



WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI
I INFORMATYKI

Imię i nazwisko studenta: Paweł Stolarski

Nr albumu: 176659

Poziom kształcenia: studia drugiego stopnia

Forma studiów: niestacjonarne

Kierunek studiów: Informatyka

Specjalność: Systemy i sieci komputerowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Analiza i rozwój metod reprezentacji licznych obiektów w symulatorach i grach wideo

Tytuł pracy w języku angielskim: Analysis and Development of Methods for Representing Numerous Objects in Simulators and Video Games

Opiekun pracy: dr inż. Mariusz Szwoch

OŚWIADCZENIE dotyczące pracy dyplomowej zatytułowanej: Analiza i rozwój metod reprezentacji licznych obiektów w symulatorach i grach video

Imię i nazwisko studenta: Paweł Stolarski
Data i miejsce urodzenia: 07.08.1998, Puck
Nr albumu: 176659

Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki

Kierunek: Informatyka

Poziom kształcenia: studia drugiego stopnia

Forma studiów: niestacjonarne

Typ pracy: praca dyplomowa magisterska

1. Oświadczenie dotyczące samodzielności pracy

Świadoma(y) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2022 r. poz. 2509, z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (t.j. Dz.U. z 2024 r. poz. 1571, z późn. zm.),¹ a także odpowiedzialności cywilnoprawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

2. Oświadczenie o stosowaniu narzędzi GenAI

wykorzystywałam/wykorzystywałem narzędzia o potencjalnie niskim stopniu ingerencji

Oświadczam, że treści wygenerowane przy pomocy GenAI poddałam / poddałem krytycznej analizie i zweryfikowałam / zweryfikowałem.

27.10.2025, Paweł Stolarski

Data i podpis lub uwierzytelnienie w portalu uczelnianym Moja PG

**) Dokument został sporządzony w systemie teleinformatycznym, na podstawie §15 ust. 3b Rozporządzenia MNiSW z dnia 12 maja 2020 r. zmieniającego rozporządzenie w sprawie studiów (Dz.U. z 2020 r. poz. 853). Nie wymaga podpisu ani stempla.*

¹ Ustawa z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce:

Art. 312. ust. 3. W przypadku podejrzenia popełnienia przez studenta czynu, o którym mowa w art. 287 ust. 2 pkt 1–5, rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 312. ust. 4. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 5, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez Komisję dyscyplinarną oraz składa zawiadomienie o podejrzeniu popełnienia przestępstwa.

STRESZCZENIE

Celem niniejszej pracy była ocena i rozwój metod przechowywania i renderowania dużych zbiorów prostych obiektów graficznych, ze szczególnym uwzględnieniem środowisk wokselowych. Zadaniem było nie tylko porównanie różnych struktur danych i technik optymalizacyjnych, ale także opracowanie oraz implementacja prototypu, który pozwoliłby na przeprowadzenie badań w realistycznych warunkach i ocenę praktycznej przydatności wybranych rozwiązań.

Na potrzeby badań przygotowano prototyp systemu generowania, renderowania i przetwarzania wokselowych światów napisany w języku *Rust* z wykorzystaniem biblioteki *Vulkan*. Prototyp został podzielony na moduły odpowiadające za: zarządzanie programem i główną pętlą, implementację generatorów światów testowych, obsługę pomiarów, implementację struktur danych, moduł renderowania wraz z shaderami, a także definicję podstawowych elementów świata wokselowego. Integralną częścią prototypu było jedenaście światów testowych – od prostych (pusty, pojedynczy woksel, świat połowicznie wypełniony) po proceduralne i realistyczne (oparte o szum Simplex i generowanie proceduralne). Prototyp stanowił jednocześnie platformę badawczą i narzędzie walidacji zastosowanych rozwiązań.

Badania przeprowadzono na dwóch platformach sprzętowych: komputerze klasy wyższej (wydajny komputer osobisty z dedykowaną kartą graficzną) oraz niższej (laptop ze zintegrowaną kartą graficzną). Analizy obejmowały: czas inicjalizacji, wypełniania i usuwania danych w strukturach, koszty pamięciowe, wydajność w scenariuszach renderowania światów oraz wpływ wybranych technik optymalizacyjnych (zastosowanie bryły widzenia i selekcji ścian, organizacja geometrii, wielowątkowość). Wyniki porównano także z referencyjnymi pomiarami z gry *Minecraft*, co pozwoliło na ocenę osiągniętego poziomu wydajności względem rozwiązania komercyjnego.

Najważniejsze wyniki pokazują, że struktura drzewa ósemkowego *octree* najlepiej spełnia wymagania stawiane w kontekście reprezentacji licznych obiektów w trójwymiarowych światach. Badania wykazały, że techniki optymalizacyjne – zwłaszcza bryła widzenia/obcinania oraz wielowątkowe generowanie danych – istotnie zwiększą wydajność, a przygotowany prototyp był w stanie osiągać wyniki porównywalne, a w niektórych przypadkach lepsze, niż zarejestrowane w grze referencyjnej.

Praca kończy się omówieniem ograniczeń badań oraz wskazaniem możliwych kierunków dalszego rozwoju, takich jak integracja bardziej zaawansowanych metod renderowania, wprowadzenie adaptacyjnych poziomów szczegółowości LOD (ang. *Level Of Details*) czy rozszerzenie modeli generowania światów proceduralnych.

Słowa kluczowe: symulacje, gry wideo, woksele, struktury danych, octree, drzewo ósemkowe, bryła widzenia, optymalizacja, Rust, Vulkan

Dziedzina nauki i techniki, zgodne z wymogami OECD: 1.2. Nauki o komputerach i informatyka

ABSTRACT

The aim of this thesis was to evaluate and develop methods for storing and rendering large sets of simple graphical objects, with particular emphasis on voxel-based environments. The task was not only to compare different data structures and optimization techniques but also to design and implement a prototype that would allow conducting experiments under realistic conditions and assessing the practical applicability of the chosen solutions.

For research purposes, a prototype system for generating, rendering, and processing voxel worlds was developed in the *Rust* programming language using the *Vulkan* library. The prototype was divided into modules responsible for: program management and the main loop, implementation of test world generators, performance measurement, data structure implementation, rendering (including shader modules), and the definition of fundamental elements of the voxel world. An integral part of the prototype consisted of eleven test worlds — ranging from simple ones (empty, single voxel, partially filled) to procedural and realistic ones (based on Simplex noise and procedural generation). The prototype served both as a research platform and as a validation tool for the applied techniques.

The experiments were conducted on two hardware platforms: a high-end computer (a desktop PC with a dedicated graphics card) and a lower-end system (a laptop with an integrated GPU). The analyses covered initialization time, data population and removal performance in different structures, memory usage, rendering performance in various scenarios, and the impact of selected optimization techniques (frustum culling and face selection, geometry organization, multithreading). The results were also compared with reference measurements from the game *Minecraft*, which allowed assessing the achieved performance relative to a commercial implementation.

The key findings show that the *octree* structure best meets the requirements for representing large numbers of objects in three-dimensional worlds. The study demonstrated that optimization techniques — particularly frustum culling and multithreaded data generation — significantly improve performance, and the developed prototype was capable of achieving results comparable to, and in some cases exceeding, those recorded in the reference game.

The thesis concludes with a discussion of the limitations of the conducted research and outlines possible directions for further development, such as integrating more advanced rendering methods, introducing adaptive Levels of Detail (LOD), and extending procedural world generation models.

Keywords: simulations, video games, voxels, octree, octree tree, data structures, frustum culling, optimization, Rust, Vulkan

Field of science and technology, according to OECD classification: 1.2. Computer and information sciences.

SPIS TREŚCI

Wykaz ważniejszych oznaczeń i skrótów	7
1. Wstęp i cel pracy	8
2. Wprowadzenie do dziedziny	9
2.1. Inspiracje	9
2.2. Badane środowiska	11
2.2.1. Pojęcia "symulacja" i "gra wideo"	11
2.3. Podstawy grafiki komputerowej	12
2.3.1. Grafika wektorowa a rastrowa	12
2.3.2. Grafika trójwymiarowa i jej związek z rasteryzacją	13
2.3.3. Problem wewnętrznie pustych modeli trójwymiarowych	15
2.3.4. Proces renderowania	15
2.4. Siatka świata	17
2.4.1. Siatki dwuwymiarowe (2D)	17
2.4.2. Siatki trójwymiarowe (3D)	17
2.4.3. Siatka świata gry Minecraft	18
2.4.4. Blok gry Minecraft	20
2.5. Przechowywanie danych	20
2.5.1. Badane wokselowe struktury danych	21
2.5.2. Wydajność wokselowych struktur danych	21
2.5.3. Wybrane struktury danych przechowujące woksele	22
3. Technologie i narzędzia	23
3.1. Przegląd języków programowania i bibliotek graficznych	23
3.2. Biblioteki graficzne	25
3.3. Algorytmy generowania szumów	25
3.4. Techniki podnoszenia wydajności	27
4. Projekt oprogramowania	29
4.1. Założenia i wymagania	29
4.1.1. Wymagania funkcjonalne	29
4.1.2. Wymagania niefunkcjonalne	30
4.2. Wokselowy typ przechowywanego świata	30
4.2.1. Implementacja woksela	31
4.3. Wybór środowiska programistycznego i narzędzi	32
4.3.1. Biblioteki wspierające prototyp	32
4.4. Algorytmy, techniki i struktury prototypu	33
4.5. Architektura oprogramowania	34

4.6. Opis działania prototypu	36
4.6.1. Konfiguracja uruchomienia	36
4.6.2. Sterowanie i interakcja	37
4.6.3. Funkcje badawcze	37
4.7. Światy testowe	38
4.8. Zaimplementowane narzędzia pomiarowe	45
4.8.1. Pomiar czasu działania (Criterion)	45
4.8.2. Pomiar zużycia pamięci (dhat)	46
5. Analiza wydajności i właściwości struktur danych	47
5.1. Platformy sprzętowe	48
5.2. Porównanie referencyjne z grą Minecraft	48
5.3. Badania struktur	49
5.3.1. Pomiar czasu inicjalizacji struktur	50
5.3.2. Pomiar czasu dodawania danych	52
5.3.3. Pomiar czasu usuwania danych	54
5.3.4. Pomiar czasu wyszukiwania danych	56
5.3.5. Pomiar rozmiaru w pamięci operacyjnej	58
5.3.6. Analiza wyników i wnioski	60
5.4. Badania na podstawie światów testowych	61
5.4.1. Badania światów na strukturze octree	62
5.4.2. Badanie wpływu wybranych technik optymalizacyjnych	64
6. Podsumowanie	66
6.1. Cel pracy i jego realizacja	66
6.2. Najważniejsze wyniki badań	67
6.3. Ograniczenia badań	68
6.4. Możliwe kierunki dalszych prac	69
Wykaz literatury	76
Wykaz rysunków	82
Wykaz tabel	85
A. Wyniki pomiarów struktur na platformie klasy wyższej	86

WYKAZ WAŻNIEJSZYCH SKRÓTÓW I OZNACZEŃ

- **AABB** (ang. *Axis-Aligned Bounding Box*) – bryła brzegowa wyrównana do osi
- **API** (ang. *Application Programming Interface*) – interfejs programistyczny aplikacji
- **BSP** (ang. *Binary Space Partitioning*) – binarne dzielenie przestrzeni
- **CPU** (ang. *Central Processing Unit*) – Centralna jednostka przetwarzania, procesor
- **DRAM** (ang. *Dynamic Random-Access Memory*) – pamięć dynamiczna o dostępie swobodnym
- **FPS** (ang. *Frames Per Second*) – liczba klatek na sekundę
- **GiB** (ang. *Gibibyte*) – gibabajt, 2^{30} bajtów
- **GPU** (ang. *Graphics Processing Unit*) – jednostka przetwarzania grafiki, karta graficzna
- **GUI** (ang. *Graphical User Interface*) – graficzny interfejs użytkownika
- **KiB** (ang. *Kibibyte*) – kibabajt, 2^{10} bajtów
- **LOD** (ang. *Level Of Details*) – poziom szczegółowości
- **MOB** (ang. *Mobile Object*) – obiekt mobilny, poruszający się
- **OS** (ang. *Operating System*) – system operacyjny
- **PDF** (ang. *Probability Density Function*) – funkcja/rozkład gęstości prawdopodobieństwa
- **RAM** (ang. *Random-Access Memory*) – pamięć o dostępie swobodnym
- **RLE** (ang. *Run-Length Encoding*) – kodowanie długości serii
- **SVG** (ang. *Scalable Vector Graphics*) – skalowalna grafika wektorowa
- **TUI** (ang. *Text-based User Interface*) – tekstowy interfejs użytkownika

1. WSTĘP I CEL PRACY

Współczesne gry wideo oraz komputerowe symulacje stawiają coraz większe wymagania wobec mocy obliczeniowej i pamięci operacyjnej, stając się coraz bardziej złożonymi zarówno pod względem funkcjonalnym, jak i graficznym. Niezależnie od zastosowania, każdy widoczny czy interaktywny element musi zostać odpowiednio odwzorowany i zarządzany, aby zapewnić płynność działania oraz wrażenie autentyczności w prezentowanej, symulowanej scenie.

Rozwój metod reprezentacji obiektów w grach i symulacjach napędzany jest zarówno postępem technologicznym, jak i potrzebą sprostania oczekiwaniom użytkowników w warunkach ekonomicznych ograniczeń. Zarówno twórcy gier¹ jak i badacze² muszą nieustannie godzić oczekiwania odbiorców z ograniczeniami sprzętowymi. Dążenie do poprawy wydajności, redukcji zapotrzebowania na pamięć oraz tworzenia bardziej realistycznych wizualizacji prowadzi do innowacji w dziedzinie algorytmów i struktur danych.

Celem niniejszej pracy jest analiza, porównanie oraz rozwój wybranych struktur danych oraz technik optymalizacji wydajności przeznaczonych do przechowywania i reprezentacji obiektów w trójwymiarowych środowiskach. Praca koncentruje się na poszukiwaniu rozwiązań umożliwiających zwiększenie wydajności oraz ograniczenie zapotrzebowania na pamięć operacyjną, przy jednoczesnym uwzględnieniu możliwości współczesnych języków programowania oraz bibliotek wspierających renderowanie grafiki.

Szczególną uwagę poświęcono w pracy wokselom³, czyli przestrzennym odpowiednikom pikseli. Zyskują one coraz większą popularność jako sposób reprezentacji danych trójwymiarowych, znajdująąc zastosowanie w algorytmice (np. w kompresji i przechowywaniu danych), w medycynie (tomografia komputerowa, rezonans magnetyczny, obrazy fluoroskopowe), w grafice komputerowej, a także w grach i środowiskach symulacyjnych. Zależność pomiędzy rozmiarem wokseli a ilością przechowywanej informacji sprawia, że stanowią one niezwykle elastyczny i potężny sposób modelowania przestrzeni, dzięki czemu stanowią podstawowy budulec świata analizowanego i rozwijanego w niniejszej pracy.

W rozdziale 2 przedstawiono teoretyczne podstawy dziedziny, obejmujące zagadnienia grafiki komputerowej, sposobów reprezentacji przestrzeni oraz przechowywania danych w aplikacjach symulacyjnych. Następnie, w rozdziale 3, zaprezentowano przegląd języków programowania, bibliotek graficznych, algorytmów generatywnych oraz technik podnoszących wydajność programu. W rozdziale 4 opisano prototyp środowiska badawczego opartego na świecie wokselowym, wykorzystującego wybrane technologię i narzędziach przedstawione w poprzedzających rozdziałach. W rozdziale 5 zaprezentowano wyniki badań porównawczych tych struktur i technik w różnych konfiguracjach świata, analizując ich wpływ na wydajność i zużycie zasobów. Praca kończy się omówieniem uzyskanych rezultatów, ograniczeń badań oraz wskazaniem możliwych kierunków dalszego rozwoju systemu.

¹Przykłady gier: [Minecraft](#), [Noita](#), [Factorio](#), [CubeWorld](#), [Teardown](#), [Terraria](#), [Super Mario Bros](#)

²Przykłady symulacji i modeli: [zbiornika](#), [przepływu wody](#), [dystrybucji danych](#), [rozchodzenia fal](#), [rośnięcia struktur](#), [MRI](#)

³<https://en.wikipedia.org/wiki/Voxel>

2. WPROWADZENIE DO DZIEDZINY

Wirtualne środowiska gier wideo i symulacji, są złożonymi systemami, których podstawowym budulcem są różnego rodzaju obiekty (ang *objects*). Pojęcie to odnosi się do każdego elementu, który istnieje w wirtualnym świecie i może być renderowany, przetwarzany lub interaktywny. Grupa obiektów – renderowanych i nierenderowanych (np. model samochodu, źródła oświetlenia, kamera) wraz ze specyfikacją ich rozmieszczenia w przestrzeni określana jest mianem sceny (ang. *scene*) [1].

Rozwój metod reprezentacji danych, w tym obiektów w grach i symulacjach od wielu lat stanowi jeden z kluczowych obszarów badań w informatyce stosowanej [2, 3, 4, 5]. Wydajność oraz jakość odwzorowania świata wirtualnego zależy nie tylko od mocy obliczeniowej współczesnych procesorów i kart graficznych, lecz także od doboru odpowiednich struktur danych, technik renderowania oraz metod optymalizacji.

Celem niniejszego rozdziału jest przedstawienie podstaw teoretycznych i technologicznych związanych z dziedziną symulacji komputerowych oraz gier wideo. Omówione zostaną kluczowe pojęcia, zasady działania grafiki komputerowej, a także sposoby organizacji przestrzeni i danych w strukturach danych.

2.1. *Inspiracje*

Inspiracją do podjęcia badań nad metodami reprezentacji były między innymi trzy gry komputerowe, które łączą wyzwania reprezentacji danych oraz wymagania wysokiej wydajności niezbędne do utrzymania symulacji o dużej proceduralnie generowanej skali lub szczegółowości. Zagadnienia te prowokują do pogłębienia wiedzy oraz przeprowadzenia badań nad sposobami obsługi takich środowisk. Przytaczanymi grami są:

- **Minecraft**¹ (rys. 2.1) – gra typu *piaskownica* (ang: *sandbox*), w której gracz swobodnie eksploruje i modyfikuje świat zbudowany z bloków. Inspiruje możliwością dowolnej ingerencji w teren oraz obserwacja problemów związanych z jego reprezentacją.
- **Factorio**² (rys. 2.2) – gra oparta na budowaniu i automatyzacji fabryk, charakteryzująca się generowaniem ogromnej liczby obiektów w krótkim czasie. Stanowi przykład systemu wymagającego wysokiej wydajności obliczeniowej.
- **Noita**³ (rys. 2.3) – dwuwymiarowa gra typu *roguelite*, w której "każdy piksel jest symulowany". Oddaje potencjał jak i trudność połączenia obu powyższych produkcji w ograniczonej skali.

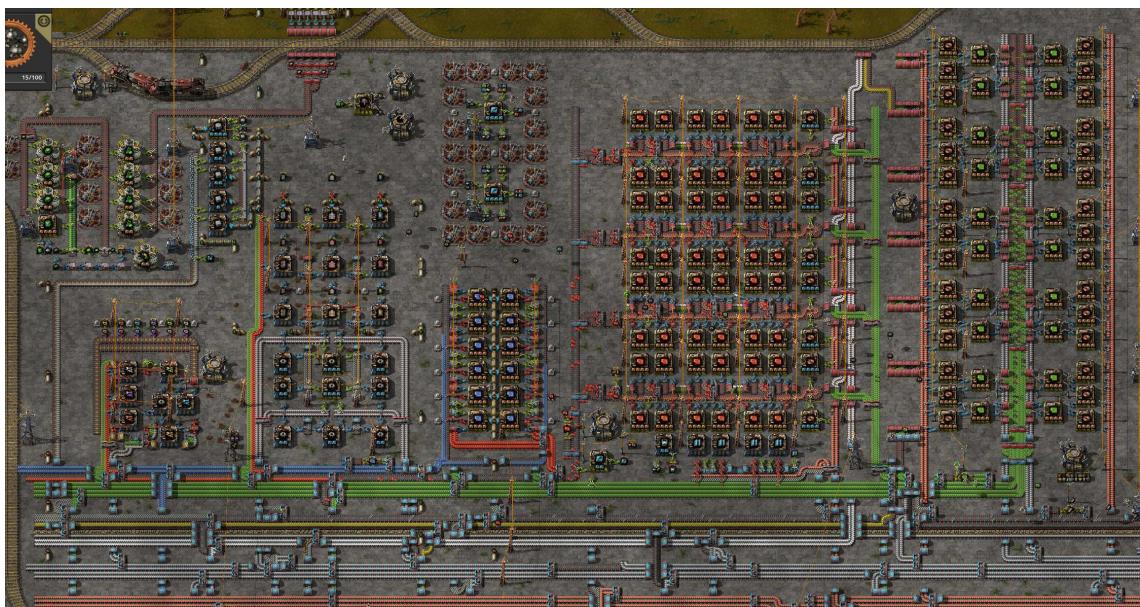
¹<https://www.minecraft.net/pl-pl/article/what-minecraft>

²<https://factorio.com>

³<https://noitagame.com>



Rys. 2.1: Panorama świata gry *Minecraft* z widocznymi budowlami graczy oraz rozległym horyzontem.
[Opracowanie własne na podstawie gry *Minecraft* [6]]



Rys. 2.2: Gra *Factorio*, z widocznymi setkami bytów/przedmiotów poruszającymi się i przetwarzanymi w czasie rzeczywistym. [Opracowanie własne na podstawie gry *Factorio* [7]]



Rys. 2.3: Zrzut ekranu z gry *Noita* z widocznymi symulowanymi pikselami cieczy (pomarańczowa lava, łososiowa mikstura), gazów (łososiowe piksele nad miksturą), portalu (czerwone koło), szczątków (zalewany pod portalem lawą kolor oliwkowy), głównej postaci (postać w centrum ilustracji), korodującego otoczenia (okołobieżowe platformy), tła. [Opracowanie własne na podstawie gry *Noita* [8]]

2.2. Badane środowiska

Podstawą prowadzenia badań nad metodami reprezentacji obiektów jest odpowiednio zdefiniowane środowisko, w którym przeprowadzane są eksperymenty. Środowisko to należy rozumieć jako cyfrową przestrzeń wraz z regułami jej funkcjonowania, oprogramowaniem realizującym logikę symulacji lub gry, a także sposobem prezentacji wyników na ekranie komputera. W niniejszej sekcji sprecyzowane zostaną pojęcia, zakres oraz charakterystyka świata, który w dalszej części pracy będzie stanowił podstawę analiz.

2.2.1. Pojęcia "symulacja" i "gra wideo"

W literaturze pojęcia *symulacji* oraz *gra wideo* posiadają szerokie znaczenie, które w niniejszej pracy należy zawęzić:

Symulacja (ang. *simulation*) – proces odwzorowania rzeczywistych zjawisk na podstawie określonego modelu (matematycznego, fizycznego, myślowego lub komputerowego). W dalszych rozważaniach przyjmuje się znaczenie **symulacji komputerowej** (ang. *computer simulation*) – komputerowo implementowanej metody służącej do badania rzeczywistości. Przydaje się na przykład w sytuacjach, gdy metody analityczne są niemożliwe lub niepraktyczne do zastosowania [9, 10].

Gra wideo (ang. *video game*) – program komputerowy służący rozrywce, charakteryzujący się określonym zestawem reguł interakcji oraz celem. W niniejszej pracy termin "gra" będzie odnosił się do produkcji wykorzystujących grafikę trójwymiarową [11].

2.3. Podstawy grafiki komputerowej

Sekcja ta omawia pojęcia oraz nakreśla środowisko graficzne będącego punktem wyjścia dla tworzonego w ramach pracy prototypu oraz prowadzenie badań.

Zarówno symulacje, jak i gry wideo działają w sposób podobny: przetwarzają logikę wewnętrzną i prezentują jej rezultaty. Prezentacja wyników może przybierać różne formy, a najbardziej charakterystyczne to:

- **interfejs tekstowy TUI** (ang. *Text-based User Interface*) – wyjście w postaci znaków i tekstu,
- **interfejs graficzny GUI** (ang. *Graphical User Interface*) – od prostej grafiki 2D po złożone sceny 3D z oświetleniem i efektami wizualnymi.

2.3.1. Grafika wektorowa a rastrowa

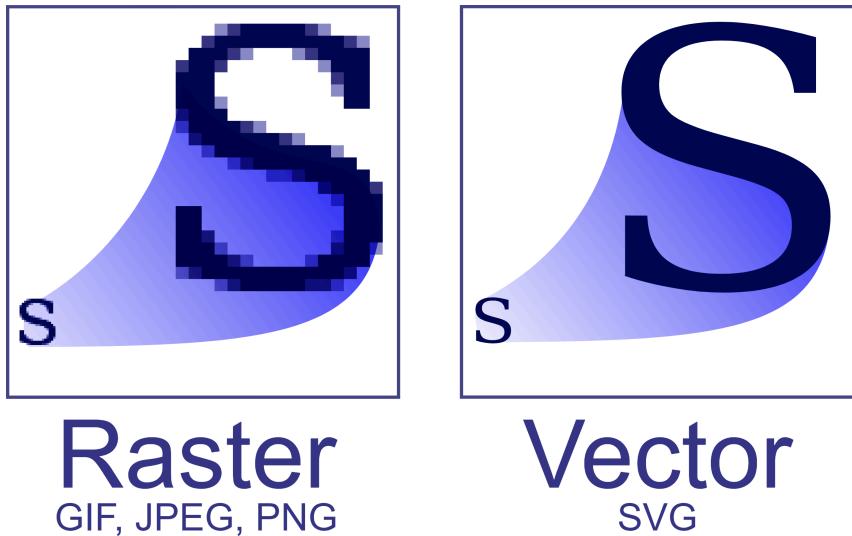
W ogólności można wyróżnić dwa podstawowe podejścia do tworzenia obrazu: grafikę rastrową oraz grafikę wektorową. Oba podejścia mają odmienne właściwości, a wybór jednego z nich jest związany z charakterem aplikacji oraz wymaganiami wydajnościowymi.

Grafika rastrowa (ang. *raster graphic*) [12] – obraz definiowany jest jako tablica (siatka) pikseli, gdzie każdy piksel posiada określony kolor. Rozdzielcość obrazu jest bezpośrednio powiązana z jego jakością – powiększanie prowadzi do utraty szczegółów i widocznej pikselizacji. Typowym przykładem zastosowania grafiki rastrowej są zdjęcia cyfrowe, tekstury w grach komputerowych oraz obrazy renderowane przez karty graficzne.

Grafika wektorowa (ang. *vector graphic*) [12] – obraz opisywany jest za pomocą obiektów geometrycznych (punktów, linii, krzywych, wielokątów) oraz atrybutów takich jak kolor czy grubość linii. Zaletą tego podejścia jest skalowalność – obrazy wektorowe mogą być dowolnie powiększane bez utraty jakości. Wektorowe metody reprezentacji znajdują zastosowanie w grafice inżynierskiej, interfejsach użytkownika czy wizualizacjach schematów.

Różnice pomiędzy obydwooma podejściami przedstawiono na rysunku 2.4. Po lewej stronie zaprezentowano fragment obrazu w formie rastrowej, natomiast po prawej – jego odpowiednik zapisany w postaci wektorowej.

Trójwymiarowe systemy symulacyjne i gry komputerowe często łączą oba podejścia. Świat trójwymiarowy renderowany jest w postaci rastrowej (piksele na ekranie), ale jego wewnętrzna reprezentacja – np. modele 3D – oparta jest na geometrii wektorowej [1, rozdz. 2.3.1, dz.6]. Taki hybrydowy model pozwala na efektywne łączenie wysokiej jakości wizualizacji z możliwością dynamicznych przekształceń obiektów [1, rozdz. 4.5].



Rys. 2.4: Porównanie obrazu rastrowego (po lewej) oraz wektorowego (po prawej). [Wikimedia Commons⁴]

2.3.2. Grafika trójwymiarowa i jej związek z rasteryzacją

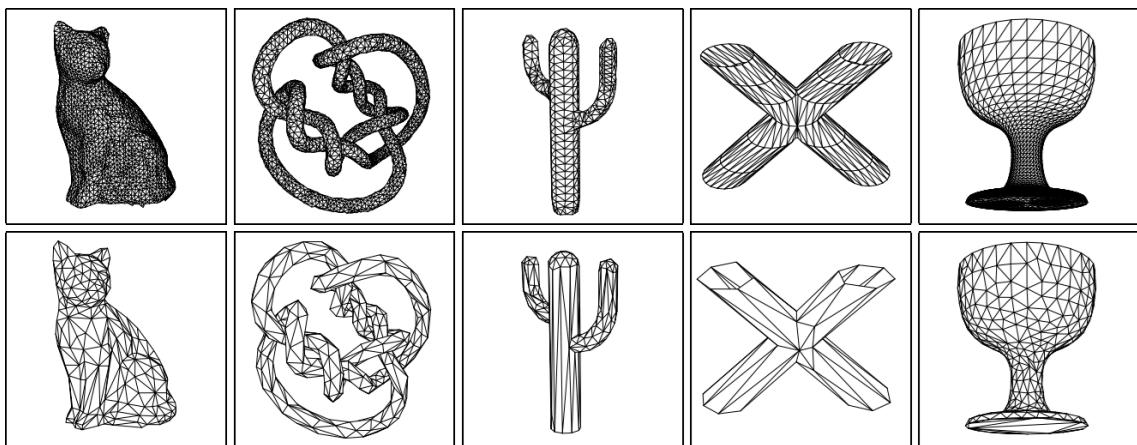
Modele 3D można traktować jako naturalne rozszerzenie grafiki wektorowej: obiekty nie są już opisywane krzywymi i prostymi, lecz siatką wielokątów (najczęściej trójkątów). Stopień złożoności siatki określa szczegółowość modelu, co przedstawiają różne liczebności wielokątów w modelach. Uproszczoną wizualizację różnic siatek o niskiej i wysokiej liczbie wielokątów przedstawiono na rysunku 2.5. Poniżej przedstawiono ogólną charakterystykę tych kategorii:

- Niska liczba wielokątów (ang. *low poly*) odpowiada za proste modele i dostrzegalne kanciaste kształty. Modele takie mogą liczyć od kilku do kilkuset lub paru tysięcy wielokątów [13] (rys. 2.6).
- Średnia liczba wielokątów (ang. *mid-poly*) stanowi balans między wydajnością a szczegółowością. Modele te posiadają wystarczającą ilość wielokątów, aby wiernie oddać kształt obiektu, ale nie są szczególnie obciążające obliczeniowo. Mogą liczyć od kilku tysięcy do kilkudziesięciu tysięcy wielokątów. Jest to najczęściej stosowany typ siatki w grach wideo⁵.
- Wysoka liczba wielokątów (ang. *high poly*) przekłada się na wysoką dokładność i bogatą szczegółowość. Modele takie mogą być zbudowane z dziesiątek/setek tysięcy, a nawet milionów wielokątów [13] (rys. 2.7).

Należy mieć na uwadze, że ekran komputera pozostaje urządzeniem rastrowym. Proces wizualizacji modeli trójwymiarowych wymaga etapu *rasteryzacji*, czyli przekształcenia opisanej geometrycznie sceny na piksele wyświetlane na monitorze. Aby zwiększyć realizm obrazu, na powierzchnie modeli nakładane są tekstury rastrowe, co sprawia, że grafika trójwymiarowa w praktyce łączy oba podejścia: wektorowe (geometria siatki) oraz rastrowe (tekstury i ostateczny obraz wyjściowy) [14, 1].

⁴https://commons.wikimedia.org/wiki/File:Bitmap_VS_SVG.svg

⁵<https://daim1993.wordpress.com/.../creating-3d-models-for-game-assets-a-comprehensive-guide/>
<https://www.artstation.com/.../midpoly-the-ultimate-guide-with-all-working-nuances>
<https://80.lv/articles/creating-assets-within-the-mid-poly-workflow-in-ue5>



Rys. 2.5: Przykłady siatek o zagęszczeniu większym (rzad górny) i mniejszym (rzad dolny) oddające różnicę między grafikami *high* oraz *low poly* [15]



Rys. 2.6: Ujęcie z gry *Poly Bridge* jako przykład grafiki *low poly*.⁶



Rys. 2.7: Ujęcie z wprowadzenia do gry *Mafia: Edycja ostateczna* jako przykład grafiki *high poly*⁷. Uwspółcześniona wersja gry wykładowczo podniosła liczbę wielokątów w używanych modelach⁸

⁶https://store.steampowered.com/app/367450/Poly_Bridge/

⁷<https://mafia.2k.com/pl-PL/mafia/>

⁸<https://gamingbolt.com/mafia-definitive-edition-vs-original...>

2.3.3. Problem wewnętrznie pustych modeli trójwymiarowych

Typowe modele trójwymiarowe stosowane w grafice komputerowej oparte są na siatkach wielokątów (ang. *polygon meshes*), które opisują jedynie powierzchnię obiektu. Tego rodzaju reprezentacja jest wystarczająca w kontekście renderowania obrazu, ponieważ dla celów wizualizacji kluczowa jest jedynie informacja o granicach widocznych dla obserwatora. W rezultacie większość modeli jest wewnętrznie pusta i nie zawiera żadnych danych o wnętrzu obiektu. Takie podejście do tworzenia modeli trójwymiarowych określa się mianem *modelowania powierzchniowego* (ang. *surface modeling*) [16].

