego pomiaru danej metody, ponieważ może on zawierać jednorazowe operacje dodatkowe, takie jak podmiana materiałów wykorzystywanych przez teren.
4. Wykonaj pomiar średniej liczby klatek na sekundę przez okres równy *czasowi badania liczby klatek na sekundę* i zapisz wynik. Zapisz również ilość wykorzystanej pamięci GPU.
5. Jeśli został osiągnięty *limit sześciianów na jednostkę odległości* lub *limit czasu pomiaru* przejdź do kolejnego kroku. W przeciwnym razie wróć do kroku 2, ale zwiększając liczbę sześciianów na jednostkę odległości o 1.
6. Jeśli zbadano wszystkie metody, zakończ test i zapisz wyniki do pliku csv. W przeciwnym razie wróć do kroku 2 i zastosuj kolejną dostępną metodę.

Parametry generowania terenu, które nie są kontrolowane przez automatyczny test, są stałe i mają taką wartość, jaką nadał im użytkownik przed uruchomieniem testu.

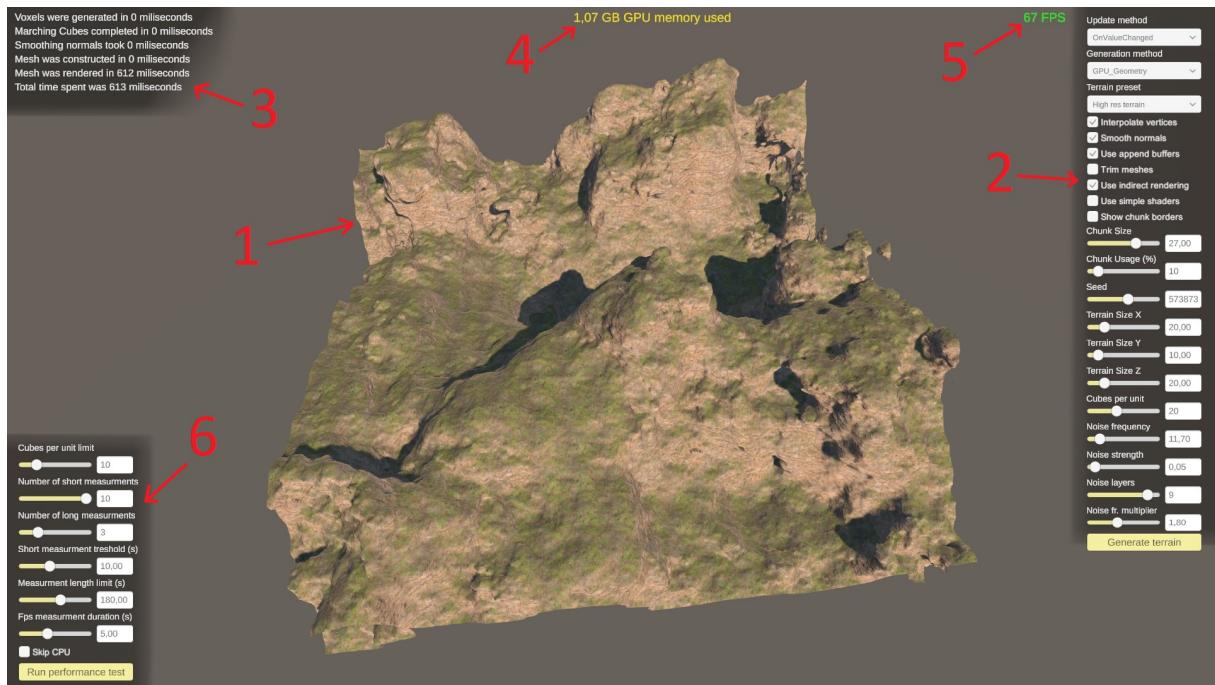
### Wyniki testu wydajności

Wynikami testu są pomiary średniego czasu wykonywania poszczególnych kroków generowania terenu dla wszystkich metod ze wzrastającą szczegółowością siatki. Oprócz średniej zapisane są również najniższy i najwyższy czas wykonania każdego etapu. Każda zastosowana konfiguracja generacji zawiera również pomiar średniej liczby klatek na sekundę rysowanych przez aplikację oraz ilość pamięci GPU wykorzystanej przez bufore. Wyniki testu zapisywane są w formie tabel, w celu ułatwienia ich analizy.

## 3.4 Finalny produkt

Po przeprowadzeniu wszystkich kroków generowania terenu według zdefiniowanych parametrów teren jest wyświetlany na ekranie, a użytkownik może oglądać go z różnych stron, przemieszczając się dzięki zaimplementowanemu kontrolerowi kamery. Kontroler umożliwia swobodny ruch i rotację kamery z wykorzystaniem myszy i klawiszy AWSD na klawiaturze. Na rysunku 23 przedstawiono okno stworzonej aplikacji z interfejsem użytkownika. Na ten interfejs składają się następujące elementy, odpowiednio ponumerowane na rysunku:

1. Wygenerowany teren.
2. Parametry generowania terenu.
3. Czas wykonania ostatniej generacji.
4. Licznik pamięci wykorzystanej przez GPU.
5. Licznik klatek na sekundę.
6. Parametry automatycznego testu.



Rys. 23. Okno aplikacji z interfejsem użytkownika po wygenerowaniu terenu.

Maksymalna wielkość i szczegółowość wygenerowanego terenu są ograniczone jedynie wydajnością sprzętową i pamięciową komputera użytkownika. Teren wygenerowany na rysunku 23 został zbudowany z około 2,8 milionów wierzchołków i powstał w wyniku analizy 32 milionów sześciianów.

# 4. Wyniki pomiarów

## 4.1 Parametry sprzętowe urządzeń

Pomiary wydajności poszczególnych metod generowania terenu wykonano na dwóch różnych urządzeniach w celu dywersyfikacji wyników. Pierwszym z nich jest komputer stacjonarny (w dalszej części pracy określany jako „komputer”) o następującej specyfikacji:

- system operacyjny: **Windows 10**
- procesor CPU: **Intel(R) Core(TM) i9-10850K**
  - częstotliwość taktowania rdzenia: **3.60 GHz**
  - liczba rdzeni: **10**
- procesor GPU: **NVIDIA GeForce RTX 3080**
  - Sterownik: **GeForce Game Ready Driver 536.23**
  - częstotliwość taktowania rdzenia: **1800 MHz**
  - liczba rdzeni (CUDA): **8704**
  - pamięć dedykowana: **10 GB**
  - pamięć współdzielona: **16 GB**
- pamięć RAM: **32 GB**

Drugim z wykorzystanych urządzeń jest podpięty do zasilania laptop (w dalszej części pracy określany jako „laptop”) o następującej specyfikacji:

- system operacyjny: **Windows 11**
- procesor CPU: **AMD Ryzen 5 5600H with Radeon Graphics**
  - częstotliwość taktowania rdzenia: **3.30 GHz**
  - liczba rdzeni: **6**
- procesor GPU: **NVIDIA GeForce RTX 3050 Ti Laptop GPU**
  - Sterownik: **GeForce Game Ready Driver 512.36**
  - częstotliwość taktowania rdzenia: **1057 - 1740 MHz**
  - liczba rdzeni (CUDA): **2048**
  - pamięć dedykowana: **4 GB**
  - pamięć współdzielona: **13,9 GB**
- pamięć RAM: **32 GB**

W czasie wykonywania pomiarów program ich dokonujący był jedyną aplikacją uruchomioną na urządzeniach, w celu minimalizacji zakłóceń spowodowanych nierównomiernym przydziałem zasobów. Wszystkie pomiary są wykonywane przy pomocy zbudowanych wersji aplikacji, poza edytorem Unity.

