Politechnika
Śląska

PRACA MAGISTERSKA

Techniki optymalizacji implementacji wokselowej reprezentacji świata gry

Dawid GROBERT
Nr albumu: ■■■■■■

Kierunek: Informatyka
Specjalność: Interaktywna Grafika Trójwymiarowa

PROWADZĄCY PRACĘ
Dr hab. inż. Agnieszka Szczęsna
KATEDRA Grafiki, Wizji Komputerowej i Systemów Cyfrowych
Wydział Automatyki, Elektroniki i Informatyki

Gliwice 2024

Tytuł pracy

Techniki optymalizacji implementacji wokselowej reprezentacji świata gry

Streszczenie

Celem niniejszej pracy jest analiza i porównanie różnych technik optymalizacji wokselowej reprezentacji świata gry, które są trójwymiarowymi siatkami umożliwiającymi tworzenie interaktywnych i szczegółowych środowisk. Praca obejmuje wprowadzenie do podstawowych pojęć, takich jak woksele i techniki renderowania. Omówiony został rozwój reprezentacji wokselowej w różnych dziedzinach. Przegląd literatury porusza algorytmy szybkiego przechodzenia przez woksele oraz struktury danych, takie jak mapy cegiełkowe, pozwalające na efektywne przechowywanie i strumieniowanie danych wokselowych, a także rzadkie drzewa ósemkowe, czy algorytm szybkiego przechodzenia przez drzewo ósemkowe. Omówiono również techniki stosowane w znanych grach komputerowych, takich jak „*Teardown*" i „*Minecraft*", oraz mniej znane techniki, takie jak wokselowy marsz promieni z efektem paralaksy. Na potrzeby badań opracowano silnik gry w oparciu o *C++* i *OpenGL API*, oraz odtworzono w nim badane techniki z podziałem na triangulację i śledzenie promieni. Zaimplementowano i porównano metody triangulacji, takie jak metoda naiwna, ukrywanie powierzchni, ukrywanie powierzchni ze wsparciem GPU, zachłanne siatkowanie oraz binarne zachłanne siatkowanie. Zbadano również zastosowanie różnych struktur danych wewnątrz brył wokseli, takie jak mapy cegiełkowe oraz drzewa ósemkowe dla śledzenia promieni.

Słowa kluczowe

Optymalizacja wokseli, śledzenie promieni, triangulacja, mapy cegiełkowe, drzewa ósemkowe, zachłanne siatkowanie, binarne zachłanne siatkowanie

Thesis title

Techniques for optimizing voxel-based representation of the game world

Abstract

The objective of this study is to analyze and compare various optimization techniques for voxel-based world representation, which are three-dimensional grids enabling the creation of interactive and detailed environments. This paper includes an introduction to fundamental concepts such as voxels and rendering techniques. The development of voxel representation in various fields is also discussed. The literature review covers algorithms for fast voxel traversal and data structures such as brick maps, which facilitate efficient storage and streaming of voxel data, as well as sparse octrees and the fast octree traversal algorithm. Techniques employed in well-known computer games such as „*Teardown*" and „*Minecraft*" are examined, along with lesser-known techniques like parallax voxel ray marching. For the purpose of the research, a game engine based on *C++* and the *OpenGL API* was developed, in which the investigated techniques were reproduced, categorized into triangulation and ray tracing. Triangulation methods such as the naive method, face culling, GPU-assisted face culling, greedy meshing, and binary greedy meshing were implemented and compared. The application of various data structures within voxel volumes, such as brick maps and octrees for ray tracing, was also examined.

Key words

Voxel optimization, ray tracing, triangulation, brick maps, octrees, greedy meshing, binary greedy meshing

Spis treści

Rozdział 1

Wstęp

1.1 Czym są woksele?

Woksele stanowią podstawową jednostkę objętości w przestrzeni trójwymiarowej. W kontekście wokselowej reprezentacji świata gry pełnią rolę podstawowego budulca, działając analogicznie do słynnych klocków *LEGO*. Pojęcie wokseli pochodzi od połączenia angielskich słów „volume" i „element", co podkreśla ich naturę elementów objętościowych. Każdy woksel posiada określone współrzędne trójwymiarowe oraz przypisane mu właściwości, takie jak kolor czy tekstura. Podobnie jak grafiki rastrowe, woksele charakteryzują się jednolitą rozdzielczością, gdzie każdy element jest niezależny od innych.

1.2 Do czego używa się woksele?

Woksele znajdują szerokie zastosowanie w różnych dziedzinach. W medycynie są wykorzystywane do obrazowania tomografii komputerowej (CT) i rezonansu magnetycznego (MRI) [15], umożliwiając szczegółowe przedstawienie wewnętrznych struktur ciała. W geologii woksele służą do modelowania trójwymiarowych struktur pod powierzchnią ziemi [21]. W architekturze i inżynierii umożliwiają dokładne przedstawienie materiałów i ich właściwości, co pozwala na przeprowadzanie analiz wytrzymałościowych oraz symulacji zachowań konstrukcji [36]. W druku 3D woksele umożliwiają tworzenie obiektów o złożonych wewnętrznych strukturach. W grach komputerowych woksele zyskały popularność dzięki możliwości tworzenia interaktywnych i szczegółowych środowisk.

1.3 Cel pracy

Celem niniejszej pracy jest analiza i porównanie różnych technik optymalizacji wokselowej reprezentacji świata gry. W ramach pracy przygotowano zaprojektowanie oraz implementację środowiska pomiarowego w *OpenGL API* oraz *C++*, implementację i pomiar

wydajności badanych technik, a także analizę otrzymanych wyników z zaimplementowanego systemu. Przeprowadzone badania mają na celu ocenę skuteczności różnych metod triangulacji i śledzenia promieni w kontekście wydajności renderowania, zajętości pamięci, czasu budowania scen oraz jakości grafiki. Ponadto, praca ma na celu zidentyfikowanie najlepszych praktyk w dziedzinie optymalizacji wokselowej reprezentacji świata gry.

1.4 Zakres pracy

W zakresie pracy zawarto:

- Zaprojektowanie oraz implementację środowiska pomiarowego w *OpenGL API* oraz *C++*.
- Implementację i pomiar wydajności badanych technik.
- Analizę otrzymanych wyników z zaimplementowanego systemu.

1.5 Charakterystyka rozdziałów

Praca składa się z pięciu rozdziałów:

1. **Wstęp** – Rozdział wprowadza do zagadnień poruszanych w przeprowadzanych badaniach, omawia dziedzinę problemu pracy, określa jej cel i ramy technologiczne. Zawiera również zwięzłą charakterystykę poszczególnych rozdziałów oraz zakres pracy.
2. **Analiza tematu** – Ten rozdział wprowadza czytelnika w szczegóły opracowywanego problemu poprzez zgłębienie jego dziedziny. Zawiera przegląd literatury naukowej oraz istniejących rozwiązań, prowadząc do wstępnego szkicu pracy badawczej.
3. **Przedmiot pracy** – Rozdział zawiera szczegółowy opis narzędzi i technik użytych w projekcie badawczym. Skupia się na technicznych aspektach działania różnych metod renderowania wokseli, takich jak triangulacja i śledzenie promieni.
4. **Badania** – W rozdziale opisano badania przeprowadzone w celu oceny wydajności różnych technik renderowania wokseli. Zawiera szczegółowe wyniki eksperymentów oraz ich interpretację.
5. **Podsumowanie** – Rozdział zawiera podsumowanie wykonanych prac oraz syntezę uzyskanych wniosków. Opisuje możliwe kierunki dalszych badań.

Rozdział 2

Analiza tematu

2.1 Wprowadzenie do dziedziny

Niniejsza praca zawiera opis i porównanie wielu technik związanych z optymalizacją modeli światów wokselowych. Modele te, oparte na trójwymiarowej siatce wokseli, pozwalają na tworzenie złożonych, łatwo modyfikowalnych i interaktywnych środowisk. Optymalizacja tych środowisk jest bardzo ważna w celu zachowania wydajności aplikacji i zapewnienia dostępności dla szerokiego grona użytkowników. Dzięki optymalizacji możliwe jest również zwiększenie rozdzielczości siatek wokseli, co skutkuje tworzeniem bardziej szczegółowych i realistycznych terenów oraz wyświetlaniem bardziej złożonych danych. Techniki optymalizacji omówione w niniejszej pracy umożliwiają redukcję obciążenia procesora i karty graficznej, jednocześnie zapewniając zachowanie porównywalnej lub identycznej jakości wizualnej. Zastosowanie tych technik znacząco wykracza poza branżę gier komputerowych, doskonale sprawdzając się również w takich dziedzinach jak geoinformatyka czy medycyna.

W tym rozdziale zostaną przedstawione kluczowe koncepcje dotyczące omawianego tematu. Na początku zostanie omówione pojęcie woksela oraz różnice pomiędzy nim a typową reprezentacją przy użyciu trójkątów. Następnie zostaną przedstawione dwie główne techniki renderowania, które zostaną wykorzystywane do generowania obrazów:

- **Rasteryzacja** – technika najczęściej używana w grach, polegająca na przekształceniu trójwymiarowych modeli na dwuwymiarowy obraz poprzez proces mapowania pikseli.
- **Śledzenie promieni** – technika, która do niedawna była głównie stosowana w przemyśle filmowym, ale obecnie znajduje coraz szersze zastosowanie również w grach oraz innych aplikacjach czasu rzeczywistego. Pozwala na uzyskanie bardziej realistycznych efektów świetlnych i cieni poprzez symulacje ścieżki światła w scenie.

2.2 Reprezentacja wokselowa

Pojęcie woksela jest często porównywane do piksela, jednak oba te terminy określają różne koncepcje. Termin „piksel" (ang. *Pixel*) powstał z połączenia dwóch angielskich słów [42]:

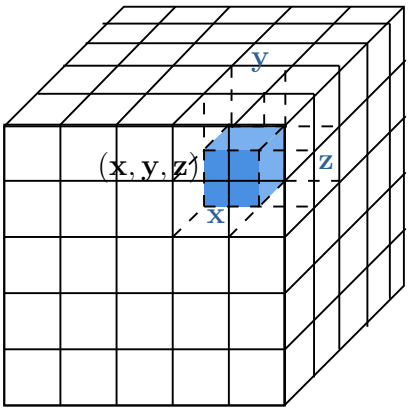
$$\text{Picture} + \text{Element} = \text{Pixel}$$

Piksel jest najmniejszą jednostką adresowalną w grafice rastrowej [42], będąc elementem dwuwymiarowej siatki, gdzie każdemu punktowi o współrzędnych x i y przypisana jest wartość, na przykład kolor. W ten sposób tworzony jest obraz cyfrowy. Analogicznie, pojęcie woksela (ang. *Voxel*) również pochodzi od złożenia dwóch słów:

$$\text{Volume} + \text{Element} = \text{Voxel}$$

Woksel jest zatem najmniejszą adresowalną jednostką, która reprezentuje objętość w przestrzeni trójwymiarowej [23], [19]. Jest to element siatki 3D, w której każdemu punktowi o współrzędnych x, y, z przypisana jest określona wartość.

Przykładowy woksel odpowiadający tej reprezentacji można zobaczyć na rysunku 2.1.



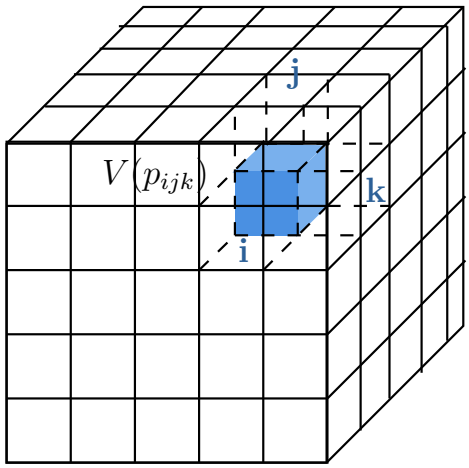
Rysunek 2.1: Pojedyńczy woksel w siatce wokseli.

Sceny wokselowe charakteryzują się jednolitą rozdzielczością, a każdy woksel jest niezależny od innych. Woksele są umieszczone w siatce o równych odstępach, bez uwzględnienia informacji topologicznych [44]. Taka struktura znacznie ułatwia modyfikowanie danych, gdzie przykładowo w przypadku siatek trójkątów wymaga to aktualizowania sąsiedztwa. Dzięki swojej prostocie umożliwia efektywne strumieniowanie danych przez podział obiektu na osobne elementy i ładowanie ich na żądanie, co pozwala na renderowanie bardzo dużych scen 3D przy niskim użyciu pamięci [3].

2.2.1 Matematyczna reprezentacja wokselowa

Matematycznie woksele można określić jako najmniejsze adresowalne jednostki objętości w przestrzeni trójwymiarowej uporządkowane w regularną, całkowitoliczbową siątkę przestrzeni $\mathbb{R}^3$. Przestrzeń ta jest reprezentowana przez zbiór $\mathbb{Z}^3$, który zawiera wszystkie punkty o całkowitoliczbowych współrzędnych. Każdy punkt w tej siatce oznaczony jest jako p_{ijk}, gdzie i, j, k są liczbami całkowitymi. Kazdy ten punkt definiuje lokalizacje narożnika sześcianu w przestrzeni. Dla każdego takiego punktu p_{ijk}, definiujemy zbiór $V(p_{ijk})$ jako jednostkowy sześcian rozciągający się od p_{ijk} do $p_{ijk} + (1, 1, 1)$. Innymi słowy jest to mały szcześcian o boku równym 1, którego jeden z narożników to punkt p_{ijk}. Zbiór $V(p_{ijk})$ zawiera wszystkie punkty (x, y, z), które spełniają warunki $i \leq x < i + 1$, $j \leq y < j + 1$, $k \leq z < k + 1$ ([3], [4], [17]). To oznacza, że każdy punkt w sześcianie $V(p_{ijk})$ ma współrzędne z przedziałów określonych przez odpowiednie wartości i, j, k. Jeśli rozpatrzymy wszystkie możliwe sześciany $V(p)$ dla każdego punktu p w siatce $\mathbb{Z}^3$ to wypełniają one całą przestrzeń $\mathbb{R}^3$ bez żadnych przerw, tworząc kompletne pokrycie całej przestrzeni trójwymiarowej, gdzie wnętrza każdego sześcianu $V(p)$ są rozłączne. Oznacza to, że żadne dwa sześciany nie mają wspólnych punktów wewnętrznych, a ich przecięcie mogłoby nastąpić jedynie na brzegach. W tym przypadku wokselem jest zbiór punktów $V(p_{ijk})$.

Przykładowy woksel odpowiadający przedstawionemu zapisowi matematycznemu można zobaczyć na rysunku 2.2.



Rysunek 2.2: Pojedyńczy woksel w siatce wokseli przedstawiony matematycznie.

2.3 Woksele czy trójkąty?

Historycznie woksele były zdecydowanie mniej popularną alternatywą dla trójkątów. Trójkąty są od dawna standardem w grafice komputerowej i przez wiele lat karty graficzne były rozwijane z myślą o pracy z nimi. W odróżnieniu od wokseli, trójkąty mogą tworzyć gładkie, ciągłe powierzchnie. Woksele natomiast nie tworzą ciągłych powierzchni, ponieważ każdy woksel jest oddzielną jednostką objętości. Prowadzi to do problemu, w którym tworzenie wrażenia gładkich powierzchni jest trudniejsze i wymaga wysokiej rozdzielczości, co z kolei przekłada się na duże zapotrzebowanie na pamięć [18]. Jest to problem, który wiele lat temu był nie do przeskoczenia. Jednym z podejść mających na celu zaradzenie problemowi nieciągłości powierzchni wokseli było stosowanie algorytmu maszerujących sześcianów, który pozwalał wygładzić przejścia między wokselami [40].

W latach 70. i 80. XX wieku woksele były głównie używane w medycynie do obrazowania tomografii komputerowej (CT) i rezonansu magnetycznego (MRI) [15]. Świetnie sprawdzały się do podglądu wolumetrycznych danych przedstawiających wewnętrzne struktury ciała. Do lat 2010 woksele były raczej stosowane głównie w specjalistycznych dziedzinach, takich jak medycyna czy geologia, w celu modelowania trójwymiarowych struktur pod powierzchnią ziemi. W międzyczasie pojawiały się eksperymenty z użyciem wokseli w grach w celu generowania terenu, jednak nie były to spektakularne sukcesy. Dopiero gra „*Minecraft*" z 2011 roku spopularyzowała woksele na niespotykaną dotąd skalę w grach komputerowych [11]. Był to również czas, gdy woksele zaczęły zyskiwać na popularności dzięki rosnącej mocy obliczeniowej oraz zwiększonej dostępności pamięci. Można powiedzieć, że pojawiła się technologia, która pozwoliła na renesans wokseli.

W momencie pisania tej pracy, stale rosnące wymagania dotyczące szczegółowości geometrycznej uzasadniają ponowne rozważenie wokseli jako podstawowego prymitywu do rysowania i edycji świata w grach komputerowych. Dzięki ogromnej ilości pamięci dostępnej w nowoczesnych systemach, dane wokselowe mogą być przechowywane w wystarczająco wysokich rozdzielczościach. Tymczasem trójkąty stają się coraz mniej wydajne wraz ze wzrostem szczegółowości. Kompaktowość trójkątów traci na znaczeniu, gdy przechowywane powierzchnie są zbudowane z dużej liczby trójkątów, co prowadzi do sytuacji, w której liczba trójkątów na piksel może być mniejsza niż jeden [45]. Woksele przechowują dane o wysokiej rozdzielczości w sposób efektywny, ze względu na ich stałą rozdzielczość. Ponadto, struktura wokseli umożliwia efektywne operacje na różnych poziomach szczegółowości dla geometrii znajdujących się w oddali. W dodatku regularny układ siatki wokseli sprzyja także wydajnym algorytmom strumieniowania [45].

2.4 Woksele jako reprezentacja świata gier

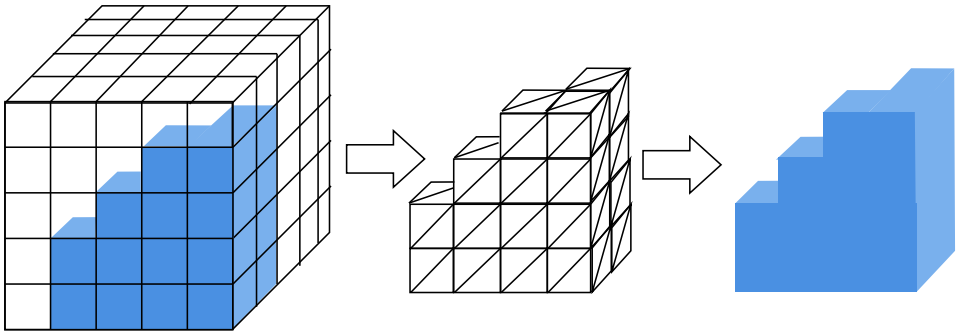
Od samego początku, zastosowanie wokseli w grach komputerowych narzuca pewne wymagania, które muszą zostać spełnione. Przede wszystkim, taki system musi działać w czasie zbliżonym do rzeczywistego na typowym sprzęcie konsumenckim. Kluczowym aspektem wokseli jest ich wysoka interaktywność; z tego powodu, byłoby dużą stratą wykorzystywanie wokseli do reprezentacji świata gry bez umożliwienia graczowi modyfikowania terenu. Gry powinny zachować wysoki poziom interaktywności między graczem a wirtualnym światem. Woksele doskonale sprawdzają się jako metoda reprezentacji danych podpowierzchniowych. Światy zbudowane przy użyciu wokseli umożliwiają generowanie rzeczywistych trójwymiarowych struktur, takich jak mosty, jaskinie czy wiszące skarpy. Naturalnym benefitem wykorzystania wokseli jest prostota edycji geometrii. Usunięcie części terenu odsłania ukryte pod nią elementy, nie powodując przy tym powstawania artefaktów czy dziur.

Edytowalność geometrii oraz wymóg wysokiej responsywności gry powodują, że nie wszystkie struktury danych zdolne do rysowania i przechowywania dużych zestawów danych wokselowych znajdują tutaj zastosowanie. Na przykład, choć popularna *rzadka struktura oktalna wokseli* (ang. *Sparse Voxel Octree*) [18] zapewnia efektywne przechowywanie danych i efektywne śledzenie promieni, wymaga kosztownej przebudowy w momencie edycji. Analogiczna sytuacja występuje w przypadku skierowanych acyklicznych grafów (ang. *Directed Acyclic Graph*), które umożliwiają eliminację redundancji w strukturze danych, jednakże związane są z wysokim kosztem budowy i aktualizacji struktury. Statyczna natura takich struktur danych stanowi znaczący problem w środowisku gier komputerowych, co potencjalnie wyklucza ich bezpośrednie zastosowanie bez zastosowania hybrydowego podejścia w połączeniu z inną techniką.

2.5 Wprowadzenie do technik renderowania

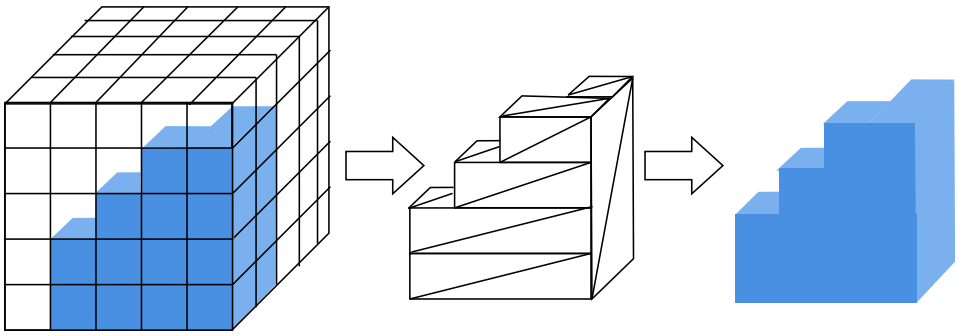
2.5.1 Triangulacja (rasteryzacja)

Triangulacja polega na przekształceniu powierzchni wokseli w siatkę trójkątów, które są następnie renderowane przy użyciu technik rasteryzacji. Przez długi czas była to dominująca metoda ze względu na specjalizację kart graficznych w rysowaniu trójkątów. Trójkąt jest kształtem, który pozwala utworzyć wielokąt przy użyciu najmniejszej liczby wierzchołków, a do tego jest niezwykle efektywny pod względem pamięci i obliczeń.



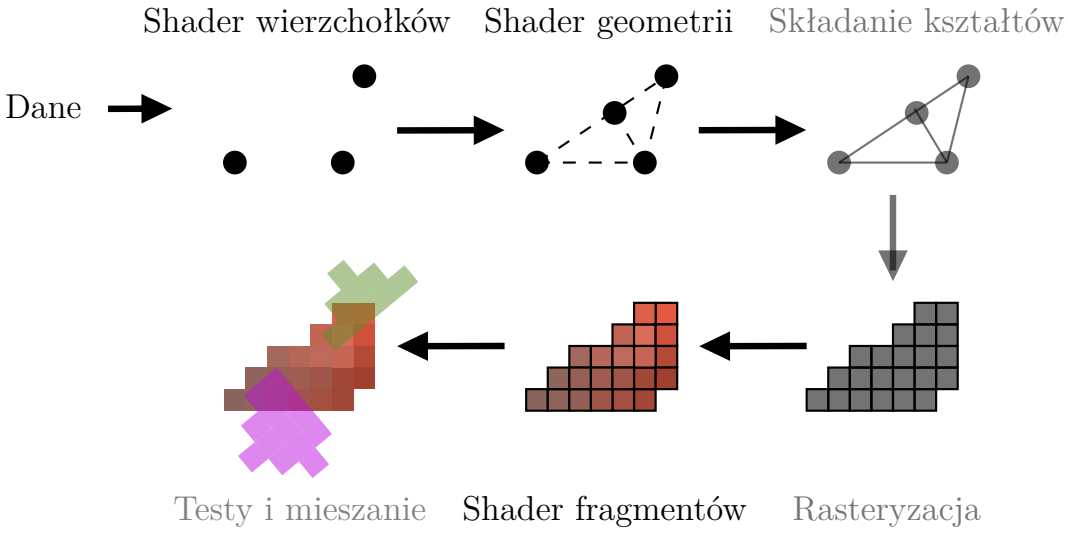
Rysunek 2.3: Przekształcanie powierzchni wokseli w siatkę trójkątów i proces rasteryzacji.