W przypadku symulacji komputerowych takie podejście okazuje się niewystarczające. Przykładowo, odwzorowanie procesów fizycznych (przepływu płynów, propagacji fal, przenoszenia ciepła czy destrukcji materiałów) wymaga wiedzy o objętości, a nie wyłącznie o powierzchni. Podobne wyzwania występują również w grach wideo o charakterze woxelowym (np. *Minecraft*), gdzie gracz może ingerować w strukturę terenu nie tylko na jego powierzchni, lecz także w jego wnętrzu – reprezentacja świata tego typu określa się *modelowaniem woxelowym* (ang. *voxel modeling*) [17, 18].

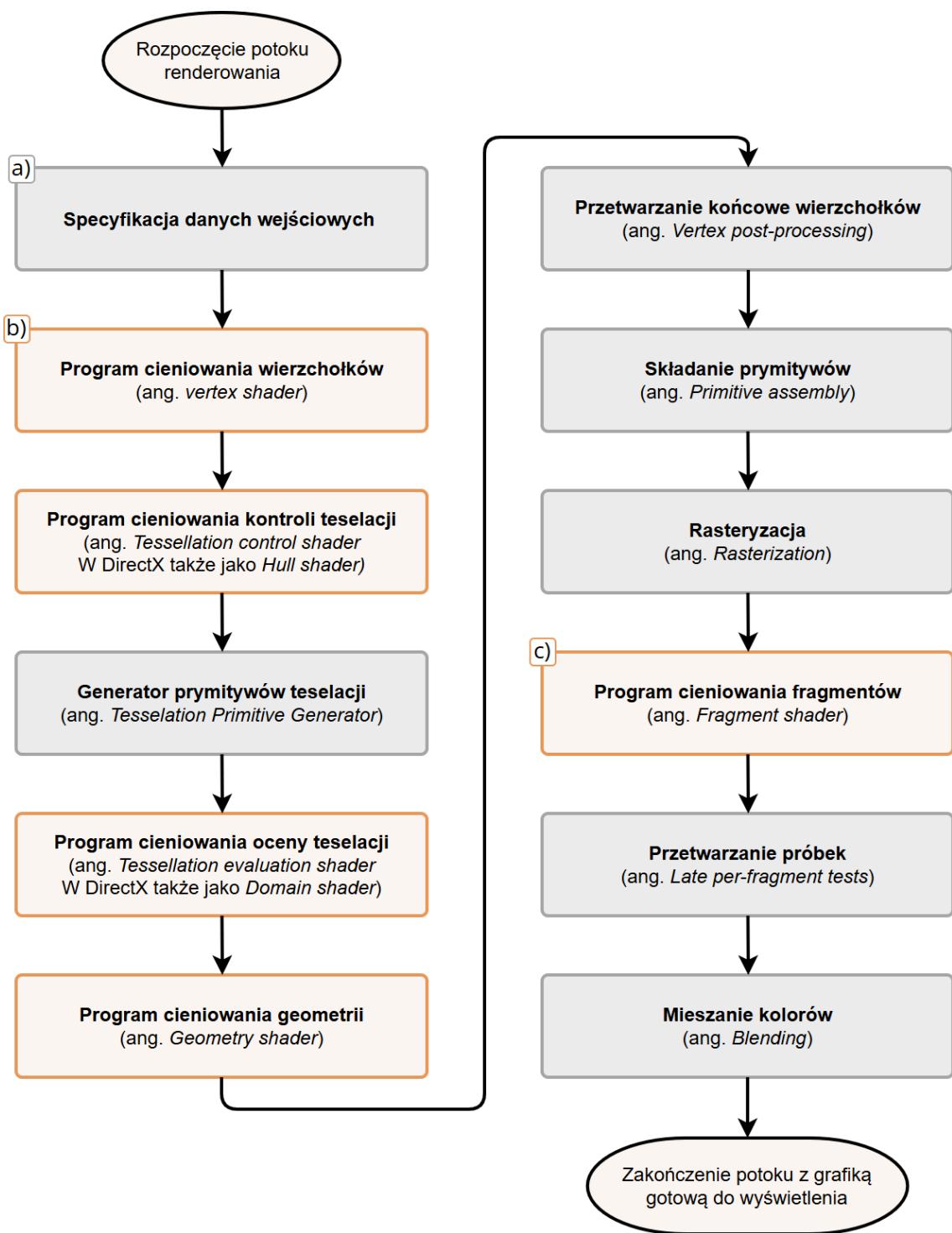
Woxel (ang. *voxel*, od *volumetric pixel*) stanowi trójwymiarowy odpowiednik piksela i opisuje niewielki fragment przestrzeni w zadanej siatce. W przeciwieństwie do klasycznych modeli siatkowych, które odwzorowują jedynie powierzchnię obiektu, woksele pozwalają na jednoznaczne odwzorowanie wnętrza bryły.

Świat woxelowy posiada charakterystyczne właściwości, które można podzielić na zalety i wady, do których należą:

- + Pełna informacja o każdej części przestrzeni, co eliminuje problem „pustych” modeli.
- + Możliwość dowolnej modyfikacji poszczególnych elementów.
- + Homogeniczny teren znacząco upraszczający wykonywanie operacji boolowskich i geometrycznych (przekształceń afiničnych).
- Ograniczona gładkość i naturalność struktur (świat jest dyskretny).
- Ograniczona jakość wizualna wynikająca z widocznej siatki wokseli.
- Wysoka złożoność pamięciowa powodowana ogromem informacji.

2.3.4. Proces renderowania

Renderowanie (ang. *rendering*) grafiki to proces przekształcania danych opisujących scenę – takich jak położenie, kolor czy parametry materiałów – w dwuwymiarowy obraz wyświetlany na ekranie [1, s. 11]. Podstawową jednostką renderowania są prymitywy renderowania (ang. *rendering primitives*) – punkty, linie, trójkąty [1, s. 8]. Po przejściu przez kolejne etapy potoku renderowania generują widok uwzględniający m.in. geometrię obiektów, oświetlenie, ustawienia kamery czy efekty wizualne [1, rozdz. 2]. Schemat wysokopoziomowego potoku renderowania przedstawiono na rysunku 2.8.



Rys. 2.8: Przykładowy potok renderowania. Wyróżnione kolorem o odcięciu pomarańczowym etapy są programowalne. Programy cieniowania wierzchołków (b) oraz fragmentów (c) są podstawowymi elementami potoku. Operacje od etapu (b) włącznie zachodzą w całości na karcie graficznej. [Opracowanie własne na podstawie materiałów grupy Khronos [19], *Graphics Programming Compendium* [20], wykładu *Interactive Graphics 18 - Tessellation Shaders* [21]]

2.4. Siatka świata

Symulacje komputerowe oraz gry wideo mogą korzystać z tak zwanej *siatki świata* (ang. *world grid*), czyli sposobu reprezentacji przestrzeni, w której umieszczane są obiekty, w tym elementy otoczenia czy dane symulacyjne. Siatka definiuje sposób podziału przestrzeni na mniejsze, dyskretne jednostki, które umożliwiają efektywne przechowywanie, wyszukiwanie, przetwarzanie i dodawanie informacji o stanie tworzonego świata. W zależności od rodzaju symulacji i wymiarowości przestrzeni, stosowane są różne typy siatek.

2.4.1. Siatki dwuwymiarowe (2D)

Komórka siatki dwuwymiarowej może reprezentować na przykład pojedyncze piksele, kafelki (ang. *tiles*) lub pola mapy. Niektóre ze spotykanych typów siatek 2D to:

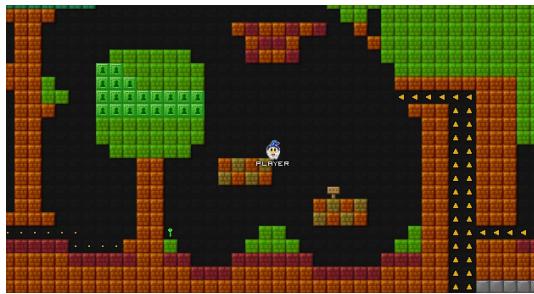
- **Siatka kwadratowa** (ang. *square grid*) [22] – przestrzeń dzielona jest na kwadraty co umożliwia łatwe obliczanie współrzędnych, prostą reprezentację danych i ich indeksację. Jest to najczęściej stosowany typ siatki w grach opartych na kafelkach (ang. *tile-based games*) [23] (rys. 2.9a).
- **Siatka trójkątna** (ang. *triangular grid*) [22] – może być stosowana w przypadku, gdy istotne jest dokładniejsze odwzorowanie położenia obiektów w przestrzeni (rys. 2.9b).
- **Siatka sześciokątna (heksagonalna)** (ang. *hexagonal grid*) [22] – często stosowana w środowiskach, w których pożądane jest bardziej naturalne czy zaaaansowane odwzorowanie sąsiedztwa. Sześciokątne komórki zapewniają równą odległość do wszystkich sąsiadów, co eliminuje problemy wynikające z różnicy dystansu w układzie prostokątnym (rys. 2.9c).
- **Siatka nieregularna** (2D) – przestrzeń dzielona jest na nieregularne wielokąty z użyciem np. triangulacji Delaunaya [24]. Umożliwia to dokładniejsze odwzorowanie złożonych kształtów, lecz wymaga bardziej złożonych metod obsługi.

2.4.2. Siatki trójwymiarowe (3D)

W przypadku siatek trójwymiarowych można wymienić następujące rodzaje:

- **Siatka sześcienna** (ang. *cube grid*) – przestrzeń dzielona jest na sześciany o identycznych wymiarach. Umożliwia prostą adresację wokseli za pomocą współrzędnych całkowitych [6].
- **Siatka czworościenna (tetrahedralna)** – przestrzeń dzielona jest na czworościany (tetraedry). Tego typu siatki stosuje się na przykład w analizach inżynierskich, gdzie wymagane jest precyzyjne modelowanie złożonych geometrii [25].
- **Siatka nieregularna** (3D) – przestrzeń dzielona jest na nieregularne wielokąty. Tego typu siatki stosuje się głównie w analizach inżynierskich (np. metoda elementów skończonych), gdzie wymagane jest precyzyjne modelowanie złożonych geometrii [24, 26]
- **Trójwymiarowe wersje siatek 2D** (3D) – figury geometryczne mogą zostać rozciągnięte w trzeci wymiar (np. wysokość)⁹

⁹<https://www.youtube.com/watch?v=b02Im1149tU&t=9s>



(a) Fragment zrzutu ekranu z gry *Everybody Edits* z widoczną kwadratową siatką świata. [Opracowanie własne na podstawie gry *Everybody Edits*¹⁰]



(b) Fragment klatki nagrania z gry *Zoo Tycoon 2* z widoczną trójkątną siatką świata¹¹.



(c) Gra *Civilization VI* z widoczną sześcienną siatką prezentowanego świata. [Opracowanie własne na podstawie gry *Civilization VI*¹²]

Rys. 2.9: Przedstawienie różnych typów siatek na przykładzie gier wideo – siatki kwadratowej (a), trójkątnej (b) i sześciennej (c).

2.4.3. Siatka świata gry *Minecraft*

Gra *Minecraft* jako najpopularniejsza gra świata¹³ stanowi dobry obiekt badawczy w temacie światów wokselowych. Świat tej gry opiera się na regularnej trójwymiarowej siatce bloków, której podstawowymi jednostkami hierarchii są:

1. **Blok** (ang. *block*) – najmniejsza jednostka budująca teren.
2. **Segment** (ang. *chunk*) – fragment świata o wymiarach $16 \times 384 \times 16$ bloków, czyli 98 304 jednostek. Gra generuje teren jedynie wzdłuż i wszerz, więc segment pokrywa całą wysokość świata

$$\underbrace{16}_{\text{szerokość}} \times \underbrace{384}_{\text{wysokość}} \times \underbrace{16}_{\text{długość}} = 98\ 304 \quad (2-1)$$

3. **Region** (ang. *region*) – zbiór 32×32 segmentów, co daje ponad 9.6×10^9 bloków.

$$\text{Bloki w segmencie} \\ \underbrace{98304^2} = 9\ 663\ 676\ 416 \quad (2-2)$$

4. **Zasięg renderowania** (ang. *render distance*) r – ustawienie określające kwadrat o boku $(2r + 1)$ segmentów, którego centrum stanowi pozycja obserwatora. Wartość ta decyduje o liczbie segmentów widocznych jednocześnie.

¹⁰<https://everybodyedits.com>

¹¹https://youtu.be/-u8Yw_26vHk?si=3Za9JyqJNNCpLF2E&t=213

¹²<https://civilization.2k.com/civ-vi/>

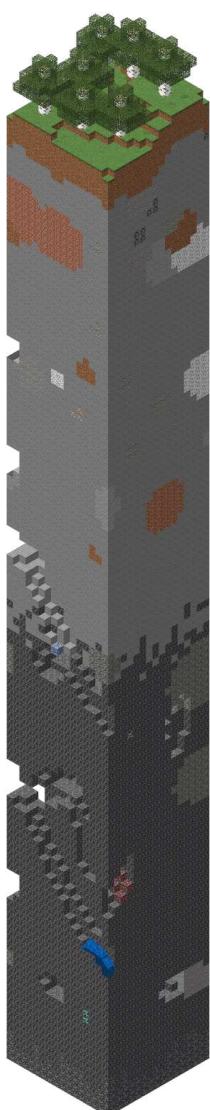
¹³https://en.wikipedia.org/wiki/List_of_best-selling_video_games

Liczبę przetwarzanych bloków możliwych do wyrenderowania w danej chwili w zależności od wartości parametru r można więc opisać wzorem:

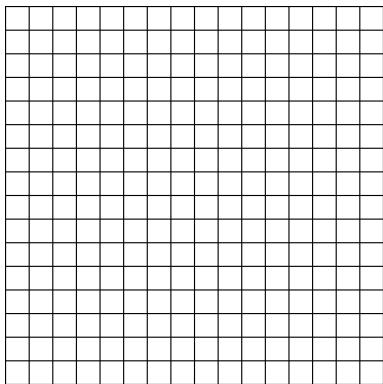
$$\text{Bloki w segmencie} \quad \overbrace{98\ 304}^{\text{ }} \times (2r + 1)^2 \quad (2-3)$$

Dla przykładu:

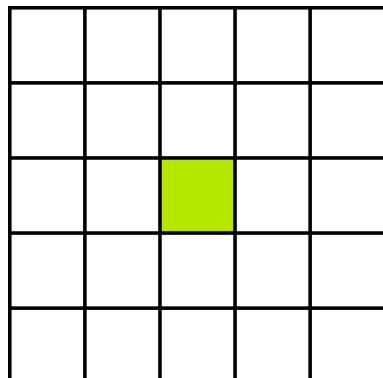
- $r = 2$: 2 457 600 bloków,
- $r = 16$: 107 053 056 bloków,
- $r = 32$: 415 334 400 bloków.



(a) Cały pojedynczy segment świata gry *Minecraft*. Segmente mają wymiary $16 \times 384 \times 16$ bloków¹⁴



(b) Siatka o wymiarach 16×16 obrazująca rzut z góry na segment świata gry *Minecraft*. [Opracowanie własne]



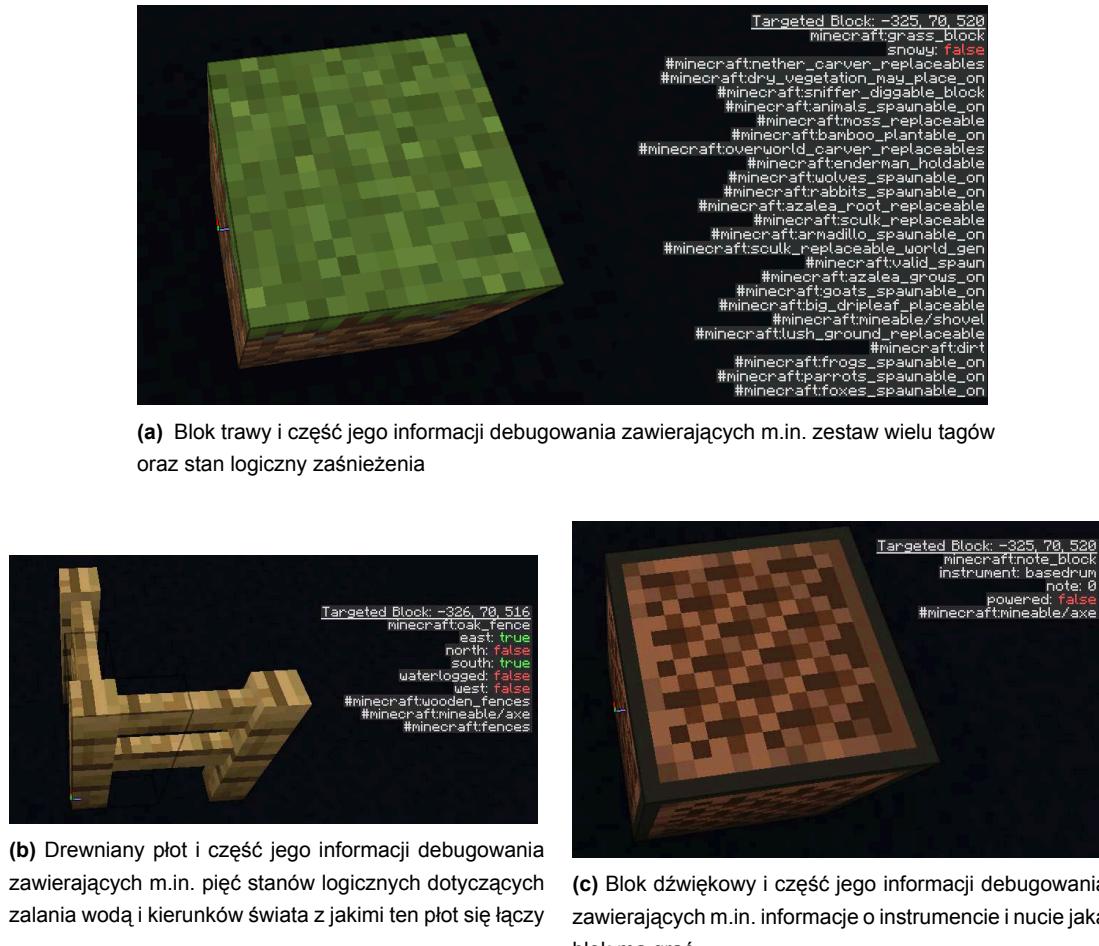
(c) Siatka ilustrująca liczbę przetwarzanych i możliwych do wyrenderowania segmentów świata gry *Minecraft* dla ustawienia zasięgu renderowania na "2". Komórka koloru limonkowego to segment w którym znajduje się obserwator/gracz. [Opracowanie własne]

Rys. 2.10: Przedstawienie segmentu świata gry *Minecraft* (a) wraz z siatką obrazującą widok z góry na segment (b) i liczbę segmentów dla zasięgu renderowania 2 segmentów (c).

¹⁴<https://minecraft.wiki/w/Chunk>

2.4.4. Blok gry Minecraft

Każdy blok w grze poza warstwą wizualną przechowuje także zestaw wewnętrznych informacji. Po włączeniu trybu debugowania gry można je podejrzeć. Kilka przykładów takich bloków i ich stanów przedstawiono na rysunku 2.11.



Rys. 2.11: Przykładowe bloki obecne w grze *Minecraft* wraz z fragmentem ich informacji debugowania. [Opracowanie własne na podstawie gry *Minecraft* [6]]

2.5. Przechowywanie danych

Przechowywanie danych cyfrowych obejmuje dwa główne obszary:

1. Pamięć dynamiczna DRAM (ang. *Dynamic Random-Access Memory*) – umożliwia zarządzanie stanem wraz z przeprowadzaniem obliczeń. Zdecydowana większość programów wymaga do swego działania np. tworzenia zmiennych.
2. Pamięć stała (dyski pamięci) – stosowana na przykład w celu przechowywania stanu między uruchomieniami programu.

W sytuacji, gdy program (symulacja lub gra) wymaga dużych ilości pamięci obu przytoczonych rodzajów, koniecznym jest umiejętne zarządzanie danymi, aby nie wyczerpać tych zasobów komputera w całości.

2.5.1. Badane wokselowe struktury danych

Do przechowywania scen 3D wykorzystujących woksele stosuje się różne struktury, z których najprostszą jest regularna (gęsta) siatka (ang. *dense regular grid*). Jest to struktura trójwymiarowej tablicy, zatem umożliwia szybki dostęp i lokalizację sąsiadów, ale wymaga ogromnych zasobów pamięci [17, rozdz. 4.2.1]. W celu zmniejszenia pamięciochłonności, używa się hierarchicznych struktur: drzew ósemkowych SVO (ang. *Sparse Voxel Octree*), które przechowują jedynie zajęte woksele i dostarczają mechanizmu poziomów szczegółowości LOD (ang. *level of detail*). SVO potrafią efektywnie obcinać niewidoczne gałęzie (ang. *culling, branch pruning*) i wspomagać mechanizmy wykrywania kolizji, jednak mają ograniczoną skalę – dla bardzo rozległych scen wymagają dużej przepustowości pamięci lub przechowywania danych poza pamięcią główną (ang. *out-of-core*) (np. budowy SVO modeli trójwymiarowych w czasie rzeczywistym na podstawie ich siatki) [17, rozdz. 4.2.1], [27].

Innym podejściem jest przestrzenne haszowanie wokseli – wykorzystanie prostej funkcji haszującej, która kompresuje przestrzeń i umożliwia szybki dostęp do wokselowych danych powierzchniowych. Przechowuje dane tylko tamdla tej przestrzeni, w której występują dane (np. pomiarowe z czujnika głębi) [28].

2.5.2. Wydajność wokselowych struktur danych

W literaturze dotyczącej współczesnych systemów wokselowych znaleźć można rozwiązania oparte na strukturze drzewa ósemkowego, które stanowią podstawę dla wydajnych frameworków renderujących i symulacyjnych. Wspólnym elementem tych podejść jest zastosowanie technik kompresji danych, adaptacyjnego podziału przestrzeni w ramach poziomów szczegółowości oraz mechanizmów strumieniowania danych, co pozwala na efektywne przetwarzanie dużych, otwartych światów [29, 30]. Analizy porównawcze różnych metod renderowania wolumetrycznego pokazują, że struktura *octree* nie tylko wspiera kompresję i LOD, ale również pełni kluczową rolę w przyspieszaniu procesu cieniowania i detekcji widoczności, co ma istotne znaczenie w kontekście gier wideo oraz symulacji trójwymiarowych [31].

W innych pracach podkreśla się znaczenie hybrydowych formatów wokselowych, które równoważą zużycie pamięci i wydajność renderowania, tworząc nowy kompromis w zakresie efektywności obliczeniowej. Szczególną uwagę poświęcono również wykorzystaniu wokseli w połączeniu z techniką śledzenia promieni, co umożliwia uzyskanie realistycznych efektów oświetleniowych przy zachowaniu wysokiej płynności generowanej grafiki [32]. Można odnaleźć także innowacyjne podejścia do tworzenia potoków renderowania – zamiast klasycznej rasteryzacji możliwym jest zastosowanie wokselowego rzutowania promieni. Podejście to pozwala na bezpośrednie odwzorowanie kierunków linii w strukturze wokselowej, zapewniając poprawną kolejność renderowania i wysoką efektywność w przypadku dużych, złożonych scen trójwymiarowych [33].

2.5.3. Wybrane struktury danych przechowujące woksele

W literaturze natrafić można na wiele odmian różnych struktur danych przechowujących dane wokselowe. Głównie są to struktury tablicowe, haszujące, drzewiaste. A do najważniejszych należą:

- **Mapa wokseli** (ang. *voxel map*) [34] – regularna n-wymiarowa tablica o stałych rozmiarach, w której każda komórka reprezentuje woksel tj. część przestrzeni. Istotnym jest, że taki woksel może reprezentować także pustą przestrzeń.
- **Siatka wolumetryczna** (ang. *Volumetric grid*) [35, 36] – szczególny przypadek mapy wokseli, która przechowuje dane objętościowe w przestrzeni 3D, takie jak gęstość, przepływy cieczy, czy inne właściwości fizyczne.
- **Lista wokseli** (ang. *voxel list*) [34] – tablica, która przechowuje tylko woksele "istotne", czyli reprezentujące zajęty teren. Tablica ta nie przechowuje więc informacji o przestrzeni niezajętej.
- **Drzewo czwórkowe** (ang. *quadtree*) [37, 38, 39] – struktura drzewiasta o rozgałęzieniach różnej głębokości, której liście symbolizują określony obiekt (np. woksele budujące teren), rekurencyjnie dzieląca się na prostokąty mapujące przestrzeń dwuwymiarową.
- **Drzewo ósemkowe** (ang. *octree*) [38, 39, 40] – struktura drzewiasta o rozgałęzieniach różnej głębokości, której liście symbolizują określony obiekt (np. woksele budujące teren), rekurencyjnie dzieląca się na graniastosłupy mapujące przestrzeń trójwymiarową. [41, 34]
- **Luźne drzewo ósemkowe/czwórkowe** (ang. *loose quadtree/octree*)¹⁵ [42] – Warianty drzew ósemkowego/czwórkowego, których liście reprezentujące przestrzeń nachodzą na siebie.
- **Polidrzewo** (ang. *polytree*)¹⁶ [40] – przeddefiniowane drzewo ósemkowe, które charakteryzuje się przechowywaniem konkretnych informacji o przestrzeni. Przechowuje takie informacje jak na przykład krawędzie, wierzchołki, czy płaszczyzny/ściany obiektów.
- **Drzewa BSP** (ang. *Binary Space Partitioning tree*) [39] – struktura dzieląca przestrzeń za pomocą płaszczyzn (lub linii w przypadku przestrzeni 2D)¹⁷
- **R-drzewo** (ang. *R-tree*)¹⁸, znane też jako "Dynamiczne drzewo AABB" (ang. *Dynamic AABB tree*) – Struktura drzewiasta, której rozgałęzienia tworzą minimalne regiony pokrywające dla obiektów wielowymiarowych. Liście tego drzewa stanowią bryłę brzegową AABB¹⁹ danego obiektu.
- **Haszowanie wokseli** (ang. *voxel hashing*) [28] – metoda przechowywania wokseli w tablicy haszującej, której indeks wyliczany jest na podstawie całkowitoliczbowych współrzędnych.

¹⁵Prezentacja opisująca "loose quadtree/octree"

¹⁶Nie należy mylić z polidrzewem w teorii grafów; <https://en.wikipedia.org/wiki/Polytree>

¹⁷Przykład BSP dla gry Quake

¹⁸Opis R-drzewa, Wizualizacja wspinania się po R-drzewie od liści do korzenia, wizualizacja poruszania się R-drzewa

¹⁹Wikipedia; Bryły brzegowe

3. TECHNOLOGIE I NARZĘDZIA

Przeprowadzenie badań nad metodami reprezentacji licznych obiektów w symulatorach i grach wideo wymaga doboru odpowiedniego zestawu technologii oraz narzędzi wspomagających proces implementacji i analizy. W niniejszym rozdziale przytoczone zostaną kwestie technologiczne, na bazie których w dalszej części pracy możliwe będzie przygotowanie prototypu aplikacji posłużącej realizacji badań oraz eksperymentów.

3.1. Przegląd języków programowania i bibliotek graficznych

Podczas przeglądu współczesnych języków programowania kierowano się wydajnością tworzonych w nich programów. Odrzucono języki skryptowe, interpretowane, ponieważ charakteryzują się niższą wydajnością w porównaniu do języków kompilowanych [43, 44]. Postawiono także nacisk na języki popularne, ponieważ ich popularność implikuje mnogość gotowych bibliotek, frameworków, czy wypracowanych rozwiązań. Na podstawie ankiety przeprowadzonej przez serwis StackOverflow [45, 46] oraz metryk serwisu GitHub [47] wyselekcjonowano najpopularniejsze języki kompilowane, ukazane w tabeli 3.1. Wybrano pozycje powyżej poziomu asemblera, uporządkowane według roku powstania.

Analiżę wydajności i bezpieczeństwa języków programowania oparto na licznych testach porównawczych, które pozwalają na wyłonienie kluczowych różnic architektonicznych i funkcjonalnych. W kwestii wydajności, języki C i C++, w różnych źródłach stanowią punkt odniesienia, często osiągając najlepsze lub porównywalne wyniki [48]. Kontrastują z nimi środowiska operujące na maszynach wirtualnych lub mocno polegające na dynamicznym zarządzaniu pamięcią, do których zaliczają się m.in. Java, C#, Dart czy Kotlin. W tych przypadkach, mechanizmy takie jak automatyczne odśmiecianie pamięci (ang. *garbage collector*) czy obsługa wyjątków (ang. *exceptions*) mogą negatywnie wpływać na stałość i szybkość osiągów, co skłania do ostrożnej klasyfikacji ich jako języków wydajnych [49, 50, 51, 52, 53]. Warto jednak zaznaczyć, że w przypadku niektórych z nich (np. Kotlin [54]) dostępne są opcje komplikacji natywnej, mogące poprawić osiągi.

Kwestia bezpieczeństwa kodu i minimalizacji podatności stanowi równie istotny element porównania [55, 56, 57, 58]. Język taki jak Rust jest projektowany z myślą o maksymalnej kontroli nad pamięcią i danymi, co czyni go najbezpieczniejszym w ogólnych zestawieniach. Natomiast starsze języki, takie jak C i C++, wymagają od programisty znacznie większej dyscypliny, co zwiększa ryzyko błędów pamięciowych [55, 56, 57, 58, 59]. Wiele popularnych języków boryka się z problemami niejednoznacznego przepływu sterowania przy obsłudze błędów oraz obecnością niebezpiecznych wartości null. Nowsze języki, na przykład Zig, starają się eliminować część tych problemów, choć nie deklarują się jako w pełni bezpieczne [59, 60]. Ogólna analiza pokazuje, że wybór języka programowania stanowi kompromis między maksymalną, niekiedy trudną do osiągnięcia, wydajnością a wbudowanymi mechanizmami bezpieczeństwa chroniącymi przed typowymi podatnościami.

Tabela 3.1: Zestawienie języków programowania

Nazwa	Rok pojawienia	Wynik komplikacji	Jest wysoce wydajny?	Zajmuje bardzo mało pamięci?	Czy bezpieczny?
C	1972 r. [61]	kod maszynowy	tak ¹	tak ¹	nie ^{9, 10, 11}
Objective-C	Około 1980 [62], brak rozwoju od roku 2012 [63]	kod maszynowy	raczej nie	brak danych raczej tak	brak pewnych danych
C++	1985 r. [64]	kod maszynowy	tak ¹	tak ¹	nie ^{9, 10, 11}
Java	1996 r. [65]	Java Bytecode, kod maszynowy [49, 66]	raczej nie ²	nie ²	raczej nie ^{10, 11}
C#	2002 r. [67]	Common Intermediate Language	raczej nie ³	nie ³	raczej nie ^{10, 11}
Dart	2011 r. [68]	kod maszynowy	raczej nie ⁵	nie ⁸	raczej nie ^{10, 11}
Kotlin	2012 r. [69]	Java Bytecode, kod maszynowy, inne [54]	raczej nie ⁶	zależy od trybu ⁶	raczej nie ¹²
Go	2012 r. [70]	kod maszynowy	raczej tak ^{1, 7}	raczej tak ^{1, 7}	raczej tak ¹²
Swift	2014 r. [71]	kod maszynowy	raczej nie ⁴	raczej nie ⁸	brak pewnych danych
Rust	2015 r. [72]	kod maszynowy	tak ¹	tak ¹	tak ¹³
Zig	2017 r. [73]	kod maszynowy	tak ¹	tak ¹	raczej nie ¹⁴

¹ We wszystkich przytaczanych porównaniach, wypada albo najwydajniej, albo zaniedbywalnie gorzej. Potwierdzają to także benchmarki [cpp-vs-c](#), [cpp-vs-go](#), [cpp-vs-rust](#), [cpp-vs-zig](#) [48].