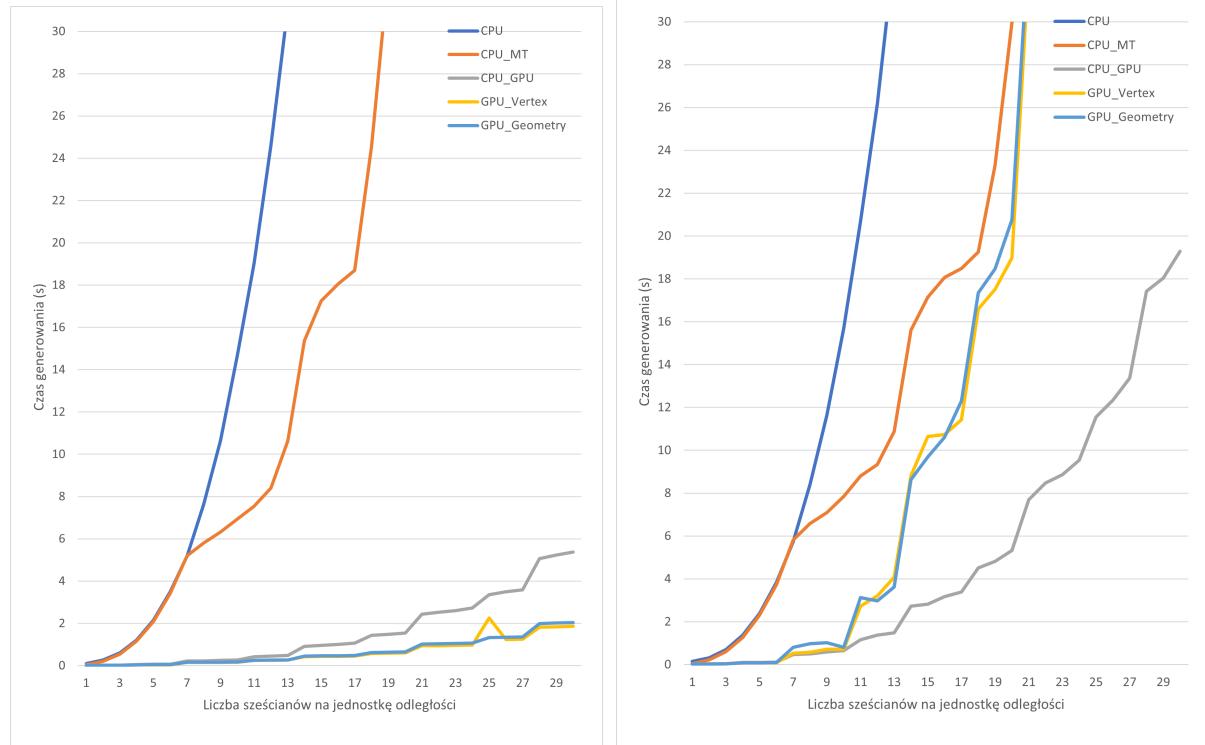
## 4.2 Porównanie wydajności metod generowania terenu

### 4.2.1 Porównanie ogólne

Pierwszym z wykonanych pomiarów jest pomiar ogólny, wykorzystujący domyślne ustawienia aplikacji. Celem tego pomiaru jest ogólne porównanie wydajności zaimplementowanych metod, w celu ustalenia pewnego standardu, do którego będą porównywane kolejne pomiary. Domyślne ustawienia aplikacji to:

- teren jest wygładzany i szczegółowy (rozmiar 20 x 10 x 20, 9 warstw szumu)
- aplikacja wykorzystuje bufory typu *append*
- aplikacja nie korzysta z renderowania pośredniego, ani uproszczonych shaderów
- aplikacja arbitralnie przycina rozmiar generycznych siatek i buforów GPU do 10% maksymalnego rozmiaru, co jest wystarczające do generowanego terenu o wskazanych parametrach
- w automatycznym teście wykonywane jest 10 krótkich albo 3 długie pomiary (granicą krótkiego pomiaru to 10 sekund). Maksymalna długość pomiaru to 60 sekund, a maksymalna liczba sześciianów na jednostkę odległości to 30. Średnia liczba klatek na sekundę jest mierzona przez 5 sekund.

#### Czas generowania terenu



(a). Pomiary wykonane na komputerze.

(b). Pomiary wykonane na laptopie.

Rys. 24. Wykresy zależności całkowitego czasu generowania terenu od liczby sześciianów na jednostkę odległości.

Na rysunku 24 zwizualizowano wyniki pomiarów czasu generacji terenu na obu badanych urządzeniach przy domyślnych ustawieniach aplikacji. Pomiary były wykonywane przez 60 sekund, ale dla klarowności wykresy zostały przycięte do 30 sekund. Całkowita liczba analizowanych sześciąanów wzrasta wykładniczo (wykładnik jest równy 3), ponieważ zwiększana jest liczba sześciąanów na jednostkę odległości, a są one ułożone w regularnej trójwymiarowej siatce. Pomiary wykonano, analizując od 4 tysięcy do 108 milionów sześciąanów, co odpowiada od 1 do 30 sześciąanom na jednostkę odległości dla terenu o wymiarach  $20 \times 10 \times 20$ .

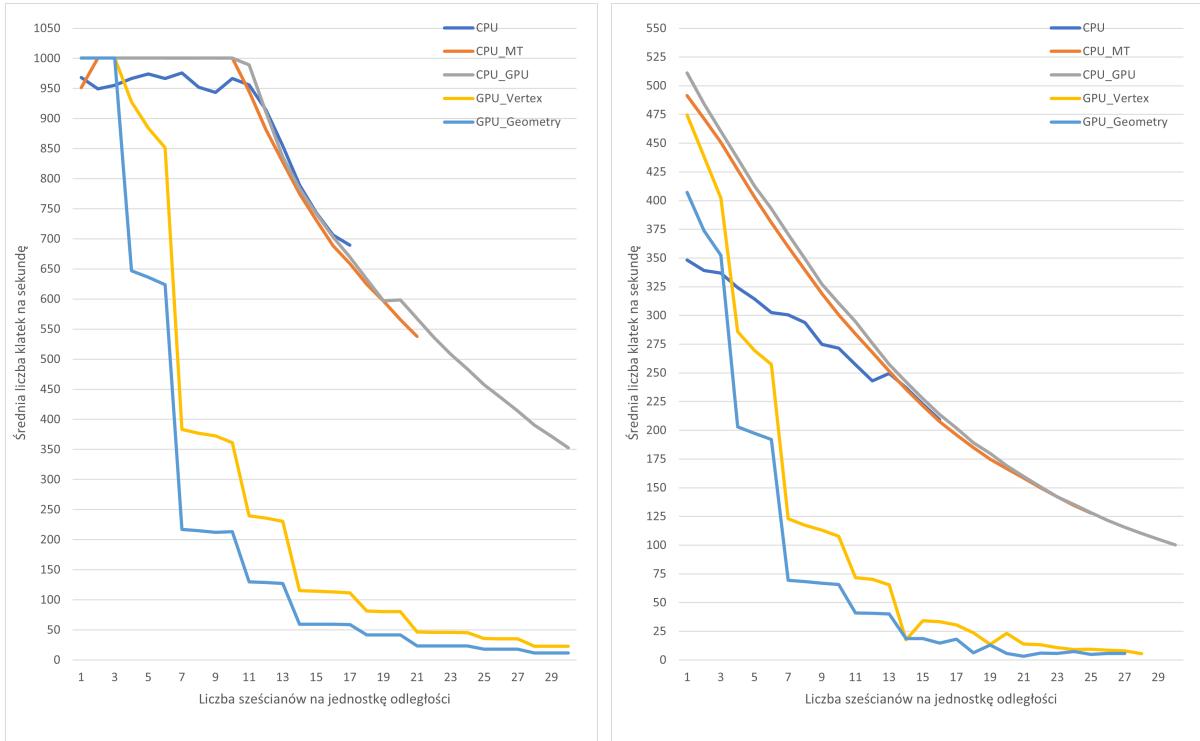
Można zaobserwować, że wykres czasu generacji metodą *CPU* przyjmuje kształt oczekiwany dla funkcji wykładniczej. Metoda *CPU\_MT* również zachowuje się w sposób oczekiwany. Dla niewielkiej liczby sześciąanów daje takie same rezultaty co metoda *CPU*, ponieważ istnieje tylko jeden fragment terenu, a zatem wykorzystany jest tylko wątek procesora. Jak tylko teren zaczyna być dzielony na więcej fragmentów, metoda *CPU\_MT* zaczyna generować teren szybciej niż metoda *CPU* (około 2-3 razy szybciej na obu urządzeniach w zależności od liczby fragmentów terenu). Wzrost wydajności metody *CPU\_MT* nie jest proporcjonalny do liczby rdzeni procesora CPU na urządzeniach najprawdopodobniej dlatego, że tylko fragment obliczeń wykonywany jest równolegle.

Wyniki metod *CPU\_GPU*, *GPU\_Vertex* i *GPU\_Geometry* silnie różnią się pomiędzy badanymi urządzeniami. Na komputerze metoda *CPU\_GPU* generuje teren średnio 41,05 razy szybciej niż metoda *CPU*. Na laptopie wartość tej przewagi wynosi jedynie 18,14. Jest to najprawdopodobniej spowodowane faktem, że procesory obu urządzeń mają podobną moc obliczeniową (wyniki metody *CPU* pomiędzy urządzeniami różnią się średnio tylko o około 11%, a procesory obu urządzeń mają zbliżoną częstotliwość taktowania), a ich karty graficzne znaczaco różnią się mocą (karta graficzna komputera ma 4,25 razy więcej rdzeni CUDA i wyższą częstotliwość taktowania niż karta laptopa). Niezależnie od urządzenia metoda *CPU\_GPU* generuje teren znacznie szybciej niż metody *CPU* i *CPU\_MT*.