Taki proces da się przeprowadzić na wiele sposobów. Niektóre techniki potrafią skutecznie zredukować liczbę trójkątów zachowując przy tym identyczny obraz wyjściowy przy zwiększonej wydajności. Na rysunku 2.4 widać zredukowaną liczbę trójkątów w porównaniu do rysunku 2.3.



Rysunek 2.4: Przekształcenie powierzchni wokseli w siatkę zredukowanych trójkątów.

Triangulacja to technika renderowania, która pozwala wykorzystać cały dostępny programowalny potok graficzny. Uproszczona wersja potoku widoczna jest na rysunku 2.5.



Rysunek 2.5: Uproszczone przedstawienie potoku graficznego, w którym jednostki cieniujące *wierzchołków*, *geometrii* i *fragmentów* stanowią etapy programowalne.

1. **Jednostka cieniująca wierzchołków** jest programem wykonywanym dla każdego wierzchołka. Transformuje pozycje wierzchołków, oraz przetwarza atrybuty wierzchołków.

2. **Jednostka cieniująca geometrii** jest programem wykonywanym dla całych prymitywów. Może je modyfikować, lub generować nowe dodając dodatkowe wierzchołki.

3. **Składanie kształtów** jest procesem składania wierzchołków w prymitywy.

4. **Rasteryzacja** jest procesem przekształcenia prymitywów na piksele. Dzieli prymitywy na fragmenty.

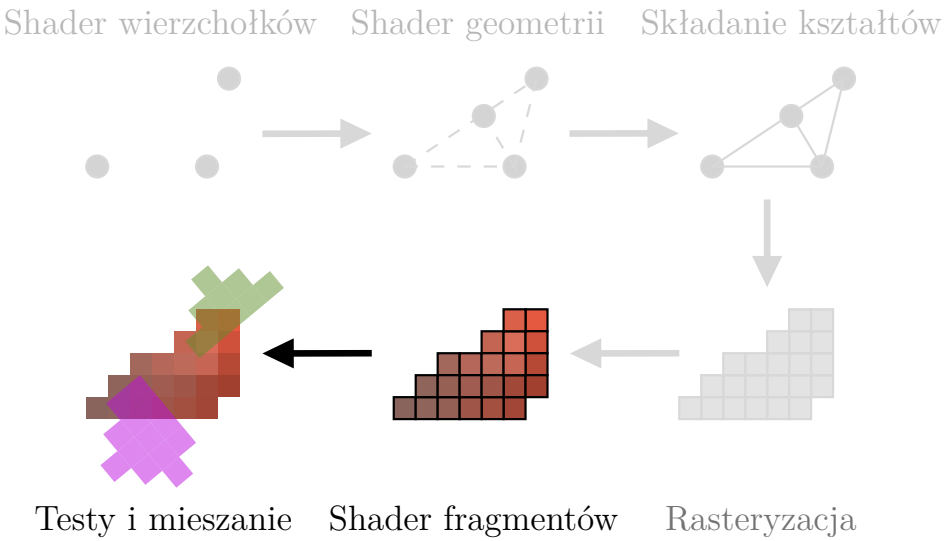
5. **Jednostka cieniująca fragmentów** jest programem wykonywanym dla każdego piksela. Określa końcowe wartości kolorów pikseli.

6. **Testy i mieszanie** są procesem sprawdzenia między innymi głębokości w celu okreslenia, które fragmenty powinny być rysowane. Łączy kolory fragmentów z kolorami w buforze ramki.

Programowalne etapy potoku graficznego obejmują jednostkę cieniującą wierzchołków, jednostkę cieniującą geometrii oraz jednostkę cieniującą fragmentów. Warto zaznaczyć, że jest to uproszczony schemat potoku. Dodatkowymi programowalnymi etapami są jednostka cieniująca kontroli teselacji i jednostka cieniująca ewaluacji teselacji, które umożliwiają zdefiniowanie poziomu teselacji, oraz jednostka cieniująca obliczeniowa, która pozwala na wykonywanie obliczeń niezwiązanych bezpośrednio z renderingiem [38], [29].

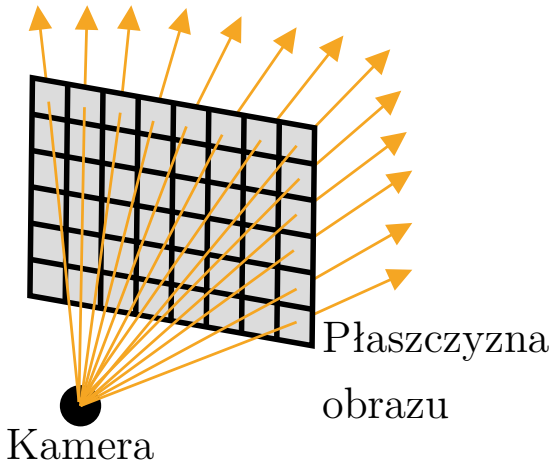
2.5.2 Śledzenie promieni

Śledzenie promieni jest techniką, która silnie koncentruje się na ostatnich etapach potoku renderowania, jak przedstawiono na rysunku 2.6.

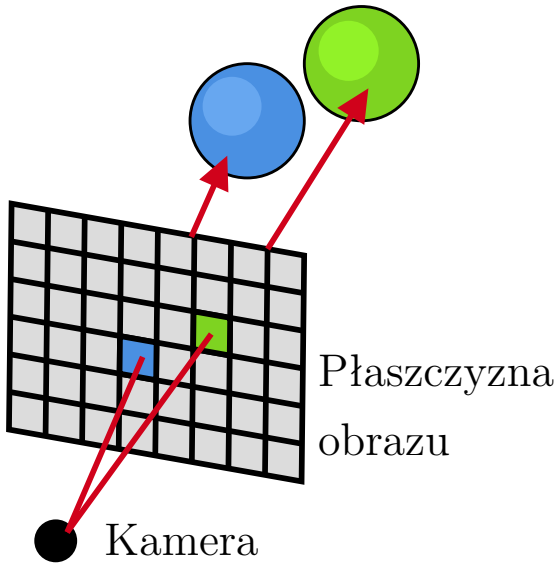


Rysunek 2.6: Istotna część potoku graficznego w technice śledzenia promieni.

W celu wykorzystania tej techniki potrzebne są tylko dwa trójkąty (cztery wierzchołki) stanowiące płaszczyznę obrazu. Promienie, odpowiadające liczbie pikseli w rozdzielczości obrazu, są wysyłane z pozycji kamery w stronę tej płaszczyzny. Promienie te przechodzą przez środek pikseli na płaszczyźnie, tworząc siatkę punktów (patrz rysunek 2.7). Każdy promień jest śledzony w celu zidentyfikowania jego przecięć z obiektami w scenie. W momencie napotkania obiektu obliczane są zjawiska świetlne, a informacje te służą określeniu koloru pikseli (patrz rysunek 2.8).



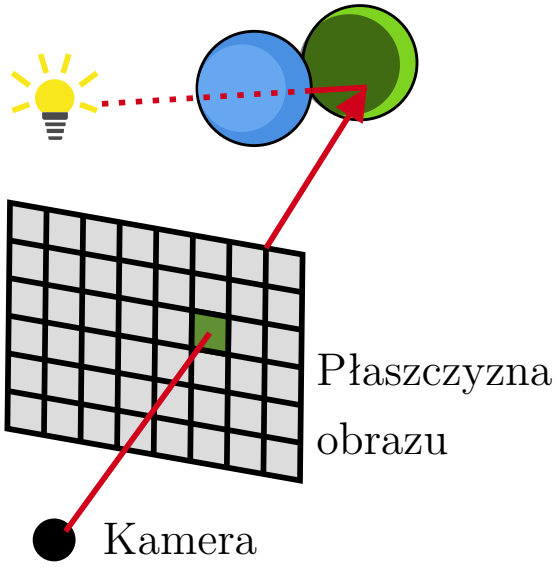
Rysunek 2.7: Promienie wysyłane z kamery w stronę płaszczyzny obrazu.



Rysunek 2.8: Promienie wysyłane z kamery trafiają w sfery, barwiąc piksele na kolor przeciętej sfery.

W najprostszym przypadku generowany jest tylko jeden promień na piksel na płaszczyź-

nie ekranu. Prowadzi to jednak do ograniczonej wizualizacji bez uwzględnienia cieni i odbić. W celu symulacji cieni generowany jest dodatkowy promień w punkcie trafienia w kierunku źródła światła. Jeśli promień jest zasłonięty to punkt znajduje się w cieniu (patrz rysunek 2.9). Za pomocą dodatkowego promienia można uzyskać idealne odbicia, czy cienie. Jednak do uzyskania efektów takich jak miękkie cienie czy połyskujące odbicia konieczne jest śledzenie wielu promieni.



Rysunek 2.9: Promień wysłany z kamery trafia w sferę, ale nie dociera do źródła światła bez przecięcia innego obiektu. Oznacza to, że obiekt jest zacieniony, więc piksel staje się ciemnozielony.

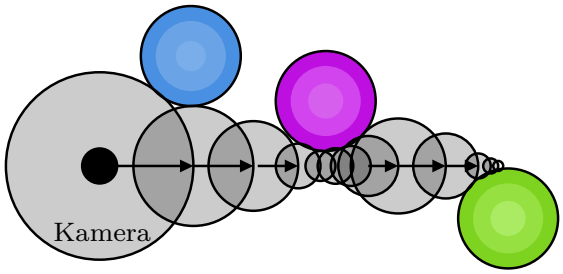
Podobnie jak w przypadku triangulacji (rasteryzacji), gdzie wydajność zwiększa się przez zmniejszenie liczby trójkątów, w śledzeniu promieni można zwiększyć wydajność poprzez redukcję liczby testów przecięcia promieni z obiektami. W tym celu skutecznie działają struktury przyspieszające. Najczęściej spotykane to:

- Drzewo k-wymiarowe (ang. *kD Tree*) – struktura danych dzieląca przestrzeń na podobszary za pomocą płaszczyzn podziału.
- Drzewo ósemkowe (ang. *Octree*) – dzieli przestrzeń na równe podobszary wzdłuż trzech głównych osi.
- Hierarchia objętości ograniczających (ang. *Bounding Volume Hierarchy*) – struktura, w której każdy węzeł przechowuje objętość ograniczającą obejmującą swoje dzieci.

2.5.2.1 Maszerowanie promieni

Maszerowanie promieni jest podrodzajem algorytmu śledzenia promieni. Różni się tym, że zamiast szukać bezpośrednio punktu przecięcia to promień jest iteracyjnie przesuwany

w stałych lub adaptacyjnych krokach. Jest to technika świetnie sprawdzająca się przy rysowaniu obiektów wolumetrycznych lub fraktali. Technika ta jest często prezentowana na przykładzie funkcji odległości (ang. *Signed Distance Function*), które zwracają minimalną odległość do najbliższej powierzchni obiektu. Na rysunku 2.10 można zaobserwować charakterystyczne „zwalnianie" promienia w pobliżu obiektu oraz jego „przyspieszanie" w obszarach odległych od figur geometrycznych. Trafienie w powierzchnię jest wykrywane, gdy krok iteracyjny staje się bardzo mały, natomiast brak trafienia jest stwierdzany po osiągnięciu maksymalnej liczby iteracji.



Rysunek 2.10: Promień przesuwany jest w adaptacyjnych krokach równych dystansowi do najbliższego obiektu.

Jest to technika bardzo często stosowana przy rysowaniu scen wokselowych, gdzie promień iteracyjnie jest przesuwany przez przestrzeń wokselową w poszukiwaniu niepustego woksela. W celu zwiększenia wydajności stosuje się struktury danych pozwalające szybko pominąć puste obszary wokselowej przestrzeni.

2.6 Przegląd literatury

2.6.1 Szybki algorytm przechodzenia przez woksele dla śledzenia promieni

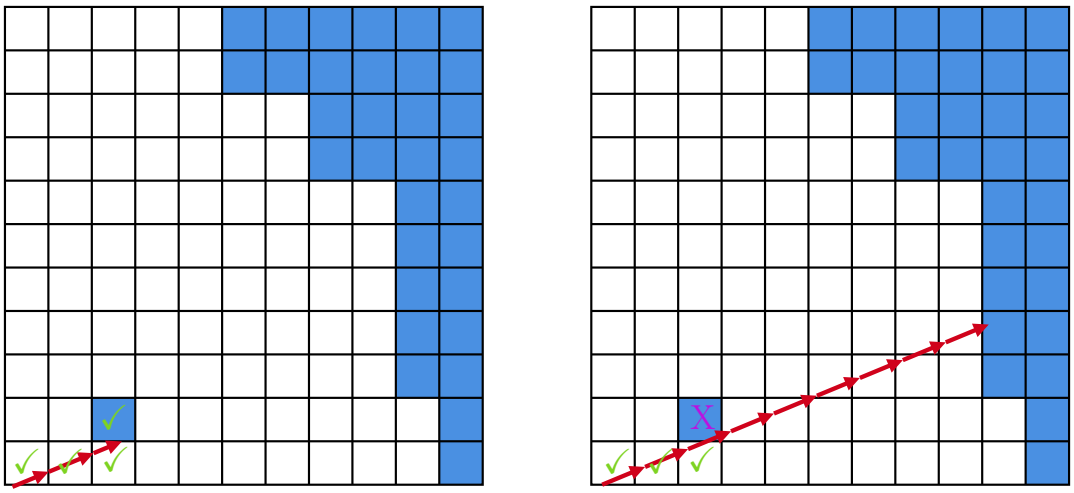
Chociaż samo śledzenie promieni jest proste w swoich założeniach i zapewnia łatwe w implementacji modelowanie odbicia, załamania i cieni, to jego naiwna implementacja jest niezwykle kosztowna obliczeniowo. Z tego powodu problem optymalizacji śledzenia promieni jest szeroko omawiany w literaturze naukowej. Klasycznym przykładem literatury naukowej w zakresie rysowania wokseli jest przełomowa praca Johna Amanatidesa i Andrew Woo, zatytułowana „*A Fast Voxel Traversal Algorithm for Ray Tracing*", która przedstawia szybki algorytm przechodzenia przez woksele [1]. Na wstępie autorzy wspominają, że śledzenie promieni potrafi zająć do 95% swojego czasu rysowania na obliczenia związane z przecięciami promieni z obiektami. W takim wypadku kluczowym aspektem optymalizowania metody śledzenia promieni jest redukcja liczby przecięć.

2.6.1.1 Objętościowe hierarchie ograniczające

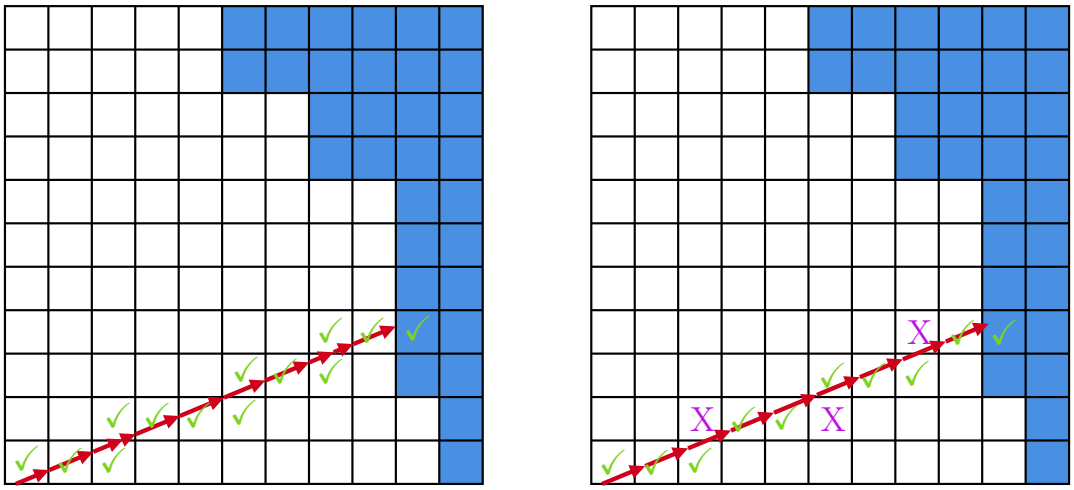
Jednym z możliwości redukcji liczby przecięć polega na otaczaniu złożonych obiektów prostszymi bryłami ograniczającymi. W ten sposób efektywnie można odrzucić promienie nieprzecinające tych brył. Jeśli nie ma przecięcia z obwiednią, nie ma potrzeby przecinania skomplikowanego obiektu. Jeśli obiekt składa się z kilku częsci, każda z tych częsci również może mieć swoją obwiednie. To samo tyczy się dużych złożonych scen, które mają swoje obwiednie i obwiednie wewnątrz. Obiekty w poddrzewie są przecinane tylko wtedy gdy obwiednia węzła nadrzędnego zostaje przecięta przez promień.

2.6.1.2 Podziały przestrzenne

Innym sugerowanym podejściem jest podzielenie przestrzeni na woksele, z których każdy zawiera listę obiektów. Promień sprawdza przecięcia tylko z obiektami znajdującymi się w danym wokselu. W tym kontekście Amanatides i Woo proponują ulepszenie algorytmu przechodzenia przez woksele, bazujące na prostym algorytmie DDA (Digital Differential Analyzer). W ich podejściu dynamicznie ustalane jest, która oś zostanie przetworzona w danym kroku. Każdy woksel na drodze promienia jest odpytywany tylko jeden raz, a żaden woksel na jego ścieżce nie zostaje pominięty (patrz rysunek 2.11 oraz rysunek 2.12).



Rysunek 2.11: Na lewym rysunku widoczne jest, jak promień algorytmu szybkiego przechodzenia przez woksele (ang. *Fast Voxel Traversal Algorithm, FVTA*) trafia w pełny woksel na swojej drodze. Natomiast na prawym rysunku, dla promienia o stałym kroku, woksel ten zostaje pominięty, a promień trafia w inny woksel znajdujący się znacznie dalej.



Rysunek 2.12: Na lewym rysunku widoczne jest, jak promień algorytmu FVTA przechodzi przez każdy woksel na swojej drodze. Natomiast na prawym rysunku, dla promienia o stałym kroku, wiele wokseli zostaje pominiętych.

Algorytm w trzech wymiarach, wewnątrz pętli, wymaga jedynie dwóch porównań zmiennoprzecinkowych, jednego dodawania zmiennoprzecinkowego, dwóch porównań całkowitych oraz jednego dodawania całkowitego na iterację, co czyni go wydajnym. Faza inicjalizacji wymaga 33 operacji zmiennoprzecinkowych (w tym porównań), jeśli początek promienia znajduje się w siatce, oraz do 40 operacji zmiennoprzecinkowych w przeciwnym przypadku. Algorytm opisuje również przypadek rysowania obiektów, które znajdują się jednocześnie w więcej niż jednym wokselu, jednak nie jest to przypadek istotny w kontekście niniejszej pracy. Silnik, który zostanie przygotowany do badań, będzie ściśle oparty na siatce wokseli, co oznacza, że nie będzie wykorzystywał wokseli opartych na sześcianach w swobodnym środowisku 3D.

2.6.1.3 Technika eliminacji wielokrotnych przecięć

Autorzy wprowadzają również technikę eliminacji wielokrotnych przecięć z obiektami znajdującymi się jednocześnie w kilku wokselach, co dodatkowo zwiększa efektywność algorytmu. Technika ta jednak nie jest przydatna w kontekście niniejszej pracy, ponieważ silnik przygotowywany na potrzeby badań będzie ściśle oparty na siatce wokseli. W takim przypadku problem wielokrotnych przecięć nie występuje. Aby jednak dopełnić analizę, warto omówić to rozwiązanie, które zakłada przypisanie do każdego promienia zmiennej całkowitej, zwanej *rayID*. Każdy obiekt na scenie przechowuje również *rayID* promienia, który ostatnio wykonał test przecięcia z tym obiektem. Przed wykonaniem testu przecięcia porównywane są *rayID* promienia i obiektu. Jeśli są równe, oznacza to, że obiekt został już przecięty przez ten sam promień i bieżące przecięcie nie jest konieczne. W przeciwnym razie wykonywany jest test przecięcia, a *rayID* obiektu zostaje ustawione na wartość *rayID* promienia.

2.6.1.4 Wyniki

Wyniki badań pokazują, że nowy algorytm znacznie redukuje liczbe przecięć, nawet dla scen zawierających dużą liczbe obiektów. Nawet przy bardzo dużej liczbie obiektów, liczba obiektów, które rzeczywiście muszą być przecięte pozostaje niewielka.

2.6.2 Śledzenie promieni w czasie rzeczywistym i edycja dużych scen wokselowych

W kontekście renderowania i edycji dużych scen wokselowych w czasie rzeczywistym, interesującą pracą jest artykuł Thijsa van Wingerdena, zatytułowany „Real-time Ray tracing and Editing of Large Voxel Scenes" [45]. Autor skupia się na innowacyjnym podejściu do renderowania dużych scen wokselowych z naciskiem na możliwość ich edycji i strumieniowania dużych zbiorów danych. W odróżnieniu od wcześniejszych rozwiązań, które często wymagają przetwarzania wstępnego danych, autor proponuje przechowywanie danych w formacie surowym, co pozwala na bezpośrednią edycję bez konieczności przetwarzania wstępnego.

W pracy autor skupił się na systemie przechowywania danych wokselowych w strukturach zwanych „Mapa cegiełkowa" (ang. *Brickmap*) oraz „Siatka Cegiełkowa" (ang. *Brickgrid*). Mapa cegiełkowa stanowi podstawową jednostkę reprezentacji danych wokselowych i jest trójwymiarową siatką o stałych wymiarach 8x8x8 wokseli. Z kolei siatka cegiełkowa jest wyższym poziomem organizacji danych, stanowiącym siatkę zawierającą wskaźniki do mapy cegiełkowej. Ważnym aspektem tego systemu jest fakt, że puste mapy cegiełkowe nie muszą być przechowywane w pamięci, co skutecznie redukuje jej zużycie. Istotną zaletą, podkreślaną przez autora, jest surowy format tych struktur, który nie wymaga przetwarzania wstępnego. Technika ta charakteryzuje się wysoką interaktywnością i minimalnym kosztem edycji. Ponadto, wiedza o tym, które mapy cegiełkowe są puste, umożliwia efektywne omijanie dużych, pustych przestrzeni, co przyczynia się do zwiększenia wydajności śledzenia promieni. Mapy cegiełkowe kodują dane w formie maski bitowej i przechowują kolory oddzielnie. Oddzielne przechowywanie kolorów przyspiesza operacje związane z edycją, a dostęp do danych kolorystycznych odbywa się tylko wtedy, gdy jest to potrzebne. Dodatkowo, podział danych na maski bitowe i kolory pozwala na bardziej efektywną kompresję kolorów.