² Język generalnie wypada słabiej w zestawieniach z językami C i C++, oraz przytacza się negatywny wpływ na wydajność takich mechanizmów jak np. dynamiczne zarządzanie pamięcią, zarządzanie wątkami czy wyjątkami [49]. Z tego względu przyjęto w tej pracy, że nie należy przydzielać językowi Java określenia języka wydajnego. Źródła wykazują różne osiągi [50, 51, 52, 53], w tym [benchmarki cpp-vs-java](#) [48].

³ Sytuacja funkcjonalności (dynamiczne zarządzanie pamięcią, zarządzanie wątkami czy wyjątkami) analogiczna jak z językiem Java [51, 52, 53], sugerują to także [benchmarki cpp-vs-csharp](#) [48].

⁴ Znalezione testy jednoznacznie wskazują na gorszą wydajność i objętość pamięciową języka Objective-C względem języka Swift, który to wypada gorzej lub porównywalnie do języków C czy C++ [74, 75, 76, 53, 77, 78], sugerują to także [benchmarki cpp-vs-swift](#) [48].

⁵ Sytuacja funkcjonalności analogiczna jak z językiem Java, sugerują to także [benchmarki cpp-vs-dart](#) [48].

⁶ Sytuacja funkcjonalności analogiczna jak z językiem Java, sugerują to także [benchmarki cpp-vs-kotlin](#) [48]. Na korzyść języka Kotlin wpływa użycie komplikacji "kotlin/native"

⁷ Język Go ma zauważalne skoki w zużyciu pamięci [79], oraz skoki spadku wydajności spowodowane działaniem mechanizmem odśmiecania [80].

⁸ Na podstawie benchmarków [cpp-vs-csharp](#) / [cpp-vs-dart](#) / [cpp-vs-swift](#)

⁹ Na podstawie porównań języków w których trudniej napisać podatny kod (C/C++ a Rust [55, 56, 57]; dokumentacja języka Rust wspominająca problemy z C/C++ [58])

¹⁰ Wartości mogą być wartością "null" (dokumentacja języka [C](#), [C++](#), [Kotlin](#) [w tym [java](#)], [C#](#))

¹¹ Niejednoznaczny przepływ sterowania, błąd nie jest wartością (dokumentacja języka [C++](#), [Kotlin](#), [C#](#), [Dart](#); kurs języka [Java](#))

¹² Nie znaleziono jednoznacznych potwierdzeń, że język Go nie jest bezpiecznym językiem, natomiast przeszukując Internet można zauważać niemal jednoznaczne stanowisko, że Go niekoniecznie jest bezpieczny, a na pewno nie tak bardzo jak język Rust [81, 82]

¹³ Wszystkie przytaczane zestawienia/porównania (w których występował język Rust) wskazywały język Rust jako najbezpieczniejszy. Z zestawień, w których języka Rust nie ma, można za pośrednictwem innych porównać dojść do rozstrzygnięcia który język jest bezpieczniejszy – ponownie na tym podium stoi język Rust.

¹⁴ Oficjalna witryna języka Zig mówi o tym, że język ten nie należy do najbezpieczniejszych [59] ("Please note that Zig is not a fully safe language."). Język ten nie został określony jasno jako niebezpieczny, ponieważ nie dotyczą go m.in. problemy niekontrolowanego przepływu sterowania czy nieoznaczone wartości "null" [60]

3.2. Biblioteki graficzne

Przegląd dostępnych bibliotek graficznych (ang. *graphics APIs*) wykazał, że rynek jest zdominowany przez kilka wiodących technologii, które przedstawiono w tabeli 3.2. Kolejność podykutowano datą wydania. Kolumna systemów operacyjnych pełni rolę orientacyjną w skali dostępności danej biblioteki.

Tabela 3.2: Zestawienie bibliotek graficznych [83]

Nazwa	Rok pojawienia	Na jakie systemy operacyjne
OpenGL	1992	Windows, Xbox, koniec wsparcia dla systemów Apple, Linux, Android
Metal	2014	MacOS, iOS
DirectX12	2015	Windows, Xbox
Vulkan	2016	Windows, MacOS i iOS (MoltenVK ¹), Linux, Android

Biblioteki *Metal* oraz *DirectX12* to rozwiązania wysokowydajne, jednak ograniczone do ekosystemów odpowiednio firmy Apple oraz Microsoft, co ogranicza ich przenośność. Największą elastyczność i neutralność względem platform zapewnia *Vulkan*, który wspiera systemy operacyjne Windows, Linux i Android, a dzięki projektowi *MoltenVK* także macOS oraz iOS.

Biblioteka *OpenGL* natomiast, mimo swojej długiej historii, jest rozwiązaniem mniej perspektywicznym ze względu na stopniowe wycofywanie. Wynika to m.in. z jej starszej architektury klient-serwer, która skutkuje większym narzutem pracy dla procesora, oraz trudność w uzyskaniu skalowania przy programowaniu równoległym/wielowątkowym [84, 85].

Biblioteka *Vulkan* jest rozwiązaniem nowszym i dającym większe możliwości. Jego niskopoziomowy interfejs daje większą i bezpośrednią kontrolę nad procesorem graficznym, pozwala na efektywne wykorzystanie wielu rdzeni procesora oraz umożliwia precyzyjne zarządzanie potokiem renderowania. Ta większa kontrola odbywa się kosztem wyższej złożoności implementacji [84, 85].

3.3. Algorytmy generowania szumów

Generowanie proceduralne jest techniką szeroko wykorzystywaną w symulacjach i grach wideo w celu tworzenia deterministycznych, dużych i zróżnicowanych światów. Zamiast przechowywać każdą jednostkę danych w pamięci, stosuje się deterministyczne algorytmy, które w oparciu o dane wejściowe (np. współrzędne) wytwarzają wynik zawsze w ten sam sposób. Najpopularniejszym podejściem do tworzenia pozornie nieskończonych środowisk/tekstur jest użycie szumu (ang. *noise*) rozumianego jako [86, 87, 88]:

¹<https://github.com/KhronosGroup/MoltenVK?tab=readme-ov-file#introduction-to-moltenvk>

- Generującego powtarzalną pseudolosową wartość dla każdej pozycji wejściowej.
- Posiadającego znany zakres (zwykle $[-1, 1]$).
- Nie wykazującego oczywistych powtarzających się wzorów.
- Jego częstotliwość przestrzenna jest niezmienna względem przesunięć.

Niektóre z podejść to:

- **Szum losowy** (ang. *random noise*)² – tworzony z losowych wartości, jest to szum chaotyczny bez zauważalnych ciągłości. Szum ten można znaleźć także pod nazwami "szum biały" (ang. *white noise*) czy "szum statyczny" (ang. *static noise*).
- **Szum Gaussa** (ang. *Gauss noise*)³ [89] – szum podobny do szumu losowego różniący się częstotliwością występowania różnych wartości, których rozkład pokrywa się z rozkładem normalnym – najczęściej jest wartości środkowych a najmniej skrajnych.
- **Szum wartości** (ang. *value noise*)⁴ – tworzona poprzez wygenerowanie i przypisanie losowych wartości na prostokątnej siatce (jej wierzchołkach), a następnie interpolowanie tych wartości pomiędzy wierzchołkami dla konkretnej pozycji.
- **Szum Perlina** (ang. *Perlin noise*)⁵ [86, 90] – opracowany przez Kena Perlina szum podobny do szumu wartości ale zamiast operacji na wartościach operuje się na gradientach. Każdemu wierzchołkowi siatki przypisywane są losowe wektory gradientu (kierunki) i interpoluje się iloczyny skalarne pomiędzy wektorami gradientu a wektorami odległości do punktu.
- **Szum Simplex** (ang. *Simplex noise*)⁶ [91, 92] – poprawiony przez Kena Perlina algorytm szumu Perlina. Zapewnia gładsze i naturalniejsze przejścia przy mniejszym koszcie obliczeniowym operując na siatce simpleksów (ang. *simplex*)⁷ (trójkątów dla szumu dwuwymiarowego).
- **Szum kłębiasty** (ang. *Billow noise*)⁸ – wybrana funkcja szumu z efektem kłebistości
- **Szum rzadkiej konwolucji** (ang. *Sparse convolution noise*)⁹ [93, 94] – algorytm generowania proceduralnego szumu, który zamiast opierać się na regularnej siatce (jak *Perlin Noise*), wykorzystuje rzadki (ang. *sparse*) rozkład impulsów i konwolucję wybranego jądra (najczęściej Gabora¹⁰)
- **Szum Worleya** (ang. *Worley noise*)¹¹ [95] – komórkowy szum generowany na podstawie diagramu Voronoi, gdzie kolor zależy od odległości do punktu centralnego komórki diagramu Voronoi.
- **Szum Voronoia** (ang. *Voronoi noise*)¹² [96] – zmodyfikowany szum Worleya przypisujący każdej komórce diagramu Voronoi konkretną wartość koloru.

²<https://gameidea.org/2023/12/16/noise-functions/>, sekcja "White Noise"

³<https://radartopix.com/en/what-is-the-difference-between-white-and-gaussian-noise/#gsc.tab=0>

⁴<https://www.youtube.com/watch?v=K110FoUnKhU>

⁵<https://mini.gmshaders.com/p/gm-shaders-mini-noise-1437243>

⁶<https://thebookofshaders.com/11/>, sekcja "Simplex Noise"

⁷<https://en.wikipedia.org/wiki/Simplex>

⁸<https://www.construct.net/en/tutorials/getting-started-advanced-30>

⁹https://www.youtube.com/watch?v=1_Ss2dUvaW8

¹⁰<https://www.geeksforgeeks.org/python-opencv-getgaborkernel-method/>

¹¹https://en.wikipedia.org/wiki/Worley_noise

¹²<https://godotshaders.com/snippet/voronoi/>

3.4. Techniki podnoszenia wydajności

Renderowanie dużych i złożonych scen trójwymiarowych, takich jak światy wokselowe, wymaga stosowania technik optymalizacyjnych. Celem tych metod jest ograniczenie liczby przetwarzanych danych lub przyspieszenie wykonywanych obliczeń tak, aby utrzymać wysoką płynność animacji przy zachowaniu jakości obrazu. Wydajność zależy nie tylko od procesora czy karty graficznej, ale również od kosztów komunikacji między tymi komponentami. Poniżej przedstawiono wybrane podejścia:

- **Pojedyncza Instrukcja, Wiele Danych SIMD** (ang. *Single Instruction Multiple Data*)¹³ [97]
 - wykonywanie w ramach jednej instrukcji tej samej operacji jednocześnie na wielu elementach danych upakowanych w jednym rejestrze.
- **Pojedyncza Instrukcja, Wiele Wątków SIMT** (ang. *Single Instruction Multiple Threads*) [97]
 - wykonywanie pojedynczej instrukcji na wielu wątkach jednocześnie. Wątki te są grupowane w jednostki, które wykonują tę samą instrukcję, ale na różnych danych, co jest istotne dla masowo równoległych obliczeń. Jest to technika charakterystyczna dla np. kart graficznych.
- **Wielowątkowość** (ang. *multithreading*) [98] – rozdzielenie zadań obliczeniowych (np. generowanie geometrii, przygotowywanie siatek segmentów świata, zarządzanie pamięcią) na wiele wątków procesora. Pozwala to równolegle przygotowywać dane dla karty graficznej, co skraca czas oczekiwania na wyrenderowanie kolejnych klatek.
- **Obinanie bryłą widzenia** (ang. *frustum culling*) [99, 100] – odrzucanie obiektów znajdujących się zbyt blisko i zbyt daleko kamery oraz poza polem jej widzenia. Dzięki temu karta graficzna przetwarza tylko te fragmenty sceny, które faktycznie mogą być widoczne w danym momencie.
- **Eliminacja niewidocznych ścian** (ang. *hidden face culling*) [101] – technika stosowana przy generowaniu siatek wokselowych, polegająca na pomijaniu w budowie geometrii wszystkich ścian, które są całkowicie ukryte wewnętrz bryły (np. przylegają do innych wokseli). Dzięki temu do karty graficznej trafiają tylko ściany mające potencjał bycia widocznymi, co znacząco zmniejsza rozmiar buforów i liczbę rysowanych trójkątów.
- **Odrzucanie tylnych ścian** (ang. *back-face culling*) [100] – odrzucanie fragmentów ścian skierowanych w stronę przeciwną do kamery (np. tylnych stron trójkątów). Jest to technika wspierana sprzętowo przez większość układów graficznych.
- **Obcinanie okluzji** (ang *occlusion culling*) [100, 102] – eliminacja obiektów zasłoniętych przez inne, znajdujące się bliżej kamery. Dzięki temu karta graficzna nie marnuje zasobów na generowanie fragmentów, które i tak zostaną przykryte.

¹³<https://www.youtube.com/watch?v=ulmjD6Y4do>

- **Programowalne generowanie geometrii** (ang. *programmable geometry generation*) [103, 104] – wykorzystanie specjalistycznych programów cieniowania – na przykład geometrii GS (ang. *geometry shader*, teselacji TS (ang. *tesselation shaders*) – do modyfikowania i generowania nowej geometrii bezpośrednio na karcie graficznej, oszczędzając czas pracy procesora oraz odciążając magistralę danych.
- **Format danych dla karty graficznej** – dobór odpowiedniego formatu danych odczytywanych przez kartę graficzną, które mogą zmniejszyć rozmiar wysyłanych do niej danych lub przyspieszyć proces rasteryzacji. Dobór formatu danych może uwzględniać wybranie odpowiedniej topologii danych (np. *TRIANGLE_STRIP* zamiast *TRIANGLE_LIST*, techniki indeksacji wierzchołków¹⁴ lub pobierania wierzchołków (ang. *vertex pulling*)¹⁵.
- **Instancjonowanie** (ang. *instancing*) [102] – wielokrotne renderowanie tego samego obiektu przy użyciu jednej definicji geometrii wysłanej do karty graficznej jednokrotnie. Technika szczególnie przydatna w przypadku powtarzalnych elementów (np. wokseli, drzew, roślinności).
- **Redukcja liczby wywołań rysowania** (ang *draw call reduction*) [102] – łączenie wielu obiektów w jeden większy bufor danych, aby zmniejszyć liczbę wywołań funkcji rysujących.
- **Poziomy szczegółowości LoD** (ang. *Level of Detail*) [105] – stosowanie modeli o różnym stopniu szczegółowości w zależności od odległości od kamery. Odległe obiekty mogą być reprezentowane uproszczonymi siatkami, co zmniejsza liczbę wierzchołków i trójkątów do przetworzenia.
- **Zachłanne siatkowanie** (ang. *greedy meshing*) [106, 107] – technika polegająca na łączeniu sąsiadujących, identycznych powierzchni (ścian wokseli) w jedną większą, redukując liczbę geometrii podlegającej renderowaniu.

¹⁴https://vulkan-tutorial.com/Vertex_buffers/Index_buffer

¹⁵<https://voxel.wiki/wiki/vertex-pulling/>

<https://www.youtube.com/watch?v=IoS5opco9LA>

4. PROJEKT OPROGRAMOWANIA

W celu przeprowadzenia badań nad metodami reprezentacji obiektów w grach i symulacjach konieczne było przygotowanie dedykowanego środowiska badawczego. Środowisko to zostało zaprojektowane jako prototyp w postaci samodzielnej aplikacji, której zadaniem nie jest dostarczenie produktu końcowego, lecz stworzenie platformy eksperymentalnej (ang. *test bed*)¹ umożliwiającej weryfikację różnych struktur danych, algorytmów oraz technik optymalizacyjnych.

Przygotowany prototyp zapewnia kontrolowane warunki eksperymentów, w tym możliwość powtarzalnego uruchamiania testów, zbierania pomiarów oraz wizualizacji wyników. Jego budowa pozwala na łatwą wymianę komponentów (np. generatorów światów, struktur danych, modułów renderujących), co umożliwia porównywanie alternatywnych rozwiązań w jednolitym środowisku. W niniejszej sekcji opisano przyjęte założenia oraz wymagania, które prototyp musiał spełniać, a także wybory dotyczące technologii i architektury, na których oparto jego implementację.

4.1. Założenia i wymagania

Projekt środowiska badawczego został oparty na zestawie wymagań funkcjonalnych oraz niefunkcjonalnych, które określają jego możliwości oraz oczekiwane własności jakościowe.

4.1.1. Wymagania funkcjonalne

- **Reprezentacja świata 3D** – prototyp powinien tworzyć, przetwarzać i renderować trójwymiarowe światy oraz umożliwiać poruszanie się po wygenerowanym terenie z użyciem myszy i klawiatury.
- **Weryfikacja wizualna** – prototyp powinien umożliwiać zatrzymanie procesu generowania i renderowania siatki w wybranym momencie, a następnie oglądanie jej z różnych perspektyw.
- **Generatory zawartości** – środowisko musi wspierać generowanie terenów przy użyciu hermetycznie zmodylaryzowanych generatorów
- **Moduł pomiarowy** – prototyp powinien umożliwiać zbieranie danych eksperymentalnych, takich jak na przykład czas generowania danych czy siatki terenu, średnia liczba klatek na sekundę, zajmowana objętość pamięci.
- **Prezentowanie wyników** – wyniki działania prototypu powinny być prezentowane w formie wizualnej (grafika 3D) oraz w postaci tekstowej możliwej do analizy.
- **Wymienność komponentów** – system musi umożliwiać łatwe przełączanie wybranych modułów (struktury danych, generatory terenu) w celu przeprowadzenia porównań w identycznych warunkach.
- **Światy testowe** – aplikacja powinna pozwalać na definiowanie i uruchamianie różnych scenariuszy badawczych, np. światów o różnym rozmiarze i stopniu złożoności.

¹<https://istqb-glossary.page/test-bed/>
<https://en.wikipedia.org/wiki/Testbed>

4.1.2. Wymagania niefunkcjonalne

- **Możliwość wizualizacji** – prototyp powinien zapewniać użytkownikowi możliwość obserwacji działania systemu w czasie rzeczywistym.
- **Wydajność** – prototyp powinien umożliwiać przeprowadzanie eksperymentów na dużych wolumenach danych w akceptowalnym czasie – pojedyncze sekundy dla prostych światów, dziesiątki sekund dla światów proceduralnych.
- **Środowisko operacyjne** – oprogramowanie powinno działać na najpopularniejszym systemie operacyjnym dla komputerów osobistych (Windows²)
- **Modularność** – system powinien być zbudowany w sposób umożliwiający łatwą rozbudowę o kolejne algorytmy i techniki.
- **Powtarzalność wyników** – badania muszą być możliwe do wielokrotnego odtworzenia w identycznych warunkach.
- **Transparentność pomiarów** – mechanizmy pomiarowe muszą działać niezależnie od logiki eksperymentu, aby nie zakłócać jego wyników.

4.2. Wokselowy typ przechowywanego świata

W ramach prototypu zdecydowano się na reprezentację świata w postaci **wokselowej** przytoczonego w sekcji 2.3.3. Rozwiążanie to zostało wybrane przede wszystkim ze względu na wysoką złożoność problemu, jaki narzuca. Świat wokselowy charakteryzuje się następującymi cechami:

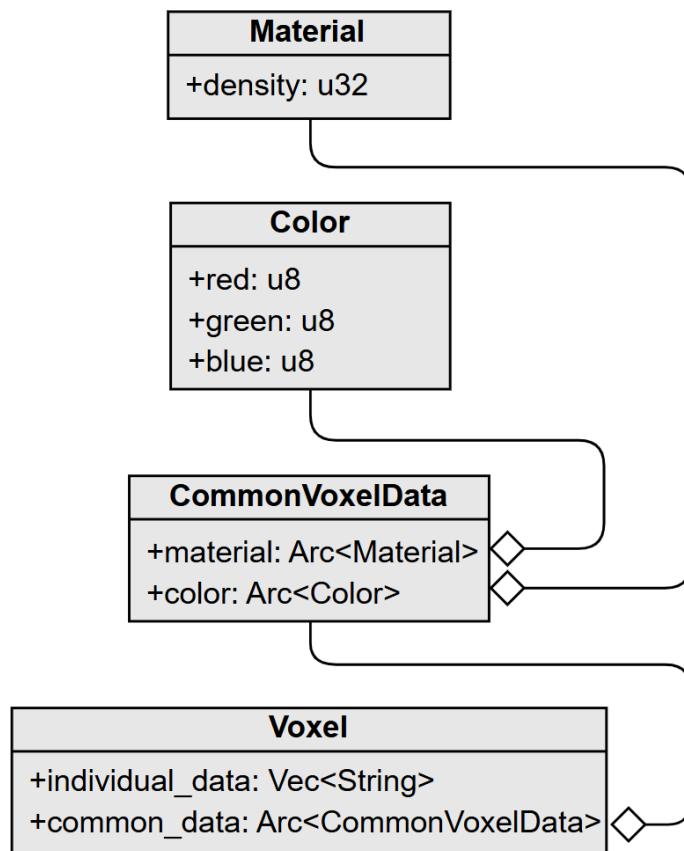
- **Wysokie wymagania pamięciowe** – nawet niewielki obszar przestrzeni opisany w rozdzielczości wokselowej generuje ogromną liczbę elementów do przechowywania i przetwarzania.
- **Złożoność obliczeniowa** – konieczne jest stosowanie technik optymalizacji (takich jak eliminacja niewidocznych ścian czy hierarchiczne struktury danych), aby zachować płynność działania systemu.
- **Elastyczność reprezentacji** – woksele umożliwiają realistyczne modelowanie zjawisk wewnętrz obiektów, takich jak propagacja fal, płynów czy rozchodzenie się uszkodzeń materiałów, co nie jest możliwe w klasycznych modelach siatkowych.
- **Potencjał badawczy** – ze względu na swoją złożoność, środowisko wokselowe pozwala weryfikować skuteczność różnych struktur danych i technik renderowania w trudnych warunkach obliczeniowych.

Wybór świata wokselowego był świadomym wyborem ukierunkowanym na prowadzenie rzetelnych badań nad efektywnością reprezentacji i optymalizacji przechowywanych danych w przestrzeni. Ważnym do wyróżnienia jest to, że ze względu na ograniczenia czasowe przechowywany świat jest światem stacjonarnym – statycznym, bez ruchomych elementów.

²<https://ranking.gemius.com/pl/ranking/systems/>

4.2.1. Implementacja woksela

Struktura woksela w ramach prototypu (rys. 4.1) zakłada jego zdolność do przechowywania danych fizycznych (np. gęstości materiału, temperatury, czy przepuszczalności światła) i wizualnych (np. koloru, tekstury, czy typu materiału). Praktyczne znaczenie takiego podejścia można odnaleźć w grze *Minecraft* (zob. 2.4.4). Na potrzeby pracy założono podstawowe lecz łatwo rozszerzalne informacje – kolor, materiał i jego gęstość, listę indywidualnych cech.



Rys. 4.1: Diagram klas przedstawiający agregację struktur składowych w obrębie struktury głównej `Voxel`.

Tak zdefiniowany model umożliwia tworzenie złożonych i różnorodnych środowisk, w których każdy woksel może posiadać własny stan wewnętrzny — analogicznie do wspomnianych bloków z gier typu *Minecraft*. Rozwiążanie to pozwala na budowę zarówno prostych scen wizualnych, jak i zaawansowanych symulacji fizycznych czy logicznych, w których woksele mogą reagować na bodźce z otoczenia lub zmieniać swój stan w czasie.

4.3. Wybór środowiska programistycznego i narzędzi

Języki uchodzące za wysoce wydajne takie jak C oraz C++ posiadają dobre alternatywy w postaci języków Go, Rust, czy Zig. Każdy z tych języków zaliczany jest do języków programowania systemowego³. Na podstawie tabeli 3.1 wytypowano **Rust** jako najbardziej perspektywiczny w ramach pisania programu obsługującego wiele prezentowanych użytkownikowi obiektów. Cechuje się wysoką wydajnością, dużym poziomem bezpieczeństwa pamięci oraz nowoczesnym systemem typów, co pozwala ograniczyć ryzyko popełnienia błędów. Jest to dodatkowo język aktywnie rozwijany, wspierany przez rosnącą społeczność, oraz adaptowany jako zamiennik języka C++ w różnych popularnych projektach⁴.

Z branych pod uwagę bibliotek graficznych (tab. 3.2) wybrano bibliotekę **Vulkan** jako docelową bibliotekę graficzną. Wybór ten podyktowany był nowoczesnością, dużą przenośnością, wysoką wydajnością i szeroką popularnością tej technologii.

Dzięki połączeniu języka *Rust* oraz biblioteki *Vulkan* prototyp środowiska badawczego zyskuje cechy kluczowe dla tej pracy: wysoką wydajność, bezpieczeństwo pamięci, szeroką przenośność, a także możliwość bezpośrednią kontroli nad procesem renderowania i zarządzaniem zasobami.

4.3.1. Biblioteki wspierające prototyp

W ramach wykonania prototypu środowiska badawczego wykorzystano szeregu bibliotek dostarczających funkcjonalności niskopoziomowych oraz ułatwiających obsługę elementów infrastrukturalnych, takich jak zarządzanie grafiką, wejściem/wyjściem czy logowaniem. Szczególne znaczenie w tym zestawie ma biblioteka *vulkanalia*, pełniąca rolę warstwy dostępowej do wybranej interfejsu graficznego Vulkan. Najważniejsze z użytych bibliotek to:

- **vulkanalia**⁵ [108] – wysokopoziomowe i idiomatyczne w języku *Rust* wiązanie do niskopoziomowego API *Vulkan*. Biblioteka ta umożliwia bezpośredni dostęp do funkcji *Vulkan* jest oznaczany blokami *unsafe*⁶ ponieważ nie implementuje mechanizmów bezpieczeństwa związanych z zarządzaniem pamięcią czy wskaźnikami. Stanowi ona główne narzędzie do obsługi grafiki 3D i zarządzania zasobami karty graficznej.
- **winit**⁷ – biblioteka do zarządzania oknami oraz zdarzeniami systemowymi, takimi jak interakcje z klawiaturą czy myszką. Pozwala na tworzenie przenośnych aplikacji działających w różnych systemach operacyjnych.
- **cgmath**⁸ – biblioteka matematyczna dostarczająca wektory, macierze, kwaterniony i inne struktury używane w obliczeniach geometrycznych oraz transformacjach 3D.
- **png**⁹ – biblioteka do obsługi grafiki rastrowej, wykorzystywana do tworzenia grafik testujących działanie wybranego algorytmu szumu.
- **rand**¹⁰ – generator liczb pseudolosowych, używany m.in. w generatorach terenu.

³https://en.wikipedia.org/wiki/Systems_programming

⁴<https://github.com/uutils/coreutils>

<https://github.com/microsoft/windows-rs>

⁵<https://docs.rs/vulkanalia/latest/vulkanalia/>

⁶<https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html>

⁷<https://docs.rs/winit/latest/winit/>

⁸<https://docs.rs/cgmath/latest/cgmath/>

⁹<https://docs.rs/png/latest/png/>

¹⁰<https://docs.rs/rand/latest/rand/>

- **dhat**¹¹ – narzędzie do profilowania pamięci (heap profiler), pozwalające badać zużycie zasobów podczas działania prototypu.
- **criterion**¹² – narzędzie do benchmarków, umożliwiający przeprowadzanie pomiarów wydajnościowych, generowanie raportów w postaci *HTML*¹³ oraz diagramów *SVG*¹⁴

4.4. Algorytmy, techniki i struktury prototypu

Podczas implementacji prototypu konieczne było dokonanie wyborów zarówno w zakresie technik optymalizacyjnych, algorytmów i struktur danych, którym poświęcono tę sekcję.

- **Wybór algorytmu generowania szumu.** Do generowania terenów proceduralnych wybrano, z pośród różnych przytoczonych algorytmów w sekcji 3.3, algorytm **Simplex Noise**, który stanowi rozwinięcie klasycznego szumu *Perlin*. W odróżnieniu od swojego poprzednika, charakteryzuje się on lepszą wydajnością obliczeniową, mniejszą złożonością obliczeniową oraz mniejszymi artefaktami siatkowania. Do niedawna algorytm ten podlegał ochronie patentowej¹⁵, przez co nie mógł być tak szeroko stosowany jak jego poprzednik. Teraz natomiast dzięki wspomnianym cechom powinien bardzo dobrze znaleźć zastosowanie w programach wymagających realistycznego odwzorowania zjawisk naturalnych.
- **Wybór technik optymalizacyjnych.** W prototype zastosowano zestaw technik, które mają kluczowe znaczenie dla uzyskania wysokiej wydajności w środowiskach trójwymiarowych. Ze względu na ograniczenia czasowe do zaimplementowania wyselekcjonowano następujące techniki opisane szerzej w sekcji 3.4: *wielowątkowość, obcinanie bryły widzenia, eliminacja niewidocznych ścian, instancjonowanie, redukcja liczby wywołań rysownia* oraz zastosowano optymalizację rozmiaru danych poprzez wybór topologii danych *TRIANGLE_STRIP*.
- **Selekcja badanych struktur danych.** Na potrzeby prototypu zaimplementowano cztery charakterystyczne struktury danych spośród wymienionych w sekcji 2.5.3 służące do reprezentacji świata wokselowego. Każda z tych struktur pełni określoną rolę w badaniach, a ich wybór podyktowany był potrzebą stworzenia porównywalnego środowiska, które pozwala analizować wpływ reprezentacji danych oraz algorytmów na wydajność (zarówno pamięciową jak i obliczeniową) programu.
 - **Mapa wokseli** – jako przykład struktury tablicowej dokładnie mapującej całą przestrzeń, zarówno wartości puste jak i istotne.
 - **Lista wokseli** – jako strukturę tablicową przechowującą tylko wartości istotne.
 - **Tablica haszująca wokseli** – jako strukturę opartą na haszowaniu.
 - **Drzewo octree** – jako strukturę drzewiastą mapującą przestrzeń.