Metody *GPU\_Vertex* i *GPU\_Geometry* są najbardziej nieprzewidywalne ze wszystkich metod. Na obu urządzeniach te metody generują teren w bardzo podobnym do siebie czasie, ale na komputerze robią to około 2 razy szybciej niż metoda *CPU\_GPU*, a na laptopie około 2,7 razy wolniej. Ta różnica może być spowodowane faktem, że metody *GPU\_Vertex* i *GPU\_Geometry* w znacznie szerszym zakresie korzystają z pamięci karty graficznej niż inne metody (wymagają stałej alokacji buforów z danymi wierzchołków terenu), a laptop posiada znacznie mniej pamięci dedykowanej karcie graficznej niż komputer i musi korzystać z wolniejszej pamięci współdzielonej. Przykładowo analiza 108 mln sześciąanów wymaga stałej alokacji około 3,6 GB pamięci GPU tylko na wyniki obliczeń (a obliczenia wykorzystują też bufore tymczasowe, które są zwalniane z pewnym opóźnieniem). Ta teoria wymaga sprawdzenia dodatkowymi pomiarami. Pozostaje faktem, że metody *GPU\_Vertex* i *GPU\_Geometry* generują teren szybciej niż metody *CPU* i *CPU\_MT*, jednak na laptopie ta różnica jest niewielka.

Podsumowując, ogólny pomiar całkowitego czasu generowania terenu pokazał, że metody wykorzystujące głównie GPU mają potencjał do znacznie wydajniejszego generowania terenu niż metody bazujące głównie na CPU, ale to jaka konkretnie metoda jest preferowana, silnie zależy od specyfikacji urządzenia. Istnieje też duże prawdopodobieństwo, że zastosowanie innych wartości parametrów aplikacji znaczaco wpłynie na wydajność poszczególnych metod, zwłaszcza tych w znacznym stopniu wykorzystujących GPU.

## Średnia liczba klatek na sekundę



(a). Pomiary wykonane na komputerze.

(b). Pomiary wykonane na laptopie.

Rys. 25. Wykresy zależności średniej liczby klatek rysowanych na sekundę od liczby sześciąników na jednostkę odległości.

Na rysunku 25 zwizualizowano liczbę średniej liczby klatek rysowanych na sekundę zarejestrowanych w ramach ogólnych pomiarów wydajności wykorzystanych wcześniej. Preferowana jest jak największa liczba klatek w celu utrzymania jak największej płynności obrazu. Maksymalna liczba klatek rysowanych na sekundę wspierana przez Unity to 1000. Wykresy niektórych metod urywają się, ponieważ te metody przekroczyły limit 60 sekund na wygenerowanie terenu i pomiar średniej liczby klatek na sekundę nie mógł zostać wykonany.

Na wykresach można zaobserwować, że metody *CPU*, *CPU\_MT* i *CPU\_GPU* dają bardzo podobne rezultaty i oferują znacznie większą płynność obrazu po wygenerowaniu terenu niż metody *GPU\_Vertex* i *GPU\_Geometry*. Jest to efekt oczekiwany, ponieważ pierwsza grupa metod buduje siatkę terenu raz, a metody z drugiej grupy generują siatkę terenu ponownie w każdej klatce, co znacznie spowalnia proces renderowania. Dodatkowym obciążeniem metod *GPU\_Vertex* i *GPU\_Geometry* jest fakt, że w zastosowanej konfiguracji, siatki przez nie generowane posiadają znacznie więcej wierzchołków, niż jest to konieczne do reprezentacji terenu. Wynikowe siatki terenu są zbudowane średnio z jedynie około 1,5% maksymalnej możliwej liczby trójkątów, a siatki generyczne wykorzystywane przez metody *GPU\_Vertex* i *GPU\_Geometry* są zbudowane z 10% maksymalnej możliwej liczby trójkątów. Zmniejszenie tej liczby jest możliwe, ale ryzykowne, ponieważ niektóre fragmenty terenu mogą być zbudowane z większej niż średnia liczby trójkątów i nie zostaną poprawnie wyświetcone. Możliwe jest również, że generując teren o innych parametrach, procentowe wykorzystanie maksymalnej możliwej liczby trójkątów się zwiększy. Konieczność ręcznej zmiany parametru w takim przypadku jest niekorzystna.

Metoda *GPU\_Vertex* oferuje średnio 1,8 razy większą liczbę klatek rysowanych na sekundę od metody *GPU\_Geometry*. Jest to najprawdopodobniej spowodowane faktem, że obie metody budują siatki o identycznym rozmiarze, ale *GPU\_Vertex* korzysta z gotowych trójkątów a metoda *GPU\_Geometry* tworzy nowe trójkąty w każdej klatce.

### **Względny czas wykonania poszczególnych etapów generowania terenu**

Na podstawie wyników metody *CPU* możliwe jest określenie proporcjonalnej złożoności obliczeniowej poszczególnych etapów generowania terenu. Metody wykorzystujące GPU niestety nie oferują takiej możliwości, ponieważ procesor nie ma możliwości zmierzenia czasu wykonania poszczególnych operacji wykonywanych przez GPU.

Tabela 1. Rozkład procentowy uśrednionego czasu generowania terenu metodą *CPU* na komputerze na poszczególne etapy.

Etap generacji	Czas wykonania (% całkowitego czasu)
Generowanie pola skalarnego	66,67
Algorytm Marching Cubes	7,60
Wygładzanie powierzchni	17,83
Budowanie siatki	0,74
Renderowanie	7,16

Na podstawie wyników zawartych w tabeli 1 można zaobserwować, że generowanie pola skalarnego jest najbardziej wymagającym obliczeniowo etapem generowania terenu. W przyszłości warto rozważyć możliwość pominięcia tego kroku poprzez czytanie wartości pola skalarnego z wygenerowanej wcześniej trójwymiarowej tekstury. Takie podejście ogranicza elastyczność generowania terenu, ale może być przydatne jeśli elastyczność nie jest potrzebna.

Wygładzanie powierzchni również jest bardzo wymagającym obliczeniowo krokiem. Warto zbadać czy wyłączenie tej funkcji wpłynie na porównanie wydajności poszczególnych metod generowania terenu.

### **Stabilność metod generowania terenu**

W ramach ogólnego pomiaru, oprócz średniego czasu generacji terenu, zostały również zebrane czasy minimalne i maksymalne. Na podstawie tych wartości można zbadać, jak bardzo stabilnie zachowują się poszczególne metody generacji. Wysoka stabilność jest korzystną cechą metody, ponieważ gwarantuje konsekwentne doświadczenie użytkownika aplikacji. W tabeli 2 przedstawiono średnie wartości maksymalnego odchylenia czasu generowania terenu od średniej na komputerze. Można zaobserwować, że metody wykorzystujące wyłącznie CPU są bardziej stabilne od metod korzystających z GPU.

Tabela 2. Niestabilność metod generowania terenu wyznaczona jako średnia wartość maksymalnych procentowych odchyleń czasu generowania terenu od średniej.

Metoda generacji	CPU	CPU_MT	CPU_GPU	GPU_Vertex	GPU_Geometry
Niestabilność (%)	2,94	2,57	10,26	12,16	6,30

#### 4.2.2 Uproszczony teren

Aplikacja udostępnia trzy podstawowe metody na zmniejszenie złożoności wykonywanych obliczeń przy zachowaniu takiego samego rozmiaru terenu:

- zmniejszenie liczby warstw szumu wykorzystanych do generowania wartości pola skalarnego
- wyłączenie funkcji wygładzania terenu
- wykorzystanie uproszczonych shaderów

Każda z wymienionych powyżej metod ogranicza złożoność obliczeń w innym etapie generacji terenu i może różnie oddziaływać na wydajność poszczególnych metod generowania terenu.

##### Zmniejszenie liczby warstw szumu

Przy porównaniu ogólnym zastosowano szczegółowy teren, do wygenerowania którego wykorzystano 9 warstw szumu Perlina. W ramach badania wpływu zmniejszenia liczby warstw szumu ponowiono pomiary, wykorzystując 3 warstwy szumu, pozostawiając wszystkie inne parametry bez zmian. Wyniki pomiaru przedstawiono w tabeli 3.

Tabela 3. Średnie zmniejszenie całkowitego czasu generacji poszczególnych metod generowania terenu poprzez wykorzystanie 3 warstw szumu Perlina zamiast 9 warstw.