Autor wprowadza także system strumieniowania danych oparty na technice odrzucania zasłoniętych obiektów (ang. *Occlusion Culling*). W ramach tego systemu zdefiniowano jasny podział zadań między jednostki GPU (Graphics Processing Unit) i CPU (Central Processing Unit). CPU jest odpowiedzialny za ładowanie i przetwarzanie danych wokselowych z dysku, ich kompresję oraz dekompresję. Ponadto, CPU zajmuje się dynamiczną alokacją pamięci dla mapy cegiełkowej i aktualizacją wskaźników w siatki ciegiełkowej. Procesor wykonuje również testy widoczności wokseli, obsługuje strumieniowanie danych

z dysku do pamięci RAM i zarządza buforowaniem danych. GPU, z kolei, odpowiada za śledzenie promieni oraz generowanie buforów sprężenia zwrotnego, które informują CPU o brakujących danych wokselowych niezbędnych do procesu renderingu.

W ostatniej części pracy autor porównał swoją technikę z rzadką strukturą oktalną wokseli (ang. *Sparse Voxel Octree*). Dla bardziej losowych i złożonych geometrii, takich jak scena Hairball, mapy cegiełkowe okazały się być znacznie bardziej efektywne niż rzadka struktura oktalna wokseli. Wynika to z faktu, że liczbę niepustych regionów jest znacznie większa, co prowadzi do dużego rozrostu drzewa i zwiększenia zużycia pamięci. Dodatkowo, prosta implementacja map cegiełkowych umożliwia łatwe zaimplementowanie techniki odrzucania zasłoniętych obiektów, co w przypadku rzadkiej struktury ósemkowej wokseli jest znacznie trudniejsze.

2.6.3 Algorytm przecięcia promienia z pudełkiem i wydajne dynamiczne renderowanie wokseli

W artykule opublikowanym w „Journal of Computer Graphics Techniques" [22] autorzy przedstawiają nowy algorytm przecięcia promienia z pudełkiem, zoptymalizowany zarówno dla pudełek osiowo-wyrównanych, jak i zorientowanych. Zaproponowana metoda stanowi ulepszenie tradycyjnej metody płytowej (ang. *Slab Method*), polegającej na dzieleniu przestrzeni pudełka na trzy pary równoległych płaszczyzn i obliczaniu momentów przecięcia promienia z tymi płaszczyznami. Nowa metoda opiera się na zastosowaniu maski bitowej, co pozwala na odrzucanie tylnych ścianek i redukcję rozgałęzień w kodzie.

Maski bitowe w algorytmie zawierają informacje o przecięciach, wektorze normalnym i umożliwiają łatwe wyznaczenie odległości do punktu przecięcia. Dzięki temu algorytm unika tradycyjnych instrukcji warunkowych, co znacząco poprawia wydajność na architekturach GPU, gdzie rozgałęzienia są kosztowne.

Algorytm rozpoczyna się od przekształcenia promienia do lokalnego układu współrzędnych pudełka, po czym identyfikuje potencjalne płaszczyzny przecięcia. Następnie obliczana jest maska bitowa, która określa, które płaszczyzny są przecięte przez promień. Maska bitowa składa się z trzech elementów odpowiadających trzem osiom (x, y, z), z których każdy zawiera informację o przecięciu (prawda/fałsz). Na podstawie maski bitowej algorytm wyznacza odległość do punktu przecięcia oraz wektor normalny odpowiedniej płaszczyzny.

Algorytm został porównany z istniejącymi metodami, takimi jak metoda płytowa (ang. *Slab Method*), i okazał się być szybszy, szczególnie w przypadku pudełek zorientowanych. Dodatkowo, algorytm znacząco przewyższa inne metody tam, gdzie wymagane jest obliczanie przecięć, wektorów normalnych i odległości. Jest szczególnie przydatny w kontekście dynamicznego rysowania wokseli, gdzie scena musi być ciągle aktualizowana bez znacznego spadku wydajności.

2.6.4 Wydajne rzadkie wokselowe drzewa ósemkowe

W niniejszej pracy autorzy prezentują kompaktową strukturę danych do przechowywania wokseli oraz wydajny algorytm rzutowania promienia (ang. *Ray Casting*) z wykorzystaniem tej struktury [18]. Rzadka struktura oparta na drzewach oktalnych (ang. *Sparse Voxel Octree, SVO*) hierarchicznie dzieli przestrzeń trójwymiarową na osiem równych części na każdym poziomie drzewa. Na najwyższym poziomie (ang. *Root*) reprezentowana jest cała przestrzeń, a każdy kolejny poziom dzieli przestrzeń na coraz mniejsze fragmenty. Rzadka struktura oznacza, że przechowywane są tylko te woksele, które zawierają rzeczywistą geometrię. Obszary bez geometrii nie są dalej dzielone ani przechowywane.

Każdy węzeł w drzewie zawiera deskryptory dzieci, które przechowują informacje o swoich podwęzłach. Są to maski bitowe (valid mask) wskazujące, które z ośmiu możliwych dzieci są obecne, a które są liśćmi. Struktura ta przechowuje również wskaźniki do dzieci.

W przypadku wokseli istotne są informacje o konturach, gdzie kontur to para równoległych płaszczyzn przecinających woksel. Dzięki temu możliwe jest lepsze odwzorowanie szczegółów geometrycznych bez konieczności przemierzania kolejnych poziomów drzewa. Pozwala to również na wstępną ocenę, czy promień przecina powierzchnię, bez dalszego dzielenia wokseli.

Autorzy przedstawiają także algorytm rzutu promienia (ang. *Ray Casting*), który przechodzi przez hierarchię wokseli korzystając ze stosu, wykonując operacje przejścia do dzieci (PUSH), następnie do rodzeństwa (ADVANCE) lub powrotu do przodka (POP) w zależności od tego, jak promień przecina woksele. Stos przechowuje przodków aktualnego woksela, wartości konturów oraz odległość od punktu początkowego promienia do punktów przecięcia.

Średnie zużycie pamięci na woksel jest bliskie teoretycznemu optimum wynoszącemu 5 bajtów na woksel, co czyni strukturę SVO (Sparse Voxel Octree) bardzo efektywną w zarządzaniu pamięcią. Dla większości scen wartość ta wynosi około 5-8 bajtów na woksel. W przypadku rzutowania promieni (ang. *Ray Casting*) z użyciem wokseli z konturami i optymalizacją wiązki, wydajność przewyższa tradycyjne rzutowanie promieni trójkątów, zwłaszcza w scenach o wysokiej złożoności.

2.6.5 Marsz promieni przez woksele z efektem paralaksy

Jest to raport projektowy, który opisuje technikę nazwaną „*Parallax Voxel Raymarcher*” [20], której zbliżoną wersję stosuje studio Tuxedo Labs w swojej grze AAA „*Teardown*”, a także popularny twórca internetowy Douglass. Technika ta grupuje woksele w większe bloki 8x8x8, a następnie renderuje je przy użyciu metody podobnej do mapowania paralaksy. Każdy piksel powierzchni otrzymuje kolor odpowiadający skorygowanemu perspektywicznie obrazowi 3D. Geometria jest rzutowana na tylne ściany sześcianu, a przemierzanie wokseli odbywa się przy użyciu algorytmu „Fast Voxel Traversal Algori-

thm" (patrz sekcja 2.6.1).

Autorzy twierdzą, że czas renderowania jest ograniczony przez liczbę odczytów pamięci. Dodatkowo zaimplementowali poziom szczegółowości (ang. *Level of Details*), który w przeciętnym przypadku nie ma znaczącego wpływu na wydajność, jednak jego wpływ staje się zauważalny w najgorszych przypadkach. Autorzy wspominają także o możliwości użycia pól odległości w maszerowaniu promienia, gdzie każdy woksel zawiera informacje o liczbie kroków, jakie promień może bezpiecznie wykonać. Niemniej, złożoność utrzymania informacji o polach odległości jest bardzo wysoka, dlatego uproszczonym rozwiązaniem może być przechowywanie informacji o tym, czy najbliżsi sąsiedzi są pustymi blokami, czy też nie.

Ostatnią strukturą przyspieszającą, o której wspominają autorzy, są mipmapy, stanowiące prostszą alternatywę dla pól odległości. Dla całej tekstury 3D ręcznie tworzą kopię z połową rozdzielczości na każdej osi, gdzie texel może być powietrzem tylko wtedy, gdy nie zawiera materiału. Pozwala to na przeskakiwanie wielu indeksów bez pomijania pełnych bloków.

2.6.6 Wydajny parametryczny algorytm przechodzenia przez drzewa ósemkowe

Drzewa ósemkowe (ang. *Octree*) są używane głównie do przyśpieszenia testów przecięcia promieni z obiektami oraz do przyśpieszenia procesu rysowania modeli wolumetrycznych za pomocą rzutowania promieni (ang. *Ray Casting*). Artykuł rozpoczyna się od krótkiego wprowadzenia, w którym omawiany jest podział na metody z góry na dół i z dołu do góry [30].

2.6.6.1 Metody z góry na dół

Metody z góry na dół rozpoczynają przeszukiwanie od korzenia (ang. *Root*), który obejmuje całą przestrzeń drzewa ósemkowego. Jako przykład podano tutaj między innymi algorytmy Agate'a, Cohena i Endla, które w swojej implementacji wykorzystują podejście rekurencyjne. Zaczynając od korzenia uzyskuje sie potomki przecięte przez promień, a proces jest powtarzany tak długo aż dotrze sie do liści (węzły terminalne).

2.6.6.2 Metody z dołu do góry

Ta metoda polega na rozpoczęciu przeszukiwania od pierwszego liścia (węzła terminalnego) przeciętego przez promień. Następnie przy użyciu procesu znajdowania sąsiadów przechodzą do kolejnych liści, które również mogą być przecięte przez promień.

2.6.6.3 Nowy algorytm parametryczny

W omawianym artykule autorzy przedstawili nowy algorytm z góry na dół oparty na parametrycznej reprezentacji promienia. Algorytm rozpoczyna się od zdefiniowania promienia w formie parametrycznej, dzięki czemu jego pozycja w przestrzeni jest określana przez zmienną t. Promień można zapisać w formie

$$P(t) = P_0 + t \cdot D \tag{2.1}$$

Gdzie P_0 jest punktem początkowym promienia, a D jest jego wektorem kierunkowym. Algorytm oblicza wartości t, w których promień przecina płaszczyzny dzielące woksel na mniejsze części. Istotnym aspektem algorytmu jest unikanie kosztownego procesu znajdowania sąsiadów poprzez dokładne obliczenie punktów przecięcia na podstawie parametrów promienia. Sam algorytm działa rekurencyjnie, rozpoczynając od korzenia i na każdym etapie oblicza wartości parametrów dla wszystkich potomków, określając, które z nich są przecięte przez promień.

Wyniki pokazały, że nowy algorytm znacząco przewyższa tradycyjne metody pod względem efektywności. Testy przeprowadzono na trzech różnych scenach, a przestrzen podzielono na drzewa ósemkowe o maksymalnej głębokości 5, 6, 7, i 8 poziomów. Metoda została porównana z metodami *SametNet*, *Samet*, *SametCorner*, oraz *Gargantini*. W każdej ze scen nowa metoda osiągała krótszy czas przetwarzania w porównaniu z innymi metodami wykazując znaczną poprawę wydajności. Dodatkowo przeprowadzono porównanie z metodą *Kelvina Sunga*. Ta sama scena była rysowana przy użyciu drzew oktalnych, oraz drzew binarnych. Wyniki pokazały, że drzewa ósemkowe z wykorzystaniem nowej metody jest bardziej efektywne w szczególności gdy drzewo jest pełne lub prawie pełne.

2.6.7 Porównanie implementacji wolumetrycznego rzutowania promieni z wykorzystaniem GPU: jednostka cieniująca fragmentów, jednostka cieniująca obliczeń, *OpenCL* i *CUDA*.

Praca autorstwa Francisco Sans i Rhadamés Carmona porusza temat porównania wydajności różnych implementacji rzutowania promieni (ang. *Ray Casting*) na GPU przy użyciu jednostki cieniującej fragmentów, jednostki cieniującej obliczeniowej (ang. *Compute Shader*), *OpenCL* i *CUDA* [31]. Na wstępie autorzy zwracają uwagę na to, że rzutowanie promieni jest obecnie standardem w renderowaniu wolumetrycznym ze względu na swoją równoległą naturę i jakość. Z racji, że metoda jest wysoce równoległa to implementacja tej metody na GPU staje się bardzo atrakcyjną opcją. Warto też zwrócić uwagę na dwie główne metody obliczania przecięcia promienia z wolumenem. Są to rasteryzacja i przecięcie promień-pudło. Mając już punkt wejścia i wyjścia promienia problem sprowadza się już tylko do próbkowania wolumenu i komponowania próbek. W celu im-

plementacji programów równoległych na GPU można skorzystać z interfejsów API takich jak *OpenCL* czy *CUDA*. Można też korzystać z potoku *OpenGL*, który jest zoptymalizowany pod kątem problemów graficznych, ale nie wszystkie jego etapu są programowalne. Szczęśliwie *OpenGL 4.5* pozwala na ogólne przetwarzanie przy użyciu jednostek cieniujących obliczeniowych i jego przewagą nad *CUDA* czy *OpenCL* jest to, że jest bezpośrednio zintegrowany z *OpenGL*.

Następnie autorzy robią przegląd literatury, oraz opisują technikę rzutowania promieni oraz dwie metody obliczania przecięcia promieni z wolumenem. Przedstawiają też szczegóły implementacyjne i sposób realizacji rzutowania promieni za pomocą każdego z czterech podejść: jednostki cieniującej fragmentów, jednostki cieniującej obliczeń, *CUDA* i *OpenCL*. Testy zostały przeprowadzone na trzech różnych zestawach danych z różnymi rozdzielczościami i funkcjami transferu, mierząc czas renderowania w milisekundach na klatkę.

Wyniki wskazują, że jednostka cieniująca obliczeń osiąga najwyższą wydajność pod względem czasu renderowania we wszystkich przypadkach, z czasem renderowania od 1.03x do 1.85x szybszym niż jednostka cieniująca fragmentów, od 1.5x do 7.1x szybszym niż *OpenCL* oraz od 1.05x do 1.3x szybszym niż *CUDA*. *CUDA* wykazuje drugą najlepszą wydajność, szczególnie dla większych zestawów danych (około 600 MB), osiągając czas renderowania od 1.1x do 5.3x szybszy niż *OpenCL*. Jednostka cieniująca fragmentów przewyższa OpenCL w większości przypadków, szczególnie dla mniejszych zestawów danych (mniej niż 40 MB). *OpenCL* wykazuje najniższą wydajność w większości przypadków, z wyjątkiem największych zestawów danych, gdzie jest do 1.4x szybszy od jednostki cieniującej fragmentów.

Dodatkowo, testy z oświetleniem rozproszonym wykazują, że obciążenie związane z obliczeniami oświetlenia jest mniejsze dla jednostki cieniującej obliczeń i *CUDA* niż dla jednostki cieniującej fragmentów oraz *OpenCL*. W przypadku jednostki cieniującej fragmentów obciążenie wzrasta od 1.3x do 2.8x, dla jednostki cieniującej obliczeń od 1.3x do 1.7x, dla *OpenCL* od 1.08x do 2.3x, a dla *CUDA* od 1.2x do 1.7x. Wyniki te wskazują również, że czas renderowania jest dłuższy dla większych zestawów danych, ponieważ promienie muszą przemieszczać się przez większe obszary objętości.

Testy porównujące dwie metody obliczania przecięcia promieni z wolumenem: rasteryzację i test przecięcia promień-pudło, wykazały, że metoda przecięcia promień-pudło jest bardziej wydajna dla jednostki cieniującej obliczeń oraz *OpenCL*, podczas gdy metoda rasteryzacji wykazuje lepszą wydajność dla *CUDA*. W przypadku jednostki cieniującej fragmentów wybór metody nie był konieczny, ponieważ jest ona zintegrowana z potokiem graficznym *OpenGL*.

Podsumowując wyniki, autorzy stwierdzają, że jednostka cieniująca obliczeń jest najbardziej wydajną metodą do implementacji rzutowania promieni na GPU. *CUDA* wykazuje dobrą wydajność w postaci średniej ilości klatek na sekundę dla większych zestawów

danych, podczas gdy jednostka cieniująca fragmentów jest bardziej efektywna dla mniejszych zestawów danych. *OpenCL* prezentuje najgorszą wydajność w większości przypadków, z wyjątkiem największych zestawów danych. Dodatkowo, wprowadzenie oświetlenia dyfuzyjnego powoduje zwiększenie czasu renderowania, przy czym jednostka cieniująca obliczeń i *CUDA* mają mniejszy narzut czasowy niż jednostka cieniująca fragmentów i *OpenCL*. Najlepsze wyniki osiągnięto, łącząc jednostkę cieniującą obliczeń z testem przecięcia promień-pudło.

2.6.8 Nowatorska rozszerzona mapa cegiełkowa dla śledzenia promieni w czasie rzeczywistym

Autor tej pracy rozwija funkcjonalność struktury danych map cegiełkowych (ang. *Brickmap*) z 2015 roku [45]. Celem pracy jest implementacja tej struktury w *API Vulkan* oraz rozszerzenie jej o dane materiałowe bazujące na fizyce, przy jednoczesnym zachowaniu jej użyteczności w aplikacjach czasu rzeczywistego [9]. W przedstawionej strukturze woksele są przechowywane w formie maski bitowej, a mapy ceiegłkowe zawierają również informacje o materiałach i kolorach. Autor przeprowadził testy wydajności na różnych kartach graficznych, wykazując, że nawet starsze GPU mogą efektywnie wykonywać śledzenie promieni, jeśli zastosowane są odpowiednie optymalizacje.

Praca rozpoczyna się od przedstawienia podstawowej wiedzy na temat rasteryzacji, wielokątów, śledzenia promieni, wokseli i struktur danych dla wokseli. Następnie wprowadzone jest pojęcie rozszerzonej mapy cegiełkowej (ang. *Extended Brickmap*), która opiera się na oryginalnej strukturze mapy cegiełkowej, ale dodaje kilka usprawnień. Kolory i materiały są przechowywane w sposób skompresowany, aby zmniejszyć zużycie pamięci, przy czym kompresja opiera się na metodzie DXT1. W pracy zastosowano algorytmy zarządzania pamięcią, które minimalizują dostęp do pamięci, stosując lokalność danych i sekwencyjny dostęp. Sama mapa cegiełkowa zorganizowana jest hierarchicznie, gdzie niższe poziomy zawierają bardziej szczegółowe informacje wokselowe (poziomy szczegółowości). Każdy poziom zawiera maskę bitową określającą obecność wokseli oraz indeksy do danych kolorów i materiałów. W implementacji użyto jednostek cieniujących obliczeniowych, a zmniejszenie opóźnienia związanego z dostępem do pamięci osiągnięto poprzez odpowiednie buforowanie i strumieniowanie danych.

Przeprowadzono test wydajności pięciu kart graficznych. GTX 1650M, jako najsłabsza z nich, osiągnęła średnio 74 klatki na sekundę przy podstawowych ustawieniach, ale nie radziła sobie z bardziej wymagającymi ustawieniami, uzyskując średni czas klatki 108 ms. GTX 1080ti wykazywała dobrą wydajność, osiągając poniżej 8 ms przy prostych ustawieniach i 43 ms przy najbardziej wymagających. RTX Titan i RTX 3090 uzyskały akceptowalną wydajność na wszystkich poziomach ustawień, przy czym na najwyższych ustawieniach średni czas klatki wyniósł 13,753 ms, co odpowiada 72 klatkom na sekundę.

RX 6800XT osiągnęła najlepsze wyniki, z najgorszym czasem klatki wynoszącym 11 ms, co przekłada się na 91 klatek na sekundę. Kompresja DXT1 zredukowała zużycie pamięci o około 40%. Starsze GPU efektywnie korzystały z rozszerzonej mapy cegiełkowej, a nowsze uzyskiwały jeszcze lepsze wyniki.

2.7 Przegląd istniejących rozwiązań

2.7.1 Gra *Teardown*

Gra „*Teardown*" to pionierska gra piaskownicowa, która wykorzystuje woksele, oferując graczom ogromny poziom destrukcji i interakcji z otoczeniem. Łączy realistyczną fizykę z zaawansowanymi technikami renderowania oraz dynamicznym oświetleniem.

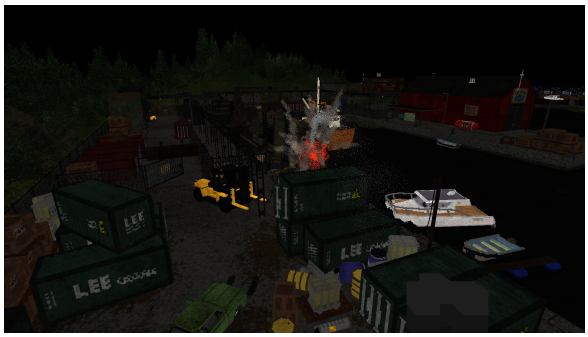
Jednym z najbardziej imponujących aspektów gry „*Teardown*" jest mechanika destrukcji, stanowiąca znak rozpoznawczy tej gry. Woksele są tu idealnym rozwiązaniem, umożliwiając niszczenie, modyfikowanie i interakcję z otoczeniem. Niszczone obiekty rozpadają się na mniejsze fragmenty, które podlegają prawom fizyki, takim jak grawitacja, siła tarcia czy impakt. Obiekty wrzucane do wody generują fale i załamania oraz gaszą ogień, a każdy materiał w grze ma swoje unikalne właściwości: drewno można spalić, a szkło tłucze się na kawałki.

W systemie fizycznym w celu zredukowania liczby porównań i operacji obliczeniowych, woksele klasyfikowane są jako krawędzie, narożniki lub wewnętrzne części. Wewnętrzne części nie muszą być sprawdzane pod kątem kolizji z zewnętrznymi obiektami, ponieważ są całkowicie otoczone innymi wokselami. Krawędzie i narożniki są sprawdzane tylko tam, gdzie faktycznie mogą wystąpić kolizje.

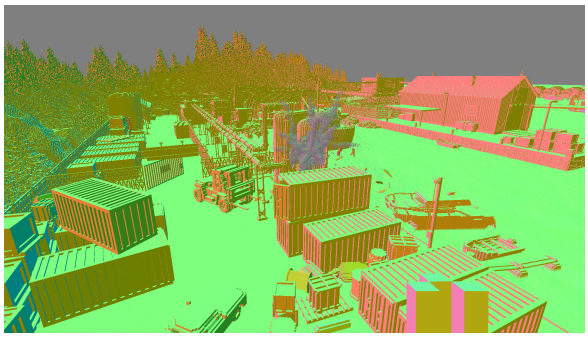
Gra „*Teardown*" charakteryzuje się również wysoce realistyczną grafiką dzięki zastosowaniu mikrowokseli. Podczas gdy woksele w innych grach mają zazwyczaj wielkość około jednego metra, w „*Teardown*" wynoszą one około 10 centymetrów. Takie podejście pozwala na stworzenie szczegółowej mapy z precyzyjną geometrią. Silnik graficzny „*Teardown*" opiera się na implementacji śledzenia promieni w pełni programowo, bez wykorzystania technologii RTX. Podstawą procesu renderowania jest wykorzystanie tzw. G-buffer, który przechowuje informacje o głębokości, kolorze i normalnych [14] [33] (patrz Tabela 2.1).