¹¹<https://docs.rs/dhat/latest/dhat/>

¹²<https://docs.rs/criterion/latest/criterion/>

¹³<https://developer.mozilla.org/en-US/docs/Web/HTML>

¹⁴<https://developer.mozilla.org/en-US/docs/Web/SVG>

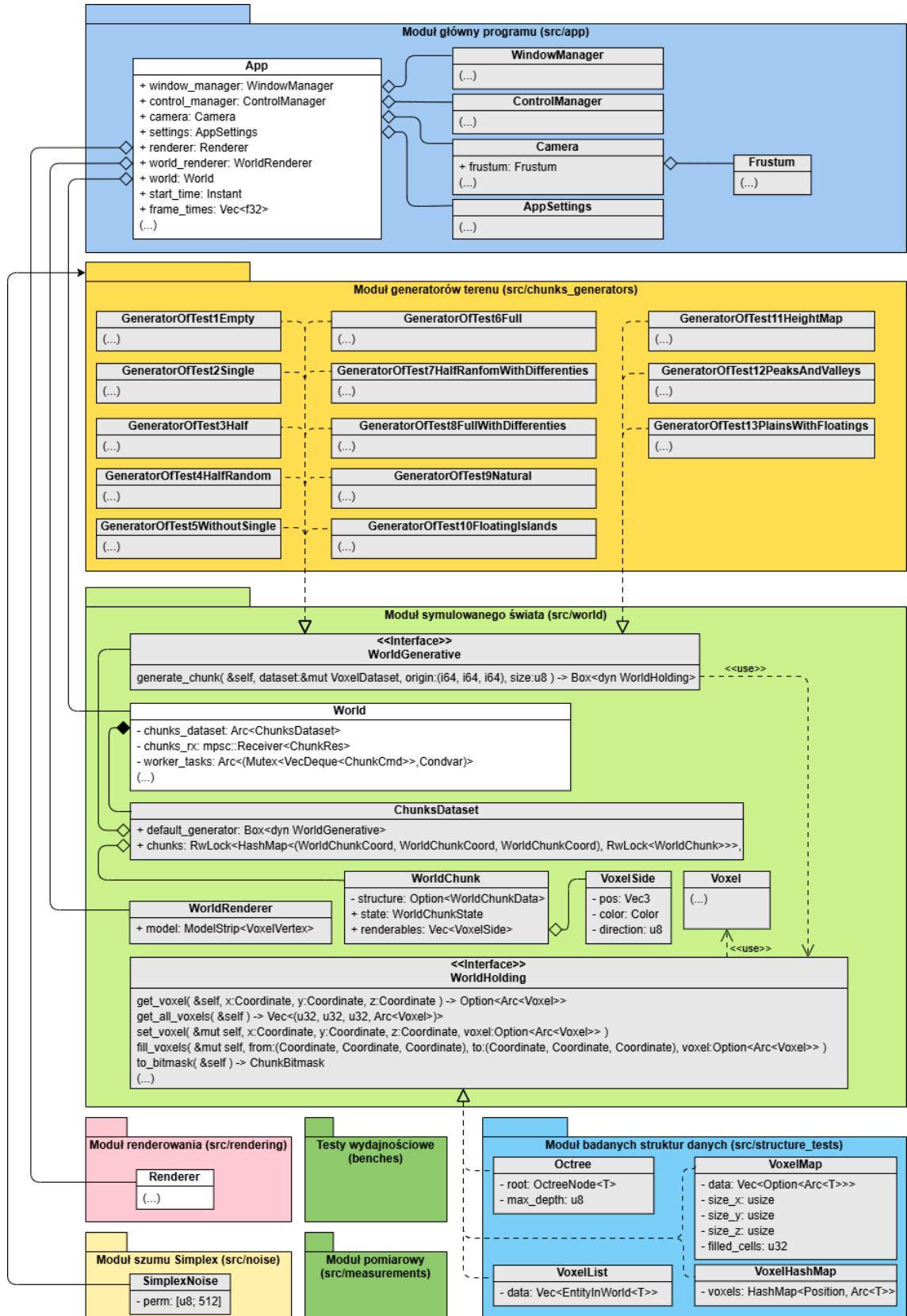
¹⁵<https://patents.google.com/patent/US6867776>

4.5. Architektura oprogramowania

Kod źródłowy przygotowanego prototypu został podzielony na szereg modułów, z których każdy pełni scisłe określona rolę. Podział ten zapewnia modularność i hermetyczność, ułatwia rozwój oraz umożliwia wymianę poszczególnych komponentów bez konieczności ingerencji w pozostałe elementy programu. Strukturę logiczną w postaci diagramu klas najważniejszych elementów prototypu przedstawiono na rysunku 4.2, a strukturę plików na rysunku 4.3. Widoczne na tych rysunkach obszary mają następującą charakterystykę:

- **Moduł główny programu** (`src/app`) – odpowiedzialny za pętlę główną, obsługę sterowania użytkownika, zarządzanie oknem i kamerą, a także konfigurację ustawień programu.
- **Generatory terenu** (`src/chunks_generators`) – każdy świat badawczy posiada swój odrębny generator. Generatory zostały zaimplementowane w postaci "cech" (ang. *traits*)¹⁶ języka Rust, co pozwala traktować je jako wymienialne komponenty i ułatwia dodawanie nowych metod generowania.
- **Części pomiarowe** (`benches`, `src/measurements`) – obejmują integrację narzędzia dhat oraz użycie funkcji `size_of` do pomiarów pamięciowych, oraz `criterion` do pomiarów wydajnościowych struktur danych. Dzięki temu możliwe jest prowadzenie eksperymentów w sposób ujednolicony i powtarzalny.
- **Moduł szumu Simplex** (`src/noise`) – wykorzystywany przy generowaniu proceduralnego terenu, zapewniający naturalną i zróżnicowaną strukturę świata.
- **Renderer** (`src/rendering`) – odpowiada za proces renderowania. W module tym znajdują się definicje potoków renderujących, programów cieniowania (shaderów), a także hermetyczne mechanizmy komunikacji z kartą graficzną przy użyciu biblioteki Vulkan.
- **Struktury danych** (`src/structure_tests`) – implementacje struktur (np. octree), które realizują trait `WorldHolder`. Dzięki takiemu podejściu poszczególne struktury są wymienialne i mogą być bezpośrednio porównywane w ramach badań.
- **Moduł świata** (`src/world`) – obejmuje definicje podstawowych elementów: świata, chunków, wokseli oraz wierzchołków siatki. Jest to warstwa łącząca logikę reprezentacji danych z warstwą renderującą.
- **Ustawienia prototypu** (`src/flags.rs`) – zawiera stałe (flagi) umożliwiające włączanie i wyłączanie logów związanych z pomiarami czasu oraz innymi diagnostycznymi funkcjonalnościami.
- **Inicjacja programu** (`src/main.rs`) – główny plik programu, inicjalizujący wszystkie niezbędne moduły oraz uruchamiający logikę aplikacji.
- **Testy wydajnościowe** (`clear_test.sh`) – skrypt uruchamiający testy wydajnościowe (`benches`), czyszczący uprzednio artefakty ewentualnych poprzednich uruchomień testów.

¹⁶<https://doc.rust-lang.org/book/ch10-02-traits.html>



Rys. 4.2: Diagram klas prototypu przedstawiający główne moduły wraz z najważniejszymi strukturami i kluczowymi atrybutami. Trzy struktury wyróżnione kolorem białym stanowią podstawę działania aplikacji – App zarządza aplikacją, a World Światem, Renderer zaś odpowiada za proces renderowania.

```

src > flags.rs > ...
1  pub const SIMULATED_TEST_WORLD_SLICE:bool = false;
2  pub const SIMULATED_TEST_WORLD_ID:u8 = 11;
3  pub const RENDER_DISTANCE:u8 = 16;
4  pub const CPUS_COUNT:u8 = 16;
5
6  pub const FULLSCREEN:bool = true;
7
8  pub const FLAG_PROFILING_SHOW_FPS:bool = true;
9  pub const FLAG_PROFILING_WORLD_GENERATION:bool = false;
10 pub const FLAG_PROFILING_WORLD_GENERATION_QUEUE:bool = false;
11 pub const FLAG_PROFILING_WORLD_RENDERING:bool = false;
12 pub const FLAG_PROFILING_WORLD_HOLDER_INITIALIZATION:bool = false;
13

```

(a) Struktura plików kodu stworzonego prototypu.

(b) zawartość pliku `flags.rs` determinującego konfigurację komplikacji prototypu

Rys. 4.3: Struktura plików stworzonego prototypu (a) oraz główny plik konfiguracyjny `flags.rs` (b).

4.6. Opis działania prototypu

Prototyp został zaprojektowany jako aplikacja interaktywna, w której użytkownik może zarówno obserwować proces generowania i renderowania świata, jak i wpływać na sposób jego działania.

4.6.1. Konfiguracja uruchomienia

Podstawowa konfiguracja systemu odbywa się w pliku `flags.rs`, gdzie można ustawić między innymi:

- generowany świata;
- odległość renderowania (promień widoczności wokół aktora);
- liczbę procesorów logicznych wykorzystywanych w obliczeniach;
- zakres oraz rodzaj logowanych i zbieranych danych.

Po dokonaniu konfiguracji prototyp uruchamiany jest za pomocą skryptu `run.sh`. Po starcie aplikacji otwiera się okno graficzne, a w tle rozpoczyna się proces generowania świata. Po chwili użytkownik może dostrzec pojawiający się wyrenderowany teren.

4.6.2. Sterowanie i interakcja

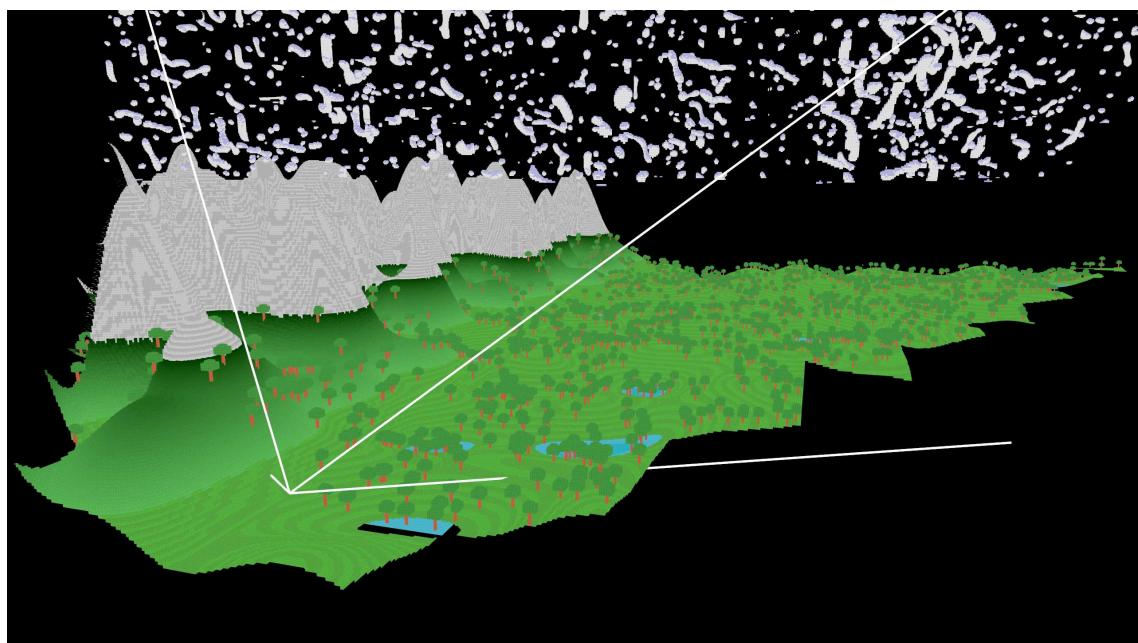
Aplikacja udostępnia możliwość swobodnego poruszania się w świecie za pomocą klawiatury. Obsługiwane są następujące operacje:

- **Klawisze strzałek** lub klawisze W, S, A, D – sterowanie ruchem aktora w płaszczyźnie poziomej;
- **Spacja** – wznoszenie;
- **Shift** – opadanie;
- **Ctrl** – przyspieszenie.

4.6.3. Funkcje badawcze

Prototyp zawiera również zestaw mechanizmów wspierających badania:

- **Klawisz R** – reset danych dotyczących pomiarów liczby klatek na sekundę.
- **Klawisz F** – zatrzymanie procesu generowania terenu oraz zamrożenie bryły widzenia w aktualnym położeniu. Dodatkowo, w tym trybie wyświetlane są kontury bryły widzenia, co umożliwia swobodny przelot wokół sceny i weryfikację poprawności działania mechanizmu obcinania (rys. 4.4)



Rys. 4.4: Zrzut ekranu z działającego prototypu przy zamrożonym przetwarzaniu i renderowaniu. Białe linie obrazują kontury bryły widzenia a miejsce, w którym wydaje się jakoby były połączone przedstawia małe prostokąt – ekran monitora odbiorcy. Wyrenderowane zostały wszystkie segmenty zawierające się lub przecinające bryłę widzenia.

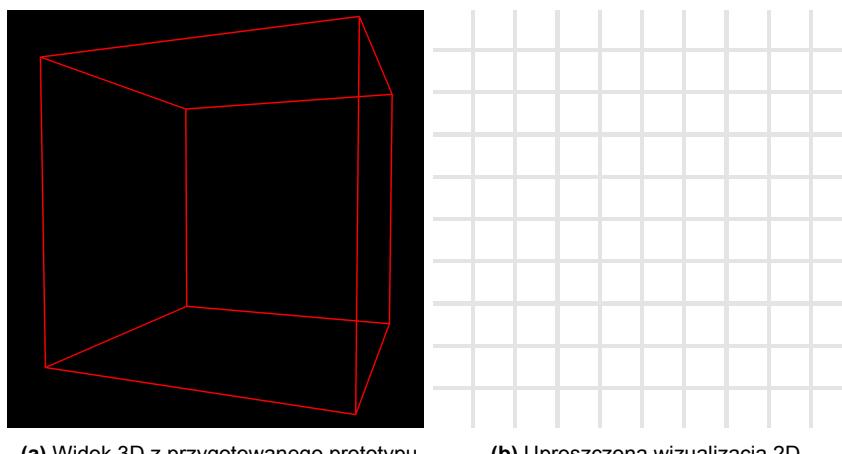
4.7. Światy testowe

Każda ze struktur winna być przetestowana w różnych środowiskach, aby w sposób kontrolowany, powtarzalny i możliwie zróżnicowany ocenić ich adekwatność w reprezentowaniu dużych przestrzeni. W tym celu przygotowano zestaw jedenastu typów wokselowych światów, różniących się stopniem i sposobem wypełnienia oraz różnorodnością danych. W celu zachowania staranności światy testowe próbują przedstawać różnorodne przypadki – od skrajnych, po te bliższe rzeczywistym.

Światy od 1 do 9 zostały przygotowane jako 8 symulowanych segmentów sześciennych, w zakresie $[0;1]$ na każdej z osi. Pozostałą przestrzeń stanowią segmenty wyłączone i niedostępne dla symulacji. Ograniczenie to widoczne jest jako czerwony kontur graniastosłupa (przykład takiego ograniczenia można zaobserwować na rysunku 4.5 (a)). Pozostałe światy testowe nie posiadają takiego ograniczenia i proces symulacji terenu odbywa się bez ograniczeń na każdej osi przestrzeni trójwymiarowej.

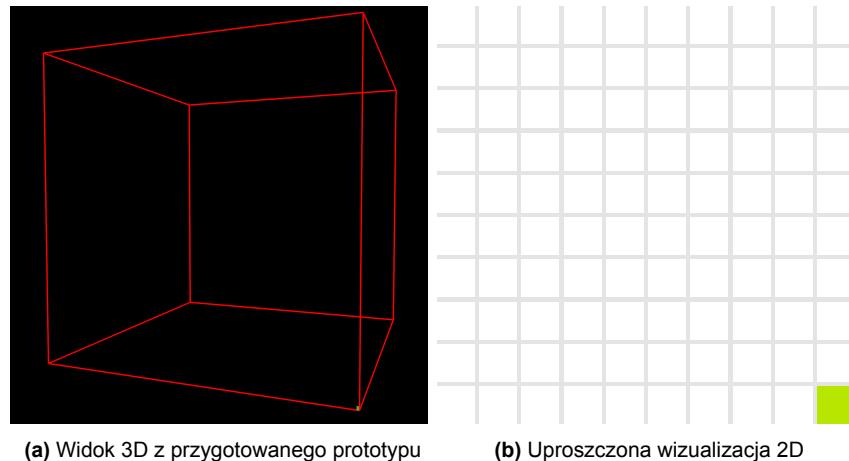
Charakterystyka światów testowych jest następująca:

1. **Pusty** (rys. 4.5) – świat nie zawiera żadnych obiektów. Jest przydatny jako przypadek testowy do oceny narzutu pamięciowego zainicjowanych struktur danych przechowujących świat (segmentów symulowanych oraz wyłączenych) i ogólnego funkcjonowania komponentów programu.



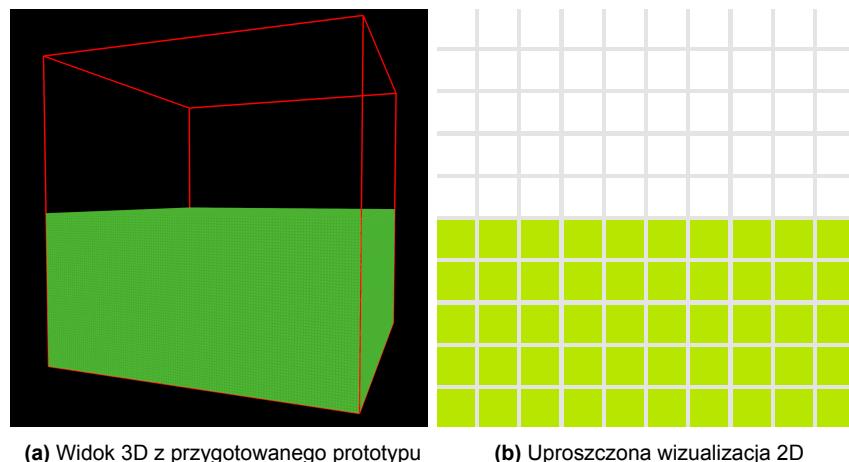
Rys. 4.5: Przedstawienie świata testowego o pustej strukturze (a), wraz z uproszczoną wizualizacją jego struktury (b).

2. **Pojedynczy woksel** (rys. 4.6) – świat zawiera dokładnie jeden woksel i testuje zachowanie struktury danych przy minimalnej liczbie przechowywanych danych, testujący strukturę przy bardzo rzadkich danych oraz jej zdolność do rozszerzania/kompresji.



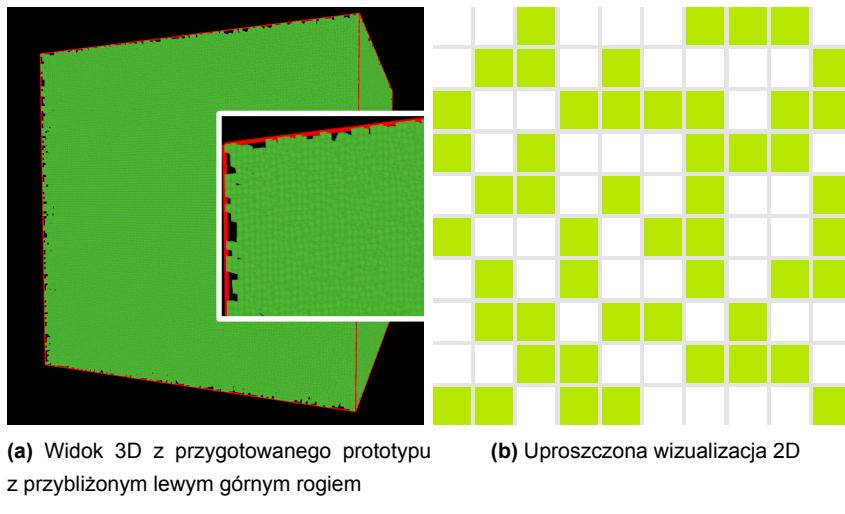
Rys. 4.6: Przedstawienie świata testowego z jednym woksem – minimalną ilością danych (a), wraz z uproszczoną wizualizacją jego struktury (b).

3. **Połowiczne wypełnienie** (rys. 4.7) – świat jednorodnie wypełniony do połowy swej wysokości, sprawdzający elastyczność struktury w przypadku wyraźnego podziału na część wypełnioną oraz pustą.



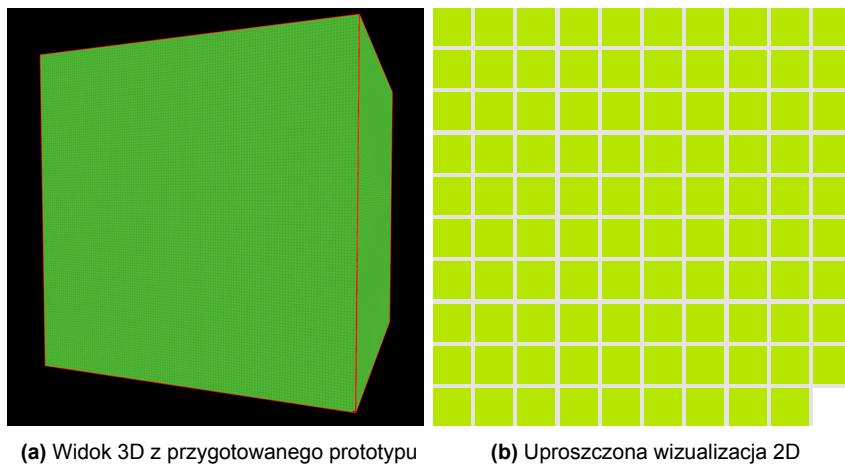
Rys. 4.7: Przedstawienie jednorodnie wypełnionego do połowy świata (a), wraz z uproszczoną wizualizacją jego struktury (b).

4. **Połowiczne losowe wypełnienie** (rys. 4.8) – świat losowo w połowie wypełniony, testujący wydajność oraz obciążenie pamięci w przypadku gęsto upakowanych wokseli.



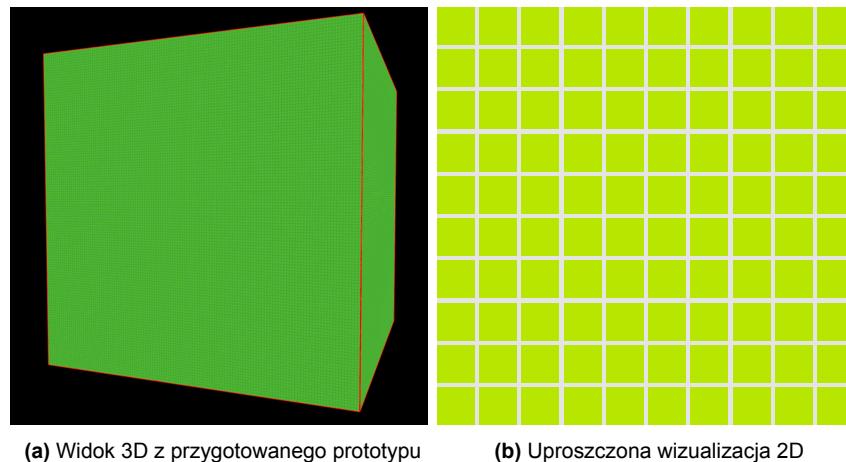
Rys. 4.8: Przedstawienie świata wypełnionego losowo wybraną połową komórek (a), wraz z uproszczoną wizualizacją jego struktury (b). Losowe wypełnienie jest najwidoczniejsze przy granicy generowanego terenu, gdzie widoczne są prześwity pustki, co przedstawiono na przybliżeniu.

5. **Bez woksela** (rys. 4.9) – świat jednorodnie wypełniony z jedną pustą komórką, testujący obciążenie wydajnościowe i pamięciowe przy braku możliwości do ewentualnej kompresji.



Rys. 4.9: Przedstawienie jednorodnie wypełnionego do połowy świata bez jednego woksela (a), wraz z uproszczoną wizualizacją jego struktury (b).

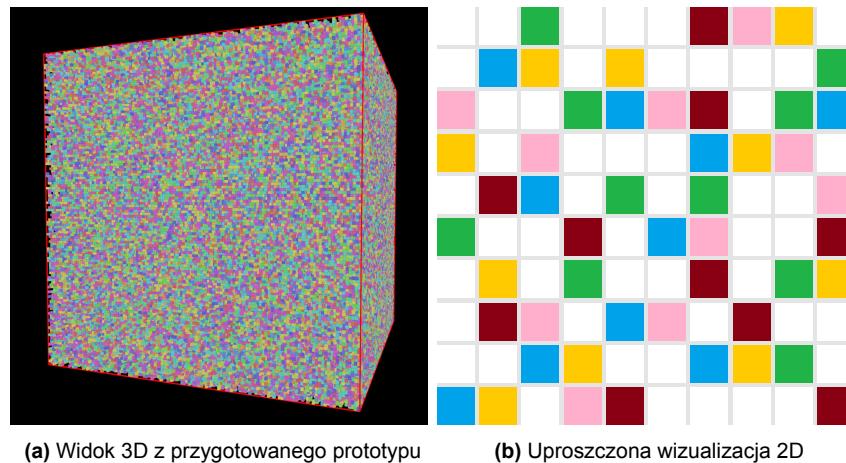
6. **Wypełniony** (rys. 4.10) – świat jednorodnie wypełniony, testujący zdolność do pełnej kompresji struktury świata.



(a) Widok 3D z przygotowanego prototypu **(b)** Uproszczona wizualizacja 2D

Rys. 4.10: Przedstawienie jednorodnie wypełnionego (a), wraz z uproszczoną wizualizacją jego struktury (b).

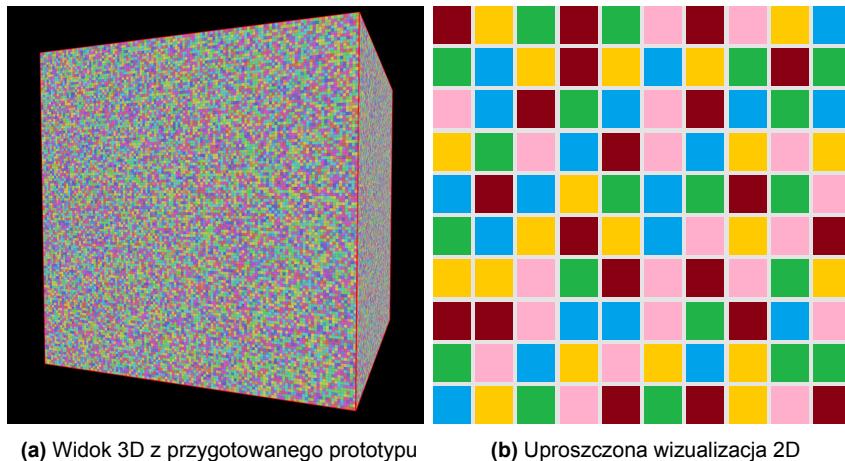
7. **Połowiczne losowe wypełnienie różnokolorowymi wokselami** (rys. 4.11) – świat losowo w połowie wypełniony wokseli o kilkuset różnych kolorach, testujący wydajność oraz obciążenie pamięci w przypadku gęsto upakowanych wokseli i konieczności przechowywania definicji różnych kolorów.



(a) Widok 3D z przygotowanego prototypu **(b)** Uproszczona wizualizacja 2D

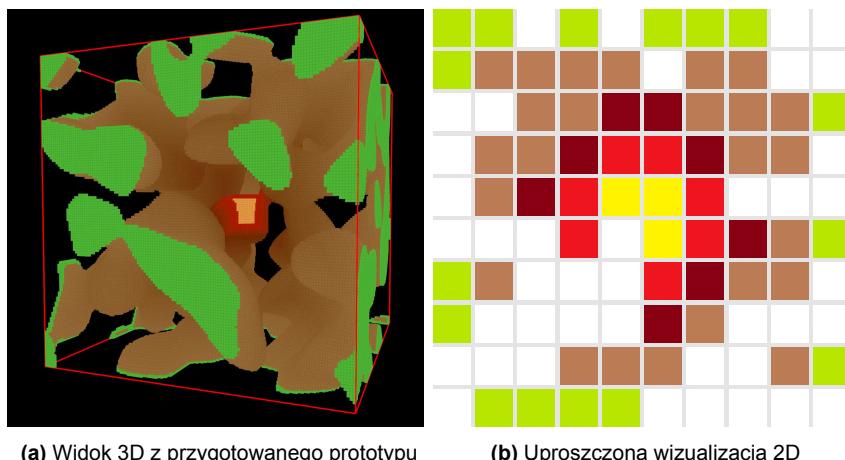
Rys. 4.11: Przedstawienie losowo wypełnionego w połowie świata różnokolorowymi wokselami (a), wraz z uproszczoną wizualizacją jego struktury (b). Losowe wypełnienie jest najwidoczniejsze przy granicy generowanego terenu, gdzie widoczne są prześwity pustki.

8. **Wypełniony różnokolorowymi wokselami** (rys. 4.12) – świat w pełni wypełniony wokselami o kilkuset różnych kolorach, sprawdzający zachowanie struktury przy pełnym wypełnieniu, muszącej przechowywać definicje różnych kolorów.



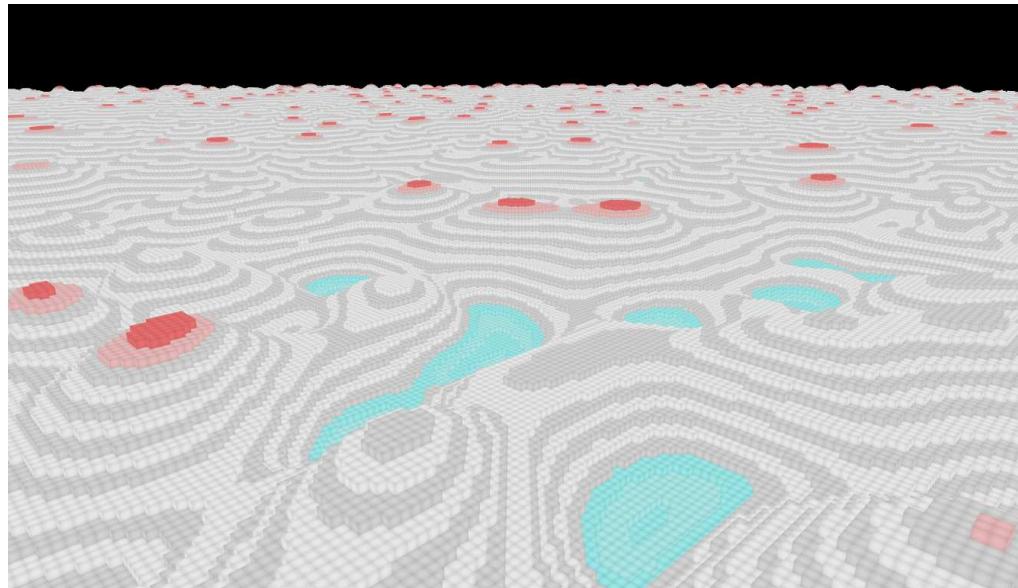
Rys. 4.12: Przedstawienie świata w pełni wypełnionego różnokolorowymi wokselami (a), wraz z uproszczoną wizualizacją jego struktury (b).

9. **Simplex** (rys. 4.13) – świat wygenerowany z użyciem trójwymiarowego szumu Simplex o rzadkim, skupionym rozkładzie wokseli. Ciągłe wartości szumu są dyskretyzowane do siatki wokselowej i odpowiednio progowane w celu uzyskania częściowo pustej przestrzeni.

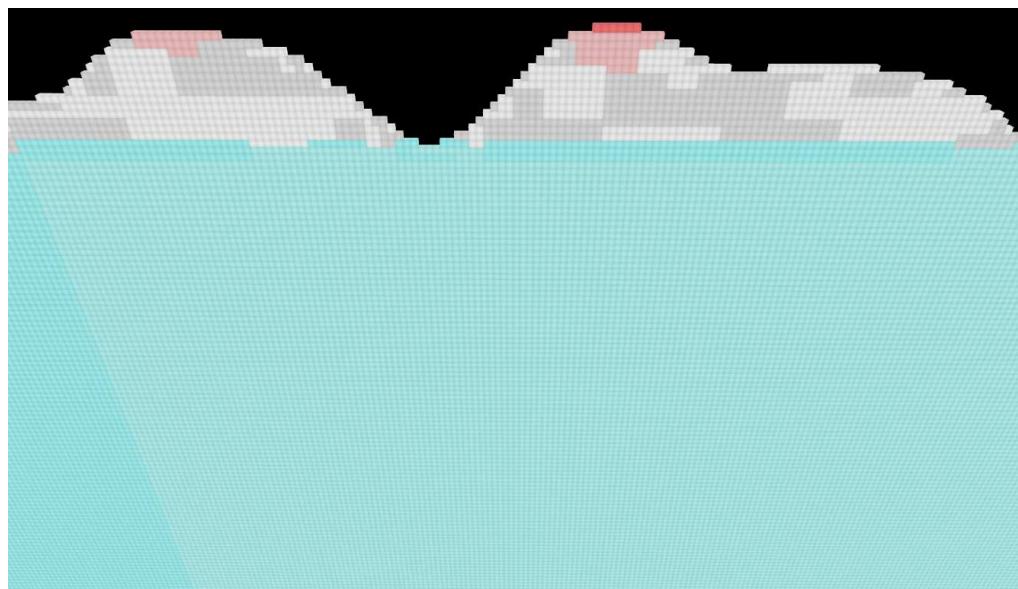


Rys. 4.13: Przedstawienie realistycznego świata z użyciem trójwymiarowego szumu Simplex o niskim procencie wypełnienia (a), wraz z uproszczoną wizualizacją jego struktury (b).

10. **Warstwicowy** (rys. 4.14) – świat proceduralnie generowany, budowany na bazie drzewa czwórkowego. Wysokość wzgórz oraz kolory zależą od wynikającej z wartości szumu Simplex pozycji pionowej węzłów drzewa czwórkowego.