Metoda generacji	Średnie zmniejszenie całkowitego czasu generacji na komputerze (%)	Średnie zmniejszenie całkowitego czasu generacji na laptopie (%)
CPU	49,36	49,22
CPU_MT	29,10	31,53
CPU_GPU	3,92	4,90
GPU_Vertex	8,33	1,53
GPU_Geometry	6,59	14,18

Na podstawie wyników pomiarów zawartych w tabeli 3 można stwierdzić, że czas wykonania metod wykorzystujących GPU zależy w znacznie mniejszym stopniu od złożoności obliczeniowej generowania wartości pola skalarnego niż metody korzystające głównie z CPU. Czas wykonania metody *CPU* zmniejszył się o prawie 50%, co jest oczekiwane, ponieważ na podstawie tabeli 1 wiadomo, że generowanie pola skalarnego jest zdecydowanie najbardziej czasochłonnym etapem generacji w tej metodzie. Najbardziej prawdopodobnym powodem, dla którego czas wykonania metod wykorzystujących GPU nie uległ znaczącej poprawie, jest to, że w przeciwieństwie do metody *CPU*, generowanie pola skalarnego nie jest w nich najbardziej czasochłonnym etapem generacji. Zastanawiająca jest różnica w wynikach metod *GPU\_Vertex* i *GPU\_Geometry* pomiędzy komputerem a laptopem. Niestety ze względu na brak dostępu do szczegółów operacji wykonywanych przez GPU, trudno jest wyciągnąć na tej podstawie wnioski.

W ramach nowych pomiarów nie zaobserwowano zmiany w średniej liczbie klatek na sekundę po wygenerowaniu terenu w żadnej metodzie. Jest to spodziewane, ponieważ generowanie wartości pola skalarnego jest operacją jednorazową i jej złożoność nie wpływa na złożoność renderowania terenu.

## Wyłączenie funkcji wygładzania terenu

Przy porównaniu ogólnym skorzystano z funkcji wygładzania terenu poprzez uśrednianie wektorów normalnych jego wierzchołków. Pomiar został ponowiony po wyłączeniu tej funkcji, a jego wyniki przedstawiono w tabeli 4.

Tabela 4. Średnie zmniejszenie całkowitego czasu generacji poszczególnych metod generowania terenu poprzez wyłączenie funkcji wygładzania terenu.

Metoda generacji	Średnie zmniejszenie całkowitego czasu generacji na komputerze (%)	Średnie zmniejszenie całkowitego czasu generacji na laptopie (%)
CPU	19,17	19,62
CPU_MT	19,54	19,95
CPU_GPU	14,38	19,52
GPU_Vertex	33,88	58,73
GPU_Geometry	32,11	56,37

Na podstawie wyników pomiarów zawartych w tabeli 4 można zauważyć, że czas wykonania metod *GPU\_Vertex* i *GPU\_Geometry* w znacznie większym stopniu zależy od złożoności etapu wygładzania terenu niż pozostałe metody. Oznacza to, że wygładzanie terenu tych metodach zajmuje proporcjonalnie więcej czasu. Jest tak najprawdopodobniej dlatego, że etap wygładzania terenu wykorzystuje więcej pamięci podręcznej GPU niż inne etapy (każdy analizowany wierzchołek, musi uzyskać dostęp do wszystkich wierzchołków w sąsiednich sześciianach). Ta hipoteza również tłumaczyłaby, dlaczego poprawa czasu generacji jest znaczaco większa na laptopie, gdzie dostępne jest mniej pamięci podręcznej GPU.

Ponownie nie zaobserwowano znaczącej zmiany w średniej liczbie klatek na sekundę po wygenerowaniu terenu w żadnej metodzie.

## Wykorzystanie uproszczonych shaderów

Przy porównaniu ogólnym wykorzystano shadery o złożonej strukturze, implementujące wszystkie funkcje Unity. W celu zbadania wpływu złożoności shaderów na wydajność poszczególnych metod ponowiono pomiar wykorzystując shadery uproszczone, zawierające tylko jedną iterację, niezbędną do pokazania terenu na ekranie z podstawowym cieniowaniem. Wyniki pomiaru przedstawiono w tabeli 5.

Tabela 5. Średnie zmniejszenie całkowitego czasu generacji poszczególnych metod generowania terenu po zastosowaniu uproszczonych shaderów.

Metoda generacji	Średnie zmniejszenie całkowitego czasu generacji na komputerze (%)	Średnie zmniejszenie całkowitego czasu generacji na laptopie (%)
CPU	2,68	0,96
CPU_MT	1,60	2,64
CPU_GPU	0,05	4,10
GPU_Vertex	6,26	6,97
GPU_Geometry	10,61	14,47

Na podstawie wyników pomiarów zawartych w tabeli 5 można zaobserwować, że zmniejszenie złożoności zastosowanego shadera znacząco zmniejsza czas generacji terenu metodami *GPU\_Vertex* i *GPU\_Geometry*, a nie ma dużego znaczenia w innych metodach. Ta różnica najprawdopodobniej spowodowana jest zmniejszeniem czasu potrzebnego na wyrenderowanie terenu po jego wygenerowaniu, ponieważ metody *GPU\_Vertex* i *GPU\_Geometry* przeprowadzają dodatkowe operacje w każdej iteracji shadera.

W przeciwnieństwie do dwóch poprzednich pomiarów średnia liczba klatek na sekundę uległa znacznej zmianie w stosunku do pomiaru ogólnego. Wyniki tych pomiarów przedstawiono w tabeli 6. Wyznaczając wartość zwiększenia liczby klatek rysowanych na sekundę, pominięto pomiary, w których osiągnięto limit 1000 klatek, ponieważ są one niemiarodajne (osiągnięty wynik zależy od arbitralnego limitu, a nie właściwości metod generacji). Z tego powodu tabela nie zawiera wyników osiągniętych metodami *CPU*, *CPU\_MT* i *CPU\_GPU* na komputerze, ponieważ wszystkie wykonane w ten sposób pomiary osiągnęły limit 1000 klatek na sekundę.

Tabela 6. Średnie zwiększenie liczby klatek rysowanych na sekundę dla poszczególnych metod generowania terenu po zastosowaniu uproszczonych shaderów.

Metoda generacji	Średnie zwiększenie liczby klatek na sekundę na komputerze (%)	Średnie zwiększenie liczby klatek na sekundę na laptopie (%)
CPU	-	122,93
CPU_MT	-	121,83
CPU_GPU	-	107,19
GPU_Vertex	332,12	156,92
GPU_Geometry	365,96	218,63

Z wyników pomiarów zawartych w tabeli 6 wynika, że zmniejszenie złożoności wykorzystanych shaderów znacząco zwiększa liczbę klatek rysowanych na sekundę wszystkimi metodami, jednak metody *GPU\_Vertex* i *GPU\_Geometry* korzystają na tym najbardziej. Jest to efekt spodziewany, ponieważ złożoność shadera bezpośrednio przekłada się na ilość obliczeń niezbędnych do wykonania w celu wyświetlenia terenu w każdej klatce, a metody *GPU\_Vertex* i *GPU\_Geometry* tworząc siatkę terenu w każdej klatce na nowo, jeszcze bardziej zwiększają ilość koniecznych obliczeń.

### Podsumowanie pomiarów przy uproszczonym terenie

Z wykonanych pomiarów można wywnioskować, że stosunek wydajności poszczególnych zaimplementowanych metod silnie zależy od złożoności obliczeniowej poszczególnych etapów generowania terenu. Oznacza to, że wybór najlepszej metody może zależeć od specyfikacji problemu, do którego zostanie zastosowana.

Metody wykorzystujące GPU znacznie lepiej reagują na zwiększenie złożoności obliczeniowej operacji niewymagających alokacji dużej ilości pamięci, takich jak wyznaczanie wartości pola skalarnego, niż metody korzystające głównie z CPU. W przypadku dodania operacji wymagających alokacji dużej ilości pamięci efekt jest odwrotny.

Liczba klatek rysowanych na sekundę zależy głównie od złożoności zastosowanych shaderów. W przypadku metod *GPU\_Vertex* i *GPU\_Geometry* jest to specyfikacja kluczowa, ponieważ ich wyniki są najbardziej od niej zależne. Zbyt niska częstotliwość odświeżania obrazu może sprawiać, że te metody staną się nieprzydatne.

### 4.2.3 Renderowanie pośrednie

Technika renderowania pośredniego pozwala zdefiniować liczbę wierzchołków generowanej siatki po stronie karty graficznej, co pozwala na generowanie precyzyjnej liczby wierzchołków metodami *GPU\_Vertex* i *GPU\_Geometry* (zamiast stosowania siatki generycznej). W teorii zastosowanie tej techniki nie powinno zmienić rezultatów generowania terenu innymi metodami, jednak w dalszym ciągu została ona zaimplementowana w celu porównania. W tabelach 7 i 8 przedstawiono wyniku pomiaru przeprowadzonego dla domyślnych parametrów aplikacji, ale stosując technikę renderowania pośredniego.

Tabela 7. Średni czas generacji poszczególnych metod generowania terenu względem pomiaru ogólnego po zastosowaniu techniki renderowania pośredniego.