	R	G	B	A	Rozmiar
RT0 - Albedo	Czerwony	Zielony	Niebieski		16-bitowy uint
RT1 - Normalna	Normalna X	Normalna Y	Normalna Z		8-bitowy int
RT2 - Materiał	Refleksyjność	Chropowatość	Metaliczność	Emisyjność	16-bitowy float
RT3 - Ruch	Ruch X	Ruch Y	Maska Wody		16-bitowy float
RT4 - Głębia	Liniowe Z				16-bitowy uint
Z-Bufor	Odwrócone Z				32-bitowy float

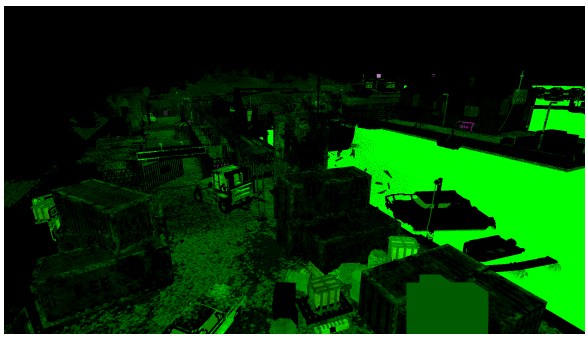
Tabela 2.1: Odroczony G-Buffer. Źródło: `https://acko.net`.



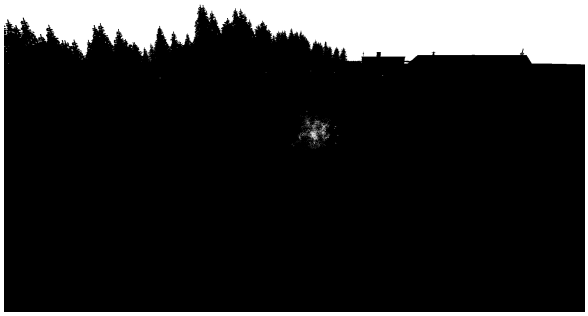
Rysunek 2.13: Albedo (RT0).
Źródło: `https://acko.net`.



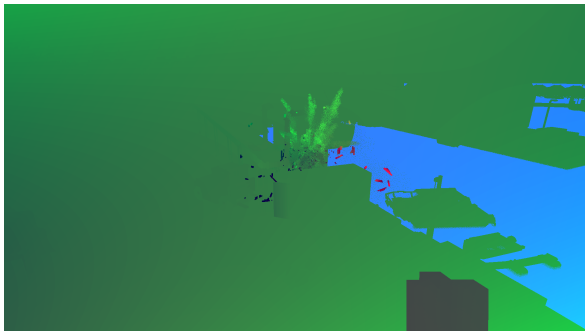
Rysunek 2.14: Normalna (RT1).
Źródło: `https://acko.net`.



Rysunek 2.15: Materiał (RT2 RGB).
Źródło: `https://acko.net`.



Rysunek 2.16: Emisyjność (RT2 Alpha).
Źródło: `https://acko.net`.

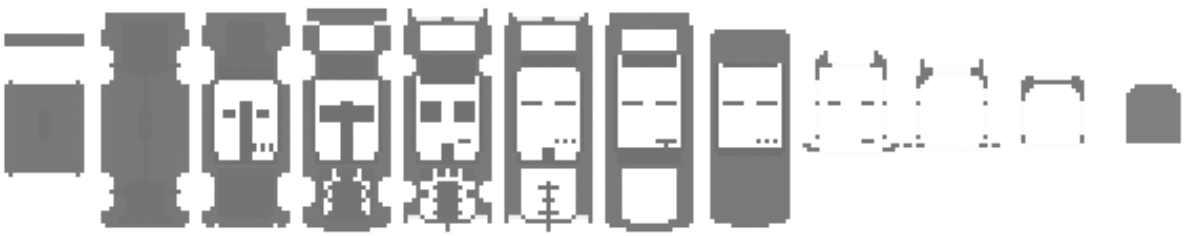


Rysunek 2.17: Ruch + Woda (RT3).
Źródło: `https://acko.net`.

Rysunek 2.18: Liniowa głębia (RT4).
Źródło: `https://acko.net`.

Gra korzysta z renderowania odroczonego (ang. *Deferred Rendering*), co umożliwia oddzielenie procesu renderowania geometrii od oświetlenia, minimalizując liczbę obliczeń poprzez jednorazowe renderowanie geometrii i późniejsze dodanie efektów świetlnych, takich jak globalna iluminacja. Główna technika renderowania wykorzystywana w „*Teardown*" to śledzenie promieni. Jest ono używane do globalnej iluminacji, odbić i cieni w czasie rzeczywistym. Oświetlenie realizowane jest poprzez kombinację sferycznego oświetlenia imitującego naturalne światło nieba oraz lokalnych źródeł światła. Zacienienie środowiskowe (ang. *Ambient occlusion*) jest realizowane poprzez wysyłanie promieni o długości do 32 metrów w różnych kierunkach, symulując rozpraszanie się światła w scenie. Promienie te sprawdzają, czy w danym kierunku znajduje się przeszkoda (np. ściana) blokująca światło, co pozwala uzyskać realistyczne cienie i przyciemnienia w miejscach, gdzie światło jest blokowane. Aby zredukować szumy generowane przez śledzenie promieni i inne techniki, „*Teardown*" wykorzystuje algorytmy odszumiania, takie jak Temporal Anti-aliasing (*TAA*), który wykorzystuje bufor prędkości (patrz rysunek 2.17), aby wygładzić obraz na przestrzeni czasu.

Każde zawołanie rysujące renderuje 12 trójkątów tworzących obwiednie wokół trójwymiarowego obiektu (pudełko). Obiekty zbudowane są z trójwymiarowych tekstur tworzących objętość, gdzie na każdy woksel przeznaczony jest jeden bajt. Przykładem trójwymiarowej tekstury jest zielony samochód przedstawiony na rysunku 2.19.



Rysunek 2.19: Trójwymiarowa tekstura tworząca obiekt w świecie gry.
Źródło: `https://acko.net`.

2.7.2 Projekt Voxplat

Projekt „Voxplat" [16] wykorzystuje algorytm przecięcia promienia z pudełkiem (ang. *A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering*) jako część swojej hybrydowej metody rysowania wokseli. Woksele znajdujące się blisko kamery są rysowane jako siatki (ang. *Meshes*), podczas gdy dalsze woksele są rysowane z użyciem techniki rozpraszania (ang. *Splatting*), w której każdy woksel jest reprezentowany jako kwadrat lub punkt. Takie podejście znacząco redukuje liczbę wierzchołków do przetworzenia. Algorytm przecięcia promienia z pudełkiem jest zaimplementowany w jednostce cieniującej fragmentów i służy do ustalania, czy promień przecina kwadrat reprezentujący woksel.

2.7.3 Gra *Minecraft*

Gra „*Minecraft*" jest uznawana za jedną z najbardziej rozpoznawalnych gier wszech czasów, co przyczyniło się do znaczącej popularyzacji wokseli w grafice komputerowej [41]. Oryginalna wersja gry *Minecraft* została stworzona przy użyciu technologii *OpenGL* i języka *Java* [37]. Gra wykorzystuje proces triangulacji do konwersji wokseli na siatki trójkątów. *Minecraft* stosuje technikę eliminacji niewidocznych ścian, co oznacza, że renderowane są jedynie te ściany wokseli, które są widoczne dla gracza, pomijając ściany wewnętrzne, niewidoczne z perspektywy użytkownika.

Pomimo zastosowania tych technik, wiele rozwiązań technologicznych w *Minecraft* pozostawało stosunkowo prostych. Było to spowodowane początkowym charakterem projektu, który nie był nastawiony na osiągnięcie globalnego sukcesu. Na początku *Minecraft* był tworzony przez jedną osobę, wspieraną przez kilku wczesnych współpracowników. Dopiero pod koniec 2010 roku zespół rozwijający grę zwiększył się do 12 pracowników [10]. Głównym celem zespołu było projektowanie rozgrywki, a zastosowane rozwiązania technologiczne były wybierane ze względu na ich prostotę implementacji. W rezultacie, mimo ogromnej rozpoznawalności gry, jej aspekty techniczne nie są uważane za wyjątkowo zaawansowane.

2.8 Wnioski z analizy tematu

Analiza różnorodnych technik optymalizacji wokseli, znajdujących zastosowanie w tworzeniu modeli światów wokselowych, ujawnia brak uniwersalnego rozwiązania. Wybór odpowiednich metod powinien być uzależniony od specyficznych celów projektowych. Przegląd literatury wskazuje na szereg efektywnych technik optymalizacji wokseli. Niektóre techniki są stosowane wyłącznie do triangulacji, podczas gdy inne znajdują zastosowanie jedynie w śledzeniu promieni. Przykładem jest algorytm szybkiego przechodzenia przez woksele Johna Amanatidesa i Andrew Woo, który znacznie redukuje liczbę obiektów

przeciętych przez promień, zwiększając wydajność śledzenia promieni. Redukcja liczby potrzebnych iteracji promieni może być osiągnięta przez zastosowanie hierarchii objętości ograniczających, drzew oktalnych, drzew k-wymiarowych, lub struktur map cegiełkowych i siatek cegiełkowych, które oferują podział przestrzeni umożliwiający efektywne pominięcie pustych przestrzeni. Dodatkowo, badania nad strukturami map cegiełkowych i siatek cegiełkowych koncentrują się również na efektywnym strumieniowaniu danych oraz dynamicznej edycji scen. Te struktury mogą być rozwijane o poziomy szczegółowości, materiały, kompresję, bądź techniki odrzucania zasłoniętych obiektów. W przypadku mniej interaktywnych, a bardziej statycznych światów, można zastosować rzadkie drzewa ósemkowe, które doskonale sprawdzają się przy szczegółowych, statycznych scenach, zarówno pod względem zajętości pamięci, jak i wydajności. W kontekście scen o wysoce dynamicznym charakterze, gdzie scena musi być ciągle aktualizowana, bardzo przydatny jest algorytm przecięcia promienia z pudełkiem. Projekt „*Parallax Voxel Raymarcher*" proponuje technikę zbliżoną do mapowania paralaksy, rzutującą geometrie na tylne ściany sześcianu, która może być hybrydą triangulacji i maszerowania promieni.

Alternatywnie, można zastosować maszerowanie promieni, gdzie woksele zawierają informacje o najbliższym dystansie do niepustego woksela. Takie dane byłyby zawarte jedynie w wokselach będących na widocznej części modelu wokselowego, co wymagałoby odpowiedniej aktualizacji przy każdej zmianie, co jest trudne do osiągnięcia.

W kontekście triangulacji można zastosować techniki minimalizujące liczbę wynikowych wierzchołków tworzących siatkę. Pomocne może być usuwanie niewidocznych powierzchni, otoczonych przez inne woksele, bądź łączenie kilku ścian w jedną większą.

Dostępnych technik jest wiele, a odpowiednia kombinacja zależy od pożądanego efektu. W przypadku światów wokselowych, które z reguły powinny być interaktywne i charakteryzować się wysoką wydajnością renderowania, uzasadnione wydaje się użycie struktur mapy cegiełkowej i siatki cegiełkowej. Ciekawą opcją jest również wokselowy marsz promienia z paralaksą, łączący marsz promieni z rasteryzującą naturą kart graficznych. Jednak odpowiedź na to pytanie nie jest jednoznaczna i pozostaje też wiele hybrydowych rozwiązań czego przykładem jest projekt „Voxplat", który renderuje woksele blisko kamery jako siatki, a te dalsze z użyciem techniki rozpryskiwania (ang. *Splatting*). Wciąż pozostaje wiele pytań i niewiadomych. Niektóre techniki mogą być bardziej odpowiednie dla dużych wokseli (około metra), jak triangulacja, podczas gdy przy mikrowokselach (około 10 centymetrów) liczba wierzchołków wzrasta na tyle, że konieczne jest zastosowanie innych technik, takich jak odpowiednio zoptymalizowane śledzenie promieni.

Rozdział 3

Przedmiot pracy

W niniejszym rozdziale przedstawiono przedmiot pracy, koncentrując się na opisie zastosowanych narzędzi i technik.

3.1 Opis wykorzystanych narzędzi

3.1.1 Język *C++20*

C++ jest językiem niskopoziomowym, co pozwala na bezpośrednie zarządzanie zasobami sprzętowymi, jednocześnie oferując wysoki poziom abstrakcji. Doskonale integruje się z *OpenGL* i ma szeroką dostępność bibliotek wspierających, takich jak *GLM* (*OpenGL Mathematics*). Społeczność i zasoby edukacyjne wokół *C++* i *OpenGL* są bardzo rozbudowane. Przez wiele lat, *C++* był, obok języka *C*, standardem w połączeniu z *OpenGL*.

3.1.2 Interfejs *OpenGL*

OpenGL (*Open Graphics Library*) [28] jest specyfikacją definiującą wieloplatformowy interfejs API do renderowania grafiki, umożliwiającą bezpośrednie zarządzanie sprzętem graficznym. Dostępna jest obszerna ilość źródeł edukacyjnych, w tym zajęcia oferowane na wielu uczelniach. *OpenGL* jest ceniony za swoją prostotę i niższy próg wejścia w porównaniu z innymi API. Pomimo długiej historii i zachowanej kompatybilności wstecznej, od wersji *OpenGL 3.0* wprowadzono tzw. *Modern OpenGL*, który zastąpił stały potok renderowania programowalnym potokiem. W tym projekcie minimalną przewidywaną wersją jest 3.3, jednak nie wyklucza się użycia nowszych rozszerzeń, takich jak *ARB_sparse_texture* z wersji 4.4.

3.1.3 Biblioteka *SFML*

SFML (*Simple and Fast Multimedia Library*) [43] jest biblioteką dostarczającą prosty w użyciu interfejs do tworzenia aplikacji multimedialnych. Składa się ona z pięciu

modułów:

- **Audio** – obsługuje dźwięki, streaming (muzyka lub niestandardowe źródła), oraz przestrzenne rozmieszczenie dźwięku.
- **Graphics** – moduł graficzny 2D, obejmujący obsługę obrazów, tekstów oraz kształty.
- **Network** – umożliwia komunikację opartą na gniazdach i dostarcza narzędzia oraz protokoły sieciowe wyższego poziomu (HTTP, FTP).
- **System** – podstawowy moduł *SFML*, definiujący różne narzędzia, takie jak zegar, czas oraz wektory.
- **Window** – moduł do zarządzania oknami opartymi na *OpenGL*, zapewniający abstrakcję zdarzeń oraz obsługę wejścia.

W ramach niniejszej pracy planowane jest wykorzystanie jedynie dwóch spośród dostępnych modułów: **System** oraz **Window**. Moduł graficzny nie zostanie użyty, ponieważ zostanie stworzony własnoręcznie. Biblioteka *SFML* będzie wykorzystywana wyłącznie do tworzenia okna oraz obsługi wejścia.

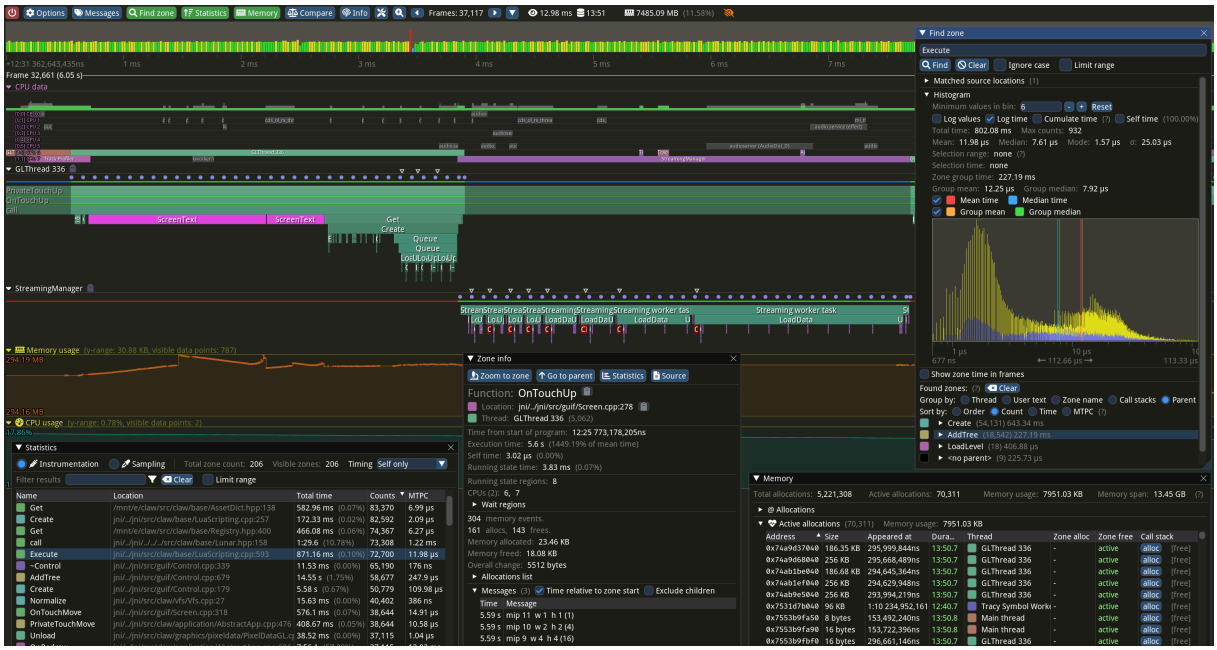
3.1.4 Narzędzie *CMake*

CMake jest narzędziem do zarządzania procesem kompilacji oprogramowania, charakteryzującym się wieloplatformowością oraz niezależnością od kompilatora [39]. W niniejszej pracy *CMake* jest wykorzystywany do pobierania wszystkich niezbędnych zależności, ich kompilacji oraz łączenia z aplikacją.

Takie podejście umożliwia zastosowanie ciągłej integracji, co pozwala upewnić się, że każda zmiana w projekcie nie zakłóca jego kompilowalności i działania na różnych maszynach.

3.1.5 Narzędzie *Tracy Profiler*

Tracy Profiler to narzędzie do profilowania wydajności aplikacji w czasie rzeczywistym [2]. Umożliwia śledzenie i analizę zachowania kodu oraz identyfikację wąskich gardeł. Integracja narzędzia z kodem odbywa się za pomocą prostych makr. *Tracy Profiler* umożliwia profilowanie zarówno procesora (CPU), jak i karty graficznej (GPU), wspierając najpopularniejsze interfejsy programowania aplikacji, takie jak *OpenGL*, *Vulkan* oraz *Direct3D*.



Rysunek 3.1: Zrzut ekranu z narzędzia Tracy
Źródło: `https://github.com/wolfpld/tracy`.

W niniejszej pracy narzędzie to będzie wykorzystywane w części badawczej do analizy wąskich gardeł różnych rozwiązań. Zebrane statystyki pozwolą na ocenę ogólnej wydajności tych rozwiązań.

3.1.6 Narzędzie *Google Benchmark*

Google Benchmark jest narzędziem przeznaczonym do pomiaru wydajności i testowania kodu [7]. Umożliwia dokładne mierzenie czasu wykonywania fragmentów kodu, definiując testy wydajnościowe za pomocą prostego interfejsu programistycznego. Posiada również zaawansowane funkcje, takie jak pomiar złożoności algorytmicznej.

W tej pracy *Google Benchmark* będzie używany, podobnie jak *Tracy Profiler*, do analizy wąskich gardeł algorytmów oraz zbierania informacji o wydajności badanych rozwiązań.

3.1.7 Inne biblioteki

- **Biblioteka *stb*** [32] – wszechstronna biblioteka służąca do łatwego ładowania, zapisywania i manipulowania obrazami. Zapewnia proste w użyciu funkcje umożliwiające pracę z różnymi formatami graficznymi, takimi jak PNG, JPEG, BMP i inne.
- **Biblioteka *result*** [24] – lekka i wydajna alternatywa dla obsługi błędów w stylu Rust, oferująca bezpieczne i czytelne zarządzanie błędami poprzez użycie typów Result i Option.

- **Biblioteka *glew*** [26] – zapewnia dynamiczne mechanizmy run-time do określania, które rozszerzenia *OpenGL* są obsługiwane na platformie docelowej.
- **Biblioteka *minitrace*** [8] – narzędzie do precyzyjnego pomiaru czasu wykonywania fragmentów kodu, umożliwiające szczegółową analizę wydajności aplikacji. Pozwala na identyfikację wąskich gardeł i optymalizację kodu poprzez śledzenie operacji i zdarzeń.
- **Biblioteka *imgui*** [27] – wszechstronna biblioteka do tworzenia interfejsów użytkownika w czasie rzeczywistym. Umożliwia szybkie i intuicyjne tworzenie dynamicznych i interaktywnych elementów UI, takich jak przyciski, okna, paski postępu, wykresy i inne.
- **Biblioteka *fastnoiselite*** [13] – biblioteka do generowania szumów proceduralnych, wspierająca różnorodne techniki szumów, takie jak Perlin, Simplex, Cellular i inne.
- **Biblioteka *yaml-cpp*** [12] – biblioteka umożliwiająca parsowanie i generowanie plików *YAML* w języku *C++*. Zapewnia prosty i elastyczny sposób na zarządzanie danymi konfiguracyjnymi w formacie *YAML*.
- **Biblioteka *spdlog*** [6] – szybka i wydajna biblioteka do logowania, oferująca różnorodne funkcje do zarządzania logami w aplikacjach. Umożliwia łatwe tworzenie, formatowanie i rotację logów, wspierając wielowątkowość i różne cele logowania, takie jak pliki, konsola i inne.

3.1.8 Autorskie rozwiązania

Na potrzeby przeprowadzenia badań opracowano silnik gry napisany w języku *C++20* z wykorzystaniem *OpenGL API*. Projekt zawiera odpowiednie mechanizmy otaczające wywołania do *OpenGL*, aby zapewnić przestrzeganie wzorca projektowego „pozyskiwanie zasobów jest inicjalizacją” (*RAII*). Dodatkowo implementuje stos stanów, który umożliwia przełączanie scen wewnątrz aplikacji.

3.2 Opis wykorzystanych technik

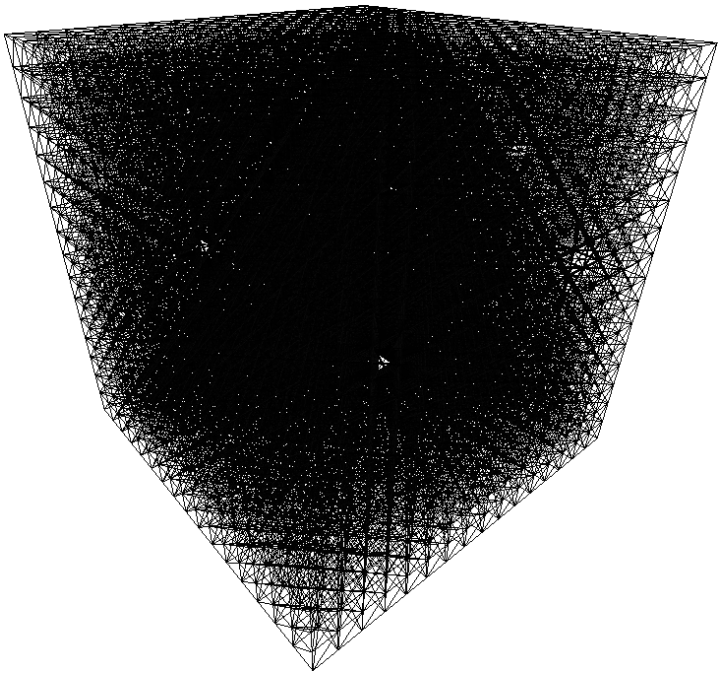
Badania zostaną przeprowadzone przy użyciu dwóch różnych typów technik renderowania obrazów: technik opartych na triangulacji oraz technik śledzenia promieni. Analizie poddane zostaną zarówno algorytmy oparte na triangulacji, jak i te oparte na śledzeniu promieni. Biorąc pod uwagę, że wybór między tymi dwiema technikami nie jest oczywisty, warto porównać różnice w ich wydajności.