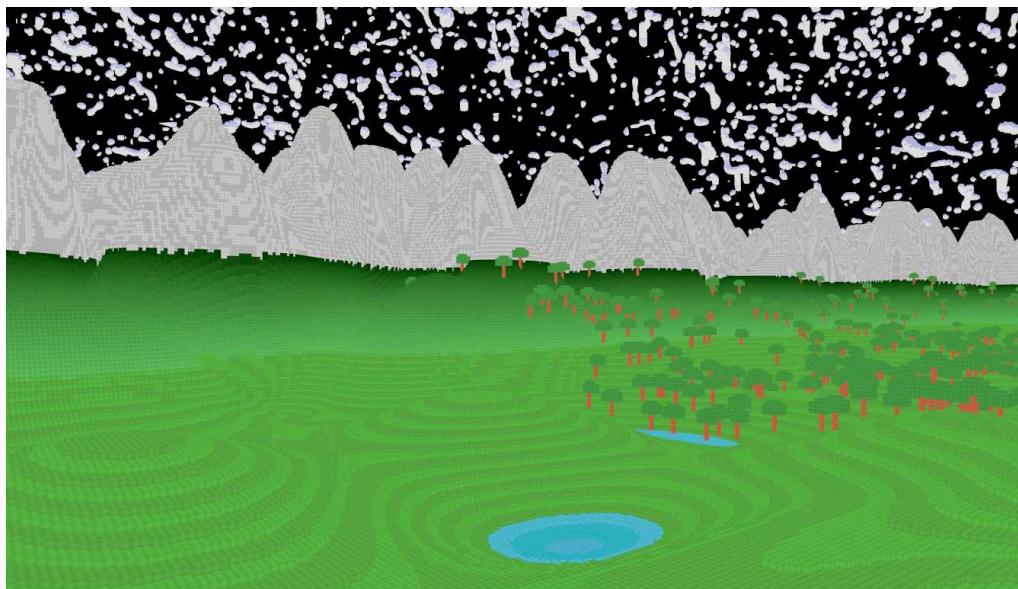
(a) Widok 3D z przygotowanego prototypu



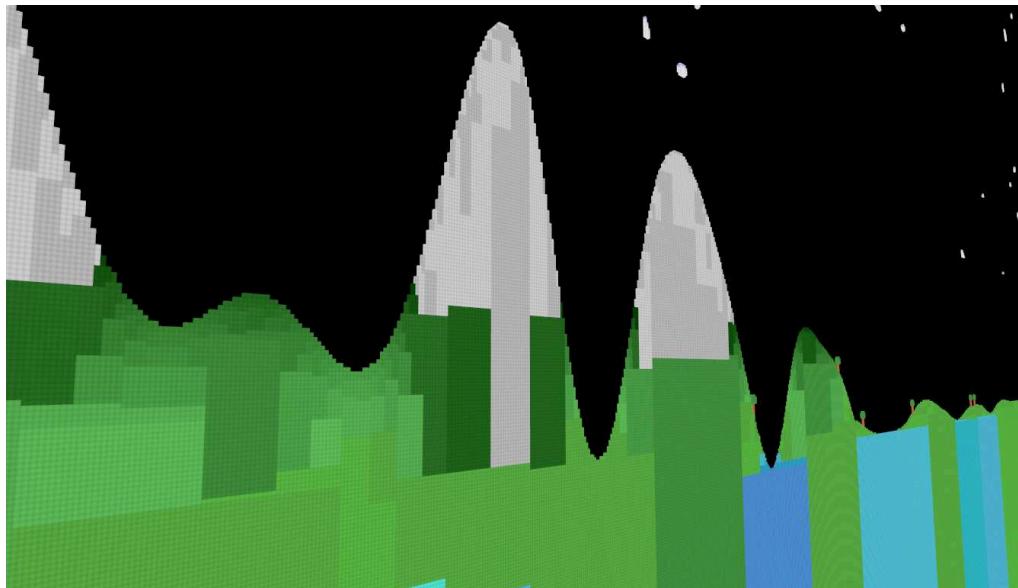
(b) Przekrój pionowy świata

Rys. 4.14: Przedstawienie świata proceduralnie generowanego budowanego na bazie drzewa czwórkowego (a), wraz z widocznym przekrojem pionowym (b).

11. **Realistyczny** (rys. 4.15) – świat proceduralnie generowany, którego teren budowany na bazie drzewa czwórkowego (podobnie jak w przypadku świata warstwicowego). Dla nieujemnych pozycji osi Z segmentów świata generowane są góry, dla ujemnych równiny. Dla ujemnych pozycji osi X segmentów świata generowane są drzewa. Dla pozycji większej bądź równej 2 osi Y generowane są chmury.



(a) Widok 3D z przygotowanego prototypu



(b) Przekrój pionowy świata

Rys. 4.15: Przedstawienie świata proceduralnie generowanego z górami, drzewami i chmurami (a), wraz z widocznym przekrojem pionowym (b).

4.8. Zaimplementowane narzędzia pomiarowe

Aby przeprowadzić badania wydajnościowe oraz pamięciowe niezbędne było wybranie narzędzi, które możliwie dokładnie ocenią działanie programu. Z tego względu odrzucono narzędzia zewnętrzne i zdecydowano się na takie, będące modułami dostępnymi w repozytorium paczek dla wybranego środowiska programistycznego – Criterion.rs [109, 110] oraz dhat-rs [111]. Wybór tych bibliotek podyktowany był ich dojrzałością, powtarzalnością uzyskanych wyników oraz szerokim wykorzystaniem w ekosystemie języka Rust.

4.8.1. Pomiar czasu działania (Criterion)

Biblioteka Criterion.rs służy do wykonywania statystycznie istotnych benchmarków. W odróżnieniu od prostych pomiarów czasu (np. z użyciem obiektu Instant¹⁷ służącemu operowaniu na czasie), Criterion automatycznie powtarza testy wielokrotnie, stosuje metody statystyczne (m.in. regresję liniową, *bootstrap*) oraz eliminuje szum wynikający z czynników zewnętrznych (np. harmonogramu systemu operacyjnego). Dzięki temu otrzymane rezultaty zawierają nie tylko wartość średnią czy medianę, ale również odchylenie standardowe i przedziały ufności.

Testowanie wykonywane przez narzędzie Criterion opiera się na wielokrotnym uruchamianiu badanej funkcji, zbieraniu próbek czasowych oraz analizie statystycznej co zapewnia stabilność wyników – w sposób automatyczny przeprowadza strojenie aby dostosować liczbę iteracji dla zadanego funkcji. Narzędzie to potrafi wykonywać tysiące iteracji (dla małych funkcji) oraz skalować ich liczbę tak, aby całkowity czas pomiaru pozostał akceptowalny. Po rzeprowadzeniu testów generuje raport, w którym znajduje się seria diagramów oraz wyniki w formie tekstuowej.

Do poprawnego przeprowadzenia pomiarów konieczne jest ograniczenie optymalizacji kompilatora na testowanych wartościach (np. usuwających dany kod uznając go za nieistotny dla działania programu). Rolę taką spełnia funkcja std::hint::black_box¹⁸, wykorzystana w przygotowanym module pomiarowym, którego fragment przedstawiono na rysunku 4.16.

```
let mut group = c.benchmark_group( "Structs initialization" );

group.bench_function( "VoxelMap (size 2)", |b| b.iter( || black_box( VoxelMap::<Voxel>::from_max_size( black_box( 2 ) ) ) ) );

group.bench_function( "Octree (size 1000)", |b| b.iter( || black_box( Octree::<Voxel>::from_max_size( black_box( 1000 ) ) ) ) );

group.bench_function( "VoxelList", |b| b.iter( || black_box( VoxelList::<Voxel>::new() ) ) );

group.bench_function( "VoxelHashMap", |b| b.iter( || black_box( VoxelHashMap::<Voxel>::new() ) ) );
```

Rys. 4.16: Przykładowy kod testowy napisany w języku *Rust* z użyciem biblioteki Criterion. Kod ten odpowiada za wygenerowanie raportu dla grupy testów mierzących czas inicjalizacji różnych struktur danych.

¹⁷<https://doc.rust-lang.org/std/time/struct.Instant.html>

¹⁸https://doc.rust-lang.org/beta/std/hint/fn.black_box.html

4.8.2. Pomiar zużycia pamięci (*dhat*)

Do mierzenia obciążenia pamięci wykorzystano dwa podejścia:

- **Metoda `std::mem::size_of<T>`**¹⁹ języka Rust pozwalająca zwracającą rozmiar pamięci pojedynczej instancji danego typu (w tym opisywanych struktur danych) na stosie, lecz nie uwzględnia alokacji na stercie (np. wektorów, danych za wskaźnikami).
- **Profiler pamięci `dhat-rs`**²⁰ będącego portem narzędzia `dhat`²¹, który rejestruje rzeczywiste alokacje podczas wykonywania programu. Dzięki niemu możliwe jest uchwycenie pełnego obrazu wykorzystania pamięci na stercie, łącznie z alokacjami dynamicznymi i zwolnieniami.

Biblioteka `dhat` rejestruje wszystkie dynamiczne alokacje i dealokacje pamięci sterty pozwalając na określenie:

- liczby wykonanych alokacji,
- szczytowego (ang. *peak*) użycia pamięci,
- pamięci utrzymanej po zakończeniu testu.

¹⁹https://doc.rust-lang.org/std/mem/fn.size_of.html

²⁰<https://docs.rs/dhat/latest/dhat/>

²¹<https://valgrind.org/docs/manual/dh-manual.html>

5. ANALIZA WYDAJNOŚCI I WŁAŚCIWOŚCI STRUKTUR DANYCH

Celem niniejszego rozdziału jest empiryczne porównanie wybranych struktur danych oraz technik optymalizacyjnych w kontekście prezentacji wielu obiektów w symulacjach i grach wideo. Przeprowadzone eksperymenty mają umożliwić wybór suboptymalnego rozwiązania do reprezentacji gęsto wypełnionych przestrzeni symulowanych (w tym światów wokselowych), a także ocenę wpływu wybranych metod optymalizacji na wydajność i zużycie pamięci.

Badania podzielono na dwa zasadnicze etapy. W pierwszym etapie przeprowadzono **mikrotesty** podstawowych operacji na strukturach danych: inicjalizacji, dodawania i usuwania elementów oraz wyszukiwania. Wyniki mikrotestów pozwoliły wskazać strukturę, która w ujęciu ogólnym najlepiej sprawdza się w zastosowaniach do przetwarzania wielu danych w przestrzeni.

W drugim etapie skupiono się na **testach makro**, obejmujących całe światy wokselowe przygotowane w formie zestawu scenariuszy testowych. W tym przypadku analizie poddano wyłącznie strukturę wyłonioną w mikrotestach jako najbardziej obiecującą, natomiast pozostałe struktury prezentowane są jedynie w sytuacjach, w których ich ograniczenia stają się szczególnie widoczne. Światy testowe obejmują zarówno proste przypadki graniczne (świat pusty, jednorodnie pełny), jak i bardziej złożone scenariusze proceduralne, co pozwala ocenić zachowanie badanych rozwiązań w warunkach zbliżonych do rzeczywistych zastosowań.

W testach makro uwzględniono również wybrane techniki optymalizacyjne, w tym **obcinanie bryłą widzenia**, **eliminacja niewidocznych ścian** oraz **przetwarzanie wielowątkowe**. Zamiast analizować je w oderwaniu od pozostałych eksperymentów, zostały one zastosowane kontekstowo – w ramach poszczególnych światów testowych. Dzięki temu możliwe jest ukazanie, w jakich sytuacjach dana technika przynosi realne korzyści, a kiedy jej wpływ jest marginalny. Takie podejście pozwala na lepsze zrozumienie praktycznej wartości poszczególnych technik – nie tylko w teorii, ale także w realistycznych scenariuszach, w których będą one faktycznie stosowane.

Wnioski uzyskane na podstawie przeprowadzonych badań stanowią podstawę do sformułowania rekomendacji dotyczących wyboru struktury danych oraz technik wspomagających jej praktyczne wykorzystanie w silnikach gier i symulacjach.

5.1. Platformy sprzętowe

Aby wszelkie badania były odtwarzalne i weryfikowalne, były przeprowadzane na tych samych konfiguracjach sprzętowych spełniających warunek środowiska operacyjnego (zob. 4.1.2). Zdecydowano się na użycie dwóch maszyn z różnych klas wydajnościowych – wyższej i niższej. Najważniejsze cechy istotne dla przebiegu badań obu platform zostały spisane w tabeli 5.1.

Tabela 5.1: Opis istotnych dla pracy podzespołów platformy sprzętowej klasy niższej

Platforma sprzętowa	Procesor	Karta graficzna	RAM
Klasa wyższa Komputer stacjonarny	AMD Ryzen 7 7800X3D Taktowanie: 4,20GHz Rdzenie: 8 Procesory logiczne: 16	NVIDIA GeForce RTX 4070 Ti SUPER	32 GB ¹
Klasa niższa Laptop	Intel(R) Core(TM) i7-1165G7 Taktowanie: 2.80GHz Rdzenie: 4 Procesory logiczne: 8	Intel(R) Iris(R) Xe Graphics (Zintegrowana)	16 GB

5.2. Porównanie referencyjne z grą Minecraft

Umiejscawiając wyniki uzyskane w ramach niniejszej pracy w kontekście praktycznym, przeprowadzono pomiary wydajności w popularnej grze *Minecraft* (zob. 2.4.3), stanowiącej naturalny punkt odniesienia. Testy wykonano na zdefiniowanych badawczych platformach sprzętowych (zob. 5.1). Pomiary obejmowały namierzenie najmniejszej, największej oraz stałej wartości liczby klatek na sekundę FPS (ang. *frames per second*).

Do przeprowadzania testów referencyjnych przygotowany został specjalny świat w grze, ustawiając globalnie jeden biom "równiny", usuwając wszystkie obiekty mobilne MOB (ang. *mobile object*) (np. zwierzęta) aby przetwarzany był jedynie teren oraz znajdując miejsce oddalone od wód aby ograniczyć używanie różnych programów cieniowania i możliwe zbliżyć wygląd terenu do przygotowanych światów testowych (rys. 5.1). Świat testowano szybkimi ruchami kamery w okolicach płaskiego terenu – aby był widoczny jak największy kawałek terenu wraz z niebem.

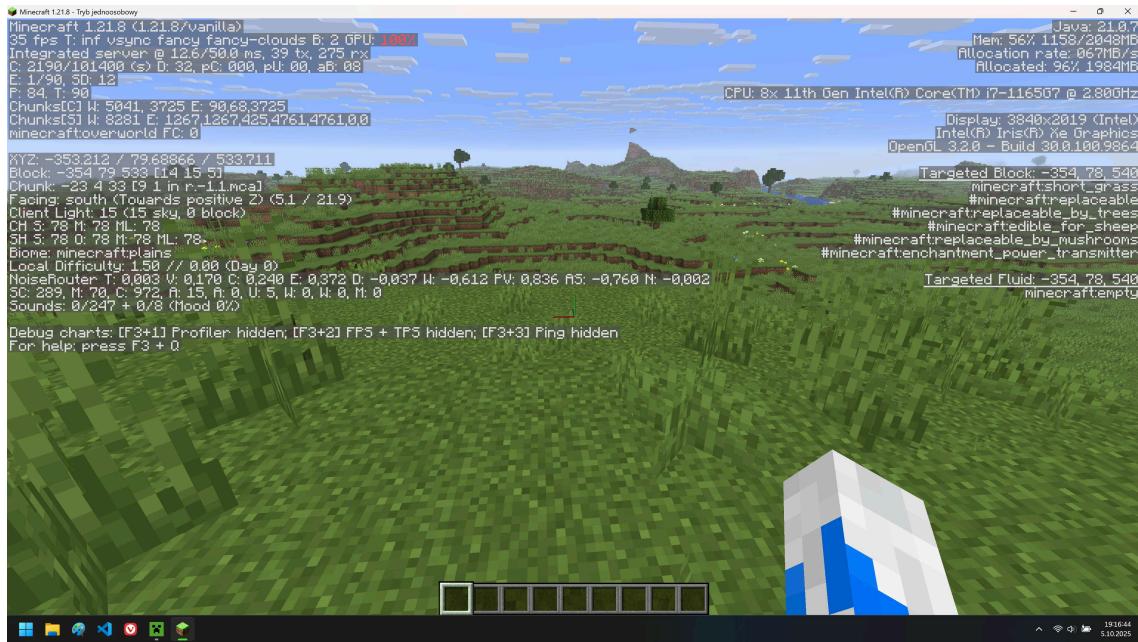
Zaobserwowane dane zapisano w tabeli 5.2 i stanowią one punkt odniesienia do późniejszych badań własnych, umożliwiając porównanie uzyskanej wydajności systemu wokselowego z rozwiązaniem stosowanym w jednej z najpopularniejszych gier komercyjnych.

Tabela 5.2: Zaobserwowane empirycznie wartości klatek na sekundę trybu debugowania gry *Minecraft* dla zdefiniowanych platform sprzętowych, odrzucając pojedyncze skokowe wartości skrajne.

Platforma sprzętowa	Min. FPS	Stabilny FPS	Maks. FPS
PC	288	340	379
Laptop	24	32	37

¹Jednostka GB została użyta dla klarowności przekazu w zamian za jednostkę GiB

https://figshare.com/.../SI_and_Binary_Prefixes_Clearing_the_Confusion_DOI-10_11453572027/26125696



Rys. 5.1: Zrzut ekranu pozyskany podczas gry w grę Minecraft na platformie sprzętowej klasy niższej (laptop) z włączonymi informacjami debugowania. Grafika obejmuje widoczny wskaźnik liczby klatek na sekundę pokazujący 35 FPS, oraz poziom użycia karty graficznej pokazujący 100%.

5.3. Badania struktur

Ta sekcja badań skupia się na przeprowadzonych mikrotestach podstawowych operacji na wybranych strukturach danych (zob. 4.4) – mapy wokseli, listy wokseli, tablice hashującej woksele, oraz drzewa ósemkowego *octree*. Badania tych struktur obejmowały następujące aspekty:

- **Inicjalizacja struktury** – czas potrzebny do utworzenia pustej instancji.
- **Wypełnianie** – czas wypełniania danymi wokselowymi.
- **Usuwanie** – czas usuwania elementów.
- **Zapytania** – czas wyszukiwania pojedynczego elementu oraz zestawu elementów.
- **Zużycie pamięci** – całkowity rozmiar struktury w pamięci operacyjnej.

Struktura mapa wokseli oraz drzewo ósemkowe są strukturami, które wymagają podania rozmiaru przy inicjalizacji. Z tego powodu w tabelach z wynikami oznaczano każdorazowo jaki rozmiar został przekazany strukturze do zainicjalizowania. Przykładowo, zapis "Mapa wokseli (dla 30³ wokseli)" oznacza wywołanie wywołanie metody tworzącej strukturę z rozmiarem 30 (co wyraża kod: `VoxelMap::<Voxel>::from_max_size(30)`).

Pomiary przeprowadzono narzędziem Criterion (zob. 4.8.1) a wyniki przedstawiono w postaci tabel z najważniejszymi danymi oraz w formie rysunków diagramów wiolinowych pezentujących rozkład gęstości prawdopodobieństwa PDF (ang. *probability density function*)² badanych operacji.

²https://bheisler.github.io/criterion.rs/book/user_guide/plots_and_graphs.html#pdf
https://amsi.org.au/ESA_Senior_Years/SeniorTopic4/4e/4e_2content_3.html
https://en.wikipedia.org/wiki/Probability_density_function

Pomiary wykonywano na obu przygotowanych platformach sprzętowych (tab. 5.1), lecz skupiono się na wynikach uzyskanych na platformie klasy niższej – jako słabsza platforma osiągała gorsze wyniki o rzędy wielkości, ale wyniki na niej uzyskane wyraźniej oddają trend (dane są bardziej rozrzucone na rozkładzie gęstości). Wyniki pomiarów dla platformy klasy wyższej zostały osadzone w załączniku A. Analiza i wnioski płynące z przeprowadzonych eksperymentów zostały ujęte w podrozdziale 5.3.6.

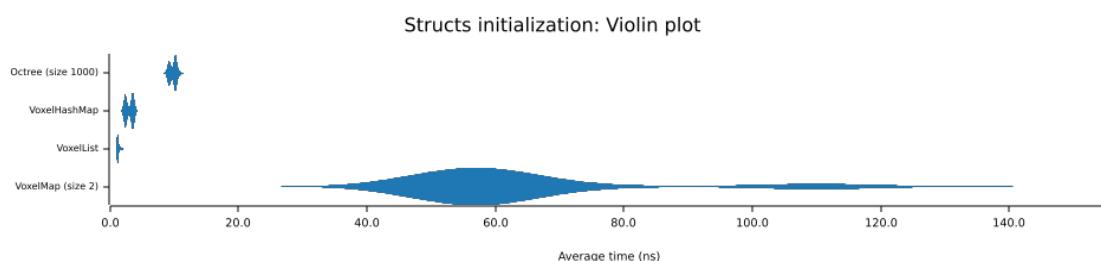
5.3.1. Pomiar czasu inicjalizacji struktur

Pierwszym etapem badań było zmierzenie czasu potrzebnego do zainicjalizowania pustej struktury danych. Czas inicjalizacji jest istotny szczególnie w przypadku aplikacji, gdzie struktury mogą być tworzone i niszczone dynamicznie (np. podczas ładowania nowych fragmentów świata). Wyniki przedstawiono na rysunkach 5.2, 5.3, 5.4 oraz zestawiono w tabeli 5.3. Wszelkie czasy podano w nanosekundach.

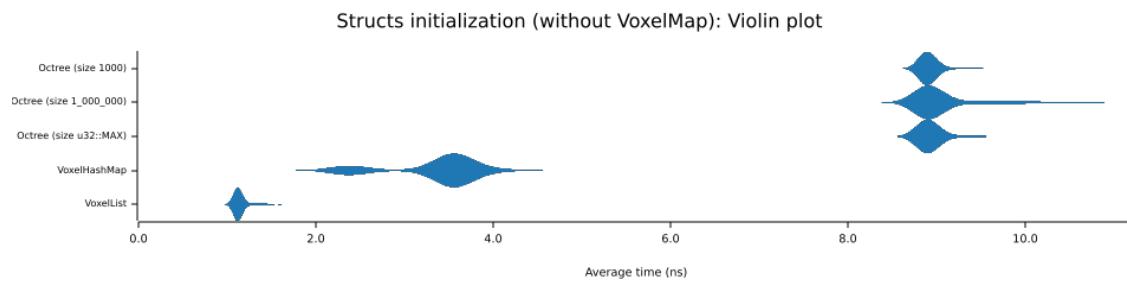
Ze względu na to, że mapa wokseli o przechowywanym rozmiarze świata zaledwie 8 wokseli (2 na 2 na 2) znaczco odbiega swoim wynikiem od pozostałych struktur danych (rys. 5.2), badania te zostały powtórzone bez uwzględnienia mapy wokseli (rys. 5.3) oraz zostały przeprowadzone dodatkowe pomiary tylko dla mapy wokseli z różną konfiguracją (tab. 5.4, rys. 5.4). W wynikach można zauważyć, że czas inicjalizacji mapy wokseli powiązany jest z jej rozmiarem, gdzie największy użyty tam rozmiar świata (30 na 30 na 30 wokseli) jest o wiele wyższy niż największe rozmiary innych struktur.

Tabela 5.3: Wyniki pomiarów średniego czasu inicjalizacji badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [ns]	Odchylenie standardowe [ns]
Mapa wokseli (dla 2^3 wokseli)	64.7501	19.5444
Lista wokseli	1.3022	0.2000
Tablica hashująca woksele	3.0660	0.5842
Octree (dla $1\ 000^3$ wokseli)	9.8835	0.5496
Octree (dla $1\ 000\ 000^3$ wokseli)	9.0750	0.3874



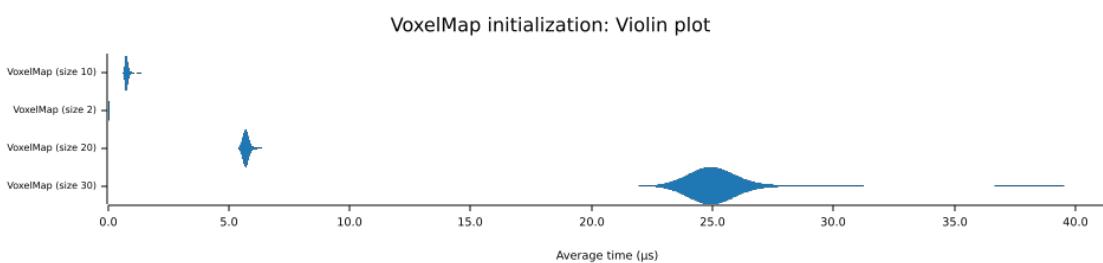
Rys. 5.2: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa PDF (ang. *Probability Density Function*) dla pomiarów średniego czasu inicjalizacji badanych struktur danych. Na diagramie widnieje znaczco odstający PDF dla mapy wokseli, co utrudnia ocenę czasu dla pozostałych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.



Rys. 5.3: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu inicjalizacji badanych struktur danych. Z badanych struktur wyłączona została *mapa wokseli* a *octree* zostało powtórzone dla różnych rozmiarów przechowywanych danych. W porównaniu struktura *octree* wypada najgorzej, natomiast dla wszystkich ocenianych rozmiarów tej struktury (stosunkowo małych oraz skrajnie dużych) czas inicjalizacji pozostaje taki sam. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia *Criterion* w środowisku *Rust* na maszynie klasy niższej.

Tabela 5.4: Wyniki pomiarów średniego czasu inicjalizacji **mapy wokseli**. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [μs]	Odchylenie standardowe [μs]
Mapa wokseli (dla 2^3 wokseli)	0.0501	0.0031
Mapa wokseli (dla 10^3 wokseli)	0.7991	0.0994
Mapa wokseli (dla 20^3 wokseli)	5.7554	0.2229
Mapa wokseli (dla 30^3 wokseli)	25.3799	2.0284



Rys. 5.4: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów średniego czasu inicjalizacji struktury **mapy wokseli** o różnych rozmiarach. Widoczny jest na diagramie trend potęgowy rosnącego czasu inicjalizacji względem zadanego (bardzo małego) rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia *Criterion* w środowisku *Rust* na maszynie klasy niższej.

5.3.2. Pomiar czasu dodawania danych

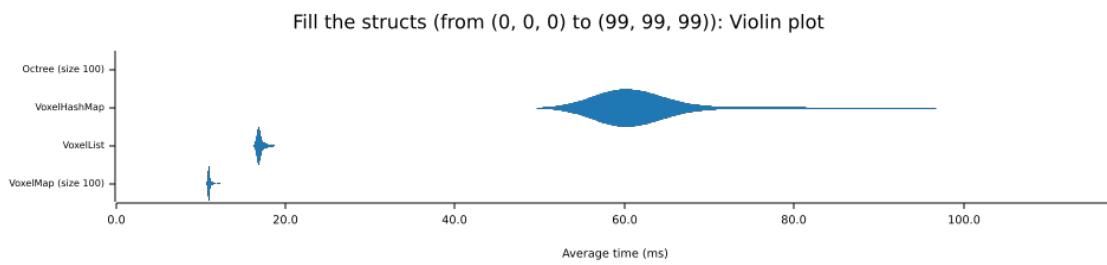
Kolejnym etapem badań było zmierzenie czasu potrzebnego do zbudowania struktury danych poprzez wypełnienie jej danymi wokselowymi. Pomiary te odzwierciedlają sytuacje typowe dla aplikacji i gier wideo, w których podczas generowania lub ładowania świata konieczne jest szybkie utworzenie odpowiednich struktur pamięciowych, przechowujących dużą liczbę obiektów.

W testach odróżniono pojęcie "wypełniania" oznaczającego pojedynczą instrukcję masowego wstawiania danych do struktury, oraz "wstawiania" oznaczającego dodawanie po jednej danej do struktury. Testy dodawania danych rozpisano następująco:

1. całkowite wypełnienie świata;
2. wypełnienie świata z marginesem;
3. wstawianie stu wokseli w rzędzie;
4. wstawianie wokseli w losowe, niepowtarzające się komórki. Istotnym dla badania jest zaznaczenie, że losowość zdefiniowano deterministycznie (z ziarnem) i dla każdej badanej struktury wybrane zostały takie same pozycje do wstawiania.

Tabela 5.5: Wyniki pomiarów średniego czasu **całkowitego wypełniania** badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop), przy użyciu narzędzia Criterion.

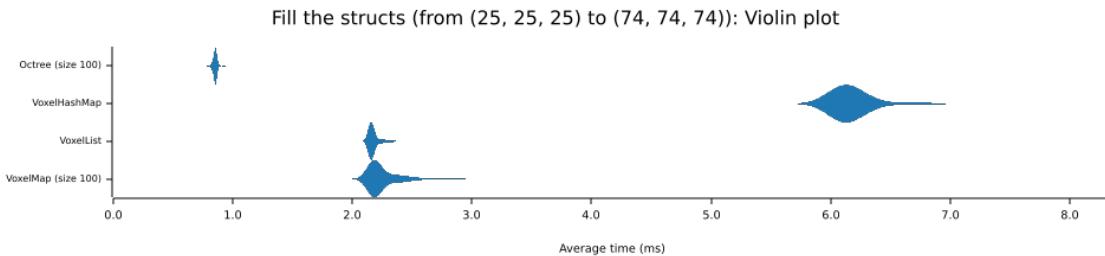
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	11.0467	0.2161
Lista wokseli	17.0686	0.5023
Tablica haszująca woksele	63.1969	8.5214
Octree (dla 100^3 wokseli)	0.1145	0.0034



Rys. 5.5: Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **całkowitego wypełniania** badanych struktur danych. Na diagramie charakterystyczne są wyniki dla octree, które są niewidoczne ze względu na swoją wielkość oraz dla *struktury haszującej* mającej znacznie dłuższy czas wypełniania względem pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.6: Wyniki pomiarów średniego czasu **częściowego wypełniania** badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

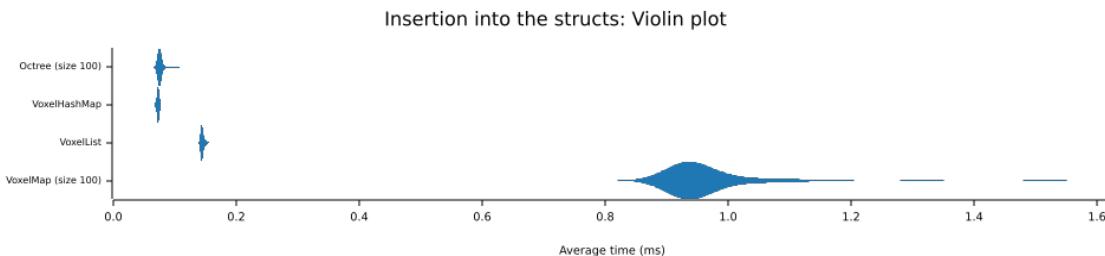
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	2.2536	0.1321
Lista wokseli	2.1807	0.0541
Tablica haszująca woksele	6.2096	0.2904
Octree (dla 100^3 wokseli)	0.8560	0.0181



Rys. 5.6: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **częściowego wypełniania** badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.7: Wyniki pomiarów średniego czasu **wstawiania rzędu** stu wokseli do badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

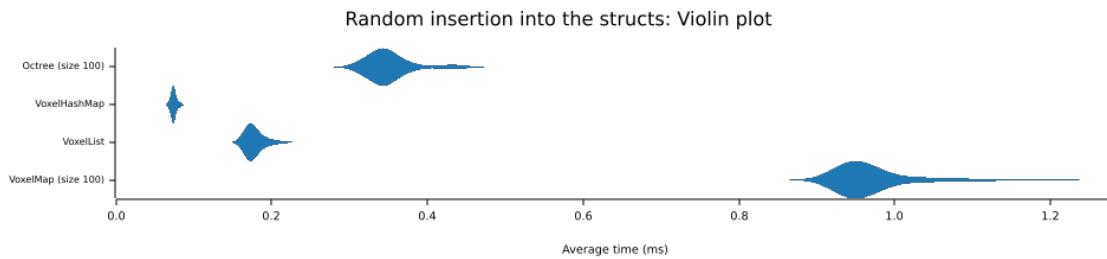
Struktura danych	Średni czas [μ s]	Odchylenie standardowe [μ s]
Mapa wokseli (dla 100^3 wokseli)	967.3438	83.0148
Lista wokseli	146.2720	4.0980
Tablica haszująca woksele	73.6895	1.9668
Octree (dla 100^3 wokseli)	77.2811	5.1017



Rys. 5.7: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **wstawiania rzędu** stu wokseli do badanych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.8: Wyniki pomiarów średniego czasu wstawiania wokseli w tysiąc **deterministycznie losowych** pozycji do badanych struktur danych. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [μs]	Odchylenie standardowe [μs]
Mapa wokseli (dla 100^3 wokseli)	975.1290	57.3585
Lista wokseli	181.1534	20.2489
Tablica haszująca woksele	75.1274	5.4231
Octree (dla 100^3 wokseli)	352.1219	31.0360



Rys. 5.8: Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów wstawiania wokseli w tysiąc **deterministycznie losowych** pozycji, identycznych dla każdej struktury. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

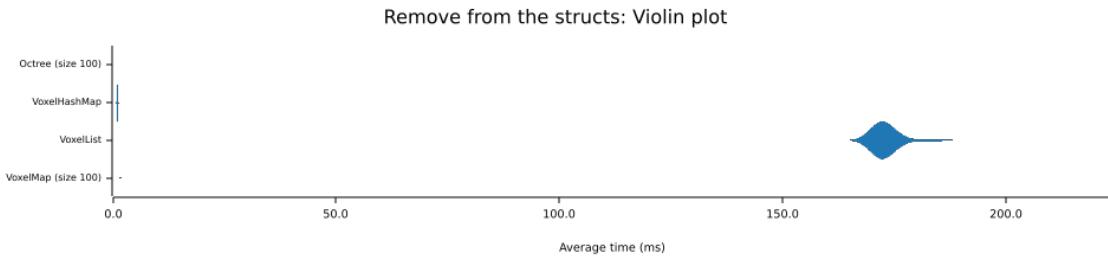
5.3.3. Pomiar czasu usuwania danych

Następnym etapem badań było zmierzenie czasu usuwania elementów ze struktur danych. Każda ze struktur przed rozpoczęciem testów usuwania była inicjalizowana dla rozmiaru 100^3 wokseli i była wypełniana w zakresie od pozycji (5, 5, 5) do (94, 94, 94). Badania przeprowadzono w dwóch wariantach:

- **usuwanie pojedyncze** – usuwanie całych wierszy kasując po jednym wokselu
- **usuwanie masowe** – polegające na usunięciu ze struktury dużego obszaru;

Tabela 5.9: Wyniki pomiarów średniego czasu **pojedynczego usuwania** wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

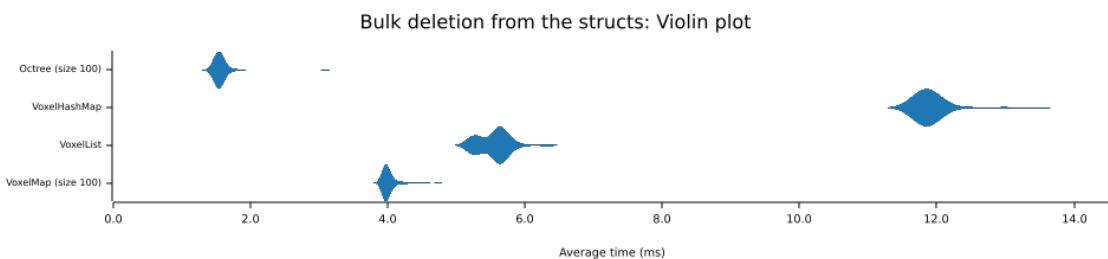
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	1.4951	0.0720
Lista wokseli	173.8683	5.6739
Tablica hashująca woksele	1.0121	0.1109
Octree (dla 100^3 wokseli)	0.4430	0.0252



Rys. 5.9: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **pojedynczego usuwania** wokseli z badanych struktur. Wyniki *listy wokseli* zmarginalizowały wyniki pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.10: Wyniki pomiarów średniego czasu **masowego usuwania** wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

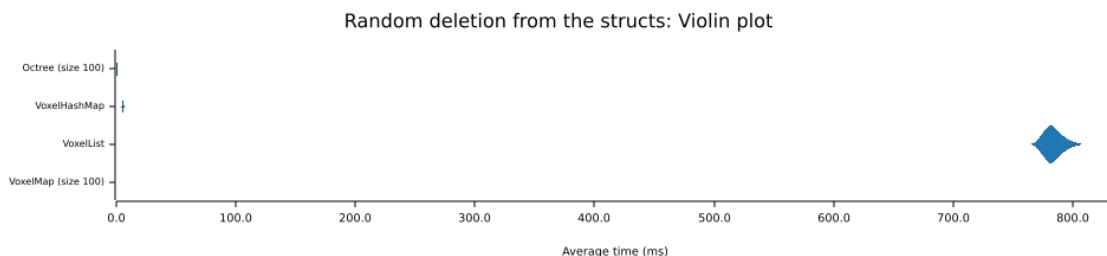
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	4.0248	0.1250
Lista wokseli	5.5630	0.2428
Tablica hashująca woksele	11.9698	0.4050
Octree (dla 100^3 wokseli)	1.5762	0.1661



Rys. 5.10: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **masowego usuwania** obszaru z badanych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.11: Wyniki pomiarów średniego czasu deterministycznie losowego usuwania tysiąca wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	1.4970	0.0714
Lista wokseli	783.8320	8.0803
Tablica hashująca woksele	5.7711	0.4969
Octree (dla 100^3 wokseli)	1.0759	0.0841



Rys. 5.11: Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie losowego usuwania tysiąca wekseli z badanych struktur. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

5.3.4. Pomiar czasu wyszukiwania danych

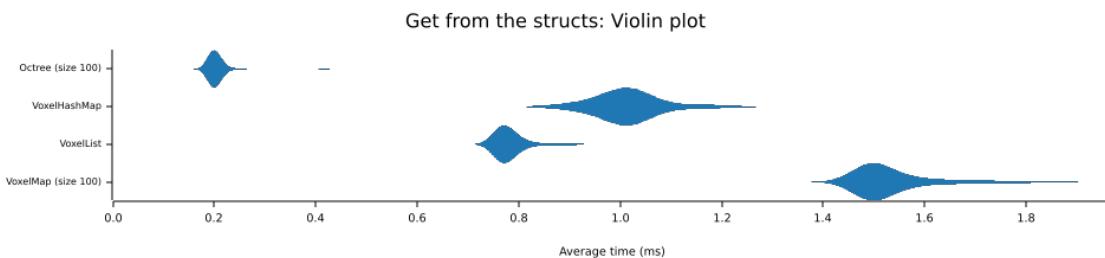
Ostatnim etapem badań wydajnościowych dla poszczególnych struktur danych było zmierzenie czasu potrzebnego do ich przeszukiwania. Operacja wyszukiwania jest jedną z najczęściej wykonywanych w kontekście symulacji i gier wideo, gdyż odpowiada m.in. za sprawdzanie obecności danego woksela w świecie, pobieranie jego właściwości lub określenie kolizji.

Test polegał na przygotowaniu struktur danych wypełnionych ustaloną liczbą wokseli, a następnie przeprowadzaniu powtarzanych zapytań o woksele występujące po sobie oraz o woksele w losowej lokalizacji. Pomiar przeprowadzono narzędziem Criterion (zob. 4.8.1), a uzyskane wartości obejmowały średni czas operacji oraz odchylenie standardowe. Wyniki zestawiono w tabelach 5.12 i 5.13 oraz zilustrowano na rysunkach 5.12 i 5.13.

Struktura mapy wokseli będąca tablicą trójwymiarową, która intuicyjnie mogłaby uchodzić za najbardziej wydajną ze względu na prosty dostęp indeksowany, w praktyce okazała się nie być najwydajniejszą, a wręcz uzyskała najslabszy wynik przy dostępie ciągłym. Takie zachowanie sprowokowało przeprowadzenie pogłębionego badania jedynie struktur tablicowych. Wyniki zebrane w tabeli 5.14 oraz przedstawiono je na rysunku 5.14.

Tabela 5.12: Wyniki pomiarów średniego czasu **wyszukiwania następujących po sobie** stu wokseli w strukturach danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

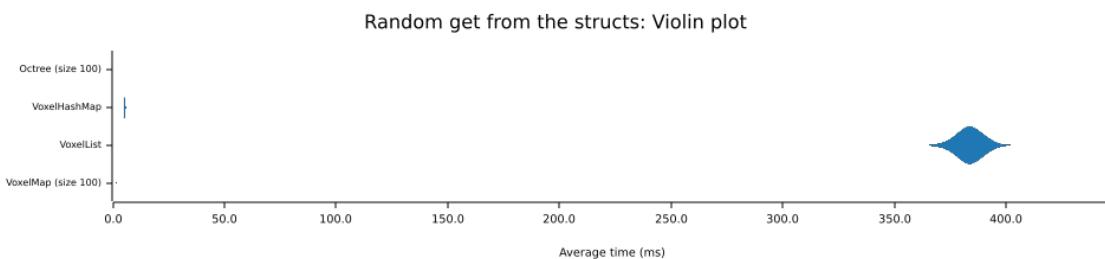
Struktura danych	Średni czas [μs]	Odchylenie standardowe [μs]
Mapa wokseli (dla 100^3 wokseli)	1 540.2146	84.5184
Lista wokseli	787.9627	46.3643
Tablica hashująca woksele	1 026.6862	101.6829
Octree (dla 100^3 wokseli)	204.3862	24.4586



Rys. 5.12: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **wyszukiwania następujących po sobie** stu wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.13: Wyniki pomiarów średniego czasu deterministycznie **losowego wyszukiwania** tysiąca wokseli w strukturach danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

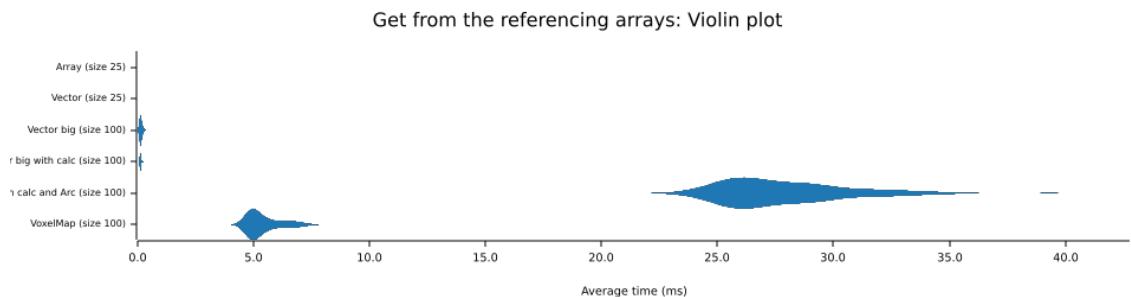
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	1.4895	0.0716
Lista wokseli	384.5694	8.3695
Tablica hashująca woksele	5.5230	0.3060
Octree (dla 100^3 wokseli)	0.6750	0.0517



Rys. 5.13: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie **losowego wyszukiwania** tysiąca wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

Tabela 5.14: Wyniki pomiarów średniego czasu **przeszukiwania tablicach i wektorach** o różnych konfiguracjach. Wyniki te wskazują, że powodem drastycznego spowolnienia czasu przeszukiwania są najprawdopodobniej przechowywane wartości atomowe (`Arc`). Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Tablice (dla 25^3 wokseli)	0.0049	0.0011
Wektor (dla 100^3 wokseli)	0.0025	0.0018
-/- z obliczanym indeksem	0.1467	0.0421
-/- i licznikami atomowymi	27.9269	2.7124
Mapa wokseli (dla 100^3 wokseli)	5.4492	0.7238



Rys. 5.14: Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **przeszukiwania tablic i wektorów** o różnych konfiguracjach. Na diagramie uwidoczniono konfigurację powodującą największe obciążenie czasowe podczas przeszukiwania – wektor przechowujący liczniki atomowe (`Arc`). Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy niższej.

5.3.5. Pomiar rozmiaru w pamięci operacyjnej

Oprócz czasu działania istotnym parametrem badanych struktur danych jest ich zapotrzebowanie na pamięć operacyjną. W przypadku symulacji i gier komputerowych, w których w pamięci mogą znajdować się miliony wokseli, rozmiar struktur danych bezpośrednio wpływa na wydajność systemu oraz możliwość uruchomienia aplikacji na urządzeniach o ograniczonych zasobach.

Pomiar przeprowadzono narzędziami Criterion i dhat (zob. 4.8.2) w następujących dwóch etapach:

1. Najpierw oszacowano rozmiar pojedynczej instancji każdej ze struktur na stosie przy pomocy funkcji `std::mem::size_of..`
2. Następnie wykorzystano profiler dhat do zarejestrowania wykorzystanej pamięci na stercie. Dla każdej struktury zmierzono całkowity przydział pamięci, liczbę alokacji oraz szczytowe użycie w trzech scenariuszach: strukturze pustej, wypełnionej w 50% oraz 100%.

Wyniki zestawiono w tabelach 5.15, 5.16, 5.17 oraz 5.18, a także zilustrowano wybranymi wykresami generowanymi przez narzędzie dhat.

Tabela 5.15: Rozmiary pojedynczej instancji struktur danych oraz narzut na każdą dodawaną wartość zmierzony funkcją `std::mem::size_of`

Struktura danych	Rozmiar instancji [B]	Narzut na wartość [B]
Mapa wokseli	56	0
Lista wokseli	24	24
Tablica hashująca	48	12
Octree	24	16

Tabela 5.16: Maksymalne zużycie pamięci, liczba alokacji i ich sumaryczny rozmiar dla pustych, zainicjowanych struktur zmierzone narzędziem `dhat`

Struktura danych	Maks [kB] ³	Liczba alokacji	Suma [kB]
Mapa wokseli (dla 100^3 wokseli)	8 000	1	8 000
Lista wokseli	0	0	0
Tablica hashująca	0	0	0
Octree (dla 100^3 wokseli)	0	0	0

Tabela 5.17: Maksymalne zużycie pamięci, liczba alokacji i ich sumaryczny rozmiar dla struktur wypełnionych w 50% zmierzone narzędziem `dhat`

Struktura danych	Maks [kB]	Liczba alokacji	Suma [kB]
Mapa wokseli (dla 100^3 wokseli)	8 000	17	8 000
Lista wokseli	12 000	17	12 000
Tablica hashująca	26 214	17	26 215
Octree (dla 100^3 wokseli)	135	1 076	136

Tabela 5.18: Maksymalne zużycie pamięci, liczba alokacji i ich sumaryczny rozmiar dla struktur wypełnionych w 100% wg `dhat`. Pomiary uzupełniono o dwa skrajne dodatkowe przypadki dla struktury octree, aby pokazać jaki wpływ na obciążenie pamięci ma wypełnienie całej struktury w jej wewnętrznej, binarnej hierarchii oraz dla tego samego wypełnienia z marginesem jednej komórki.

Struktura danych	Maks [kB]	Liczba alokacji	Suma [kB]
Mapa wokseli (dla 100^3 wokseli)	8 000	17	8 000
Lista wokseli	24 000	17	24 000
Tablica hashująca	52 428	17	52 429
Octree (dla 100^3 wokseli)	82	657	82
Octree (dla 128^3 wokseli)	0.802	16	0.812
-/- od (1,1,1) do (126,126,126)	4 006	31 313	4 006

³Jednostka *kB* została użyta dla klarowności przekazu w zamian za jednostkę *KiB*

https://figshare.com/.../SI_and_Binary_Prefixes_Clearing_the_Confusion_DOI-10_11453572027/26125696

5.3.6. Analiza wyników i wnioski

Przeprowadzone pomiary ujawniły istotne różnice w zachowaniu badanych struktur danych. Zebrano je w postaci wniosków odnoszących się do każdej z badanych struktur danych:

- **Mapa wokseli** – intuicyjnie mogłaby uchodzić za najbardziej wydajną obliczeniowo strukturę ze względu na prosty dostęp indeksowany, w praktyce natomiast uzyskała jedne z najsłabszych wyników. Przyczyną może być konieczność rezerwacji dużych obszarów pamięci, co skutkuje wysokim kosztem inicjalizacji i obniżeniem lokalności odwołań w pamięci podrzędnej procesora. Pewnym powodem jest natomiast użycie licznika atomowego jako przechowywanej wartości, co dobrze oddają wyniki inicjalizacji (tab. 5.3, rys. 5.4) i przeszukiwania (tab. 5.14). Problem inicjalizacji uwidaczniają także testy inicjalizacji, gdzie struktura ta jako jedyna zajmuje przestrzeń na stosie (tab. 5.16). Pozostałe testy alokacji pamięci prezentują tę strukturę jako najlepszą – rozmiar zmieniał się minimalnie (zaniedbywalnie), zaś liczba alokacji pozostaje stałą bez względu na liczbę elementów (tab. 5.17, 5.18).
- **Lista wokseli** – głównym problemem obniżającym wydajność tej struktury okazała się potrzeba przeszukiwania wszystkich przechowywanych danych w przypadku braku posiadania szukanej danej – problem ten objawia się w sytuacji dodawania (tab. 5.7, 5.8), usuwania (tab. 5.9, 5.11), oraz pobierania (tab. 5.13) danych. Hipotetyczna implementacja sortowania przechowywanych danych mogłaby przyspieszyć niektóre operacje, lecz znaczco mogłaby spowolnić dodawanie i usuwanie ze względu na konieczność rozszerzania rozmiaru i przenoszenia danych w pamięci. Podobnie jak w przypadku mapy wokseli liczba alokacji pozostaje niezmienna, natomiast rozmiar zwiększa się proporcjonalnie do ilości przechowywanych danych (tab. 5.16, 5.17, 5.18).
- **Tablica hashująca** – okazała się strukturą wyważoną w stosunku do wyżej opisanych struktur tablicowych – badania wykazują, że jej wydajność wypada pomiędzy nimi. Dzięki zastosowaniu funkcji mieszającej dostęp do elementów jest średnio stałoczasowy (tab. 5.12, 5.13), natomiast stanowi główny koszt obliczeniowy tego rozwiązania. Czynnikiem ograniczającym jest narzucona pamięciowa związana z samą tablicą i jej mechanizmem rozszerzania – struktura ta wypada najgorzej w pomiarach objętości pamięci (tab. 5.17, 5.18).
- **Octree** – charakteryzowało się najlepszą równowagą pomiędzy czasem operacji a zużyciem pamięci. Dzięki hierarchicznemu podziałowi przestrzeni pozwala szybko odrzucać duże fragmenty przestrzeni przy wyszukiwaniu komórki (tab. 5.12) oraz dynamicznie rozszerzać się i kurczyć w ramach kompresji (tab. 5.3, 5.18). Struktura ta potrafi osiągnąć gorsze (lecz nie najgorsze) wyniki przy operacjach na pojedynczych danych (tab. 5.8).

Na podstawie przeprowadzonych eksperymentów można stwierdzić, że najlepszym wyborem dla implementacji świata wokselowego o dużych rozmiarach o zróżnicowanym wypełnieniu jest struktura *octree*. Struktura ta powinna sprawdzić się bardzo dobrze także w innych typach światów, ponieważ jej objętość pamięciowa jest zależna od ilości przechowywanych danych i to przy gęstych światach (takich jak wokselowe) wydajność powinna być najgorsza.

5.4. Badania na podstawie światów testowych

Dotychczasowe analizy obejmowały pomiary podstawowych operacji na strukturach danych, takich jak inicjalizacja, dodawanie, usuwanie czy wyszukiwanie elementów. Na ich podstawie wyłoniono strukturę, która najlepiej spełnia wymagania stawiane w kontekście niniejszej pracy — **octree**. W kolejnej fazie badań skupiono się więc wyłącznie na tej strukturze, aby sprawdzić jej zachowanie w bardziej złożonych i praktycznych scenariuszach.

W tym celu przygotowano jedenaście światów *testowych* (zob. 4.7). Światy te różnią się stopniem zapełnienia, rozkładem danych oraz charakterystyką przestrzenną (np. świat pusty, świat losowy, świat z dominacją powietrza, świat proceduralnie generowany). Stanowią one modelowe przypadki środowisk, w jakich może być wykorzystywana implementacja systemu wokselowego. Dzięki nim możliwe jest przejście od testów syntetycznych do bardziej realistycznych pomiarów.

Badania przeprowadzono dwuetapowo:

1. **Badania wszystkich światów na strukturze octree** — dla każdego ze światów zmierzono wydajność poprzez analizę 1000 ostatnich pomiarów klatek na sekundę działającego programu. W tabelach wynikowych zestawiono średnią liczbę klatek na sekundę, odchylenie standardowe, minimalny oraz maksymalny FPS z uwzględnieniem wartości skrajnych. Przedstawienie wyników z oraz bez wartości skrajnych ma pokazać zarówno zakłócenia wynikające z działania aplikacji oraz jej charakter ogólny.

2. **Badania wybranego świata proceduralnego** — na bardziej złożonym świecie proceduralnym porównano:

- działanie octree z innymi (wcześniej odrzuconymi) strukturami danych, co pozwoliło ostatecznie uzasadnić wybór,
- wpływ zastosowania technik optymalizacyjnych (wielowątkowość, frustum culling, renderowanie ze spodem opisu danych *TRIANGLE_STRIP*, eliminacja niewidocznych ścian).

Taki wybór uzasadniono tym, że techniki optymalizacyjne mają największe znaczenie w przypadku światów złożonych i bogatych w dane, dlatego ich badanie w prostszych środowiskach (np. pustych) nie wnosiłoby wartościowych wniosków.

Przedstawione w tej sekcji wyniki pozwalają nie tylko ocenić efektywność struktury octree w różnych modelach światów, ale także wykazać praktyczne znaczenie wybranych technik optymalizacyjnych w kontekście prezentacji dużych i złożonych scen.

5.4.1. Badania światów na strukturze octree

Pierwszy etap badań koncentruje się na ocenie wydajności wybranej struktury *octree* w jedenastu różnych światach testowych opisanych w rozdziale 4.7. Każdy ze światów charakteryzuje się odmiennym stopniem wypełnienia oraz złożonością geometryczną, co pozwala sprawdzić działanie systemu zarówno w przypadkach skrajnie prostych (np. świat pusty, pojedynczy woksel), jak i bardziej realistycznych (np. świat proceduralny czy świat oparty na mapie wysokościowej).

Dla każdego ze światów wykonano pomiar liczby klatek na sekundę FPS podczas przemieszczania się kamery po scenie. Analiza oparta została na ostatnich 1000 próbkach, a w tabelach zestawiono następujące statystyki:

- średnią liczbę FPS,
- odchylenie standardowe,
- wartości minimalne i maksymalne,
- wartości minimalne i maksymalne po odrzuceniu pojedynczych skrajnych obserwacji, które mogły wynikać z zakłóceń systemowych lub chwilowych opóźnień niezwiązanych z implementacją.

Pomiar został przeprowadzony na każdej z przedstawionych w tabeli 5.1 platformie sprzętowej. Takie porównanie pozwala ocenić, w jakim stopniu złożoność świata wpływa na wydajność w zależności od zasobów sprzętowych oraz jak skalowalna jest zastosowana metoda reprezentacji.

Tabela 5.19: Statystyki FPS dla przemieszczania się po światach testowych z sekcji 4.7 dla komputera klasy wyższej (tab. 5.1).

Świat testowy	Średni FPS	Odch. std.	Min. FPS	Maks. FPS	Min. odst. FPS	Maks. odst. FPS
Pusty	1831.45	312.65	912.99	2142.25	705.07	2160.76
Pojedynczy woksel	1631.21	332.07	674.90	1913.88	469.81	2016.94
Poł. wypełniony	866.35	172.83	508.23	1456.66	390.24	1802.45
Poł. losowy	63.00	34.03	29.64	161.04	24.74	310.60
Bez woksela	720.50	110.03	449.86	868.21	373.59	891.74
Wypełniony	1453.55	387.86	639.39	1943.63	415.77	1986.49
Poł. losowo różnorodny	35.72	9.73	24.86	77.91	22.74	99.01
Wypełniony różnorodnie	689.67	120.21	421.34	869.94	327.65	911.08
Simplex	378.91	41.47	289.75	459.47	233.11	486.71
Mapa wysokościowa	364.09	340.44	71.77	1271.29	61.26	1587.30
Realistyczny	241.33	205.86	22.47	760.40	16.50	902.85

Tabela 5.20: Statystyki FPS dla przemieszczania się po światach testowych z sekcji 4.7 dla komputera klasy niższej (tab. 5.1)

Świat testowy	Średni FPS	Odch. std.	Min. FPS	Maks. FPS	Min. odst. FPS	Maks. odst. FPS
Pusty	61.20	9.14	45.12	91.83	27.99	148.26
Pojedynczy woksel	60.69	7.41	46.01	78.15	18.98	150.49
Poł. wypełniony	45.26	12.09	23.74	73.62	16.69	141.56
Poł. losowy	6.34	3.64	2.38	23.39	2.01	34.13
Bez woksela	27.92	5.23	19.00	44.20	14.76	65.66
Wypełniony	26.55	5.34	17.76	42.83	15.59	52.22
Poł. losowo różnorodny	5.60	2.31	2.40	11.46	2.17	14.57
Wypełniony różnorodnie	23.97	3.80	16.44	36.24	11.45	44.83
Simplex	20.34	4.93	11.85	33.60	8.48	44.88
Mapa wysokościowa	22.77	4.53	15.09	34.56	6.47	45.05
Realistyczny	22.33	8.05	11.56	42.69	9.58	53.90

Na komputerze klasy wyższej w prostych scenariuszach (*świat pusty*, *pojedynczy woksel*) osiągane wartości znaczco przekraczały 1000 FPS, co świadczy o bardzo niskich kosztach renderowania przy minimalnym zapełnieniu struktury. Na komputerze klasy niższej analogiczne przypadki pozwalały na utrzymanie stabilnych wartości w granicach 60 FPS, a więc na poziomie umożliwiającym komfortową interakcję.

Wraz ze wzrostem złożoności świata wartości FPS spadały, przy czym największe obciążenie generowały światy o losowym rozmieszczeniu wokseli (*połowicznie losowy*, *połowicznie losowo różnorodny*). Na komputerze wyższej klasy średnie wartości w tych przypadkach spadały do ok. 40 FPS, natomiast na sprzęcie słabszym osiągały poziom jedynie 5–7 FPS, co w praktyce oznaczało całkowitą utratę płynności. Wynik ten potwierdza, że nieregularna i nieprzewidywalna struktura danych jest szczególnie trudnym przypadkiem dla wydajności systemu. W takich scenariuszach przydatną techniką optymalizacyjną mogłyby okazać się obcinanie okluzji (ang. *occlusion culling*) polegające na odrzucaniu zasłoniętych obiektów przez inne obiekty.

Światy o bardziej uporządkowanej strukturze, takie jak *mapa wysokościowa* czy *świat realistyczny*, dawały wyniki pośrednie. W przypadku mapy wysokościowej FPS był stosunkowo niski, ale stabilny, co sugeruje równomierne obciążenie. Natomiast świat realistyczny charakteryzował się bardzo dużymi wahaniem klatek na komputerze klasy wyższej (od 22 do ponad 760 FPS), co odzwierciedla znaczną niejednorodność sceny i uzależnienie wydajności od aktualnego pola widzenia oraz dynamiki kamery.

5.4.2. Badanie wpływu wybranych technik optymalizacyjnych

Dotychczasowe pomiary dla różnych światów testowych pozwoliły ocenić zachowanie struktury Octree w zróżnicowanych scenariuszach. Do przeprowadzenia badań nad technikami optymalizacyjnymi wybrano świat *mapa wysokościowa*.

W tej części pracy przeprowadzono analizę wpływu wybranych technik optymalizacyjnych na wydajność systemu wokselowego. Badania oparto na świecie testowym *mapa wysokościowa*, który charakteryzuje się dużą lecz prostą złożonością i akceptowalnie odwzorowuje praktyczne scenariusze użycia.

Pomiary podzielono na dwie grupy:

- techniki związane z prezentowaniem obiektów na ekranie,
- techniki związane z równoległym generowaniem danych na wątkach roboczych. brak wątków roboczych oznacza zawieszenie programu na czas generowania danych, dlatego nie przedstawiono tego scenariusza w badaniach.

Pierwsza grupa została zbadana w oparciu o statystyki FPS dla wariantów z wyłączeniem wybranych metod renderingu (tab. 5.21), natomiast druga na podstawie czasu generowania świata przy różnej liczbie wątków roboczych (tab. 5.22).

Tabela 5.21: Wpływ wybranych technik prezentowania obiektów na ekranie, na bazie świata testowego "mapa wysokościowa".

Zastosowana konfiguracja	Średni FPS	Odch. std.	Min.	Maks.	Min. odst.	Maks. odst.
Wszystkie techniki	155.21	73.37	85.97	345.02	74.20	364.54
Bez <i>frustum culling</i>	12.10	0.55	10.85	13.15	9.45	13.33
Bez <i>hidden face culling</i>	42.49	3.48	33.32	50.28	20.48	135.00
Z <i>TRIANGLE_LIST</i>	162.97	66.68	78.87	310.87	72.80	391.63

Tabela 5.22: Wpływ liczby wątków na czas generowania świata.

Liczba wątków roboczych	Czas generowania [s]
1	109.4704596
5	27.6315853
11	18.460942
12	20.3584567

Analiza wyników pozwala na sformułowanie kilku istotnych obserwacji:

- **Obcinanie bryła widzenia** jest absolutnie kluczową techniką. Jego brak prowadzi do spadku średniego FPS z ponad 150 do zaledwie 12, co praktycznie uniemożliwia płynną interakcję ze światem.
- **Eliminacja niewidocznych ścian** również odgrywa dużą rolę – wyłączenie tej metody obniża średni FPS prawie czterokrotnie, a wartości minimalne spadają poniżej 35 FPS.
- **Zmiana trybu interpretowania topologii** z *TRIANGLE_LIST* (pracującego wraz z buforem indeksów) na *TRIANGLE_STRIP* (rezygnując z bufora indeksów) nieznacznie pogarsza wydajność renderowania. Może to oznaczać, że karty graficzne są tworzone z myślą o optymalizowaniu buforów indeksów – konieczność poszukiwania danego wierzchołka w pamięci karty graficznej okazuje się być sprawniejsza, niż obliczenie domykania trójkątów.
- W przypadku **równoległego generowania świata**, wzrost liczby wątków znacząco skraca czas potrzebny na inicjalizację. Przykładowo, przejście od jednego wątku do jedenastu pozwala zmniejszyć czas z 109s do 18s (ponad 5-krotne przyspieszenie).
 - Zwiększenie liczby wątków ponad liczbę procesorów logicznych udostępnianych przez procesor prowadzi do spadku wydajności – prawdopodobnym scenariuszem jest bardziej chaotyczne przełączanie kontekstów przez system operacyjny i brak przydziału wszystkich wątków do struktur procesora.
- Łączne zastosowanie wszystkich technik daje efekt synergii – system utrzymuje średnio ponad 150 FPS, a więc zapewnia płynną i stabilną rozgrywkę nawet w złożonych światach.

Uzyskane wyniki jednoznacznie pokazują, że techniki optymalizacyjne są niezbędne dla utrzymania wysokiej wydajności zarówno w procesie generowania, jak i renderowania światów. Największe znaczenie mają techniki związane z redukcją liczby obiektów niewidocznych dla obserwatora oraz wielowątkowość, które w praktyce decydują o możliwości płynnej pracy systemu wokselowego.

6. PODSUMOWANIE

Rozdział ten stanowi syntetyczne podsumowanie przeprowadzonych badań oraz uzyskanych wyników w ramach pracy pt. „*Analiza i rozwój metod reprezentacji licznych obiektów w symulatorach i grach wideo*”. W kolejnych sekcjach zestawiono osiągnięcia względem przyjętych celów, omówiono najważniejsze wyniki badań, wskazano ograniczenia przeprowadzonych eksperymentów oraz zaproponowano możliwe kierunki dalszego rozwoju.

6.1. Cel pracy i jego realizacja

Celem niniejszej pracy było zbadanie, porównanie oraz rozwój metod przechowywania i prezentowania dużych zbiorów obiektów w środowiskach trójwymiarowych, w szczególności w kontekście systemów wokselowych uznanych za te najbardziej złożone. Zadanie to obejmowało zarówno analizę teoretyczną istniejących metod, jak i przeprowadzenie badań eksperymentalnych pozwalających na wybór najefektywniejszych rozwiązań.

Na etapie wprowadzającym w rozdziałach 2 i 3 przedstawiono podstawy teoretyczne i technologiczne związane z działaniem grafiki komputerowej, sposobami organizacji przestrzeni i danych w strukturach danych, przytoczono rynek współczesnych języków programowania oraz bibliotek graficznych, a także omówiono metody generowania proceduralnej zawartości.

W rozdziale 4 opisano przygotowane środowisko badawcze obejmując wybrany język programowania z zestawem bibliotek, strukturę projektu i przygotowane światy testowe.

Zasadniczą część pracy stanowiły badania opisane w rozdziale 5. W pierwszej kolejności porównano różne struktury danych pod kątem operacji podstawowych (inicjalizacja, wypełnianie, usuwanie, pobieranie, zajętość pamięciowa). Analiza ta pozwoliła wyłonić strukturę *octree* jako najbardziej odpowiednią dla przyjętych założeń. Następnie zbadano zachowanie tej struktury w praktycznych scenariuszach, opartych o jedenaście światów testowych o zróżnicowanej charakterystyce. Ostatni etap badań obejmował ocenę wpływu wybranych technik optymalizacyjnych (m.in. bryły widzenia/obcinania i eliminacji niewidocznych ścian), na wydajność renderowania czy ogólnie prototypu.

Wszystkie etapy pracy zostały zrealizowane zgodnie z przyjętymi celami. Dokonano zarówno przeglądu teoretycznego, jak i weryfikacji empirycznej, co pozwoliło na kompleksową ocenę badanych metod. Zestawienie wyników z referencyjnymi pomiarami uzyskanymi w grze *Minecraft* osadzają badania w kontekście praktycznym – badania te potwierdzają, że opracowane rozwiązania pozwalają osiągnąć wydajność porównywalną, a nawet wyższą, niż w programie komercyjnym, spełniając tym samym główny cel pracy.