Metoda generacji	Czas generacji na komputerze (% wartości pomiaru ogólnego)	Czas generacji na laptopie (% wartości pomiaru ogólnego)
CPU	98,68	100,24
CPU_MT	95,92	96,37
CPU_GPU	101,30	99,21
GPU_Vertex	91,78	65,50
GPU_Geometry	88,13	91,27

Tabela 8. Średnia liczba klatek rysowanych na sekundę względem pomiaru ogólnego po zastosowaniu techniki renderowania pośredniego.

Metoda generacji	Liczba klatek na sekundę na komputerze (% wartości pomiaru ogólnego)	Liczba klatek na sekundę na laptopie (% wartości pomiaru ogólnego)
CPU	95,90	115,88
CPU_MT	98,65	99,10
CPU_GPU	92,18	98,61
GPU_Vertex	665,76	499,67
GPU_Geometry	1123,11	564,01

Na podstawie wyników pomiarów zawartych w tabelach 7 i 8 można zaobserwować, że zgodnie z oczekiwaniami, zastosowanie techniki renderowania pośredniego nie mają znaczącego wpływu na metody *CPU*, *CPU\_MT* i *CPU\_GPU*. Znacząca różnica pojawiła się za to w wynikach metod *GPU\_Vertex* i *GPU\_Geometry*. Czas generowania terenu tymi metodami uległ tylko nieznacznej poprawie, ale średnia liczba klatek rysowanych na sekundę zwiększyła się diametralnie.

Czas generacji terenu metodami *GPU\_Vertex* i *GPU\_Geometry* zmniejszył się ze względu na zmniejszony czas przeprowadzania etapu renderowania terenu. Zwiększenie średniej liczby klatek rysowanych na sekundę związane jest z ograniczeniem liczby wierzchołków, z których zbudowane są siatki terenu. Zamiast wykorzystywać siatki generyczne, zbudowane z 10% maksymalnej możliwej liczby wierzchołków (przy domyślnych parametrach programu), siatka tworzona jest w całości po stronie GPU, dokładnie z takiej liczby wierzchołków ile jest zawartych w danych wyjściowych procesu generacji. Zastosowanie renderowania pośredniego znacznie zwiększa potencjalną przydatność metod *GPU\_Vertex* i *GPU\_Geometry*, zwiększając średnią zaobserwowaną liczbę klatek rysowanych na sekundę w najgorszym przypadku złożoności terenu z 11,6 do 165,6 na komputerze i z 3,2 do 11,2 na laptopie.

#### 4.2.4 Przycinanie rozmiaru generycznych siatek i buforów

W aplikacji zaimplementowano dwie metody ograniczania rozmiaru generycznych siatek i buforów. Jedną z nich jest zastosowane w pomiarze ogólnym, arbitralne ograniczenie ich rozmiarów (wartość 10%). Drugą metodą jest pobieranie informacji o dokładnej liczbie potrzebnych wierzchołków z GPU. Obie te techniki mają zastosowanie tylko w metodach *GPU\_Vertex* i *GPU\_Geometry*.

##### Arbitralne ograniczenie rozmiaru

W celu zbadania istotności arbitralnego ograniczenia rozmiaru generycznych siatek i buforów ponowiono pomiar o takich samych parametrach jak pomiar ogólny, ale wartość ograniczenia równą 5% (było 10%). Wyniki pomiarów przedstawiono w tabelach 9 i 10.

Tabela 9. Średni czas generacji terenu metodami *GPU\_Vertex* i *GPU\_Geometry* względem pomiaru ogólnego po ograniczeniu rozmiaru siatek i buforów do 5%.

Metoda generacji	Czas generacji na komputerze (% wartości pomiaru ogólnego)	Czas generacji na laptopie (% wartości pomiaru ogólnego)
GPU_Vertex	95,17	73,89
GPU_Geometry	92,91	68,93

Tabela 10. Średnia liczba klatek rysowanych na sekundę względem pomiaru ogólnego po ograniczeniu rozmiaru siatek i buforów do 5%.

Metoda generacji	Liczba klatek na sekundę na komputerze (% wartości pomiaru ogólnego)	Liczba klatek na sekundę na laptopie (% wartości pomiaru ogólnego)
GPU_Vertex	179,94	161,62
GPU_Geometry	185,6	169,12

Na podstawie wyników pomiarów zawartych w tabelach 9 i 10 można zaobserwować, że zmniejszenie wartości limitu rozmiaru siatek generycznych i buforów znaczco poprawia wydajność metod *GPU\_Vertex* i *GPU\_Geometry*, zarówno pod kątem czasu generacji, jak i średniej liczby klatek rysowanych na sekundę.

Średni czas generacji terenu został zmniejszony zarówno na komputerze, jak i na laptopie, ale w drugim przypadku poprawa jest bardziej znacząca. Jest to najprawdopodobniej związane z faktem, że w wyniku zmiany wartości badanego parametru wykorzystywane jest znacznie mniej pamięci podręcznej GPU, co ma większe znaczenie na laptopie, gdzie tej pamięci jest mniej.

Zwiększenie średniej liczby klatek rysowanych na sekundę nastąpiło w podobnym zakresie na obu urządzeniach. Na komputerze osiągnięto niemal dwukrotne zwiększenie liczby klatek, co jest oczekiwane, ponieważ wykorzystane siatki terenu i bufory są dwukrotnie mniejsze niż w przypadku pomiaru ogólnego.

Niestety, zmniejszenie wartości limitu rozmiaru siatek generycznych i buforów wiąże się z ryzykiem wyświetlenia niepełnego terenu, jeśli liczba wierzchołków konieczna do jego reprezentacji przekroczy wartość parametru. W przypadku wykonanego pomiaru nie zaobserwowano niepełnego wyświetlenia terenu, jednak w przypadku innych parametrów terenu byłoby ono możliwe.

## Odczyt liczby wierzchołków z GPU

W celu zbadania konsekwencji zastosowania odczytu liczby wierzchołków z GPU w metodach *GPU\_Vertex* i *GPU\_Geometry* ponowiono pomiar o takich samych parametrach jak pomiar ogólny, ale stosując odczyt. Wyniki pomiarów przedstawiono w tabelach 11 i 12.

Tabela 11. Średni czas generacji terenu metodami *GPU\_Vertex* i *GPU\_Geometry* względem pomiaru ogólnego po zastosowaniu odczytu liczby wierzchołków z GPU.

Metoda generacji	Czas generacji na komputerze (% wartości pomiaru ogólnego)	Czas generacji na laptopie (% wartości pomiaru ogólnego)
GPU_Vertex	236,14	64,00
GPU_Geometry	225,82	57,28

Tabela 12. Średnia liczba klatek rysowanych na sekundę względem pomiaru ogólnego po zastosowaniu odczytu liczby wierzchołków z GPU.

Metoda generacji	Liczba klatek na sekundę na komputerze (% wartości pomiaru ogólnego)	Liczba klatek na sekundę na laptopie (% wartości pomiaru ogólnego)
GPU_Vertex	678,24	465,35
GPU_Geometry	1201,07	733,22

Na podstawie wyników pomiarów zawartych w tabelach 9 i 10 można zaobserwować, że efekt zastosowania odczytu liczby wierzchołków z GPU w znacznym stopniu zależy od urządzenia. Oba urządzenia odnotowały znaczny wzrost średniej liczby klatek rysowanych na sekundę, ale wpływ na całkowity czas generowania terenu jest mieszany.

W przypadku komputera średni czas generacji terenu znacząco się zwiększył. Ten spadek wydajności jest tak duży, że średni czas generacji metodą *GPU\_Vertex* stał się średnio 1,36 raza większy niż czas generacji metodą *CPU\_GPU*, a czas generacji metodą *GPU\_Geometry* stał się średnio 1,34 raza większy. W porównaniu ogólnym metoda *GPU\_Vertex* osiągnęła średni czas generacji 2,47 razy mniejszy niż metoda *CPU\_GPU*, a metoda uzyskała czas 2,41 razy mniejszy. Oznacza to, że odczyt rozmiaru bufora i przygotowanie nowej siatki terenu jest podobnie czasochłonne od pobrania wszystkich danych z bufora, a zysk z ograniczenia rozmiarów siatki i buforów jest minimalny.

Pomiarystwo wykonane na laptopie pokazują efekt odwrotny do pomiarów wykonanych na komputerze. W tym przypadku czas generacji terenu znacząco się poprawił. Generowanie terenu metodą *GPU\_Vertex* osiągnęło średni czas tylko 1,04 razy większy od metody *CPU\_GPU*, podczas gdy podczas ogólnego pomiaru ten czas był aż 3,05 razy większy. Metoda *GPU\_Vertex* osiągnęła średni czas 1,03 razy większy, podczas gdy wcześniej była aż 2,88 razy wolniejsza. Oznacza to to, że w przypadku laptopa zysk z ograniczenia wykorzystywanej przez bufore pamięci jest tak znaczący, że rekompensuje z nadwyżką czas potrzebny na odczyt wartości. Jest to najprawdopodobniej spowodowane małą ilością pamięci podręcznej GPU dostępnej na laptopie.