3.2.1 Triangulacja

W przypadku triangulacji można wyróżnić różne metody budowania siatki z reprezentacji wokselowejale [25]. Ważne są zarówno prędkość generowania siatki, jak i wynikowa liczba wierzchołków. Istotne jest zminimalizowanie liczby wierzchołków, jednak należy pamiętać, że świat zbudowany z wokseli cechuje się wysoką interaktywnością. Każda ingerencja w teren wymaga przebudowy siatki, co oznacza, że czas budowy musi być maksymalnie skrócony. Długie oczekiwanie na aktualizację świata gry w odpowiedzi na modyfikacje terenu przez gracza jest nieakceptowalne. W związku z tym konieczny może być kompromis pomiędzy jakością a wydajnością. Możliwe jest również zastosowanie podejścia hybrydowego, w którym początkowo generowana jest siatka za pomocą najszybszej metody, a następnie stopniowo zastępowana siatką o mniejszej liczbie wierzchołków, wymagającą dłuższego czasu budowy. Takie rozwiązanie jednak wiąże się z koniecznością wykonania pracy dwukrotnie.

3.2.1.1 Metoda naiwna

Metoda naiwna polega na generowaniu siatki dla każdego woksela osobno. Oznacza to iterowanie po każdym bloku i generowanie po dwa trójkąty na ścianę bloku. W takim wypadku każdy blok składa się z 36 trójkątów. Dla bryły (ang. *Chunk*) o rozmiarach 16x16x16 pełnych wokseli jest to $16 \cdot 16 \cdot 16 \cdot 36 = 147456$ trójkątów. Wynikową siatkę takiej metody można zobaczyć na rysunku 3.2.

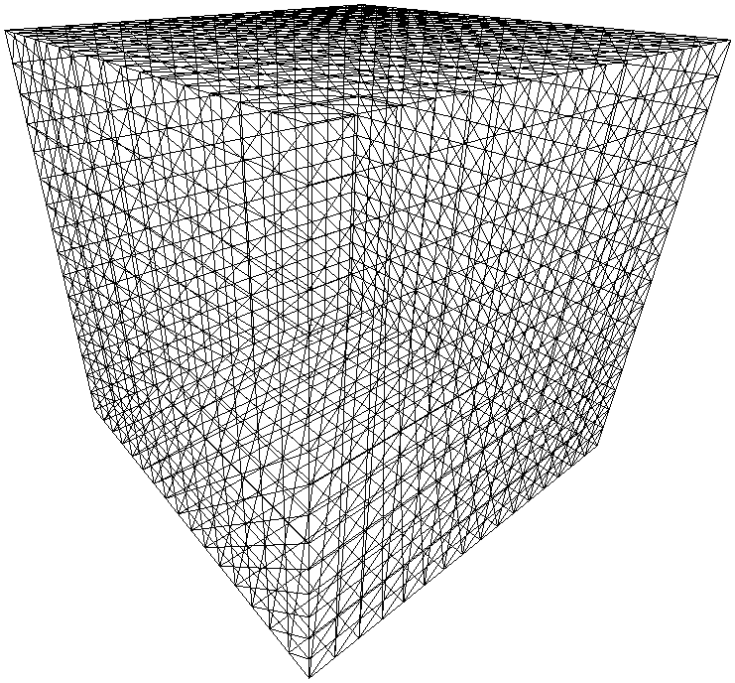


Rysunek 3.2: Siatka wynikowa metody naiwnej.

Metoda ta jest niepraktyczna, w związku z czym w tym badaniu zostanie wykorzystana jedynie jako baza wynikowa, poniżej której nie powinien spaść żaden algorytm.

3.2.1.2 Ukrywanie powierzchni

Bardzo prostym ulepszeniem poprzedniej metody jest usunięcie niewidocznych ścian, czyli wewnętrznych ścian przykrytych przez inne woksele. Wymaga to sprawdzenia sąsiada każdej ściany bloku w celu określenia, czy jest to blok nieprzezroczysty. W takim przypadku generowanie ściany bloku jest zbędne. Dla bryły o rozmiarze 16x16x16 pełnych wokseli jest to $16 \cdot 16 \cdot 6 \cdot 2 = 3072$ trójkątów. Stanowi to zaledwie 2.08% poprzedniej liczby trójkątów. Wynikową siatkę takiej metody można zobaczyć na rysunku 3.3.



Rysunek 3.3: Siatka wynikowa metody ukrywania powierzchni.

Wysokopoziomowy pseudokod tego algorytmu można zobaczyć w fragmencie kodu 1. Oczywiście warto pamiętać o kilku przypadkach brzegowych, których ten pseudokod nie pokaże, a mogą być zawarte w chociażby implementacji sprawdzania sąsiadów.

```
1  funkcja przygotujSiatkę(bryłaWokseli):
2      dla każdej pozycji i bloku w bryłaWokseli:
3          jeśli blok jest przezroczysty:
4              kontynuuj
5          utwórzSiatkęBloku(pozycja, blok, bryłaWokseli)
6
7  funkcja utwórzSiatkęBloku(pozycja, blok, bryłaWokseli):
8      dla każdej ściany w zakresie(0, 6):
9          jeśli ścianaBlokuMaPrzezroczystegoSąsiada(ściana, pozycja, bryłaWokseli):
10             dodajCzworokątDoSiatki(ściana, blok, pozycja)
```

Fragment kodu 1: Wysokopoziomowy pseudokod algorytmu ukrywania powierzchni,

3.2.1.3 Ukrywanie powierzchni z wsparciem GPU

Kolejnym ulepszeniem poprzedniej metody może być przeniesienie części algorytmu do potoku graficznego. Zamiast tworzyć wszystkie cztery narożniki czworokąta po stronie CPU, można wykorzystać jednostkę cieniującą geometrii, przekazując do potoku graficznego tylko środek czworokąta. Choć główna część algorytmu nie ulega zmianie (patrz fragment kodu 2), tak istotnie zmienia się już sama implementacja kodu odpowiedzialnego za tworzenie poszczególnych punktów siatki (patrz fragment kodu 3).

```
funkcja przygotujSiatkę(bryłaWokseli):
    dla każdej pozycji i bloku w bryłaWokseli:
        jeśli blok jest przezroczysty:
            kontynuuj
        utwórzSiatkęBloku(pozycja, blok, bryłaWokseli)

funkcja utwórzSiatkęBloku(pozycja, blok, bryłaWokseli):
    dla każdej ściany w zakresie(0, 6):
        jeśli ścianaBlokuMaPrzezroczystegoSąsiada(ściana, pozycja, bryłaWokseli):
            dodajPunktDoSiatki(ściana, blok, pozycja)
```

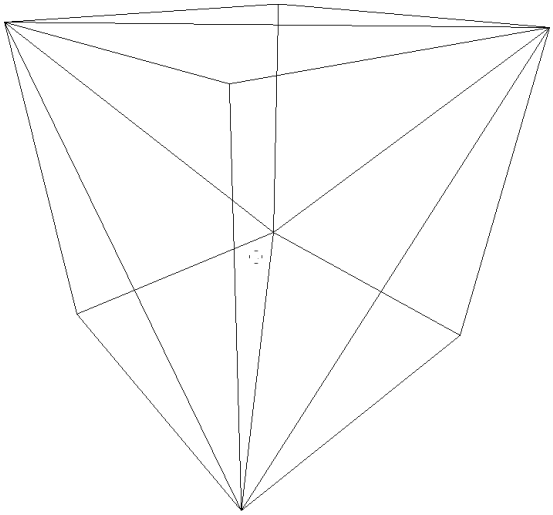
Fragment kodu 2: Wysokopoziomowy pseudokod algorytmu ukrywania powierzchni,

```
funkcja dodajPunktDoSiatki(ścianaBloku, identyfikatorTekstury, pozycjaBloku):
    oblicz pozycję wierzchołka
    utwórz nowy wierzchołek
    ustaw pozycję, identyfikator tekstury, stronę
    dodaj wierzchołek do siatki

funkcja dodajCzworokątDoSiatki(ścianaBloku, identyfikatorTekstury, pozycjaBloku):
    zainicjuj współrzędne tekstury dla czworokąta
    dla każdego z czterech wierzchołków w czworokącie:
        oblicz pozycję wierzchołka
        utwórz nowy wierzchołek
        ustaw pozycję, współrzędne tekstury, identyfikator tekstury
        dodaj wierzchołek do siatki
    zaktualizuj indeksy siatki dla czworokąta
```

Fragment kodu 3: Wysokopoziomowy pseudokod algorytmu ukrywania powierzchni,

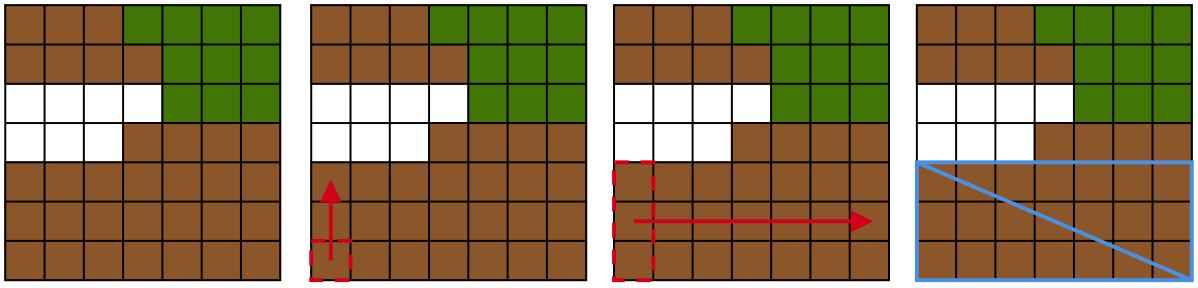
3.2.1.4 Zachłanne siatkowanie

Aby dalej zmniejszyć liczbę wierzchołków, można połączyć sąsiadujące ściany bloków wokselowych tego samego typu w większe regiony. Dzięki temu można zredukować całkowity rozmiar geometrii. Dla bryły o rozmiarach 16x16x16 pełnych wokseli tego samego typu jest to $2 \cdot 6 = 12$ trójkątów. Stanowi to zaledwie 0.008% trójkątów metody naiwnej i 0.39% trójkątów metody z ukrywaniem powierzchni. Wynikową siatkę takiej metody można zobaczyć na rysunku 3.4.

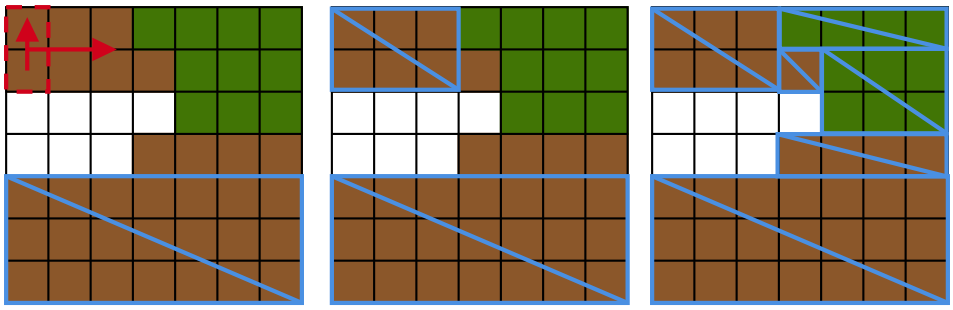


Rysunek 3.4: Siatka wynikowa metody zachłannego siatkowania.

Algorytm znacznie łatwiej przedstawić w formie dwuwymiarowej dla przekroju bryły wokseli, gdzie akurat znajdują się widoczne powierzchnie bryły, a następnie rozszerzyć go do trójwymiaru, gdzie w jednym momencie przetwarzane są dwie osie. Na rysunku 3.5 widać, jak zaczynając od lewego dolnego woksela, siatka rozszerza się w pionie, aż do napotkania pustego woksela lub woksela innego typu. Następnie algorytm próbuje rozszerzyć się w poziomie. Wynikiem tego procesu jest pierwsza stworzona siatka. Każdy woksel tej siatki powinien zostać oznaczony jako przetworzony, aby nie był ponownie brany pod uwagę w dalszym procesie. Następnie brane są kolejne nieprzetworzone woksele, które postępują zgodnie z tym samym schematem (patrz rysunek 3.6).



Rysunek 3.5: Wyznaczenie pojedyńczej ściany w algorytmie zachłannego siatkowania.



Rysunek 3.6: Wyznaczenie pozostałych ścian przekroju bryły wokseli.

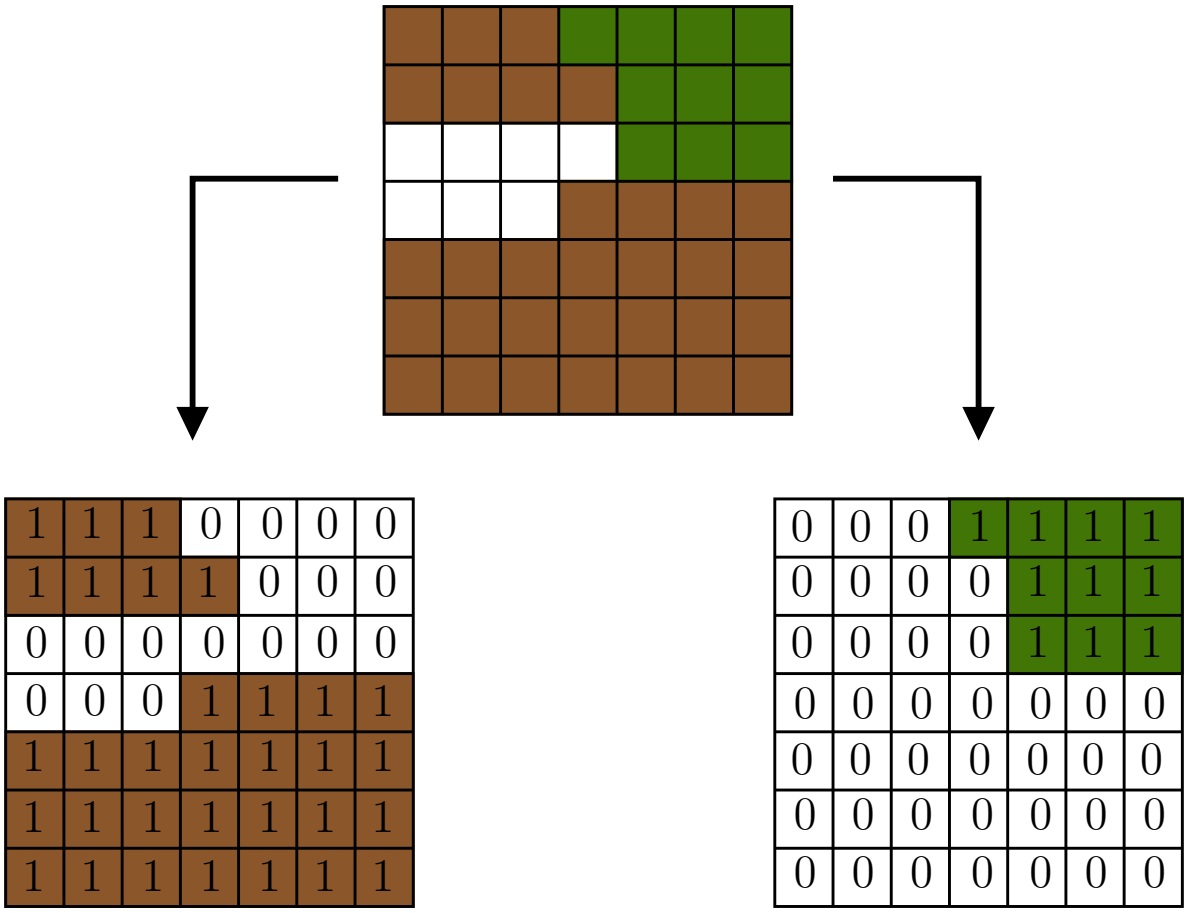
Oczywiście, podobnie jak poprzednia metoda, zachłanne siatkowanie wymaga sprawdzania sąsiadów pod kątem przeźroczystości, aby nie tworzyć zakrytych ścian. Pseudokod wysokiego poziomu opisujący ten algorytm widoczny jest w fragmencie kodu 4. Ze względu na złożoność implementacyjną, jest to bardzo ogólny zarys jego działania, bez szczegółów implementacyjnych.

```
funkcja przygotujSiatkę(bryłaWokseli):
    dla każdej ściany bloku:
        pobierz kierunki skanowania dla ściany
        dla każdej pozycji i bloku w bryłaWokseli:
            jeśli pozycja jest już przetworzona:
                kontynuuj
            jeśli blok jest przezroczysty lub nie ma przezroczystego sąsiada:
                oznacz pozycję jako przetworzoną
                kontynuuj
            zainicjuj region na podstawie pozycji, ściany i bloku
            pobierz identyfikator tekstury dla tej ściany bloku

            dopóki możliwe jest scalanie pionowe:
                oznacz pionowe bloki jako przetworzone
                zwiększ wysokość regionu

            dopóki możliwe jest scalanie poziome:
                jeśli nie można scalić całego wiersza:
                        przerwij
                zwiększ szerokość regionu
                oznacz poziome bloki jako przetworzone

            ustaw współrzędne tekstury dla regionu
            utwórz siatkę dla regionu
            oznacz pozycję jako przetworzoną

funkcja czyMożnaScalić(przetworzoneŚciany, pozycja, identyfikatorTekstury, ściana):
    jeśli pozycja jest poza zasięgiem lub jest już przetworzona
                                    lub jest przezroczystym blokiem:
        zwróć Fałsz
    jeśli blok ma inną teksturę lub nie ma przezroczystego sąsiada:
        zwróć Fałsz
    zwróć Prawda
```

Fragment kodu 4: Wysokopoziomowy pseudokod algorytmu zachłannego siatkowania.

3.2.1.5 Binarne zachłanne siatkowanie

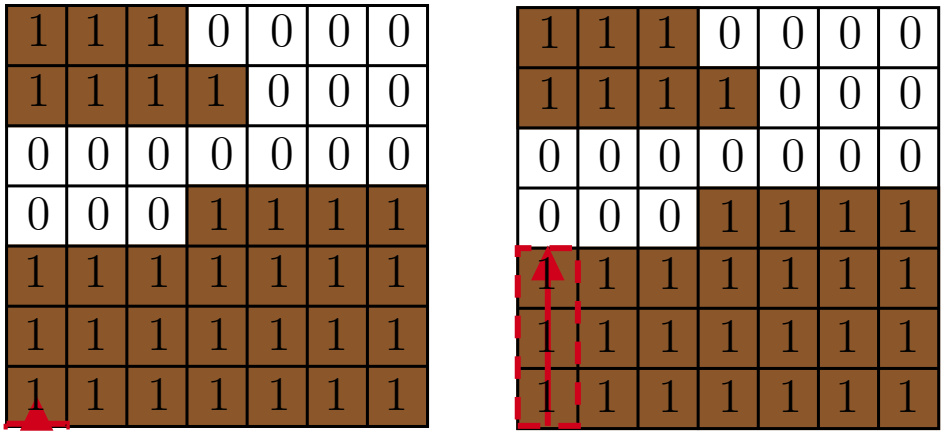
Poprzednia wersja algorytmu zachłannego siatkowania charakteryzuje się licznymi krokami, które znacząco wydłużają czas jego działania. Iteracje wykonywane są dla każdego woksela, przy czym każdy woksel jest sprawdzany pod kątem swojego typu i typu sąsiadów. Niestety, zapytania do pamięci są kosztowne, szczególnie z uwagi na fakt, iż występują one na każdej iteracji algorytmu. Problem ten można uprościć, traktując go jako problem binarny [34] [35], [5], poprzez utworzenie tablic bitowych w liczbie odpowiadającej ilości typów bloków (patrz rysunek 3.7).



Rysunek 3.7: Podział pojedynczego przekroju bryły na osobne tablice binarne w ilości równej typom wokseli.

Przekrój bryły wokseli przedstawiony na rysunku 3.7 ma wymiary 7x7. Każdy podział tablicy można zatem zapisać przy pomocy 49 bitów. Oznacza to, że jedna tablica na systemie 64-bitowym mieści się w zaledwie jednej zmiennej typu „unsigned long”, co zajmuje łącznie tylko 8 bajtów. Dla bryły wokseli o wymiarach 16x16x16 taki przekrój wymagałby 256 bitów, czyli zaledwie 32 bajty.

Dla każdej uzyskanej tablicy przeprowadzane są operacje tworzenia siatek. Proces ten rozpoczyna się od przesunięcia o liczbę pustych wokseli, co odpowiada liczbie najmniej znaczących zer. W przypadku siatki przedstawionej na rysunku 3.8 nie występują końcowe zera w punkcie początkowym, co oznacza, że aktualna wysokość wskaźnika pozostaje na poziomie zero. Następnie następuje przesunięcie o liczbę jedynek, co pozwala na jednorazowe określenie wysokości siatki, jak zilustrowano na rysunku 3.8).

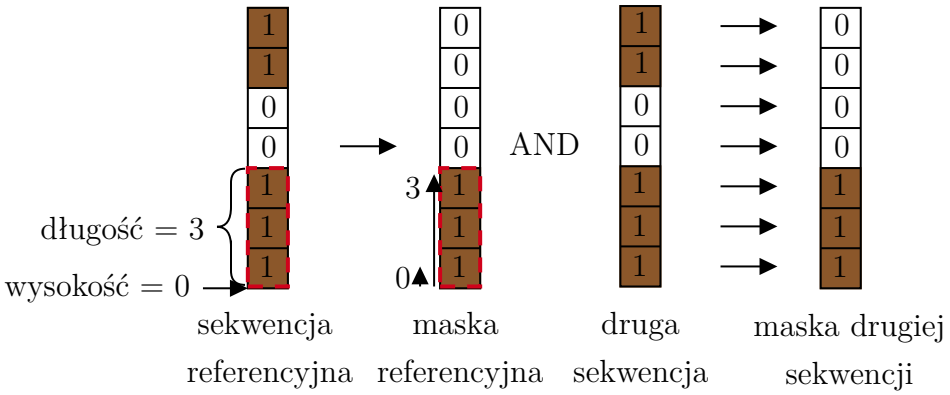


Rysunek 3.8: Rozszerzenie siatki w pionie.

Rozszerzenie siatki w poziomie jest nieco bardziej skomplikowane, ponieważ wymaga porównania dwóch sekwencji bitów. Na początku konieczne jest uzyskanie maski referencyjnej, która przedstawi bity siatki podlegającej rozszerzeniu. Znając długość i wysokość siatki, można skonstruować odpowiednią maskę:

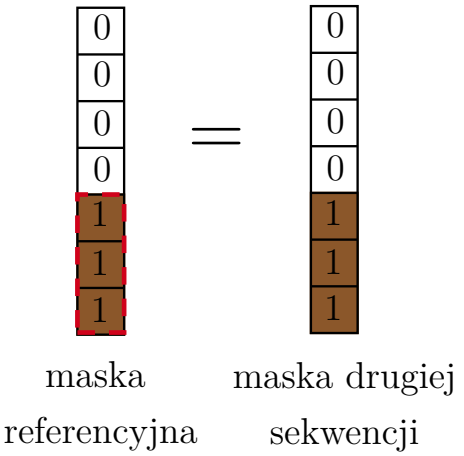
- Maska rozpoczyna się od stworzenia ciągu jedynek o długości równej długości siatki: `maska = (1 « długość) - 1`.
- Następnie przesuwa się tę maskę o wartość równą wysokości siatki: `maska = maska « wysokość`.