6.2. Najważniejsze wyniki badań

W wyniku przeprowadzonych eksperymentów uzyskano następujące najważniejsze rezultaty:

- **Struktury danych.** Analiza wykazała, że spośród testowanych podejść szczególnie korzystnym rozwiązaniem dla reprezentacji dużych światów wokselowych okazało się drzewo ósemkowe "octree". Łączy ono stosunkowo niskie zużycie pamięci z wysoką efektywnością operacji dostępu do danych. W porównaniu do struktur liniowych czy prostych siatek, *octree* lepiej radziło sobie zwłaszcza w kontekście fragmentaryczne wypełnionych światów.
- **Światy testowe.** Badania przeprowadzone na zróżnicowanych mapach (m.in. losowych, warstwowych, realistycznych) pozwoliły ocenić zachowanie systemu w różnych scenariuszach. Wyniki wskazały, że efektywność wybranej struktury i technik renderowania jest silnie zależna od rozkładu obiektów oraz gęstości wypełnienia przestrzeni.
- **Techniki optymalizacyjne.** Stosowanie metod optymalizacyjnych potrafi poprawić wydajność symulacji o kilkadziesiąt czy nawet o kilkaset klatek na sekundę. Przykładowymi zastosowanymi technikami są:
 - wielowątkowość w procesie generowania świata;
 - obcinanie bryłą widzenia ograniczające liczbę renderowanych obiektów;
 - eliminacja niewidocznych ścian redukujące liczbę generowanych ścian w siatce;
 - instancjonowanie wraz z redukcją liczby wywołań rysowania porpawiające wydajność.
- **Wielowątkowość.** Testy pokazały wyraźny spadek czasu generowania świata przy rosnącej liczbie wątków. Największy efekt uzyskano stosując maksymalną liczbę wątków roboczych odpowiadającej liczbie procesorów logicznych dla danej maszyny, a dalsze zwiększanie ich liczby wpływało negatywnie na wyniki.
- **Porównanie z grą Minecraft.** Świat przedstawiony w grze jest zdecydowanie bardziej różnorodny, jednak należy zwrócić uwagę na znaczaco niższą liczbę obiektów, jakie gra jest w stanie przetwarzać w danej chwili. Badania referencyjne wskazały, że przygotowany prototyp osiąga wydajność na poziomie porównywalnym do komercyjnej implementacji.

Zebrane wyniki potwierdzają, że zastosowanie odpowiednio dobranych struktur danych w połączeniu z zestawem technik optymalizacyjnych pozwala na skutecną reprezentację oraz wydajne renderowanie licznych obiektów w środowisku wokselowym.

6.3. Ograniczenia badań

Pomimo uzyskanych akceptowalnych rezultatów oraz osiągnięcia zakładanego celu, przygotowany prototyp nie implementuje jeszcze różnych innych złożonych technik optymalizacyjnych a przeprowadzone badania posiadają pewne ograniczenia, które należy uwzględnić przy interpretacji wyników:

- **Zakres światów testowych.** Analiza została przeprowadzona na ograniczonej liczbie procedur generowania światów. Chociaż pozwalają one ocenić wydajność w różnych scenariuszach, nie odzwierciedlają pełnej różnorodności środowisk spotykanych w grach video.
- **Skala eksperymentów.** Ze względu na ograniczenia sprzętowe oraz czasowe badania przeprowadzono dla światów o relatywnie małej skali (rozdzielncość woksela, różnorodność i wypełnienie symulowanej przestrzeni). W środowiskach rzeczywistych, o znacznie bogatszych wypełnieniach, mogą wystąpić dodatkowe problemy związane z wydajnością pamięciową czy synchronizacją obliczeń.
- **Platforma sprzętowa.** Eksperymenty wykonano na dwóch maszynach testowych, reprezentujących komputer klasy wyższej i niższej – komputer do gier oraz laptop ze zintegrowaną kartą graficzną. Wyniki mogą różnić się w przypadku innych konfiguracji sprzętowych, szczególnie w odniesieniu do kart graficznych o odmiennej architekturze czy procesorów o innej liczbie rdzeni.
- **Podstawowa optymalizacja pamięciowa.** W pracy skupiono się przede wszystkim na wydajności renderowania i technikach optymalizacyjnych. Szczegółowe badania nad zużyciem pamięci (np. techniki kompresji danych wokselowych, stronicowanie dyskowe) nie zostały rozwinięte i mogłyby wpływać na bardziej kompleksową ocenę systemu.
- **Ograniczenia implementacyjne.** Prototyp stworzony na potrzeby pracy nie posiada pełnego zestawu funkcji typowych dla symulacji naukowych czy gier komercyjnych (np. symulacji fizyki, złożonych animacji czy obsługi sieciowej). Z tego względu uzyskane wyniki dotyczą wyłącznie aspektów związanych z reprezentacją i renderowaniem obiektów.
- **Porównanie z grą Minecraft.** Analiza porównawcza miała charakter referencyjny i opierała się na pomiarach FPS w wybranym scenariuszu bez użycia dokładnego oprogramowania pomiarowego. Z uwagi na zamknięty kod źródłowy gry, nie były możliwe zarówno przeprowadzenie dokładnej analizy zastosowanych tam technik jak i pełna kontrola nad środowiskiem testowym.

Przytoczone ograniczenia nie podważają głównych wniosków pracy, lecz wyznaczają obszary, w których dalsze badania mogłyby pogłębić uzyskane rezultaty, znacząco podnieść wydajność badanych obszarów oraz poszerzyć ich zakres zastosowania.

6.4. Możliwe kierunki dalszych prac

Wypracowany prototyp oraz uzyskane wyniki badań stanowią solidną podstawę do dalszego rozwijania systemu reprezentacji i renderowania licznych obiektów. Do szeregu obszarów, które można rozbudować należą:

- **Zaawansowane techniki kompresji danych.** Zastosowana struktura drzewa ósemkowego *octree* pozwala na przechowywanie zamiast wskaźników o dużej objętości pamięciowej (dla architektury 64 bitowej wskaźniki mają rozmiar 64 bitów), lokalnych tablic mapujących lekkie identyfikatory przechowywanych obiektów (np. 8b) do globalnej tablicy obiektów (64b), co znaczająco zredukowałoby konsumpcję pamięci przez program.
- **Optymalizacje pamięciowo-obliczeniowe.** Oprócz badanych technik renderowania warto wprowadzić dodatkowe mechanizmy, takie jak poziom szczegółowości LOD dla wokseli, cieniowanie bazujące na tzw. *mesh simplification* czy bardziej zaawansowane podejścia do instancjonowania obiektów.
- **Skalowanie na większe światy.** W pracy badano światy o ograniczonej wielkości, natomiast dalsze prace mogłyby uwzględniać światy o jeszcze większych dystansach renderowania do czego konieczna byłaby implementacji wspomnianych powyżej technik. Większe światy, czyli dużo większa liczba aktywnych obiektów, zbliżyłaby przygotowywany prototyp do zaawansowanych w tym zakresie projektów takich jak *Teardown* [112]
- **Integracja z fizyką i interakcją.** Obecny prototyp skupia się na reprezentacji i renderowaniu, pomijając aspekty fizyczne, o które można go poszerzyć w przyszłości. Kolizje, dynamiczne modyfikowanie świata przez użytkownika, czy implementacja obiektów ruchomych (ang. *mobile objects*) stanowią przykładowe rozszerzenia.
- **Porównania z większą liczbą systemów referencyjnych.** W pracy punktem odniesienia była gra *Minecraft*, jednak wartościowe byłoby zestawienie wyników również z innymi silnikami i narzędziami (np. *Unreal Engine*, *Unity*) w kontekście dużych scen.
- **Rozszerzenie testów sprzętowych.** Przeprowadzenie eksperymentów na szerszej gamie konfiguracji – różne generacje kart graficznych, architektury procesorów, bardziej uśrednione platformy zamiast jedynie klas wysokiej i niskiej – pozwoliłyby lepiej ocenić skalowalność i dostępność opracowanego rozwiązania.
- **Eksperymenty z renderowaniem hybrydowym.** Połączenie tradycyjnego rasteryzera z technikami śledzenia promieni (ang. *ray tracing*) mogłyby otworzyć nowe możliwości wizualne oraz umożliwić efektywne odwzorowanie efektów świetlnych.
- **Rozbudowa renderera.** W zaawansowanych projektach stosuje się wiele różnych kombinacji programów cieniowania oraz potoków renderowania. Przygotowany prototyp obsługuje jedynie dwa proste scenariusze co stanowi pole do rozbudowy (na przykład rysowanie cieni, chmur, cieczy, interfejsu użytkownika w postaci HUD (ang. *head-up display*)).

Realizacja powyższych kierunków umożliwiłyby rozszerzenie zakresu zastosowań opracowanego rozwiązania – od prototypowych prostych gier i symulacji terenu, po zaawansowane symulatory naukowe płynów i systemy wizualizacji w grach wideo, inżynierii i medycynie.

WYKAZ LITERATURY

- [1] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering; Fourth Edition*. Taylor and Franci, 2018. [Online]. Available: http://mutantstargoat.com/~nuclear/tmp/realtimerendering_4ed.pdf.
- [2] I. V. D. Srihith, A. D. Donald, T. Aditya, and T. A. S. Srinivas, "The backbone of computing: An exploration of data structures," *International Journal of Computer Science Research*, vol. 1, 1 2023. [Online]. Available: https://www.researchgate.net/publication/369960268_The_Backbone_of_Computing_An_Exploration_of_Data_Structures.
- [3] S. Dutta, A. Chowdhury, I. Debnath, R. Sarkar, H. Dey, and A. Dutta, "The role of data structures in different fields of computer- science: A review," *Journal of Mathematical Sciences and Computational Mathematics*, 2020. [Online]. Available: https://jmscm.smartsociety.org/volume1_issue3/Paper10.pdf.
- [4] E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C*. Computer Science Press, 1992. [Online]. Available: <https://mrce.in/ebooks/Fundamentals-of-Data-Structures-in-C-Ellis-Horowitz-Sartaj-Sahni-etc.-.pdf>.
- [5] J. Gregory, *Game Engine Architecture*. Taylor and Franci, 2009. [Online]. Available: https://www.latexstudio.net/wp-content/uploads/2014/12/Game_Engine_Architecture-en.pdf.
- [6] M. '. Persson and J. '. Bergensten, *Minecraft*. [Online]. Available: <https://www.minecraft.net/pl-pl>.
- [7] W. Software, *Factorio*. [Online]. Available: <https://factorio.com>.
- [8] N. Games, *Noita*. [Online]. Available: <https://noitagame.com>.
- [9] K. Shanton and A. Goldman, "Simulation theory," *WIREs Cognitive Science*, 2010. [Online]. Available: <https://doi.org/10.1002/wcs.33>.
- [10] P. Humphreys, "Computer simulations," *Cambridge University Press*, 2023. [Online]. Available: <https://www.cambridge.org/core/services/aop-cambridge-core/content/view/A38B480072F7B19CEC28222D03FD38E/S0270864700009061a.pdf/computer-simulations.pdf>.
- [11] N. Esposito, "A short and simple definition of what a videogame is," *DiGRA*, 2005. [Online]. Available: <https://dl.digra.org/index.php/dl/article/view/177/177>.
- [12] I. B. Sinisa Mihajlovic, *(pdf) application and difference of raster and vector graphics*, Oct. 2019. [Online]. Available: https://www.researchgate.net/publication/366578092_Application_and_difference_of_raster_and_vector_graphics.
- [13] N. L. Webster, "High poly to low poly workflows for real-time rendering," *Journal of Visual Communication in Medicine*, 2017.
- [14] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Levy, *Polygon Mesh Processing*. A K Peters/CRC Press, 2010. [Online]. Available: http://staff.ustc.edu.cn/~lgliu/Courses/DGP_2014_autumn-winter/References/Book_Polygon%20Mesh%20Processing.pdf.
- [15] H. Hoppe, T. DeRose, T. Duchampy, J. McDonaldz, and W. Stuetzle, "Mesh optimization," *ACM*, 1993. [Online]. Available: <https://dl.acm.org/doi/10.1145/166117.166119>.
- [16] Y. Zhong, Z. Liu, and C. Zhou, "Free roaming of 3d stratum models based on internal and external boundary identification," *PLOS One*, 2024. [Online]. Available: <https://doi.org/10.1371/journal.pone.0300805>.

- [17] M. Aleksandrov, S. Zlatanova, and D. J. Heslop, “Voxelisation algorithms and data structures: A review,” *Sensors*, 2021. [Online]. Available: <https://doi.org/10.3390/s21248241>.
- [18] J. Zabrodzki, *Vi. modelowanie obiektów i scen 3d - grafika komputerowa (grk)*, [Online; accessed 2025-09-26]. [Online]. Available: <https://gakko.pjwstk.edu.pl/materiały/1927/lec/wykład-6.html>.
- [19] K. V. W. Group. “Pipelines.” (), [Online]. Available: <https://docs.vulkan.org/spec/latest/chapters/pipelines.html>. (data dostępu: 01.11.2024).
- [20] I. Dunn and Z. J. Wood. “Chapter 2: The graphics pipeline.” (), [Online]. Available: <https://graphicscompendium.com/intro/01-graphics-pipeline>. (data dostępu: 01.11.2024).
- [21] C. Yuksel. “Interactive graphics 18 - tessellation shaders.” (), [Online]. Available: <https://www.youtube.com/watch?v=0qRMNrvu6TE>. (data dostępu: 01.11.2024).
- [22] V. Karilainen, *Conversion between square grid and hexagon grid playing fields as a game mechanic*, 2024. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/821569/Karilainen_Veeti.pdf?sequence=2.
- [23] Y.-C. Chen and A. Jhala, *Gametilenet: A semantic dataset for low-resolution game art in procedural content generation*, 2025. [Online]. Available: <https://arxiv.org/html/2507.02941v1>.
- [24] S. Raman, “Efficient terrain triangulation and modification for game applications,” 2008.
- [25] Z. Ren, Z. Liu, and J. Tang, “3-d parallel anisotropic inversion of controlled-source electromagnetic data using nested tetrahedral grids,” *Geophysical Journal International*, 2024.
- [26] A. Wit, S. Wronski, and J. Tarasiuk, “Simulation and optimization of porous bone-like microstructures with specific mechanical properties,” *Acta Polytechnica CTU Proceedings*, 2019.
- [27] S. Laine and T. Karras, “Efficient sparse voxel octrees,” *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Association for Computing Machinery, 2010, pp. 55–63. [Online]. Available: https://research.nvidia.com/sites/default/files/pubs/2010-02_Efficient-Sparse-Voxel/laine2010i3d_paper.pdf?utm_source=chatgpt.com.
- [28] M. Nießner, M. Zollhofer, S. Izadi, and M. Stamminger, “Real-time 3d reconstruction at scale using voxel hashing,” *ACM Transactions on Graphics*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2508363.2508374>.
- [29] Y. Fang, Q. Wang, and W. Wang, “Aokana: A gpu-driven voxel rendering framework for open world games,” *Proc. ACM Comput. Graph. Interact. Tech.*, 2025. [Online]. Available: <https://doi.org/10.1145/3728299>.
- [30] S. Daubner, A. Cohen, B. Dörich, and S. Cooper, *Evoxels: A differentiable physics framework for voxel-based microstructure simulations*, 2025.
- [31] A. Knoll, “A survey of octree volume rendering methods,” *Visualization of Large and Unstructured Data Sets*, 2006. [Online]. Available: <https://www.sci.utah.edu/~knolla/octsurvey.pdf>.
- [32] R. Arbore, J. Liu, A. Wefel, S. Gao, and E. Shaffer, “Hybrid voxel formats for efficient ray tracing,” *Advances in Visual Computing: 19th International Symposium, ISVC 2024, Lake Tahoe, NV, USA, October 21–23, 2024, Proceedings, Part I*, Lake Tahoe, NV, USA: Springer-Verlag, 2024, pp. 125–138. [Online]. Available: https://doi.org/10.1007/978-3-031-77392-1_10.
- [33] M. Kanzler, M. Rautenhaus, and R. Westermann, “A voxel-based rendering pipeline for large 3d line sets,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, 2018.
- [34] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, and R. Dillmann, “Unified gpu voxel collision detection for mobile manipulation planning,” *IEEE*, 2014. [Online]. Available: <https://doi.org/10.1109/ROS.2014.6943148>.

- [35] J. R. Cebral, F. E. Camelli, and R. Löhner, “A feature-preserving volumetric technique to merge surface triangulations,” *Wiley Online Library*, 2002. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.499>.
- [36] M. Zeng, F. Zhao, J. Zheng, and X. Liu, “A memory-efficient kinectfusion using octree,” *Springer*, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-34263-9_30.
- [37] H. Samet, “The quadtree and related hierarchical data structures,” *ACM Computing Surveys*, 1984. [Online]. Available: <https://doi.org/10.1145/356924.356930>.
- [38] D. Ayala, P. Brunet, R. Juan, and I. Navazo, “Object representation by means of nonminimal division quadtrees and octrees,” *ACM Transactions on Graphics*, 1985. [Online]. Available: <https://doi.org/10.1145/3973.3975>.
- [39] G. Zachmann and E. Langetepe, “Geometric data structures for computer graphics,” 2012. [Online]. Available: <https://cg.cs.uni-bonn.de/backend/v1/files/publications/eg-tutorial-2002-2.pdf>.
- [40] I. Carlbom, I. Chakravarty, and D. Vanderschel, “A hierarchical data structure for representing the spatial decomposition of 3d objects,” *Springer*, 1985. [Online]. Available: https://doi.org/10.1007/978-4-431-68025-3_1.
- [41] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, “Octomap: An efficient probabilistic 3d mapping framework based on octrees,” *Springer*, 2012. [Online]. Available: <https://doi.org/10.1007/s10514-012-9321-0>.
- [42] S. Raschdorf and M. Kolonko, “A comparison of data structures for the simulation of polydisperse particle packings,” *Wiley Online Library*, 2002. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2988>.
- [43] K. Forsythe and L. Demers, “What is a scripting language,” 2022. [Online]. Available: <https://careerkarma.com/blog/what-is-a-scripting-language/>.
- [44] K. Forsythe and L. Demers, “What is a scripting language and what are the most common ones?,” 2022. [Online]. Available: <https://rockcontent.com/blog/scripting-languages/>.
- [45] S. Overflow, “The top programming languages,” 2023. [Online]. Available: <https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages>.
- [46] S. Overflow, “The top programming languages,” 2024. [Online]. Available: <https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages>.
- [47] S. Overflow, “Top 50 programming languages globally,” 2024. [Online]. Available: <https://innovationgraph.github.com/global-metrics/programming-languages#programming-languages-rankings>.
- [48] “Benchmarks for programming languages and compilers, which programming language or compiler is faster.” (), [Online]. Available: <https://programming-language-benchmarks.vercel.app>.
- [49] “Compile to native machine code - java performance tuning [book].” (), [Online]. Available: <https://www.oreilly.com/library/view/java-performance-tuning/0596000154/ch03s06.html>.
- [50] D. Lion, A. Chiu, M. Stumm, and D. Yuan, “Investigating managed language runtime performance: Why javascript and python are 8x and 29x slower than c++, yet and go can be faster?” *USENIX*, 2022. [Online]. Available: <https://www.usenix.org/system/files/atc22-lion.pdf>.
- [51] J. Ogala and D. V. Ojie, “Comparative analysis of c, c++, c# and java programming languages,” *Research Gate*, 2020. [Online]. Available: https://www.researchgate.net/profile/Justin-Ogala-2/publication/358368843_COMPARATIVE_ANALYSIS_OF_C_C_C_AND_JAVA_PROGRAMMING_LANGUAGES/links/61fe4aeb702c892cef063f13/COMPARATIVE-ANALYSIS-OF-C-C-C-AND-JAVA-PROGRAMMING-LANGUAGES.pdf.

- [52] M. Jankowski and J.-P. Kuska, "Connected components labeling - algorithms in mathematica, java, c++ and c#," *Research Gate*, 2020. [Online]. Available: <https://citeserx.ist.psu.edu/document?repid=rep1&type=pdf&doi=20ccd23694b7074d843a7491a66731518fd67793>.
- [53] "Measured : Which programming language is fastest? (benchmarks game)." (), [Online]. Available: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html>.
- [54] "Kotlin compiler options | kotlin documentation." (2012), [Online]. Available: <https://kotlinlang.org/docs/compiler-reference.html>.
- [55] A. Kenneth, "You can't spell trust without rust," *Carleton University Institutional Repository*, 2016. [Online]. Available: <https://doi.org/10.22215/etd/2016-11260>.
- [56] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *ACM*, 2021. [Online]. Available: <https://doi.org/10.1145/3485498>.
- [57] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed," *ACM*, 2021. [Online]. Available: <https://doi.org/10.1145/3485832.3485903>.
- [58] "Meet safe and unsafe." (2023), [Online]. Available: <https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html>.
- [59] "Overview zig programming language." (2019), [Online]. Available: <https://ziglang.org/learn/overview/>.
- [60] "Documentation - the zig programming language." (2024), [Online]. Available: <https://ziglang.org/documentation/master/>.
- [61] D. M. Ritchie, "The development of the c language," *ACM*, 1993. [Online]. Available: <https://doi.org/10.1145/155360.155580>.
- [62] C. Garling. "Iphone coding language now world's third most popular." (2012), [Online]. Available: <https://web.archive.org/web/20130909060247/http://www.wired.com/wiredenterprise/2012/07/apple-objective-c/>.
- [63] "Document revision history." (2012), [Online]. Available: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/RevisionHistory.html#/apple_ref/doc/uid/TP40011210-CH99-SW1.
- [64] B. Stroustrup. "Bjarne stroustrup's faq – when was c++ invented?" (1993), [Online]. Available: https://www.stroustrup.com/bs_faq.html#invention.
- [65] "Javasoft ships java 1.0." (2007), [Online]. Available: <https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml>.
- [66] "Graalvm." (), [Online]. Available: <https://www.graalvm.org>.
- [67] E. Dietrich, P. Smacchia, and Microsoft. "The history of c#." (2024), [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history#c-version-10-1>.
- [68] L. Bak. "Dart: A language for structured web programming." (2011), [Online]. Available: <https://googlecode.blogspot.com/2011/10/dart-language-for-structured-web.html>.
- [69] "Kotlin m1 is out!" (2012), [Online]. Available: <https://blog.jetbrains.com/kotlin/2012/04/kotlin-m1-is-out>.
- [70] "Release history - the go programming language." (2011), [Online]. Available: <https://go.dev/doc/devel/release>.
- [71] "Wwdc14 | documentation." (2014), [Online]. Available: <https://wwdcnotes.com/documentation/wwdcnotes/wwdc14>.

- [72] “1.1.0 | rust changelogs.” (2015), [Online]. Available: <https://releases.rs/docs/1.1.0>.
- [73] “Download zig programming language.” (2017), [Online]. Available: <https://ziglang.org/download>.
- [74] K. S. Banach and M. Skublewska-Paszkowska, “Comparison of objective-c and swift on the example of a mobile game,” *Journal of Computer Sciences Institute*, 2020. [Online]. Available: <https://ph.pollub.pl/index.php/jcsi/article/view/2058/1985>.
- [75] H. Singh, “Speed performance between swift and objective-c,” *International Journal of Engineering Applied Sciences and Technology (IJEAST)*, 2016. [Online]. Available: https://www.ijeast.com/papers/185-189_Tesma110_IJEAST.pdf.
- [76] S. Tylec and K. Woś, “Swift performance statistical applications,” *Journal of Computer Sciences Institute*, 2022. [Online]. Available: <https://ph.pollub.pl/index.php/jcsi/article/view/3067/2979>.
- [77] “C++ g++ vs c gcc - which programs are fastest? (benchmarks game).” (), [Online]. Available: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/cpp.html>.
- [78] “Swift vs c clang - which programs are fastest? (benchmarks game).” (), [Online]. Available: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/swift.html>.
- [79] Z. Liang, V. Jabrayilov, A. Charapko, and A. Aghayev, “The cost of garbage collection for state machine replication,” *International Journal of Engineering Applied Sciences and Technology (IJEAST)*, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2405.11182>.
- [80] “Dlaczego discord przechodzi z języka go na rust?” (2020), [Online]. Available: <https://discord.com/blog/why-discord-is-switching-from-go-to-rust>.
- [81] “>*rust and go are both touted as memory safe. while this statement is absolutely... | hacker news.” (), [Online]. Available: <https://news.ycombinator.com/item?id=9818388>.
- [82] “How to write safe and secure go code.” (), [Online]. Available: <https://www.codiga.io/blog/secure-go-code/>.
- [83] A. Mikkonen, *Graphics programming then and now*, 2021. [Online]. Available: https://www.theses.fi/bitstream/handle/10024/501343/Aatu_Mikkonen.pdf?sequence=2.
- [84] M. J. Kilgard, *Realizing opengl: Tbo implementations of one architecture*, 1997. [Online]. Available: https://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/kilgard97_realizinggl.pdf.
- [85] J. Unterguggenberger, B. Kerbl, and M. Wimmer, “Vulkan all the way: Transitioning to a modern low-level graphics api in academia,” *Elsevier*, 2023. [Online]. Available: <https://doi.org/10.1016/j.cag.2023.02.001>.
- [86] A. Lagae et al., *A survey of procedural noise functions*, 2010. [Online]. Available: <https://www.cs.umd.edu/~zwicker/publications/SurveyProceduralNoise-CGF10.pdf>.
- [87] H. Serrano, *Noise in computer graphics- a brief introduction*, 2015. [Online]. Available: <https://www.haroldsserrano.com/blog/noise-in-computer-graphics-a-brief-introduction>.
- [88] S. G. N. Corporation, *Chapter 26. implementing improved perlin noise | nvidia developer*. [Online]. Available: <https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-26-implementing-improved-perlin-noise>.
- [89] M. Colom, M. Lebrun, A. Buades†, and J.-M. Morel, “Nonparametric multiscale blind estimation of intensity-frequency dependent noise,” *ResearchGate*, 1015. [Online]. Available: https://www.researchgate.net/publication/278333834_Nonparametric_Multiscale_Blind_Estimation_of_Intensity-Frequency-Dependent_Noise.

- [90] K. Perlin, "An image synthesizer," ACM, 1985. [Online]. Available: <https://dl.acm.org/doi/10.1145/325165.325247>.
- [91] S. Gustavson, "Simplex noise demystified," *ResearchGate*, 2005. [Online]. Available: https://cgvr.cs.uni-bremen.de/teaching/cg_literatur/simplexnoise.pdf.
- [92] K. Perlin, "Improving noise | acm transactions on graphics," ACM, 2002. [Online]. Available: <https://dl.acm.org/doi/10.1145/566654.566636>.
- [93] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutre, "Procedural noise using sparse gabor convolution," ACM, 2009. [Online]. Available: <https://graphics.cs.kuleuven.be/publications/LLDD09PNSGC/>.
- [94] J. P. Lewis, "Generalized stochastic subdivision," ACM, 1987. [Online]. Available: <https://dl.acm.org/pdf/10.1145/35068.35069>.
- [95] S. Worley, "A cellular texture basis function," ACM, 1996. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/237170.237267>.
- [96] A. Wojciechowski and P. Napieralski, "Computer game innovations," *ResearchGate*, 2018. [Online]. Available: https://www.researchgate.net/publication/335524620_Computer_Game_Innovations_2018.
- [97] D. Mustafa, R. Alkhasawneh, F. Obeidat, and A. S. Shatnawi, "Mimd programs execution support on simd machines: A holistic survey," *IEEE Access*, vol. 12, pp. 34 354–34 377, 2024.
- [98] Y. Jlali, *Directx 12: Performance comparison between single- and multithreaded rendering when culling multiple lights*, 2020. [Online]. Available: <https://bth.diva-portal.org/smash/record.jsf?pid=diva2%3A1453696&dswid=-2203>.
- [99] U. Assarsson and T. Moller, "Optimized view frustum culling algorithms for bounding boxes," *Journal of Graphics Tools: JGT*, 2000.
- [100] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand, "A survey of visibility for walkthrough applications," *IEEE Transactions on Visualization and Computer Graphics*, 2003.
- [101] M. Robinson. "I optimised my game engine up to 12000 fps." (), [Online]. Available: https://youtu.be/40JzyaOYJeY?si=3W-X_GX0LogdQoby&t=72.
- [102] M. Johansson, "Integrating occlusion culling and hardware instancing for efficient real-time rendering of building information models," 2013. [Online]. Available: https://publications.lib.chalmers.se/records/fulltext/173349/local_173349.pdf.
- [103] E. Arnebäck. "Making faster fragment shaders by using tessellation shaders." (), [Online]. Available: https://erkaman.github.io/posts/tess_opt.html.
- [104] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007. [Online]. Available: <https://developer.nvidia.com/gpugems/gpugems3/contributors>.
- [105] S. Jabłoński and T. Martyn, "Real-time rendering of continuous levels of detail for sparse voxel octrees," 2016. [Online]. Available: <https://api.semanticscholar.org/CorpusID:45314387>.
- [106] "Greedy meshing algorithm." (), [Online]. Available: <https://deepwiki.com/vercidium-patreon/meshing/4.1-greedy-meshing-algorithm>.
- [107] E. Johansson. "Binary greedy meshing v2." (), [Online]. Available: <https://github.com/cgerikj/binary-greedy-meshing>.
- [108] "Vulkan tutorial (rust)." (), [Online]. Available: <https://kylemayes.github.io/vulkanalia/>.
- [109] bheisler. "Criterion.rs – rust benchmarking library." (2025), [Online]. Available: <https://github.com/bheisler/criterion.rs> (, Dostęp 2025-09-04).
- [110] bheisler. "Criterion.rs book." (2025), [Online]. Available: https://bheisler.github.io/criterion.rs/book/criterion_rs.html (, Dostęp 2025-09-04).

- [111] T. R. P. Contributors. “Dhat-rs – heap profiling for rust.” (2025), [Online]. Available: <https://docs.rs/dhat/latest/dhat/> (, Dostęp 2025-09-04).
- [112] T. Labs, *Teardown*. [Online]. Available: <https://teardowngame.com>.