Zaobserwowany wzrost średniej liczby klatek rysowanych na sekundę jest bardzo zbliżony do wzrostu przedstawionego w tabeli 8, gdzie do pomiaru zastosowano technikę renderowania pośredniego. Jest to efekt oczekiwany, ponieważ odczyt liczby wierzchołków z GPU pozwala stworzyć siatkę terenu o minimalnej niezbędnej liczbie wierzchołków, tak samo, jak zastosowanie renderowania pośredniego.

#### 4.2.5 Unikanie buforów typu *append*

Przy domyślnych ustawieniach aplikacja wykorzystuje bufory typu *append*. Zastosowanie tego typu buforów może potencjalnie spowalniać proces przetwarzania danych przez GPU, ponieważ wymaga synchronizacji działających wątków. W celu zbadania, czy wykorzystanie buforów typu *append* spowalnia działanie aplikacji, wykonano pomiar identyczny do ogólnego, ale nie wykorzystując tego typu buforów. Niestety, bez korzystania z buforów typu *append* nie jest możliwe przycinanie rozmiaru generycznych siatek i buforów, ani renderowanie pośrednie. Z tego powodu nie jest możliwe porównanie wykonanego pomiaru do pomiaru ogólnego, ponieważ wykorzystywał on parametr arbitralne ograniczający rozmiar siatek i buforów do 10% wartości maksymalnej. W celu dokonania porównania ponowiono pomiar ogólny, ale ustawiając wartość tego parametru na 100%.

Z powodu zwiększonego zużycia pamięci, w ramach automatycznego testu ustawiono maksymalną liczbę sześciianów na jednostkę odległości na 13. Większe wartości spowodowały awaryjne zatrzymanie aplikacji przez system operacyjny. Pomiary wykonano wyłącznie dla metod *GPU\_Vertex* i *GPU\_Geometry*, ponieważ metody CPU nie wykorzystują buforów GPU, a w metodzie *CPU\_GPU* bufory typu *append* są niezbędne w celu pobrania danych wierzchołków na CPU. Porównanie wyników obu pomiarów zawarto w tabelach 13 i 14.

Tabela 13. Porównanie średniego czasu generacji terenu metodami *GPU\_Vertex* i *GPU\_Geometry* w zależności od zastosowania buforów typu *append*.

Metoda generacji	Czas generacji na komputerze wykorzystując bufory <i>append</i> (s)	Czas generacji na komputerze nie wykorzystując bufory <i>append</i> (s)	Zmiana czasu generacji	Czas generacji na laptopie wykorzystując bufory <i>append</i> (s)	Czas generacji na laptopie nie wykorzystując bufory <i>append</i> (s)	Zmiana czasu generacji
GPU_Vertex	0,21	0,29	+38,10%	3,87	5,71	+47,55%
GPU_Geometry	0,27	0,34	+25,93%	4,17	6,53	+56,59%

Tabela 14. Porównanie średniej liczby klatek na sekundę w zależności od zastosowania buforów typu *append*.

Metoda generacji	Liczba klatek na sekundę na komputerze wykorzystując bufory <i>append</i> (s)	Liczba klatek na sekundę na komputerze nie wykorzystując bufory <i>append</i> (s)	Zmiana liczby klatek	Liczba klatek na sekundę na laptopie wykorzystując bufory <i>append</i> (s)	Liczba klatek na sekundę na laptopie nie wykorzystując bufory <i>append</i> (s)	Zmiana liczby klatek
GPU_Vertex	190,12	178,32	-6,21%	63,51	51,17	-19,43%
GPU_Geometry	110,66	104,2	-5,84%	35,09	31,52	-10,17%

Na podstawie porównania, którego rezultaty przedstawiono w tabelach 13 i 14 można zaobserwować, że zastosowanie buforów standardowych, zamiast buforów typu *append*, zwiększyło czas generowania terenu i zmniejszyło średnią liczbę klatek rysowanych na sekundę na obydwu urządzeniach. Oznacza to, że zastosowanie buforów typu *append* nie zmniejsza wydajności aplikacji, a nawet ją poprawia. W połączeniu z faktem, że te bufory są niezbędne do przycinania rozmiaru generycznych siatek i buforów oraz renderowania pośredniego, można jednoznacznie stwierdzić, że unikanie buforów typu *append* nie jest korzystne.

#### 4.2.6 Zastosowanie innych interfejsów graficznych

Zaletą silnika Unity jest możliwość łatwej zmiany stosowanego interfejsu graficznego (ang. graphics API). Warto zbadać, jak zaimplementowane metody sprawdzają się w połączeniu z różnymi interfejsami graficznymi. Wszystkie poprzednie pomiary były wykonane z wykorzystaniem interfejsu DirectX 11, ale Unity pozwala na zastosowanie 3 innych interfejsów:

- OpenGL Core
- OpenGL ES3
- Vulkan

W ramach testu stworzono 3 nowe wersje aplikacji wykorzystujące każdy z wymienionych interfejsów i przeprowadzono za ich pomocą pomiary z wykorzystaniem domyślnych parametrów. Pomiary są analogiczne do pomiaru ogólnego, ale zastosowano różną maksymalną liczbę sześciianów na jednostkę odległości ze względu na awarię niektórych wersji aplikacji podczas testowania metod *GPU\_Vertex* i *GPU\_Geometry*. Dane diagnostyczne aplikacji wskazują, że przyczyną awarii jest przekroczenie limitu czasu przetwarzania danych przez GPU, co system Windows interpretuje jako zawieszenie. Występowanie awarii powinno być brane pod uwagę przy ocenie przydatności poszczególnych interfejsów.

Maksymalną liczbę sześciianów na jednostkę odległości ograniczono do 18 w przypadku interfejsów *OpenGL Core* i *OpenGL ES3*. W przypadku interfejsu Vulkan konieczne było ograniczenie tej wartości do 12. W tabelach 15 i 16 przedstawiono wyniki wykonanych pomiarów w porównaniu z wynikami osiągniętymi w pomiarze ogólnym.

Tabela 15. Porównanie średniego czasu generacji terenu z wykorzystaniem alternatywnych interfejsów graficznych zamiast interfejsu DirectX 11. Podane wartości opisują stosunek procentowy wyniku nowego pomiaru do pomiaru ogólnego.

Metoda\Interfejs	OpenGL Core (komputer)	OpenGL ES3 (komputer)	Vulkan (komputer)	OpenGL Core (laptop)	OpenGL ES3 (laptop)	Vulkan (laptop)
CPU	99,19%	100,10%	98,71%	96,76%	101,05%	96,43%
CPU_MT	99,31%	102,75%	100,44%	93,90%	96,19%	99,31%
CPU_GPU	1794,45%	1768,99%	128,60%	1130,94%	1132,84%	5232,73%
GPU_Vertex	2321,99%	2357,59%	131,69%	632,27%	666,74%	10655,64%
GPU_Geometry	2251,18%	2276,35%	127,42%	603,45%	566,86%	8695,30%

Tabela 16. Porównanie średniej liczby klatek na sekundę rysowanych z wykorzystaniem alternatywnych interfejsów graficznych zamiast interfejsu DirectX 11. Podane wartości opisują stosunek procentowy wyniku nowego pomiaru do pomiaru ogólnego.

Metoda\Interfejs	OpenGL Core (komputer)	OpenGL ES3 (komputer)	Vulkan (komputer)	OpenGL Core (laptop)	OpenGL ES3 (laptop)	Vulkan (laptop)
CPU	78,94%	64,37%	99,23%	81,40%	75,04%	100,47%
CPU_MT	79,12%	65,74%	100,38%	72,15%	67,38%	87,77%
CPU_GPU	79,34%	65,60%	99,93%	70,61%	65,60%	82,51%
GPU_Vertex	59,27%	53,26%	100,30%	48,02%	43,23%	85,45%
GPU_Geometry	61,57%	56,14%	98,55%	54,47%	49,84%	88,34%

Na podstawie porównania, którego rezultaty przedstawiono w tabelach 15 i 16 można zaobserwować, że badane interfejsy graficzne mają niewielki wpływ na czas generacji terenu metodami *CPU* i *CPU\_MT*, ale znacznie zwiększaą czas generacji metodami wykorzystującymi GPU. Porównanie średniej liczby klatek rysowanych na sekundę również wypada dla nich niekorzystnie, pokazując pogorszenie dla wszystkich metod generowania terenu.

Ze względu na obniżoną wydajność, wykorzystywanie innych interfejsów graficznych niż DirectX 11 wydaje się być niekorzystne. Warto zaznaczyć, że uzyskane wyniki nie dowodzą tego, że badane interfejsy graficzne są zawsze mniej wydajne niż DirectX 11, ale tego, że są mniej wydajne w przypadku wykorzystania shaderów obliczeniowych wewnątrz silnika Unity. Jest prawdopodobne, że wyniki poszczególnych interfejsów graficznych poprawiłyby się, gdyby zastosować analogiczną implementację poza silnikiem Unity.