Ze względu na to, że długość maski wynosi 3, a jej wysokość to 0, finalnie uzyskuje się maskę referencyjną 0000111. Następnie konieczne jest uzyskanie drugiej sekwencji bitów, która będzie porównywana z maską. Przesuwając się do następnej kolumny i mając sekwencję bitów 1100111, wykonuje się operację logiczną AND na tej sekwencji oraz na mapie referencyjnej. W wyniku tej operacji otrzymuje się 0000111 (patrz rysunek 3.9).



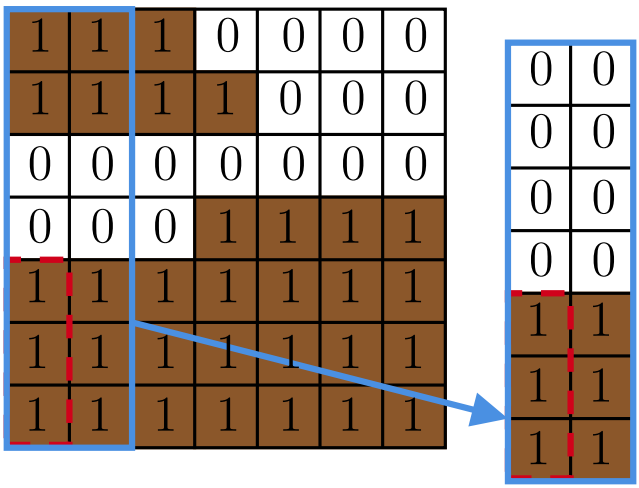
Rysunek 3.9: Stworzenie maski referencyjnej i maski drugiej sekwencji.

Porównując obie sekwencje bitów, można zauważyć, że są identyczne. Oznacza to, że możliwe jest rozszerzenie maski w poziomie.



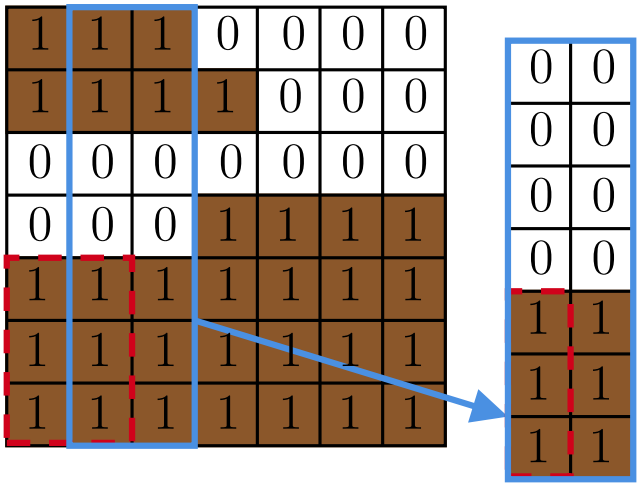
Rysunek 3.10: Porównanie maski referencyjnej i maski drugiej sekwencji.

Krok tego algorytmu można zobaczyć na rysunku 3.11.



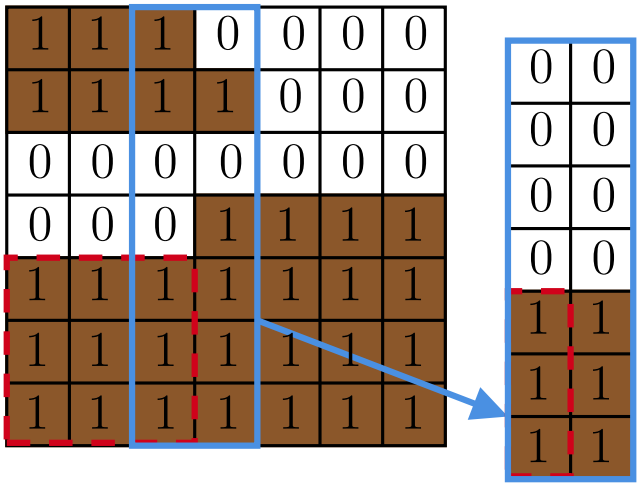
Rysunek 3.11: Pierwszy krok rozszerzenia siatki w poziomie poprzez porównanie maski referencyjnej i maski drugiej sekwencji.

Kolejny krok wygląda analogicznie.



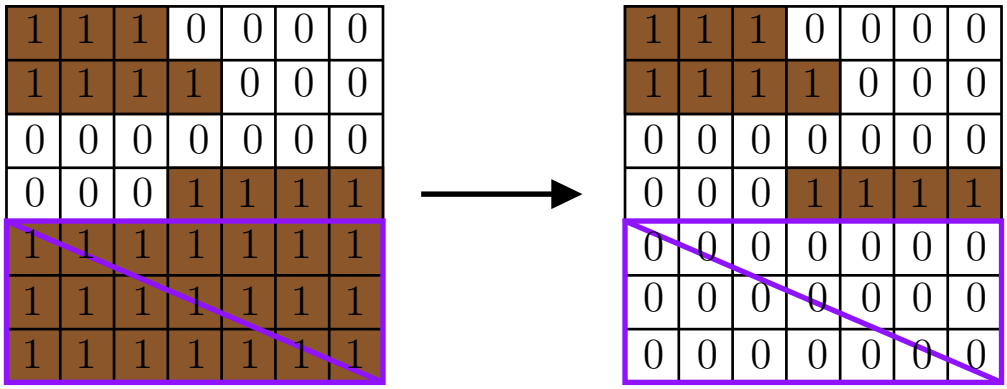
Rysunek 3.12: Drugi krok rozszerzenia siatki w poziomie poprzez porównanie maski referencyjnej i maski drugiej sekwencji.

Na rysunku 3.13 mimo że sekwencje bitowe są znacznie różne, to dzięki wcześniej przedstawionym operacjom bitowym obie maski są identyczne.



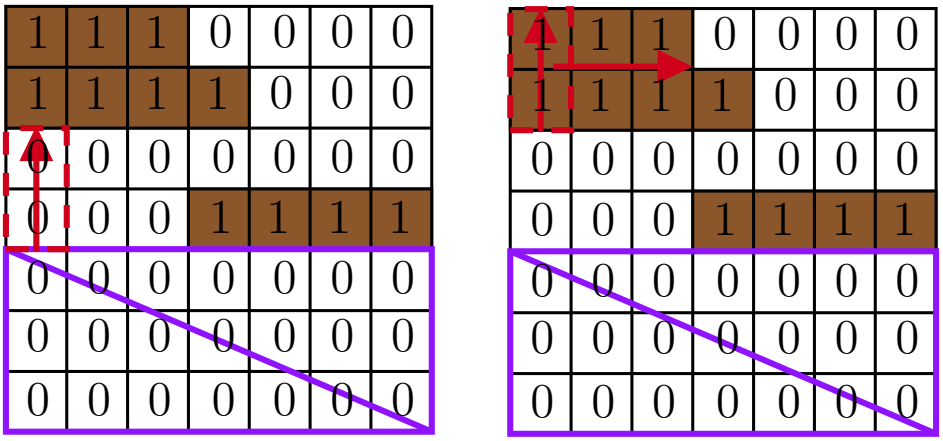
Rysunek 3.13: Trzeci krok rozszerzenia siatki w poziomie poprzez porównanie maski referencyjnej i maski drugiej sekwencji.

Dalsze kroki prowadzą do stworzenia pierwszej siatki. Bity w tablicy powinny zostać wyzerowane, aby nie wpływały na kolejne generowanie siatek (patrz rysunek 3.14).



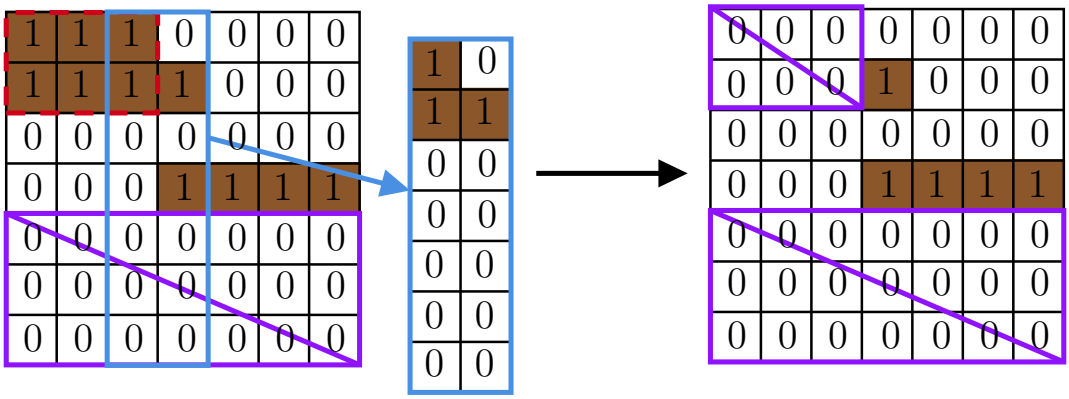
Rysunek 3.14: Finalny krok rozszerzenia siatki w poziomie.

Algorytm ponownie przesuwa się o liczbę obecnych zer, a w kolejnej operacji określa ilość pełnych wokseli w pionie poprzez zliczanie jedynek (patrz rysunek 3.15).



Rysunek 3.15: Znalezienie początku kolejnych pełnych wokseli oraz rozszerzenie siatki w pionie.

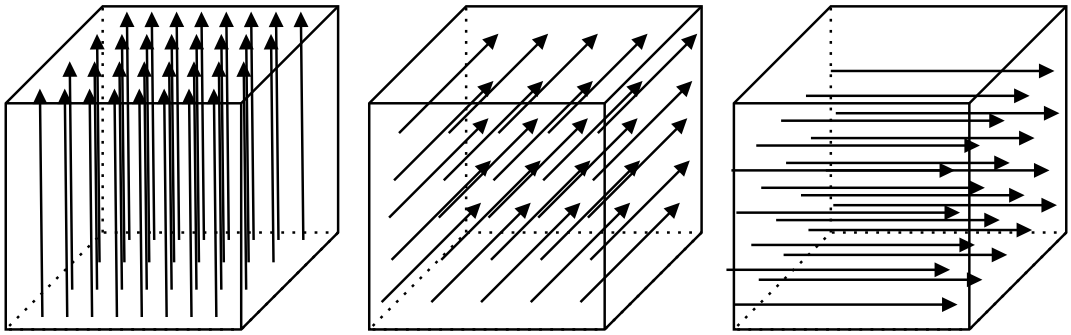
Na rysunku 3.16 widać, spełniony warunek końca tworzenia siatki gdy dwie wygenerowane maski nie są sobie równe. Reszta algorytmu działa analogicznie dla pozostałych wokseli, oraz dla tablic innych bloków.



Rysunek 3.16: Rozszerzenie siatki w poziomie.

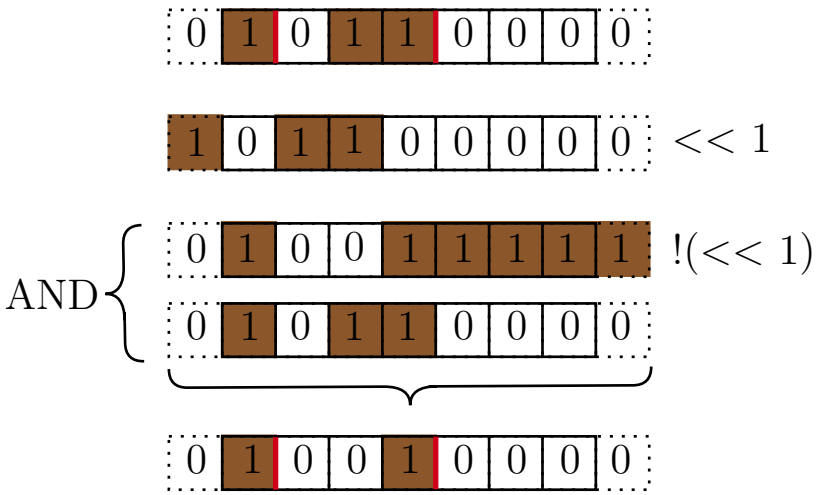
Istotną częścią tego algorytmu pozostaje jednak zbudowanie tablic bitowych, które muszą zostać zbudowane z uwzględnieniem ukrycia niewidocznych powierzchni. W fazie początkowej wymaga to również odczytania wartości wokseli poza bryłą. Oznacza to, że dla bryły o wymiarach 16x16x16, dane będą pobierane także na jej granicach, co skutkuje rozmiarem bryły 18x18x18 w początkowej fazie algorytmu. Wynika to z potrzeby uzyskania informacji o sąsiednich wokselach w celu usunięcia wewnętrznych powierzchni na krawędziach bryły.

Binarne zachłanne siatkowanie umożliwia bardzo wydajne wskazywanie widocznych powierzchni. Z perspektywy sekwencji binarnych można wykonywać przesunięcia bitowe tylko w lewo lub w prawo. Oznacza to, że bryła musi być odpowiednio przetwarzana. Sekwencje bitowe są przetwarzane dla każdej osi osobno w obu kierunkach, zgodnie z ilustracją na rysunku 3.17.



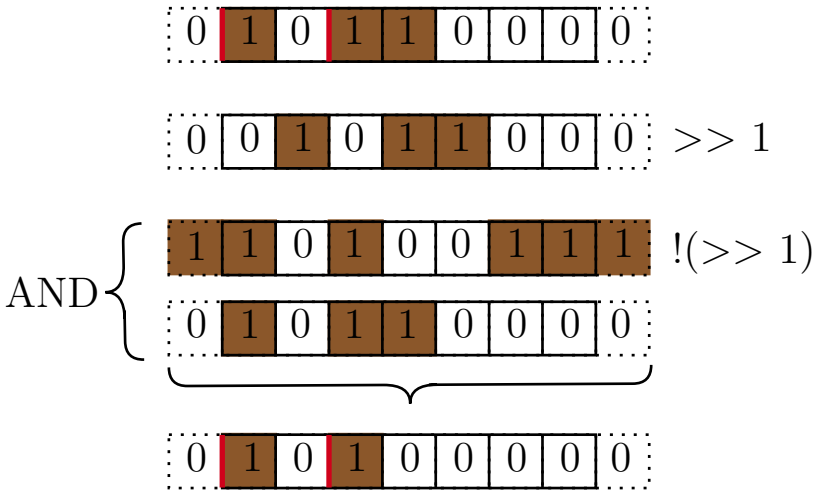
Rysunek 3.17: Sekwencje bitowe są odczytywane wzdłuż trzech osi.

Dla każdej takiej sekwencji stosunkowo łatwo jest określić widoczne powierzchnie dla lewej i prawej strony bitów. Aby określić widoczne powierzchnie z lewej strony, wykonywana jest operacja bitowego przesunięcia w prawo, następnie negacji, a na końcu operacja logiczna AND z pierwotną sekwencją bitów. Jest to widoczne na rysunku 3.18.



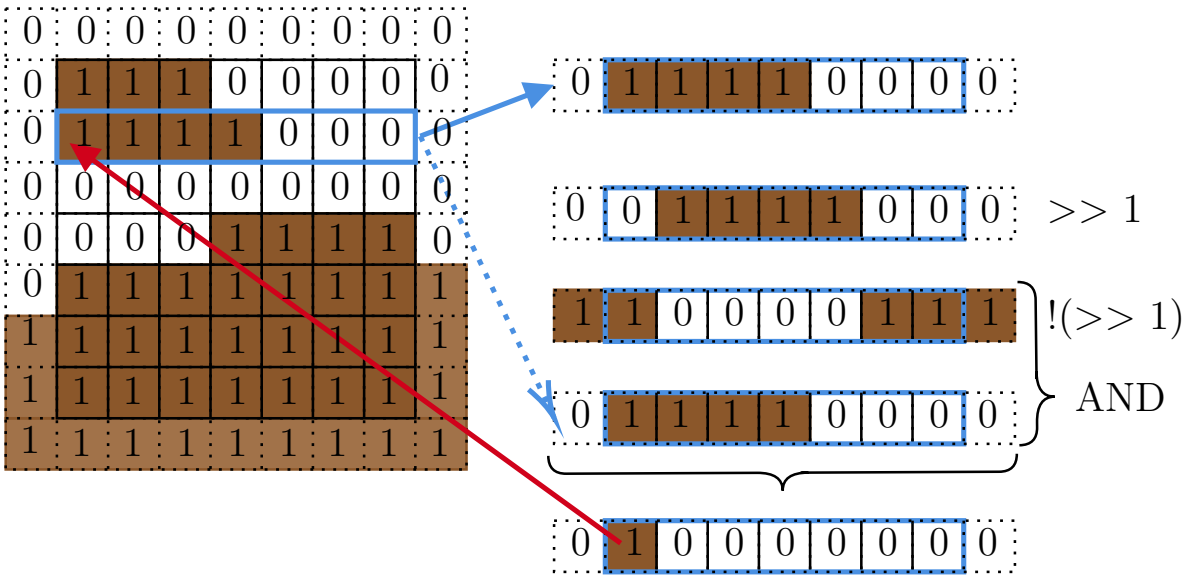
Rysunek 3.18: Oznaczenie niezakrytych powierzchni z prawej strony bitów

Aby określić widoczne powierzchnie z prawej strony, należy postępować analogicznie, tym razem wykonując przesunięcie bitowe w lewą stronę, co jest widoczne na rysunku 3.19.

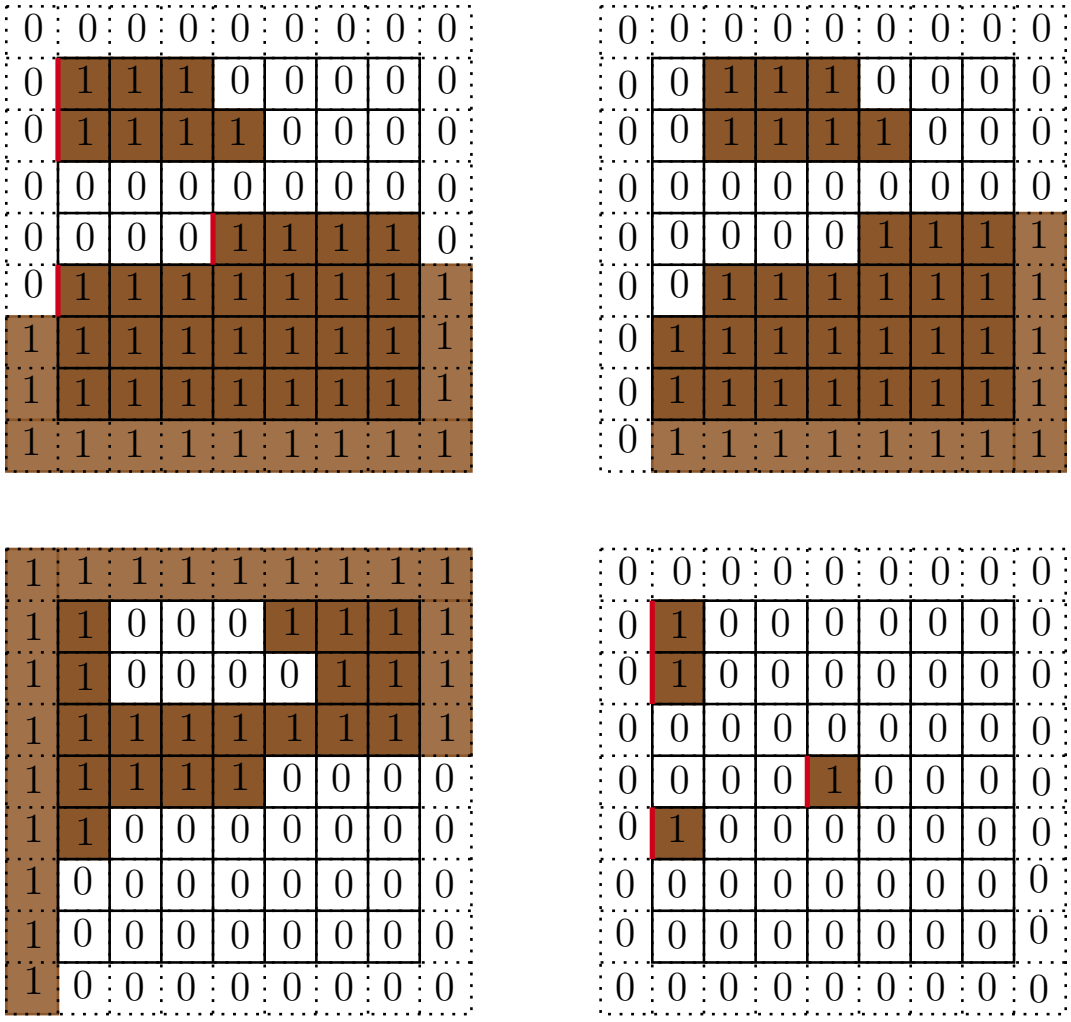


Rysunek 3.19: Oznaczenie niezakrytych powierzchni z lewej strony bitów

Proces dla pojedyńczej sekwencji jest widoczny na rysunku 3.20 A proces dla całego przekroju jest widoczny na rysunku 3.21.



Rysunek 3.20: Proces dla pojedynczej sekwencji przecięcia bryły.



Rysunek 3.21: Proces przedstawiony dla całego przekroju bryły.

Ostatnim krokiem jest umieszczenie tych informacji w kontenerze, tak aby wcześniej przedstawiony algorytm binarnego siatkowania mógł połączyć wszystkie widoczne powierzchnie tego samego typu i skierowane w tą samą stronę w jedną dużą siatkę.

```
funkcja przygotujSiatkę():
    binarnePłaszczyzny = zbudujBinarnePłaszczyzny()
    dla każdej ściany bloku:
        regionySiatki = wygenerujSiatkęDlaŚcianyBloku(binarnePłaszczyzny, ścianaBloku)
        dla każdego regionu w regionySiatki:
            dodajCzworokątDoSiatki(region)

funkcja zbudujBinarnePłaszczyzny():
    maskiUkrywaniaŚcian = zbudujMaskiUkrywaniaŚcian()
    dane = inicjalizujDaneKażdejMapyBloku()
    dla każdej ściany bloku:
        dla każdego z, oraz x w płaszczyźnie:
            indeksSekwencjiBitów = obliczIndeksSekwencjiBitów(x, z, ścianaBloku)
            maskaUkrywaniaŚcian = maskiUkrywaniaŚcian[indeksSekwencjiBitów]
            maskaUkrywaniaŚcian = usuńInformacjePoBokachMaski(maskaUkrywaniaŚcian)
            przechowajMaskęBinarnejPłaszczyźnie(dane, ścianaBloku, z, x, maskaUkrywaniaŚcian)
    zwróć dane

funkcja przechowajMaskęBinarnejPłaszczyźnie(dane, ścianaBloku, z, x, maskaUkrywaniaŚcian):
    dopóki maskaUkrywaniaŚcian nie jest pusta:
        y = znajdźPierwszyZerowyBit(maskaUkrywaniaŚcian)
        maskaUkrywaniaŚcian &= usuńNajniższyBit(maskaUkrywaniaŚcian)
        pozycjaWokselu = obliczPozycjęWokselu(ścianaBloku, x, y, z)
        identyfikatorBloku = pobierzIdentyfikatorTeksturyBloku(pozycjaWokselu, ścianaBloku)
        płaszczyzna = pobierzPłaszczyznę(dane, ścianaBloku, identyfikatorBloku, y)
        płaszczyzna[x] |= generujBitNaDanejPozycji(z)

funkcja wygenerujSiatkęDlaŚcianyBloku(binarnePłaszczyzny, ścianaBloku):
    regionySiatki = []
    dla każdego typu bloku i płaszczyzny w binarnePłaszczyzny[ścianaBloku]:
        regiony = wyodrębnijRegionyZPłaszczyzny(ścianaBloku, typBloku, płaszczyzna)
        dodajRegiony(regionySiatki, regiony)
    zwróć regionySiatki

funkcja wyodrębnijRegionyZPłaszczyzny(ścianaBloku, typBloku, płaszczyzna):
    regiony = []
    dla każdego położenia i pojedynczej płaszczyzny w płaszczyzna:
        czworokąty = zachłanneSiatkowanieJednejPłaszczyzny(pojedynczaPłaszczyzna, rozmiarPłaszczyzny)
        dla każdego czworokąta w czworokąty:
            region = stwórzRegionSiatki(ścianaBloku, typBloku, położenie, czworokąt)
            dodajRegion(regiony, region)
    zwróć regiony

funkcja zachłanneSiatkowanieJednejPłaszczyzny(dane, rozmiarPłaszczyzny):
    czworokąty = []
    dla każdego rzędu w dane:
        przetwórzRządDlaCzworokątów(dane, czworokąty, rząd, rozmiarPłaszczyzny)
    zwróć czworokąty

funkcja przetwórzRządDlaCzworokątów(dane, czworokąty, rząd, rozmiarPłaszczyzny):
    y = 0
    dopóki y < rozmiarPłaszczyzny:
        y += pomińPoczątkoweZera(dane[rząd], y)
        jeśli y >= rozmiarPłaszczyzny:
            przerwij
        szerokośćSegmentu = policzKolejneJedynki(dane[rząd], y)
        maska = wygenerujMaskęJedynkową(szerokośćSegmentu) << y
        szerokośćRozrostu = rozwińIUsuńRząd(dane, rząd, y, maskaSzerokości, maska, rozmiarPłaszczyzny)
        dodajCzworokąt(czworokąty, rząd, y, szerokośćRozrostu, szerokośćSegmentu)
        y += szerokośćSegmentu

funkcja rozwińIUsuńRząd(dane, rządPoczątkowy, kolumnaPoczątkowa, maskaSzerokości, maska, rozmiarPłaszczyzny):
    szerokość = 1
    dopóki (rządPoczątkowy + szerokość) < rozmiarPłaszczyzny:
        segmentNastępnegoRzędu = (dane[rządPoczątkowy + szerokość] >> kolumnaPoczątkowa) & maskaSzerokości
        jeśli segmentNastępnegoRzędu != maskaSzerokości:
            przerwij
        dane[rządPoczątkowy + szerokość] &= ~maska
        szerokość += 1
    zwróć szerokość
```

Fragment kodu 5: Wysokopoziomowy pseudokod algorytmu binarnego zachłannego siatkowania.