WYKAZ RYSUNKÓW

2.1 Panorama świata gry <i>Minecraft</i> z widocznymi budowlami graczy oraz rozległym horyzontem. [Opracowanie własne na podstawie gry <i>Minecraft</i> [6]]	10
2.2 Gra <i>Factorio</i> , z widocznymi setkami bytów/przedmiotów poruszającymi się i przetwarzanymi w czasie rzeczywistym. [Opracowanie własne na podstawie gry <i>Factorio</i> [7]]	10
2.3 Zrzut ekranu z gry <i>Noita</i> z widocznymi symulowanymi pikselami cieczy (pomarańczowa lawa, łososiowa mikstura), gazów (łososiowe piksele nad miksturą), portalu (czerwone koło), szczątków (zalewany pod portalem lawą kolor oliwkowy), głównej postaci (postać w centrum ilustracji), korodującego otoczenia (okołobrązowe platformy), tła. [Opracowanie własne na podstawie gry <i>Noita</i> [8]]	11
2.4 Porównanie obrazu rastrowego (po lewej) oraz wektorowego (po prawej). [Wikimedia Commons ¹]	13
2.5 Przykłady siatek o zagęszczeniu większym (rząd górny) i mniejszym (rząd dolny) oddające różnicę między grafikami <i>high</i> oraz <i>low poly</i> [15]	14
2.6 Ujęcie z gry <i>Poly Bridge</i> jako przykład grafiki <i>low poly</i> . ²	14
2.7 Ujęcie z wprowadzenia do gry <i>Mafia: Edycja ostateczna</i> jako przykład grafiki <i>high poly</i> ³ . Uwspółcześniona wersja gry wykładowiczo podniosła liczbę wielokątów w używanych modelach ⁴	14
2.8 Przykładowy potok renderowania. Wyróżnione kolorem o odcieniu pomarańczowym etapy są programowalne. Programy cieniowania wierzchołków (b) oraz fragmentów (c) są podstawowymi elementami potoku. Operacje od etapu (b) włącznie zachodzą w całości na karcie graficznej. [Opracowanie własne na podstawie materiałów grupy Khronos [19], <i>Graphics Programming Compendium</i> [20], wykładu <i>Interactive Graphics 18 - Tessellation Shaders</i> [21]]	16
2.9 Przedstawienie różnych typów siatek na przykładzie gier wideo – siatki kwadratowej (a), trójkątnej (b) i sześcienniej (c).	18
2.10 Przedstawienie segmentu świata gry <i>Minecraft</i> (a) wraz z siatką obrazującą widok z góry na segment (b) i liczbę segmentów dla zasięgu renderowania 2 segmentów (c).	19
2.11 Przykładowe bloki obecne w grze <i>Minecraft</i> wraz z fragmentem ich informacji debugowania. [Opracowanie własne na podstawie gry <i>Minecraft</i> [6]]	20
4.1 Diagram klas przedstawiający agregację struktur składowych w obrębie struktury głównej <i>Voxel</i>	31
4.2 Diagram klas prototypu przedstawiający główne moduły wraz z najważniejszymi strukturami i kluczowymi atrybutami. Trzy struktury wyróżnione kolorem białym stanowią podstawę działania aplikacji – App zarządza aplikacją, a World światem, Renderer zaś odpowiada za proces renderowania.	35
4.3 Struktura plików stworzonego prototypu (a) oraz główny plik konfiguracyjny <i>flags.rs</i> (b).	36

4.4 Zrzut ekranu z działającego prototypu przy zamrożonym przetwarzaniu i renderowaniu. Białe linie obrazują kontury bryły widzenia a miejsce, w którym wydaje się jakoby były połączone przedstawia mały prostokąt – ekran monitora odbiorcy. Wyrenderowane zostały wszystkie segmenty zawierające się lub przecinające bryłę widzenia.	37
4.5 Przedstawienie świata testowego o pustej strukturze (a), wraz z uproszczoną wizualizacją jego struktury (b).	38
4.6 Przedstawienie świata testowego z jednym woksem – minimalną ilością danych (a), wraz z uproszczoną wizualizacją jego struktury (b).	39
4.7 Przedstawienie jednorodnie wypełnionego do połowy świata (a), wraz z uproszczoną wizualizacją jego struktury (b).	39
4.8 Przedstawienie świata wypełnionego losowo wybraną połową komórek (a), wraz z uproszczoną wizualizacją jego struktury (b). Losowe wypełnienie jest najwidoczniejsze przy granicy generowanego terenu, gdzie widoczne są prześwit pustki, co przedstawiono na przybliżeniu.	40
4.9 Przedstawienie jednorodnie wypełnionego do połowy świata bez jednego woksela (a), wraz z uproszczoną wizualizacją jego struktury (b).	40
4.10 Przedstawienie jednorodnie wypełnionego (a), wraz z uproszczoną wizualizacją jego struktury (b).	41
4.11 Przedstawienie losowo wypełnionego w połowie świata różnokolorowymi wokselami (a), wraz z uproszczoną wizualizacją jego struktury (b). Losowe wypełnienie jest najwidoczniejsze przy granicy generowanego terenu, gdzie widoczne są prześwit pustki.	41
4.12 Przedstawienie świata w pełni wypełnionego różnokolorowymi wokselami (a), wraz z uproszczoną wizualizacją jego struktury (b).	42
4.13 Przedstawienie realistycznego świata z użyciem trójwymiarowego szumu Simplex o niskim procencie wypełnienia (a), wraz z uproszczoną wizualizacją jego struktury (b).	42
4.14 Przedstawienie świata proceduralnie generowanego budowanego na bazie drzewa czwórkowego (a), wraz z widocznym przekrojem pionowym (b).	43
4.15 Przedstawienie świata proceduralnie generowanego z górami, drzewami i chmurami (a), wraz z widocznym przekrojem pionowym (b).	44
4.16 Przykładowy kod testowy napisany w języku <i>Rust</i> z użyciem biblioteki <i>Criterion</i> . Kod ten odpowiada za wygenerowanie raportu dla grupy testów mierzących czas inicjalizacji różnych struktur danych.	45
5.1 Zrzut ekranu pozyskany podczas gry w grę Minecraft na platformie sprzętowej klasy niższej (laptop) z włączonymi informacjami debugowania. Grafika obejmuje widoczny wskaźnik liczby klatek na sekundę pokazujący 35 FPS, oraz poziom użycia karty graficznej pokazujący 100%.	49

5.2 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa PDF (ang. <i>Probability Density Function</i>) dla pomiarów średniego czasu inicjalizacji badanych struktur danych. Na diagramie widnieje znaczaco odstający PDF dla <i>mapy wokseli</i> , co utrudnia ocenę czasu dla pozostałych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	50
5.3 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu inicjalizacji badanych struktur danych. Z badanych struktur wyłączona została <i>mapa wokseli</i> a <i>octree</i> zostało powtórzone dla różnych rozmiarów przechowywanych danych. W porównaniu struktura <i>octree</i> wypada najgorzej, natomiast dla wszystkich ocenianych rozmiarów tej struktury (stosunkowo małych oraz skrajnie dużych) czas inicjalizacji pozostaje taki sam. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	51
5.4 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów średniego czasu inicjalizacji struktury mapy wokseli o różnych rozmiarach. Widoczny jest na diagramie trend potęgowy rosnącego czasu inicjalizacji względem zadanej (bardzo małego) rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	51
5.5 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu całkowitego wypełniania badanych struktur danych. Na diagramie charakterystyczne są wyniki dla <i>octree</i> , które są niewidoczne ze względu na swoją wielkość oraz dla <i>struktury haszującej</i> mającej znaczaco dłuższy czas wypełniania względem pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	52
5.6 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu częściowego wypełniania badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	53
5.7 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu wstawiania rzędu stu wokseli do badanych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	53
5.8 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów wstawiania wokseli w tysiąc deterministycznie losowych pozycji, identycznych dla każdej struktury. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	54
5.9 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu pojedynczego usuwania wokseli z badanych struktur. Wyniki <i>listy wokseli</i> zmarginalizowały wyniki pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	55

5.10 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu masowego usuwania obszaru z badanych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	55
5.11 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie losowego usuwania tysiąca wekseli z badanych struktur Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	56
5.12 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu wyszukiwania następujących po sobie stu wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	57
5.13 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie losowego wyszukiwania tysiąca wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	57
5.14 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu przeszukiwania tablic i wektorów o różnych konfiguracjach. Na diagramie uwidoczniono konfigurację powodującą największe obciążenie czasowe podczas przeszukiwania – wektor przechowujący liczniki atomowe (Arc). Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy niższej.	58
A.1 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa PDF (ang. <i>Probability Density Function</i>) dla pomiarów średniego czasu inicjalizacji badanych struktur danych. Na diagramie widnieje znaczaco odstający PDF dla mapy wokseli , co utrudnia ocenę czasu dla pozostałych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	86
A.2 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu inicjalizacji badanych struktur danych. Z badanych struktur wyłączona została mapa wokseli a octree zostało powtórzone dla różnych rozmiarów przechowywanych danych. W porównaniu struktura octree wypada najgorzej, natomiast dla wszystkich ocenianych rozmiarów tej struktury (stosunkowo małych oraz skrajnie dużych) czas inicjalizacji pozostaje taki sam. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	87
A.3 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów średniego czasu inicjalizacji struktury mapy wokseli o różnych rozmiarach. Widoczny jest na diagramie trend potęgowy rosnącego czasu inicjalizacji względem zadanego (bardzo małego) rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	87

A.4 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu całkowitego wypełniania badanych struktur danych. Na diagramie charakterystyczne są wyniki dla <i>octree</i> , które są niewidoczne ze względu na swoją wielkość oraz dla <i>struktury haszującej</i> mającej znacznie dłuższy czas wypełniania względem pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	88
A.5 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu częściowego wypełniania badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	88
A.6 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu wstawiania rzędu stu wokseli do badanych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	89
A.7 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów wstawiania wokseli w tysiąc deterministycznie losowych pozycji, identycznych dla każdej struktury. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	89
A.8 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu pojedynczego usuwania wokseli z badanych struktur. Wyniki <i>listy wokseli</i> zmarginalizowały wyniki pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	90
A.9 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu masowego usuwania obszaru z badanych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	90
A.10 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie losowego usuwania tysiąca weokseli z badanych struktur. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	91
A.11 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu wyszukiwania następujących po sobie stu wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	91
A.12 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie losowego wyszukiwania tysiąca wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia <i>Criterion</i> w środowisku <i>Rust</i> na maszynie klasy wyższej.	92

A.13 Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **przeszukiwania tablic i wektorów** o różnych konfiguracjach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia *Criterion* w środowisku *Rust* na maszynie klasy wyższej. 92

WYKAZ TABEL

3.1 Zestawienie języków programowania	24
3.2 Zestawienie bibliotek graficznych [83]	25
5.1 Opis istotnych dla pracy podzespołów platformy sprzętowej klasy niższej	48
5.2 Zaobserwowane empirycznie wartości klatek na sekundę trybu debugowania gry <i>Minecraft</i> dla zdefiniowanych platform sprzętowych, odrzucając pojedyncze skokowe wartości skrajne.	48
5.3 Wyniki pomiarów średniego czasu inicjalizacji badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	50
5.4 Wyniki pomiarów średniego czasu inicjalizacji <i>mapy wokseli</i> . Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	51
5.5 Wyniki pomiarów średniego czasu całkowitego wypełniania badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	52
5.6 Wyniki pomiarów średniego czasu częściowego wypełniania badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	53
5.7 Wyniki pomiarów średniego czasu wstawiania rzędu stu wokseli do badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	53
5.8 Wyniki pomiarów średniego czasu wstawiania wokseli w tysiąc deterministycznie losowych pozycji do badanych struktur danych. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	54
5.9 Wyniki pomiarów średniego czasu pojedynczego usuwania wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	55
5.10 Wyniki pomiarów średniego czasu masowego usuwania wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	55
5.11 Wyniki pomiarów średniego czasu deterministycznie losowego usuwania tysiąca wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	56
5.12 Wyniki pomiarów średniego czasu wyszukiwania następujących po sobie stu wokseli w strukturach danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	57
5.13 Wyniki pomiarów średniego czasu deterministycznie losowego wyszukiwania tysiąca wokseli w strukturach danych. Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	57

5.14 Wyniki pomiarów średniego czasu przeszukiwania tablicach i wektorach o różnych konfiguracjach. Wyniki te wskazują, że powodem drastycznego spowolnienia czasu przeszukiwania są najprawdopodobniej przechowywane wartości atomowe (<code>Arc</code>). Pomiar wykonano na maszynie klasy niższej (laptop). przy użyciu narzędzia Criterion.	58
5.15 Rozmiary pojedynczej instancji struktur danych oraz narzut na każdą dodawaną wartość zmierzone funkcją <code>std::mem::size_of</code>	59
5.16 Maksymalne zużycie pamięci, liczba alokacji i ich sumaryczny rozmiar dla pustych, zainicjowanych struktur zmierzone narzędziem <code>dhat</code>	59
5.17 Maksymalne zużycie pamięci, liczba alokacji i ich sumaryczny rozmiar dla struktur wypełnionych w 50% zmierzone narzędziem <code>dhat</code>	59
5.18 Maksymalne zużycie pamięci, liczba alokacji i ich sumaryczny rozmiar dla struktur wypełnionych w 100% wg <code>dhat</code> . Pomiary uzupełniono o dwa skrajne dodatkowe przypadki dla struktury octree, aby pokazać jaki wpływ na obciążenie pamięci ma wypełnienie całej struktury w jej wewnętrznej, binarnej hierarchii oraz dla tego samego wypełnienia z marginesem jednej komórki.	59
5.19 Statystyki FPS dla przemieszczania się po światach testowych z sekcji 4.7 dla komputera klasy wyższej (tab. 5.1).	62
5.20 Statystyki FPS dla przemieszczania się po światach testowych z sekcji 4.7 dla komputera klasy niższej (tab. 5.1)	63
5.21 Wpływ wybranych technik prezentowania obiektów na ekranie, na bazie świata testowego "mapa wysokościowa".	64
5.22 Wpływ liczby wątków na czas generowania świata.	64
A.1 Wyniki pomiarów średniego czasuinicjalizacji badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	86
A.2 Wyniki pomiarów średniego czasuinicjalizacji mapy wokseli . Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	87
A.3 Wyniki pomiarów średniego czasu całkowitego wypełniania badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	88
A.4 Wyniki pomiarów średniego czasu częściowego wypełniania badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	88
A.5 Wyniki pomiarów średniego czasu wstawiania rzędu stu wokseli do badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	89
A.6 Wyniki pomiarów średniego czasu wstawiania wokseli w tysiąc deterministycznie losowych pozycji do badanych struktur danych. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	89
A.7 Wyniki pomiarów średniego czasu pojedynczego usuwania wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	90

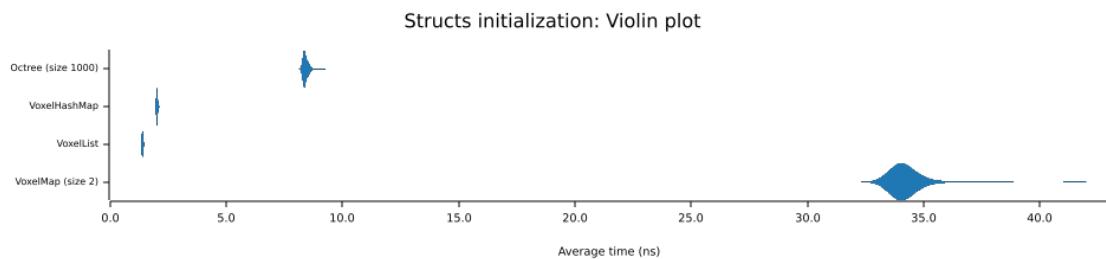
A.8 Wyniki pomiarów średniego czasu masowego usuwania wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	90
A.9 Wyniki pomiarów średniego czasu deterministycznie losowego usuwania tysiąca wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	91
A.10 Wyniki pomiarów średniego czasu wyszukiwania następujących po sobie stu wokseli w strukturach danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	91
A.11 Wyniki pomiarów średniego czasu deterministycznie losowego wyszukiwania tysiąca wokseli w strukturach danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	92
A.12 Wyniki pomiarów średniego czasu przeszukiwania tablicach i wektorach o różnych konfiguracjach. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.	92

A. WYNIKI POMIARÓW STRUKTUR NA PLATFORMIE KLASY WYŻSZEJ

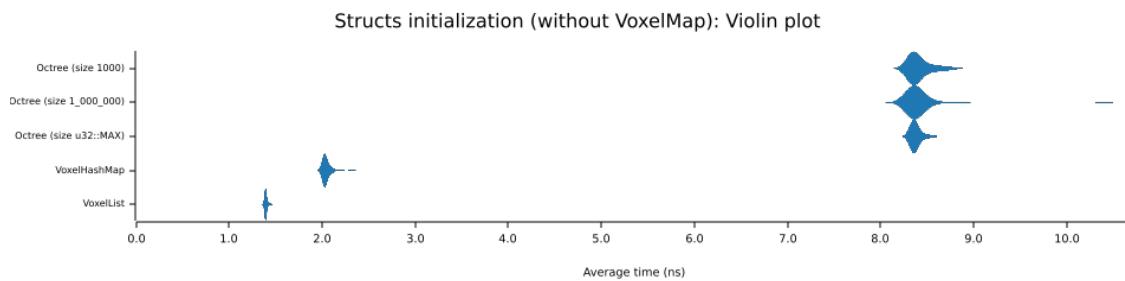
Niniejszy załącznik stanowi uzupełnienie badań przeprowadzonych w rozdziale 5.3 dla platformy sprzętowej klasy wyższej. Kolejność rysunków i tabel zostało zachowana.

Tabela A.1: Wyniki pomiarów średniego czasu inicjalizacji badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [ns]	Odchylenie standardowe [ns]
Mapa wokseli (dla 2^3 wokseli)	34.3622	1.0727
Lista wokseli	1.4130	0.0549
Tablica hashująca woksele	2.0416	0.0696
Octree (dla $1\ 000^3$ wokseli)	8.4463	0.1654
Octree (dla $1\ 000\ 000^3$ wokseli)	8.4091	0.2209



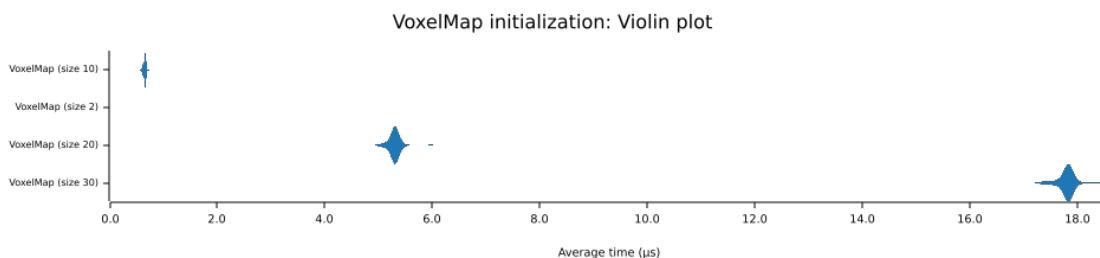
Rys. A.1: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa PDF (ang. *Probability Density Function*) dla pomiarów średniego czasu inicjalizacji badanych struktur danych. Na diagramie widnieje znaczaco odstający PDF dla mapy wokseli, co utrudnia ocenę czasu dla pozostałych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.



Rys. A.2: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasuinicjalizacji badanych struktur danych. Z badanych struktur **wyłączona została mapa wokseli** a octree zostało powtórzone dla różnych rozmiarów przechowywanych danych. W porównaniu struktura octree wypada najgorzej, natomiast dla wszystkich ocenianych rozmiarów tej struktury (stosunkowo małych oraz skrajnie dużych) czas inicjalizacji pozostaje taki sam. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.2: Wyniki pomiarów średniego czasu inicjalizacji **mapy wokseli**. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

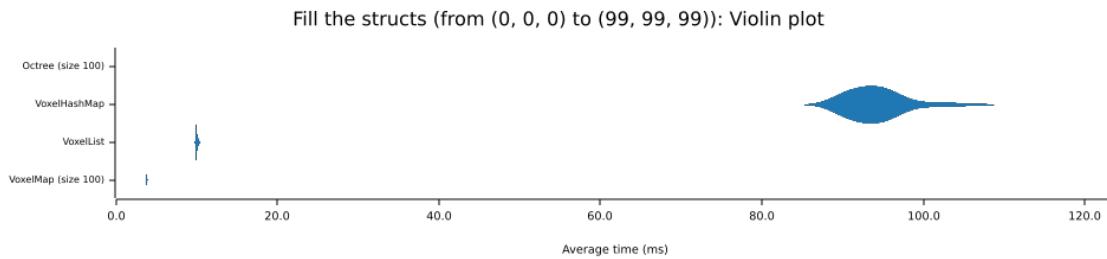
Struktura danych	Średni czas [μs]	Odchylenie standardowe [μs]
Mapa wokseli (dla 2^3 wokseli)	0.0346	0.0008
Mapa wokseli (dla 10^3 wokseli)	0.6513	0.0251
Mapa wokseli (dla 20^3 wokseli)	5.3059	0.1341
Mapa wokseli (dla 30^3 wokseli)	17.8024	0.1600



Rys. A.3: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów średniego czasu inicjalizacji struktury **mapy wokseli** o różnych rozmiarach. Widoczny jest na diagramie trend potęgowy rosnącego czasu inicjalizacji względem zadanego (bardzo małego) rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.3: Wyniki pomiarów średniego czasu **całkowitego wypełniania** badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

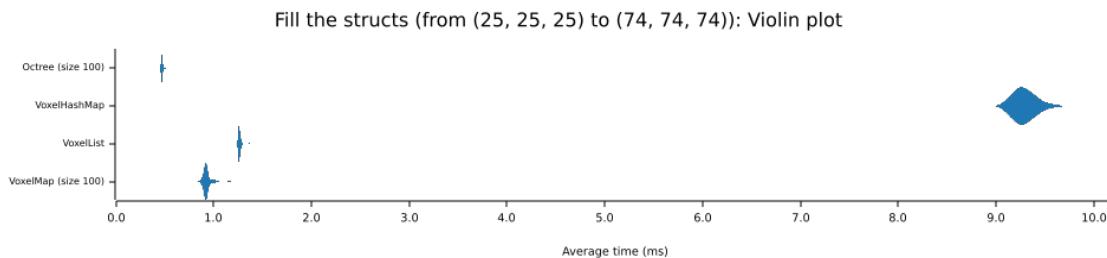
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	3.7777	0.0298
Lista wokseli	10.0558	0.1364
Tablica haszująca woksele	94.5262	4.4461
Octree (dla 100^3 wokseli)	0.0618	0.0014



Rys. A.4: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **całkowitego wypełniania** badanych struktur danych. Na diagramie charakterystyczne są wyniki dla *octree*, które są niewidoczne ze względu na swoją wielkość oraz dla *struktury haszującej* mającej znacząco dłuższy czas wypełniania względem pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.4: Wyniki pomiarów średniego czasu **częściowego wypełniania** badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

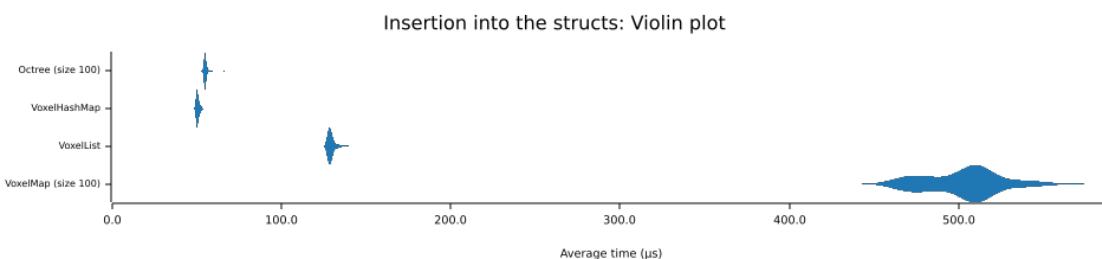
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	0.9295	0.0377
Lista wokseli	1.2673	0.0182
Tablica haszująca woksele	9.2998	0.1338
Octree (dla 100^3 wokseli)	0.4709	0.0075



Rys. A.5: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **częściowego wypełniania** badanych struktur danych – z marginesem kilkudziesięciu wokseli od krawędzi zdefiniowanego rozmiaru. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.5: Wyniki pomiarów średniego czasu **wstawiania rzędu** stu wokseli do badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

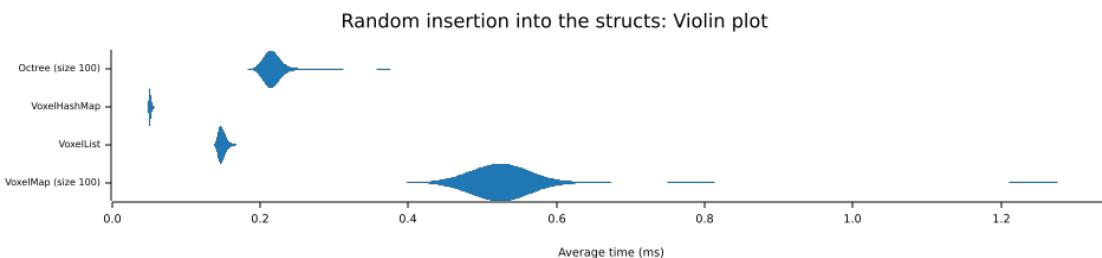
Struktura danych	Średni czas [μs]	Odchylenie standardowe [μs]
Mapa wokseli (dla 100^3 wokseli)	502.6090	21.6651
Lista wokseli	129.5310	2.3217
Tablica haszująca woksele	50.8605	1.3919
Octree (dla 100^3 wokseli)	55.3760	1.3165



Rys. A.6: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **wstawiania rzędu** stu wokseli do badanych struktur danych. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.6: Wyniki pomiarów średniego czasu wstawiania wokseli w tysiąc **deterministycznie losowych** pozycji do badanych struktur danych. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

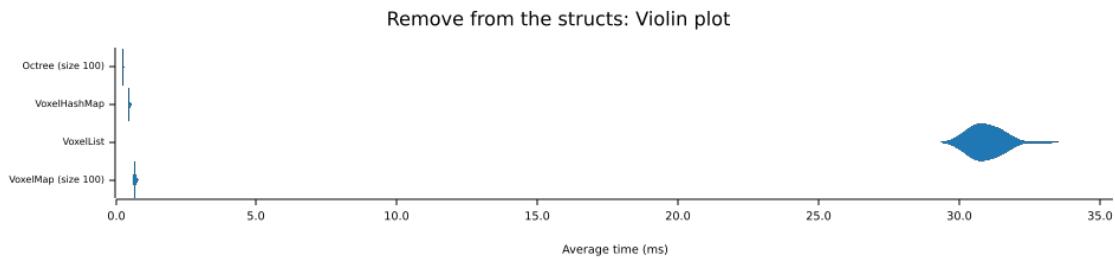
Struktura danych	Średni czas [μs]	Odchylenie standardowe [μs]
Mapa wokseli (dla 100^3 wokseli)	534.7431	80.0034
Lista wokseli	149.4789	5.6807
Tablica haszująca woksele	52.2331	2.3984
Octree (dla 100^3 wokseli)	221.0282	21.4746



Rys. A.7: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów wstawiania wokseli w tysiąc **deterministycznie losowych** pozycji, identycznych dla każdej struktury. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.7: Wyniki pomiarów średniego czasu **pojedynczego usuwania** wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

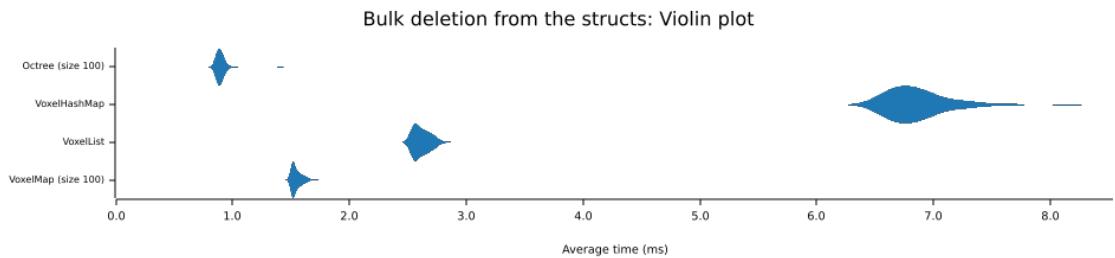
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	0.6812	0.0335
Lista wokseli	31.0077	0.7265
Tablica hashująca woksele	0.4678	0.0232
Octree (dla 100^3 wokseli)	0.2554	0.0109



Rys. A.8: Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **pojedynczego usuwania** wokseli z badanych struktur. Wyniki *listy wokseli* zmarginalizowały wyniki pozostałych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.8: Wyniki pomiarów średniego czasu **masowego usuwania** wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

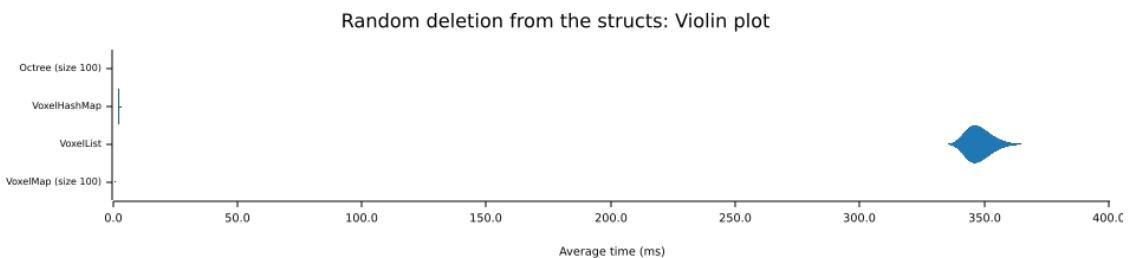
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	1.5471	0.0439
Lista wokseli	2.6227	0.0799
Tablica hashująca woksele	6.8797	0.3023
Octree (dla 100^3 wokseli)	0.8999	0.0587



Rys. A.9: Diagram wiolinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **masowego usuwania** obszaru z badanych struktur. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.9: Wyniki pomiarów średniego czasu deterministycznie **losowego usuwania** tysiąca wokseli z badanych struktur danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

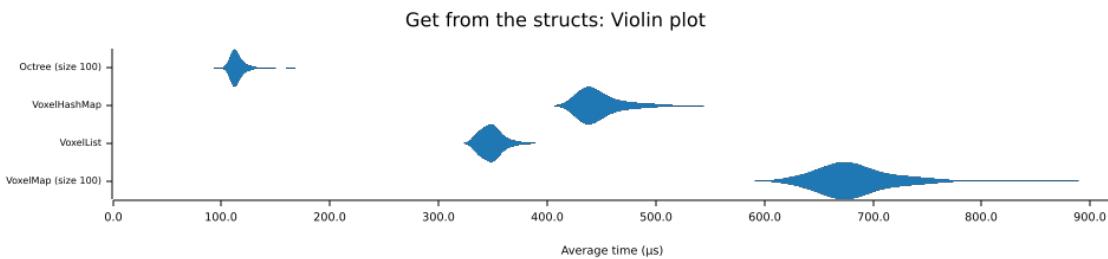
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	0.7238	0.0886
Lista wokseli	348.8322	7.2384
Tablica hashująca woksele	2.4039	0.2058
Octree (dla 100^3 wokseli)	0.6041	0.0495



Rys. A.10: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie **losowego usuwania** tysiąca weokseli z badanych struktur. Każda struktura danych otrzymała taki sam zestaw wybranych pozycji. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.10: Wyniki pomiarów średniego czasu **wyszukiwania następujących po sobie** stu wokseli w strukturach danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

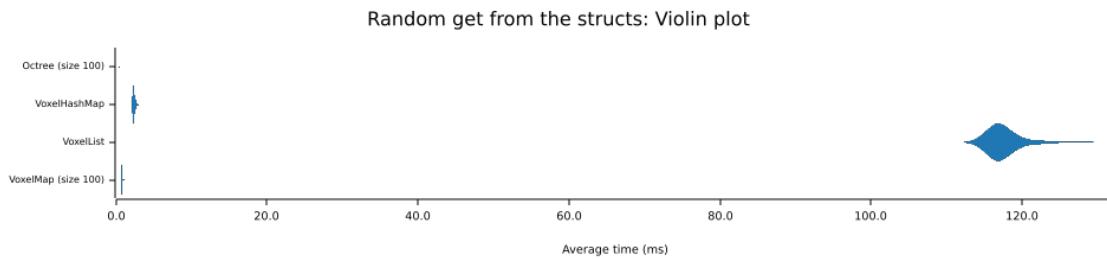
Struktura danych	Średni czas [μ s]	Odchylenie standardowe [μ s]
Mapa wokseli (dla 100^3 wokseli)	685.7700	39.9407
Lista wokseli	349.7889	13.0084
Tablica hashująca woksele	450.0020	23.2481
Octree (dla 100^3 wokseli)	115.6582	8.9014



Rys. A.11: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu **wyszukiwania następujących po sobie** stu wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.11: Wyniki pomiarów średniego czasu deterministycznie losowego wyszukiwania tysiąca wokseli w strukturach danych. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

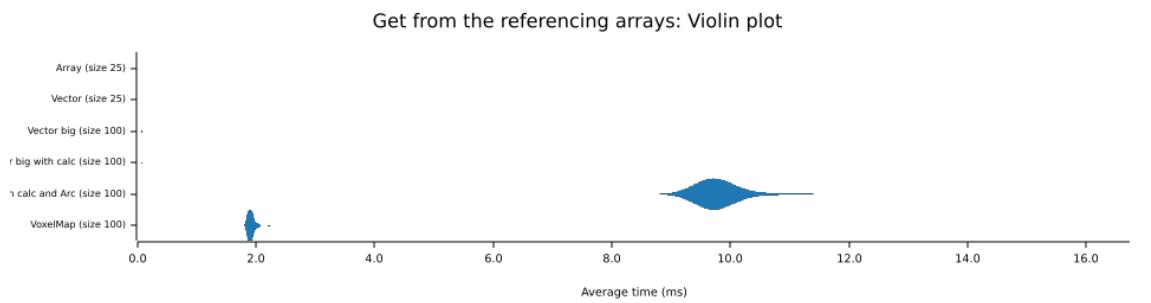
Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Mapa wokseli (dla 100^3 wokseli)	0.7809	0.0664
Lista wokseli	117.7771	2.6733
Tablica hashująca woksele	2.4044	0.1611
Octree (dla 100^3 wokseli)	0.3812	0.0232



Rys. A.12: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu deterministycznie losowego wyszukiwania tysiąca wokseli w badanych strukturach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.

Tabela A.12: Wyniki pomiarów średniego czasu przeszukiwania tablicach i wektorach o różnych konfiguracjach. Pomiar wykonano na maszynie klasy wyższej (laptop). przy użyciu narzędzia Criterion.

Struktura danych	Średni czas [ms]	Odchylenie standardowe [ms]
Tablice (dla 25^3 wokseli)	0.0028	0.0002
Wektor (dla 100^3 wokseli)	0.0003	0.0001
-// - z obliczanym indeksem	0.0692	0.0037
-// - i licznikami atomowymi	9.8827	0.7142
Mapa wokseli (dla 100^3 wokseli)	1.9202	0.0572



Rys. A.13: Diagram violinowy porównujący rozkłady gęstości prawdopodobieństwa (PDF) dla pomiarów czasu przeszukiwania tablic i wektorów o różnych konfiguracjach. Pomiary zostały wykonane i zwizualizowane za pomocą narzędzia Criterion w środowisku Rust na maszynie klasy wyższej.