W przypadku, gdyby docelowa platforma aplikacji nie wspierała interfejsu DirectX 11 (dedykowanego systemowi operacyjnemu Windows), najlepszą alternatywą wydaje się być interfejs Vulkan (dedykowany wielu platformom, włącznie z systemem operacyjnym Linux i Android). Interfejs Vulkan osiągnął wyniki zbliżone do DirectX 11 na komputerze. Niestety w przypadku pomiarów przeprowadzonych na laptopie, drastycznie zwiększyły się czas generacji, co może oznaczać, że Vulkan gorzej sprawuje się na słabszych urządzeniach. Dodatkowo maksymalną możliwą do zastosowania liczbą sześcianów na jednostkę odległości w przypadku interfejsu Vulkan było 12, co przekłada się na 6912000 przeanalizowanych sześcianów. Próba generacji bardziej złożonego terenu z wykorzystaniem tego interfejsu kończy się przekroczeniem limitu czasu wykonywania operacji przez GPU i awaryjnym zatrzymaniem aplikacji.

### 4.3 Podsumowanie wyników pomiarów

Na podstawie dokonanych obserwacji nie jest możliwe jednoznaczne wyłonienie najbardziej wydajnej metody generowania terenu, ponieważ wydajność poszczególnych metod względem siebie zależy od parametrów generacji terenu i specyfikacji urządzenia. Każda metoda posiada wady i zalety w zależności od zastosowania.

Metoda *CPU* jest najprostsza w implementacji, zachowuje się najbardziej stabilnie i pozwala wygenerować teren o bardzo dużym rozmiarze bez ryzyka awaryjnego zatrzymania aplikacji. Niestety jest to również metoda generująca teren naj wolniej.

Metoda *CPU\_MT* jest trochę trudniejsza w implementacji niż metoda *CPU*, ale pozwala generować teren szybciej (około 2-3 razy szybciej w przeprowadzonych pomiarach), zachowując wszystkie zalety tej metody. Warto stosować metodę *CPU\_MT* zamiast metody *CPU* na każdym urządzeniu udostępniającym wiele wątków procesora.

Metoda *CPU\_GPU* pozwala generować teren znacznie szybciej niż metody *CPU* i *CPU\_MT*, jednak wiąże się to z wykorzystaniem shaderów obliczeniowych, co komplikuje implementację. Zwiększenie wydajności tej metody w stosunku do metod *CPU* i *CPU\_MT* zależy od mocy obliczeniowej procesora graficznego. Na komputerze odnotowano 41,05 razy krótszy średni czas generowania terenu metodą *CPU\_GPU* niż metodą *CPU*, natomiast na laptopie ta wartość wyniosła 18,14. Dużą zaletą metody *CPU\_GPU* względem metod *GPU\_Vertex* i *GPU\_Geometry* jest zachowanie wysokiej liczby klatek rysowanych na sekundę po generacji terenu, na równi z metodami *CPU* i *CPU\_MT*. Metoda *CPU\_GPU* również nie wprowadza ryzyka awaryjnego zatrzymania aplikacji.

Wyniki osiągnięte przez metody *GPU\_Vertex* i *GPU\_Geometry* są dwuznaczne. Są to metody najtrudniejsze w implementacji i bardzo zależne od specyfikacji urządzenia

docelowego aplikacji i zastosowanych parametrów. Na komputerze obie metody pozwoliły osiągnąć około 2 razy krótszy czas generowania terenu niż metoda *CPU\_GPU*, ale na laptopie odnotowano czas 2,7 razy dłuższy. Te wyniki mogą zostać znacznie poprawione za pomocą arbitralnego ograniczania rozmiaru generycznych siatek i buforów, jednak wiąże się to z ręcznym doborem tego parametru w zależności od generowanego terenu, co nie zawsze jest możliwe.

Wydajność metod *GPU\_Vertex* i *GPU\_Geometry* zdaje się być silnie uzależniona od ilości dostępnej pamięci podręcznej procesora graficznego, co jest problematyczne, jeżeli docelowa aplikacja jest tworzona z myślą o wielu różnych urządzeniach. Silne wykorzystanie pamięci GPU generuje również szereg innych problemów, przede wszystkim fakt, że aplikacja może zostać awaryjnie zatrzymana przez system operacyjny, jeśli wykorzystane zostanie zbyt dużo pamięci oraz to, że może zabraknąć pamięci podręcznej GPU dla innych elementów aplikacji, takich jak załadowane tekstury innych obiektów na scenie.

Kolejną wadą metod *GPU\_Vertex* i *GPU\_Geometry* jest zmniejszona liczba klatek rysowanych na sekundę po zakończeniu generacji terenu, związana z koniecznością kształtuowania siatki terenu w każdej klatce. Technika renderowania pośredniego pozwala na znaczące ograniczenie tego problemu, jednak nawet po jej zastosowaniu liczba generowanych klatek pozostaje od 10% do 50% mniejsza niż w przypadku pozostałych metod.

Właściwości generowanego terenu również mają bardzo duże znaczenie w wyborze najlepszej metody. Zmniejszenie liczby warstw szumu wykorzystywanego do generowania wartości pola skalarnego znacząco zmniejsza czas generowania terenu metodami *CPU* i *CPU\_MT*, mając niewielki wpływ na pozostałe metody. Wyłączenie funkcji wygładzania terenu daje efekt odwrotny, nieproporcjonalnie skracając czas generacji metodami *GPU\_Vertex* i *GPU\_Geometry*. Zastosowanie prostszych shaderów graficznych również znacząco poprawia wydajność tych metod.

Konieczności wykorzystania innego interfejsu graficznego niż DirectX 11 również może wpływać na wybór metody generacji terenu. Wykorzystując interfejsy opierające się o OpenGL, można oczekiwąć znacznego spadku wydajności metod wykorzystujących shaderów obliczeniowe (nawet 23 razy dłuższy czas generowania terenu), co zmniejsza atrakcyjność tych metod. Wykorzystując interfejs Vulkan, można oczekiwąć porównywalnych rezultatów metod wykorzystujących shaderów obliczeniowe na silniejszych urządzeniach, ale znacznego spadku wydajności (nawet 106 razy dłuższy czas generowania terenu) na urządzeniach słabszych. Wszystkie alternatywne interfejsy spowodowały zwiększenie ryzyka awaryjnego zatrzymania aplikacji, co dodatkowo przemawia przeciwko stosowaniu metod *GPU\_Vertex* i *GPU\_Geometry* w takiej konfiguracji.

Przeprowadzone pomiary wykazały, że wykorzystanie GPU do przeprowadzania obliczeń pozwala na zmniejszenie czasu generowania terenu w większości badanych przypadków. Jedynymi sytuacjami, w których preferowane są metody bazujące wyłącznie na CPU, są sytuacje, kiedy docelowe urządzenie nie wspiera interfejsu graficznego DirectX 11. Wśród metod wykorzystujących shaderów obliczeniowe, metoda *CPU\_GPU* jest najbardziej bezpieczna. Wykorzystanie tej metody pozwala na znaczące zmniejszenie czasu generowania terenu, zachowując wysoką liczbę klatek rysowanych na sekundę i nie zajmuje przesadnie dużo pamięci podręcznej GPU. Metody *GPU\_Vertex* i *GPU\_Geometry* pozwalają w niektórych przypadkach na osiągnięcie krótszego czasu generacji niż metoda *CPU\_GPU*, jednak posiadają też wiele wad. Te rozwiązania są znacznie bardziej skomplikowane w implementacji, działają gorzej na słabszych urządzeniach, zmniejszają liczbę klatek rysowanych na sekundę i zużywają dużo pamięci podręcznej GPU, co może prowadzić do kolejnych spadków wydajności lub awaryjnego zatrzymania aplikacji.

# Podsumowanie

Celem pracy było porównanie wydajności generowania terenu 3D tworzonego za pomocą algorytmu Marching Cubes na CPU i GPU. Porównanie przeprowadzono, implementując aplikację w silniku Unity i porównując wydajność 5 różnych metod generowania terenu w różnym stopniu i w różny sposób wykorzystujących zasoby CPU i GPU. Zastosowanymi metrykami wydajności były całkowity czas generowania terenu i liczba rysowanych klatek na sekundę po zakończeniu generacji. W analizie wzięto również pod uwagę dodatkowe czynniki wpływające na praktyczność zastosowania poszczególnych metod, takie jak ilość wykorzystanej pamięci podręcznej GPU i maksymalny możliwy do uzyskania rozmiar terenu.