3.2.2 Śledzenie promieni

W przypadku śledzenia promieni analizowane będą różne metody przemieszczania się promienia przez przestrzeń wewnątrz bryły bloków. Zarówno w przypadku triangulacji, jak i śledzenia promieni, sceny będą zbudowane z brył bloków (ang. *Chunk*). Zasadniczą różnicą jest to, że w technikach triangulacji porównywana jest wydajność z wykorzystaniem tworzenia siatek. Natomiast w przypadku śledzenia promieni, porównywana jest wydajność sceny w zależności od sposobu, w jaki promień pokonuje zawartość każdej bryły. Wyjątkiem jest wokselowy marsz promieni z efektem paralaksy, który łączy śledzenie promieni z wykorzystaniem rasteryzacji.

3.2.2.1 Stały promień

Planowane jest porównanie pomiaru wydajności renderowania dla różnych stałych kroków promienia. Analiza obejmie zarówno wydajność tej metody, jak i efekty graficzne w zależności od długości kroku. Oznacza to, że promień będzie przemieszczał się w równych odstępach przez bryłę wokseli.

3.2.2.2 Algorytm szybkiego przechodzenia przez woksele

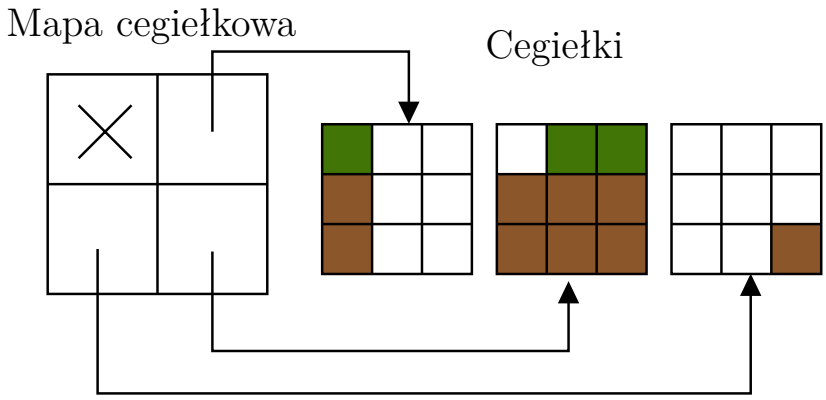
Kolejnym punktem badania będzie porównanie różnic w wydajności między stałym promieniem a algorytmem szybkiego przechodzenia przez woksele. Usprawnienie w postaci algorytmu Johna Amanatidesa i Andrew Woo powinno usunąć artefakty graficzne, które mogą wystąpić na krawędziach wokseli przy stałym promieniu, jednocześnie zachowując wydajność na poziomie w miarę sensownego stałego kroku.

3.2.2.3 Prosta siatka

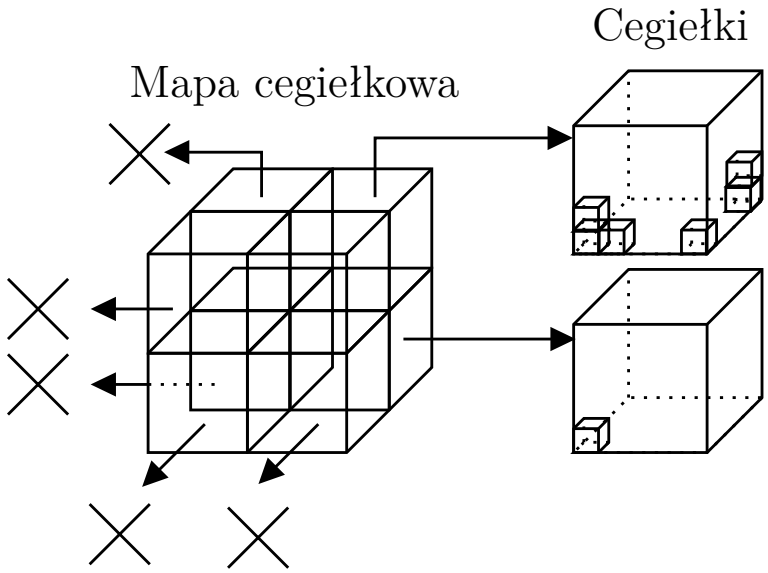
Stały promień i algorytm szybkiego przechodzenia przez woksele zostaną porównane na strukturze przedstawiającą prostą siatkę wokseli, w której promień stopniowo się porusza. Jest to najprostsza możliwa struktura danych, ponieważ przedstawia woksele w sposób surowy i bezpośredni.

3.2.2.4 Mapa cegiełkowa

Mapa cegiełkowa powinna umożliwić łatwe pomijanie pustych przestrzeni wokseli oraz zmniejszenie ilości zużytej pamięci poprzez nieprzechowywanie pustych wokseli jednocześnie próbując zachować surowy sposób przedstawienia danych. Potencjalnym problemem tej techniki jest fakt, że wystarczy zaledwie jeden woksel wewnątrz całej mapy cegiełkowej (bryły 8x8x8), aby wymusić rysowanie całej tej przestrzeni oraz jej przechowywanie w pamięci. Przykładową, uproszczoną ilustrację przedstawiającą działanie mapy cegiełkowej można zobaczyć na rysunku 3.22. Pełniejszy trójwymiarowy obraz tej struktury można zobaczyć na rysunku 3.23.



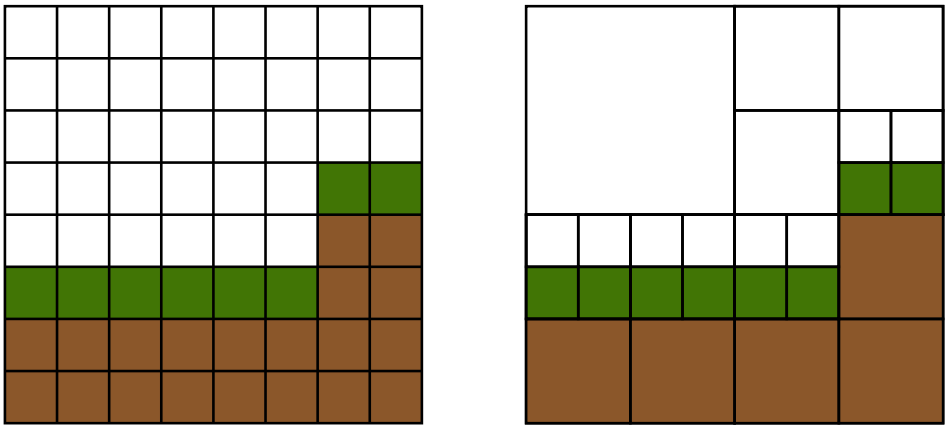
Rysunek 3.22: Uproszczone, dwuwymiarowe przedstawienie map cegiełkowych.



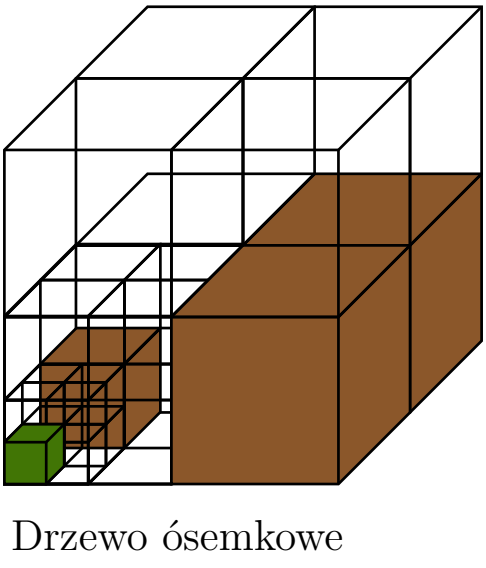
Rysunek 3.23: Trójwymiarowe przedstawienie map cegiełkowych.

3.2.2.5 Drzewo ósemkowe

Drzewo ósemkowe potencjalnie rozwiązuje problem mapy cegiełkowej, polegający na tym, że pojedynczy woksel wewnątrz mapy cegiełkowej wymusza zaalokowanie całej struktury i jej rysowanie. Drzewo ósemkowe rekursywnie dzieli przestrzeń na osiem mniejszych części i, jeżeli przestrzeń jest jednorodna (składa się z tego samego typu bloków), nie jest dalej rozdzielana. Pozwala to na bardzo efektywne pomijanie jednorodnych przestrzeni oraz nieprzechowywanie ich w pamięci. Kosztem tego jest odpowiednie skomplikowanie struktury i zwiększenie złożoności algorytmu. Na rysunku 3.24 można zobaczyć płaską siatkę oraz jej przedstawienie w postaci drzewa czwórkowego będącego uproszczeniem drzewa ósemkowego do postaci dwuwymiarowej. Przykładowe drzewo ósemkowe można zobaczyć na rysunku 3.25.



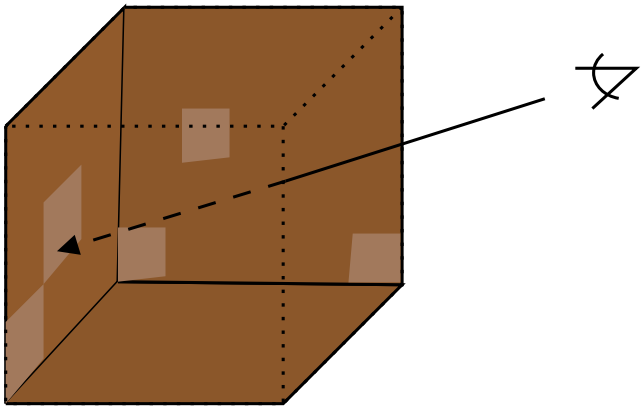
Rysunek 3.24: Płaska siatka wokseli i jej bezpośrednie przedstawienie w formie drzewa czórkowego.



Rysunek 3.25: Przykładowe drzewo ósemkowe.

3.2.2.6 Wokselowy marsz promieni z efektem paralaksy

Wokselowy marsz promieni z efektem paralaksy polega na zgrupowaniu wokseli w większe bloki 8x8x8, a następnie rysowaniu ich przy użyciu metody podobnej do mapowania paralaksy. Geometria jest rzutowana na tylne ściany sześcianu i każdy piksel powierzchni otrzymuje kolor odpowiadający skorygowanemu perspektywicznie obrazowi trójwymiarowemu. Jest to pewna hybryda między rasteryzacją a śledzeniem promieni.



Rysunek 3.26: Rzutowanie geometrii na tylne ściany sześcianu.

Rozdział 4

Badania

4.1 Opis stanowiska badawczego

Badania zostały przeprowadzone na następującej konfiguracji sprzętowej:

- **Procesor:** Intel Core i5 13600k.
- **Płyta główna:** Gigabyte Z690 Gaming X DDR4.
- **Pamięć RAM:** 48 GB DDR4.
- **Karta graficzna:** AMD Radeon RX 6700 XT.

Projekt zbudowano przy użyciu *CMake* w wersji 3.26.4 oraz kompilatora *MSVC* w wersji 19.38.33134.0 z obsługą standardu *C++20*.

4.1.1 Wykorzystane narzędzia i motywacja ich wyboru

4.1.1.1 Google Benchmark

Google Benchmark to biblioteka stworzona przez firmę *Google*, która służy do testowania i analizowania wydajności kodu w języku *C++* [7]. Umożliwia dokładne pomiary wydajności małych, specyficznych fragmentów kodu, zamiast przeprowadzania całościowego przeglądu wydajności całego systemu. Precyzja pomiarów oraz wielokrotne wykonywanie pomiarów w celu zmniejszenia błędu pomiarowego zapewniają wiarygodne wyniki. Elastyczność biblioteki pozwala na definiowanie różnych scenariuszy testowych, a dzięki łatwej integracji możliwe jest szybkie rozpoczęcie testowania.

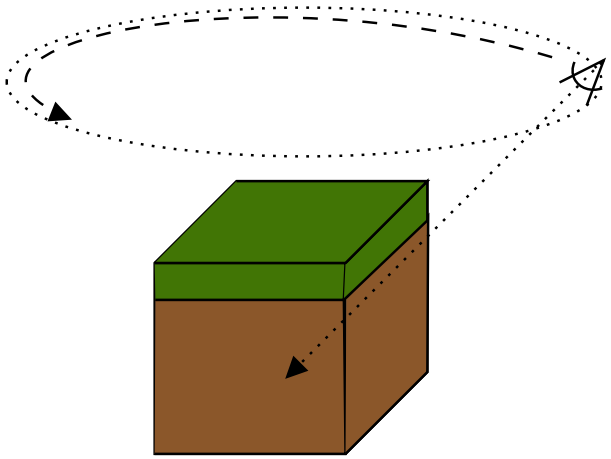
4.1.1.2 Tracy Profiler

Tracy Profiler to narzędzie do profilowania aplikacji w czasie rzeczywistym, które zawiera różnego rodzaju funkcje śledzenia i analizy pozwalające uzyskać szczegółowe informacje na temat wykorzystania zasobów systemowych oraz identyfikację wąskich gardeł

[2]. Dzięki wsparciu dla różnych API graficznych, takich jak *OpenGL* i *Direct3D*, jest szczególnie użyteczne w kontekście profilowania aplikacji graficznych. Umożliwia ono zebranie statystyk dotyczących czasu poświęcanego przez procesor na wykonanie określonych fragmentów kodu oraz czasu potrzebnego na wykonanie operacji renderowania przez kartę graficzną. Ponadto, posiada opcję gromadzenia statystyk pamięci, takich jak alokacje i użycie pamięci.

4.2 Sceny

Każda ze scen posiada kamerę poruszającą się ruchem owalnym wokół środka sceny, co jest zaprezentowane na rysunku 4.1. Sceny są podzielone na dwie główne kategorie, z różnymi wariacjami dla każdej z nich (patrz sekcja 4.2.1 i sekcja 4.2.2). Każdy pomiar trwa około minuty.



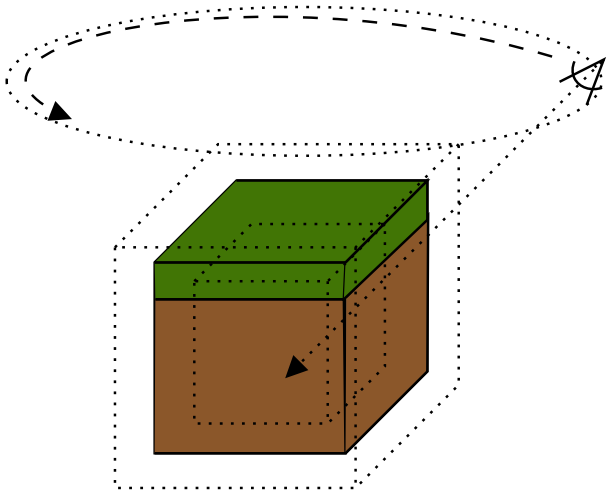
Rysunek 4.1: Kamera poruszająca się ruchem owalnym wokół sceny.

4.2.1 Jedna bryła

Każda technika jest sprawdzana pod kątem rysowania jednej bryły na scenie. Testy są przeprowadzane w pięciu wariacjach:

- Bryła o boku wielkości 8 wokseli.
- Bryła o boku wielkości 16 wokseli.
- Bryła o boku wielkości 32 wokseli.
- Bryła o boku wielkości 64 wokseli.

Scena przedstawiona jest na rysunku 4.2.

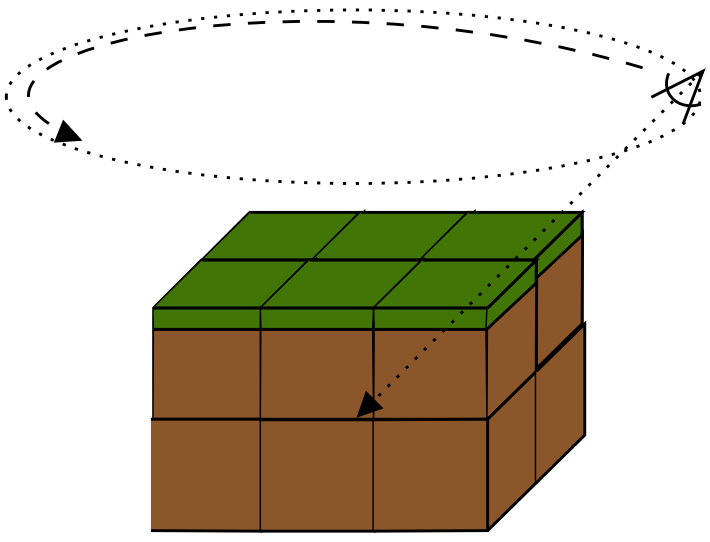


Rysunek 4.2: Kamera poruszająca się ruchem owalnym wokół jednej bryły, badanej dla kilku wielkości.

4.2.2 Wiele bryl

Każda technika jest również sprawdzana pod kątem rysowania wielu brył. W tej kategorii testów również występuje kilka wariacji:

- 18 brył o bokach wielkości 16, 32 wokseli.
- 405 brył o bokach wielkości 16, 32 wokseli.
- 1445 brył o bokach wielkości 16, 32 wokseli.



Rysunek 4.3: Kamera poruszająca się ruchem owalnym wokół wielu brył.

4.3 Mikrotestowanie wydajności

Aby przetestować małe, specyficzne fragmenty kodu, takie jak budowa siatki, zostanie zastosowane mikrotestowanie. Metoda ta jest wyjątkowo użyteczna, ponieważ pozwala uzyskać wiarygodne wyniki z minimalnym błędem pomiarowym poprzez zbieranie danych z wielu iteracji. Dzięki bibliotece *Google Benchmark* [7] scenariusze testowe można łatwo modyfikować pod kątem parametrów wejściowych, co umożliwia precyzyjne pomiary wydajności algorytmu w zależności od wielkości tych parametrów. Przykładowy test pomiarowy przedstawiono na poniższym fragmencie kodu 6.

```
1  static void BM_ChunkCullingRebuildMesh(benchmark::State& state)
2  {
3      initializeOpenGL();
4
5      auto blockPosition = Block::Coordinate{0, 0, 0};
6      auto texturePack = TexturePackArray("default");
7      auto chunk = Polygons::ChunkCulling(blockPosition, texturePack);
8
9      for (auto _: state)
10     {
11         chunk.rebuildMesh();
12     }
13 }
```

Fragment kodu 6: Test pomiaru szybkości budowania siatki dla metody ukrywania powierzchni.

Przykładowy wynik wykonania testu w bibliotece *Google Benchmark* jest przedstawiony w fragmencie kodu 7.

```
-------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations
-------------------------------------------------------------------------
BM_ChunkCullingRebuildMesh         6405564 ns      1918248 ns          896
```

Fragment kodu 7: Przkładowy wynik pomiaru szybkości budowania siatki dla metody ukrywania powierzchni.

4.4 Pomiary

4.4.1 Triangulacja

W przypadku triangulacji istotnym pomiarem jest czas budowania siatki. Czas ten zostanie zmierzony przy pomocy mikrotestowania wydajności. Ponadto, analizie zostanie poddany całkowity czas renderowania. Każda scena zostanie oceniona pod kątem liczby wygenerowanych wierzchołków, zajętości pamięci, czasu budowania oraz średniej liczby klatek na sekundę. Zostaną również zebrane statystyki dotyczące czasu poświęcanego przez procesor na wykonanie innych fragmentów kodu.

4.4.2 Śledzenie promieni

W przypadku śledzenia promieni istotnym pomiarem jest liczba iteracji promienia, które promień musi przebyć, zanim trafi na pełen woksel. Średnia liczba iteracji promienia zostanie zmierzona przy użyciu atomowego licznika, czyli specjalnego rodzaju bufora umożliwiającego atomowe operacje na wartościach całkowitych, wymagających synchronizacji między wieloma wątkami, jak to ma miejsce w przypadku jednostek cieniujących.

4.5 Badane techniki

Badane techniki podzielone są na dwie kategorie: triangulację i śledzenie promieni.

4.5.1 Triangulacja

Triangulacja obejmuje analizę następujących metod:

- Metoda naiwna opisana w podsekcji 3.2.1.1.
- Metoda ukrywania powierzchni opisana w podsekcji 3.2.1.2.
- Metoda ukrywania powierzchni ze wsparciem GPU opisana w podsekcji 3.2.1.3.
- Metoda zachłannego siatkowania opisana w podsekcji 3.2.1.4.
- Metoda binarnego zachłannego siatkowania opisana w podsekcji 3.2.1.5.

4.5.2 Śledzenie promieni

Śledzenie promieni obejmuje analizę następujących technik:

- Surowa struktura danych z zastosowaniem stałego kroku promienia opisana w podsekcji 3.2.2.1 i 3.2.2.3.

- Algorytm szybkiego przechodzenia przez woksele korzystający z surowej struktury danych opisany w podsekcji 3.2.2.2 i 3.2.2.3.

Ponadto, badane są dwie inne struktury danych:

- Mapa cegiełkowa opisana w podsekcji 3.2.2.4.
- Drzewo ósemkowe opisane w podsekcji 3.2.2.5.

4.6 Proces badawczy

4.6.1 Cel badań

Celem badań jest ocena skuteczności różnych technik optymalizacji stosowanych w wokselowej reprezentacji świata gry oraz porównanie dwóch wiodących technik: triangulacji i śledzenia promieni. Badania porównawcze obejmują różne podejścia, oceniając je pod kątem wydajności (czas renderowania, czas poświęcony przez procesor), zużycia pamięci, czas budowania oraz jakości grafiki.