Stworzona aplikacja pozwoliła na przetestowanie metod generowania terenu w rzeczywistych sytuacjach i na różnych urządzeniach. Przeanalizowano, w jaki sposób zachowują się poszczególne metody, kiedy generowany teren jest bardziej lub mniej złożony, kiedy docelowe urządzenie jest silniejsze lub słabsze oraz kiedy stosowane są różne techniki renderowania i interfejsy graficzne. Zaimplementowano również wiele wariantów poszczególnych metod oraz przeanalizowano, jaki wpływ na wydajność ma stosowanie ich.

Na podstawie przeprowadzonej analizy wynioskowano, że wykorzystanie GPU do przeprowadzania obliczeń pozwala na zmniejszenie czasu generowania terenu w większości przypadków, ale może też prowadzić do zmniejszenia liczby rysowanych klatek na sekundę i innych niekorzystnych konsekwencji w zależności od szczegółów implementacji zastosowanej metody i parametrów urządzenia. W niektórych przypadkach metody generujące teren korzystając niemal wyłącznie z GPU wypadały najlepiej, osiągając czasy generacji nawet ponad 100 razy krótsze od metod wykorzystujących głównie CPU i ponad 2 razy krótsze od metody hybrydowej, jednocześnie zachowując akceptowną liczbę klatek rysowanych na sekundę. W innych przypadkach zastosowanie metody hybrydowej było korzystniejsze, szczególnie kiedy wykorzystywane było słabsze urządzenie lub interfejs graficzny inny niż DirectX 11. Przeprowadzona analiza pomaga w wyborze najbardziej wydajnej metody generowania terenu w zależności od jego parametrów, docelowego urządzenia i technologii.

Przeprowadzona analiza może zostać rozszerzona w kolejnych badaniach w różnych kierunkach. Jednym z nich jest implementacja analogicznych metod generowania terenu poza silnikiem Unity i sprawdzenie, czy wyciągnięte wnioski są uniwersalne dla innych rozwiązań. Innym potencjalnym kierunkiem dalszych badań jest poszerzenie implementacji o nowe metody i techniki. Dwie ciekawe techniki często wykorzystywane w grach komputerowych to usuwanie niewidocznych powierzchni (ang. frustum culling) oraz dynamiczna rozdzielcość terenu (zmniejszanie jakości terenu wraz ze wzrostem odległości od położenia kamery). Obie te techniki mają potencjał znacznie zwiększyć wydajność generowania terenu i są w wysokim stopniu kompatybilne z metodami generującymi teren w oparciu o GPU. W poszukiwaniu dalszych optymalizacji warto również zwrócić się w stronę innych prac naukowych. Ciekawym kierunkiem dalszych badań byłaby porównanie zaimplementowanych metod do techniki opisanej w pracy "High-speed Marching Cubes using HistoPyramids" autorstwa Christophera Dyken'a i innych [25], oferującej wydajniejszą, ale bardziej złożoną implementację algorytmu Marching Cubes.

# Bibliografia

- [1] G. Johansson i H. Carr, „Accelerating marching cubes with graphics hardware,” sty. 2006, s. 378. DOI: [10.1145/1188966.1189018](https://doi.org/10.1145/1188966.1189018).
- [2] B. José, D. Esteban, W. Gonzalez, C. Marcelo, B. Vieira i F. R. Leta, „Procedural Terrain Generation at GPU Level with Marching Cubes,” 2007.
- [3] T. J. Rose i A. G. Bakaoukas, „Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques,” w *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, 2016, s. 1–2. DOI: [10.1109/VS-GAMES.2016.7590336](https://doi.org/10.1109/VS-GAMES.2016.7590336).
- [4] A. Fournier, D. Fussell i L. Carpenter, „Computer Rendering of Stochastic Models,” *Commun. ACM*, t. 25, s. 371–384, czer. 1982. DOI: [10.1145/358523.358553](https://doi.org/10.1145/358523.358553).
- [5] T. Archer, „Procedurally Generating Terrain,” w *3rd Int. Conf. Comput. Support. Educ.*, 2011.
- [6] K. Perlin, „An Image Synthesizer,” *SIGGRAPH Comput. Graph.*, t. 19, nr. 3, s. 287–296, lip. 1985, ISSN: 0097-8930. DOI: [10.1145/325165.325247](https://doi.org/10.1145/325165.325247).
- [7] S. Gustavson, „Simplex noise demystified,” sty. 2005.
- [8] S. Worley, „A Cellular Texture Basis Function,” w *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’96, New York, NY, USA: Association for Computing Machinery, 1996, s. 291–294, ISBN: 0897917464. DOI: [10.1145/237170.237267](https://doi.org/10.1145/237170.237267).
- [9] W. E. Lorensen i H. E. Cline, „Marching Cubes: A High Resolution 3D Surface Construction Algorithm,” w *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’87, New York, NY, USA: Association for Computing Machinery, 1987, s. 163–169, ISBN: 0897912276. DOI: [10.1145/37401.37422](https://doi.org/10.1145/37401.37422).
- [10] P. M. Sutton, C. D. Hansen, H.-W. Shen i D. Schikore, „A Case Study of Isosurface Extraction Algorithm Performance,” w *Data Visualization 2000*, W. C. de Leeuw i R. van Liere, red., Vienna: Springer Vienna, 2000, s. 259–268, ISBN: 978-3-7091-6783-0.
- [11] S. Raman i R. Wenger, „Quality Isosurface Mesh Generation Using an Extended Marching Cubes Lookup Table,” *Comput. Graph. Forum*, t. 27, s. 791–798, maj 2008. DOI: [10.1111/j.1467-8659.2008.01209.x](https://doi.org/10.1111/j.1467-8659.2008.01209.x).
- [12] G. Nielson, „Dual marching cubes,” w *IEEE Visualization 2004*, 2004, s. 489–496. DOI: [10.1109/VISUAL.2004.28](https://doi.org/10.1109/VISUAL.2004.28).
- [13] G. Treece, R. Prager i A. Gee, „Regularised marching tetrahedra: improved iso-surface extraction,” *Computers & Graphics*, t. 23, nr. 4, s. 583–598, 1999, ISSN: 0097-8493. DOI: [10.1016/S0097-8493\(99\)00076-X](https://doi.org/10.1016/S0097-8493(99)00076-X).
- [14] S. Frisken, „Constrained Elastic Surface Nets: Generating Smooth Surfaces from Binary Segmented Data.,” sty. 1998, s. 888–898.
- [15] T. Ju, F. Losasso, S. Schaefer i J. Warren, „Dual Contouring of Hermite Data,” w *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’02, San Antonio, Texas: Association for Computing Machinery, 2002, s. 339–346, ISBN: 1581135211. DOI: [10.1145/566570.566586](https://doi.org/10.1145/566570.566586).

- [16] D. Kayser, *How Ghost Ship Games found success with Deep Rock Galactic*, <https://www.unrealengine.com/en-US/spotlights/how-ghost-ship-games-found-success-with-deep-rock-galactic>, dostęp: 2023-07-10, 2019.
- [17] B. Foley, *Indie Developer Creates Breathtaking Viking Survival Experience*, [https://www.gaiusciochmagazine.com/articles/valheim\\_interview.html](https://www.gaiusciochmagazine.com/articles/valheim_interview.html), dostęp: 2023-07-10, 2018.
- [18] H. E. C. E. Lorensen, *System and method for the display of surface structures contained within the interior region of a solid body*, patent US06/741,390, 1985.
- [19] M. J. Dürst, „Re: Additional Reference to "Marching Cubes,"” *SIGGRAPH Comput. Graph.*, t. 22, nr. 5, s. 243, paź. 1988, ISSN: 0097-8930. DOI: 10.1145/378267.378271.
- [20] G. Nielson i B. Hamann, „The asymptotic decider: Resolving the ambiguity in marching cubes,” list. 1991, s. 83–91, 413, ISBN: 0-8186-2245-8. DOI: 10.1109/VISUAL.1991.175782.
- [21] R. Geiss, „Generating Complex Procedural Terrains Using the GPU,” w *GPU Gems 3*, NVIDIA Corporation, 2007, rozd. 1.
- [22] J. Peck, *FastNoise Lite*, <https://github.com/Auburn/FastNoiseLite>, dostęp: 2023-07-10, 2021.
- [23] R. Geiss i M. Thompson, „Cascades by NVIDIA,” w *Game Developers Conference*, 2007.
- [24] S. White, D. Batchelor, D. Coulter, M. Jacobs i M. Satran, *Graphics pipeline*, <https://learn.microsoft.com/pl-pl/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline?redirectedfrom=MSDN>, dostęp: 2023-07-10, 2022.
- [25] C. Dyken, G. Ziegler, C. Theobalt i H.-P. Seidel, „High-speed Marching Cubes using HistoPyramids,” *Comput. Graph. Forum*, t. 27, s. 2028–2039, grud. 2008. DOI: 10.1111/j.1467-8659.2008.01182.x.