4.6.2 Przebieg procesu badawczego

Cały proces badawczy obejmował implementację oraz testowanie dwóch kategorii renderowania. Na śledzenie promieni przypada pięć różnych technik, jak i również na triangulację pięć różnych technik. Każdy model podlegał szczegółowej ewaluacji opartej na trzech kluczowych kryteriach:

- **Czas renderowania** - mierzony dla każdej sceny osobno.
- **Czas budowania sceny** - mierzony dla różnych etapów budowania.
- **Zużycie pamięci** - mierzone od strony kodu dla buforów wysyłanych do karty graficznej.
- **Jakość wizualna** - oceniana subiektywnie, z założeniem braku jakiejkolwiek degradacji graficznej.

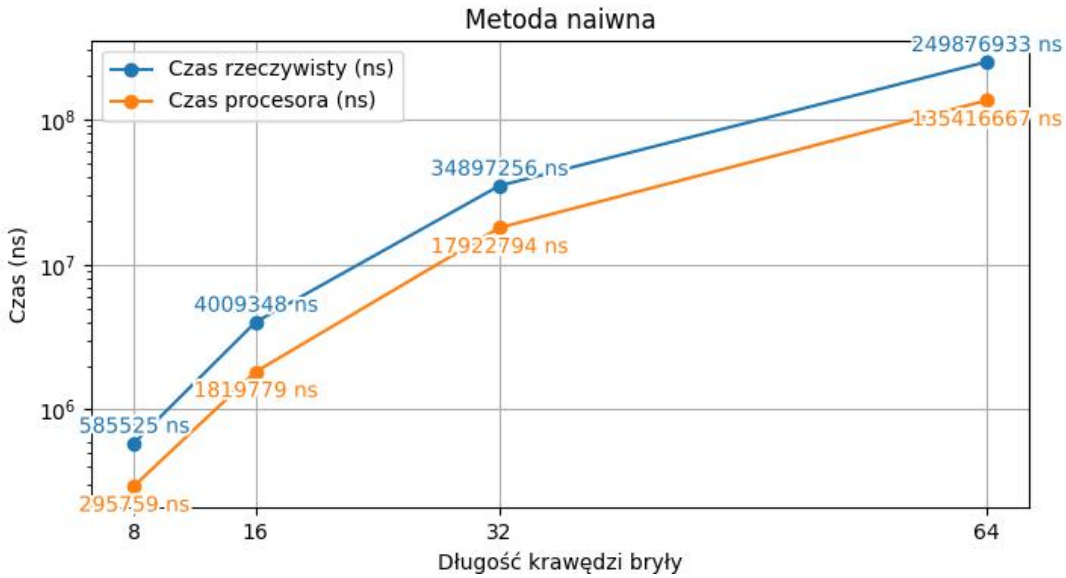
4.7 Wyniki eksperymentów

Wyniki eksperymentów podzielono na dwie główne części: triangulację i śledzenie promieni. W każdej części omówiono specyficzne techniki dla danej metody, zgodnie z informacjami przedstawionymi w sekcji 4.5.

4.7.1 Triangulacja

Triangulacja została przebadana zarówno pod kątem średniej liczby klatek na sekundę, mierzonej na przestrzeni 60 sekund, jak i szybkości budowania scen, liczby wierzchołków oraz zajętości pamięci. Testy przeprowadzono przy użyciu mikrotestowania oraz analizy statystyk zebranych z przygotowanych i uruchomionych scen.

4.7.1.1 Metoda Naiwna



Rysunek 4.4: Wynik mikrotestowania budowy siatki w metodzie naiwnej.

Z wykresu mikrotestowania metody naiwnej (patrz rysunek 4.4) wynika, że czas procesora stanowi około 50% czasu rzeczywistego (kolejno: 50.21%, 45.39%, 51.36%, 54.19%). Sugeruje to, że pozostałe 50% czasu przeznaczone jest na operacje wejścia/wyjścia, zarządzanie pamięcią i inne narzuty. Złożoność obliczeniowa jest zbliżona do $O(n^3)$, co jest typowe dla naiwnych metod operujących na trójwymiarowych danych.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy siatki**	**Średnie użycie procesora**	**Zajętość pamięci**
8x8x8	13677 (±4217)	758.8 µs	0.95%	9528 wierzchołki 228672 bajty (0.22 MB)
16x16x16	11987 (±3821)	6.58 ms	3.38%	74928 wierzchołki 1798272 bajtów (1.79 MB)
32x32x32	7254 (±1879)	48.93 ms	0.59%	607944 wierzchołki 14590656 bajtów (14.59 MB)
64x64x64	1857 (±670)	341.71 ms	2.98%	4187592 wierzchołki 100502208 bajtów (100.50 MB)

Tabela 4.1: Wyniki pomiaru wydajności dla jednej bryły rysowanej metodą naiwną w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Zgodnie z informacji zawartymi w tabeli 4.1 można zauważyć, że wraz z wzrostem wielkości bryły intensywnie spada średnia liczba klatek na sekundę.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało spadkiem średniej liczby klatek na sekundę o 12.36%.
- Przejście z bryły 16x16x16 do 32x32x32 skutkowało spadkiem średniej liczby klatek na sekundę o 39.48%.
- Przejście z bryły 32x32x32 do 64x64x64 skutkowało spadkiem średniej liczby klatek na sekundę o 74.40%.

Czas budowy i zajętość pamięci również notują ogromny wzrost.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało wzrostem czasu budowy o 767.16% oraz wzrostem zajętości pamięci o 713.64%.
- Przejście z bryły 16x16x16 do 32x32x32 spowodowało wzrost czasu budowy o 643.62% oraz wzrost zajętości pamięci o 715.08%.
- Przejście z bryły 32x32x32 do 64x64x64 przyniosło wzrost czasu budowy o 598.37% oraz wzrost zajętości pamięci o 588.83%.

Oznacza to, że przejście z bryły 8x8x8 na bryłę 64x64x64 poskutkowało spadkiem średniej liczby klatek na sekundę o 86.42%, wzostem czasu budowy siatki o 44932.95%, oraz wzrostem zajętości pamięci o 45581%.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Liczba wierzchołków**	**Zajętość pamięci**
18	16x16x16	4342 (±666)	1497192	35932608 bajtów (35.93 MB)
18	32x32x32	754 (±61)	11017584	264422016 bajtów (264.42 MB)
405	16x16x16	745 (±54)	12602160	302451840 bajtów (302.45 MB)
405	32x32x32	107 (±2.42)	99088104	2378114496 bajtów (2378.11 MB)
1445	16x16x16	237 (±4.84)	45589272	1094142528 bajtów (1094.14 MB)
1445	32x32x32	—	352482816	4164620288 bajtów (4164.62 MB)

Tabela 4.2: Wyniki pomiaru wydajności dla wielu brył rysowanych metodą naiwną w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

W tabeli 4.2 nie udało się uzyskać informacji na temat średniej ilości klatek na sekundę dla przypadku 1445 brył o wielkości 32x32x32, jednak wynik ten zdecydowanie nie spełnia wymagań aplikacji czasu bliskiego rzeczywistemu.

Na średnią liczbę klatek na sekundę wpływają zarówno wielkość bryły, jak i liczba brył. Porównując przypadek 18 brył o wielkości 32x32x32 z 405 bryłami o wielkości 16x16x16, których liczba wierzchołków jest zbliżona, można założyć, że to liczba wierzchołków ma największy wpływ na wydajność. Podobną tendencję można zaobserwować, porównując przypadek 405 brył o wielkości 32x32x32 z 1445 bryłami o wielkości 16x16x16.

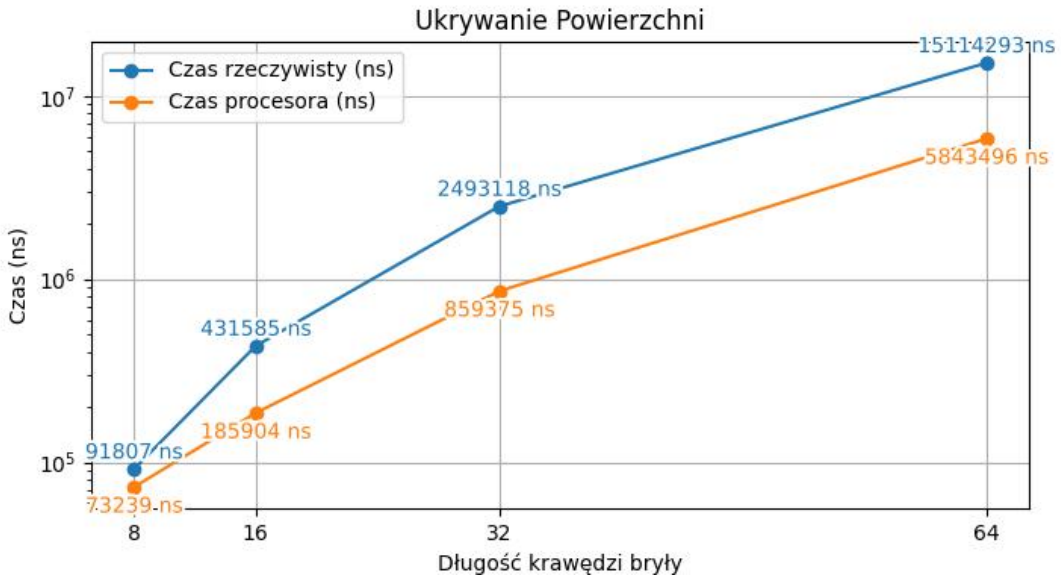
Liczba brył	Wielkość bryły	Czas tworzenia brył	Czas budowy siatek	Czas przebudowy siatek	Łączny czas budowy sceny
18	16x16x16	139.63 ms (18 brył)	343.84ms (51 siatek)	225.21 ms (33 przebudowy)	407.3 ms
18	32x32x32	1.04s (18 brył)	2.21s (51 siatek)	1.32s (33 przebudowy)	2.74s
405	16x16x16	853.59 ms (405 brył)	2.86s (1449 siatek)	2.14s (1044 przebudowy)	3.4s
405	32x32x32	7.93s (405 brył)	24.66s (1449 siatek)	18.27s (1044 przebudowy)	31.34s
1445	16x16x16	3.43s (1445 brył)	11.47s (5321 siatek)	8.59s (3876 przebudowy)	13.77s
1445	32x32x32	33.73s (1445 brył)	1m 48s (5321 siatek)	1m 21s (3876 przebudowy)	2m 18s

Tabela 4.3: Wyniki pomiaru czasu budowy sceny dla wielu brył rysowanych metodą naiwną w scenie trwającej 1 minutę.

W tabeli 4.3 zebrano dane dotyczące pomiaru czasu budowania sceny. Czas tworzenia brył odnosi się do pierwszego utworzenia każdej bryły. Zakładając dynamiczne dobudowywanie się świata, dodawanie nowych brył wymaga przebudowywania siatek sąsiadów, co wprowadza dodatkowy czas przebudowy siatek. Czas budowy siatek stanowi sumę czasu tworzenia brył i czasu przebudowy siatek. Łączny czas budowy sceny uwzględnia dodatkowy narzut związany z aktualizacją informacji niebędących bezpośrednio związanych z procesem budowania siatki.

Można zauważyć, że w procesie budowy siatek największy jest czas przebudowy siatek. Potencjalne przyspieszenie tego procesu, na przykład poprzez edycję siatki bryły bezpośrednio na jej krawędziach, mogłoby znacząco skrócić czas budowania sceny. Jest to obiecująca ścieżka do potencjalnych usprawnień.

4.7.1.2 Ukrywanie powierzchni



Rysunek 4.5: Wynik mikrotestowania budowy siatki w metodzie ukrywania powierzchni.

Z wykresu mikrotestowania metody ukrywania powierzchni (patrz rysunek 4.5) wynika, że czas podobnie jak w poprzedniej metodzie stanowi blisko tylko 49% czasu rzeczywistego. Tym razem jednak procent czasu procesora znacząco maleje wraz ze wzrostem bryły. Dla bryły o wymiarach 8x8x8 stanowi on aż 79.78% czasu rzeczywistego. Dla większych brył średnio wynosi 40% czasu procesora. Metoda ukrywania powierzchni zajmuje zaledwie pomiędzy 15.68% – 06.05% czasu rzeczywistego metody naiwnej i między 24.76% – 4.32% czasu procesora metody naiwnej. Największą poprawę względem metody naiwnej widać wraz ze wzrostem długości boku bryły.

Wielkość bryły	Średnia klatek na sekunde (± SD)	Czas budowy siatki	Średnie użycie procesora	Zajętość pamięci
8x8x8	14386 (±4675)	133.28 µs	4.65%	1352 wierzchołków 32448 bajtów (0.03 MB)
16x16x16	13670 (±4508)	707.42 µs	2.03%	5432 wierzchołków 130368 bajtów (0.13 MB)
32x32x32	13682 (±4544)	3.47 ms	3.10%	21496 wierzchołków 515904 bajtów (0.51 MB)
64x64x64	13259 (±4153)	15.67 ms	3.06%	83008 wierzchołków 1992192 bajtów (1.99 MB)

Tabela 4.4: Wyniki pomiaru wydajności dla jednej bryły rysowanej metodą ukrywania powierzchni w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Zgodnie z informacjami zawartymi w tabeli 4.4 można zauważyć ogromną poprawę w kwestii średniej liczby klatek na sekundę wraz z wzrostem wielkości bryły względem metody naiwnej. Średnia liczba klatek na sekundę dla każdej wielkości bryły pozostaje podobna i wacha się w granicach błędu pomiarowego.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało spadkiem średniej liczby klatek na sekundę o 4.98%.
- Przejście z bryły 16x16x16 do 32x32x32 skutkowało wzrostem średniej liczby klatek na sekundę o 0.09%.
- Przejście z bryły 32x32x32 do 64x64x64 skutkowało spadkiem średniej liczby klatek na sekundę o 3.09%.

Czas budowy i zajętość pamięci również spisują się sporo lepiej w porównaniu do metody naiwnej. Porównując bryłę 64x64x64 widać poprawę rzędu 95.41% w kwestii czasu budowania i 98.02% w kwestii zajętości pamięci. Zwiększanie wielkości bryły również już nie wiąże się z tak drastycznym wzrostem zajętości pamięci jak w metodzie naiwnej.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało wzrostem czasu budowy o 430.78% oraz wzrostem zajętości pamięci o 301.78%.
- Przejście z bryły 16x16x16 do 32x32x32 spowodowało wzrost czasu budowy o 390.51% oraz wzrost zajętości pamięci o 295.73%.

- Przejście z bryły 32x32x32 do 64x64x64 przyniosło wzrost czasu budowy o 351.59% oraz wzrost zajętości pamięci o 286.16%.

Oznacza to, że przejście z bryły 8x8x8 na bryłę 64x64x64 poskutkowało spadkiem średniej liczby klatek na sekundę o 7.84%, wzrostem czasu budowy siatki o 11657.20%, oraz wzrostem zajętości pamięci o 6039.64%. Nie jest to aż tak kosztowne skalowanie jak w metodzie naiwnej.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Liczba wierzchołków**	**Zajętość pamięci**
18	16x16x16	13741 (±3615)	40608	974592 bajtów (0.97 MB)
18	32x32x32	11898 (±3245)	161112	3866688 bajtów (3.86 MB)
405	16x16x16	3805 (±855)	243560	5845440 bajtów (5.84 MB)
405	32x32x32	3798 (±738)	1031344	24752256 bajtów (24.75 MB)
1445	16x16x16	1081 (±138)	767592	18422208 bajtów (18.42 MB)
1445	32x32x32	1129 (±117)	3299064	79177536 bajtów (79.17 MB)

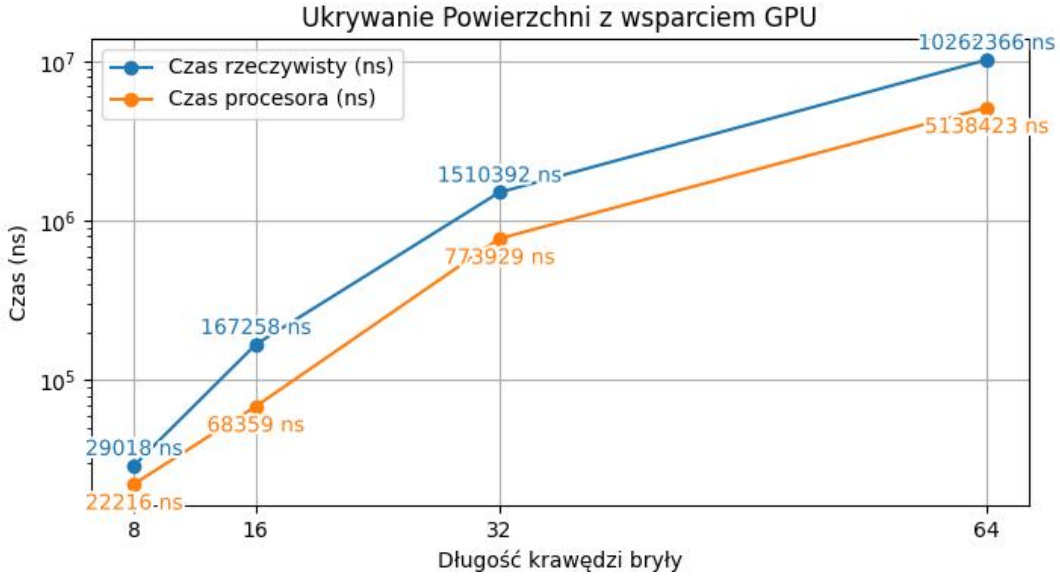
Tabela 4.5: Wyniki pomiaru wydajności dla wielu brył rysowanych metodą ukrywania powierzchni w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Tabela 4.5 ujawnia potencjalny problem ze spadkiem średniej liczby klatek na sekundę wraz ze zwiększeniem liczby brył, który nie był widoczny w pomiarze dla jednej bryły. Scena z jedną bryłą stanowi pomiar na zbyt małą skalę. Problem związany ze spadkiem wydajności zdaje się mieć inne podłoże niż w metodzie naiwnej. Porównując przypadki większych brył o tej samej liczbie brył na scenie, można zauważyć znaczący wzrost liczby wierzchołków, który niekoniecznie prowadzi do spadku średniej liczby klatek na sekundę. Sugeruje to, że w tym przypadku problemem może być nie rosnąca liczba wierzchołków, lecz liczba wywołań rysowania.

Liczba brył	Wielkość bryły	Czas tworzenia brył	Czas budowy siatek	Czas przebudowy siatek	Łączny czas budowy sceny
18	16x16x16	10.65 ms (18 brył)	22.8ms (51 siatek)	12.81ms (33 przebudowy)	31.76 ms
18	32x32x32	61.62ms (18 brył)	132.46 ms (51 siatek)	73.65 ms (33 przebudowy)	233.11 ms
405	16x16x16	96.28ms (405 brył)	248.98ms (1449 siatek)	159.72ms (1044 przebudowy)	413.29 ms
405	32x32x32	551.29 ms (405 brył)	1.71s (1449 siatek)	1.19s (1044 przebudowy)	3.96s
1445	16x16x16	372.18 ms (1445 brył)	1.09s (5321 siatek)	757.77ms (3876 przebudowy)	1.83s
1445	32x32x32	1.75s (1445 brył)	5.75s (5321 siatek)	4.11s (3876 przebudowy)	13.67 s

Tabela 4.6: Wyniki pomiaru czasu budowy sceny dla wielu brył rysowanych metodą ukrywania powierzchni w scenie trwającej 1 minutę.

4.7.1.3 Ukrywanie powierzchni z wsparciem GPU



Rysunek 4.6: Wynik mikrotestowania budowy siatki w metodzie ukrywania powierzchni z wsparciem GPU.

Z wykresu mikrotestowania metody ukrywania powierzchni z wsparciem GPU (patrz rysunek 4.6) wynika, że czas procesora stanowi blisko 54% czasu rzeczywistego. Podobnie

jak w przypadku podstawowej wersji ukrywania powierzchni, najwięcej czasu procesora poświęcone jest bryle o rozmiarze 8x8x8, co stanowi aż 76.57%. Dla większych brył średni czas procesora wynosi 50%, przy czym najmniejszy procent czasu procesora dotyczy bryły 16x16x16. Metoda ukrywania powierzchni z wsparciem GPU zajmuje od 4.96% do 4.11% czasu rzeczywistego metody naiwnej oraz od 7.51% do 3.79% czasu procesora metody naiwnej. Jest to znacząca poprawa w porównaniu do metody bez wsparcia GPU dla brył o rozmiarach 8x8x8 i 16x16x16. Dla brył 32x32x32 i 64x64x64 różnica wynosi około 0.5 punktu procentowego względem czasu wykonania metody naiwnej.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy siatki**	**Średnie użycie procesora**	**Zajętość pamięci**
8x8x8	14364 (±4608)	42.46 µs	1.40%	338 wierzchołków 6760 bajtów
16x16x16	13649 (±4473)	235.4 µs	3.73%	1358 wierzchołków 27160 bajtów (0.02 MB)
32x32x32	13690 (±4517)	1.5 ms	3.68%	5374 wierzchołków 107480 bajtów (0.1 MB)
64x64x64	12028 (±3609)	10.75 ms	2.96%	20752 wierzchołków 415040 bajtów (0.41 MB)

Tabela 4.7: Wyniki pomiaru wydajności dla jednej bryły rysowanej metodą ukrywania powierzchni z wsparciem GPU w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Zgodnie z informacjami zawartymi w tabeli 4.7 można zauważyć, podobne osiągi w stosunku do metody bez wsparcia GPU.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało spadkiem średniej liczby klatek na sekundę o 4.98%.
- Przejście z bryły 16x16x16 do 32x32x32 skutkowało wzrostem średniej liczby klatek na sekundę o 0.30%.
- Przejście z bryły 32x32x32 do 64x64x64 skutkowało spadkiem średniej liczby klatek na sekundę o 12.14%.

Istotną zmianą względem metody bez wsparcia GPU jest jednak zmniejszenie zajętości pamięci czterokrotnie, oraz znaczne zmniejszenie czasu budowy siatki.

- rzejście z siatki 8x8x8 do 16x16x16 skutkowało wzrostem czasu budowy o 454.40% oraz wzrostem zajętości pamięci o 301.78%.
- Przejście z bryły 16x16x16 do 32x32x32 spowodowało wzrost czasu budowy o 537.21% oraz wzrost zajętości pamięci o 295.73%.
- Przejście z bryły 32x32x32 do 64x64x64 przyniosło wzrost czasu budowy o 616.67% oraz wzrost zajętości pamięci o 286.16%.

Oznacza to, że przejście z bryły 8x8x8 na bryłę 64x64x64 poskutkowało spadkiem średniej liczby klatek na sekundę o 16.26%, wzrostem czasu budowy siatki o 25217.95%, oraz wzrostem zajętości pamięci o 6039.64%.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Liczba wierzchołków**	**Zajętość pamięci**
18	16x16x16	13113 (±3931)	10152	203040 bajtów (0.2 MB)
18	32x32x32	10388 (±2762)	40278	805560 bajtów (0.8 MB)
405	16x16x16	4258 (±888)	60890	1217800 bajtów (1.21 MB)
405	32x32x32	3587 (±428)	257836	5156720 bajtów (5.15 MB)
1445	16x16x16	1283 (±185)	191898	3837960 bajtów (3.83 MB)
1445	32x32x32	1300 (±83)	824766	16495320 bajtów (16.49 MB)

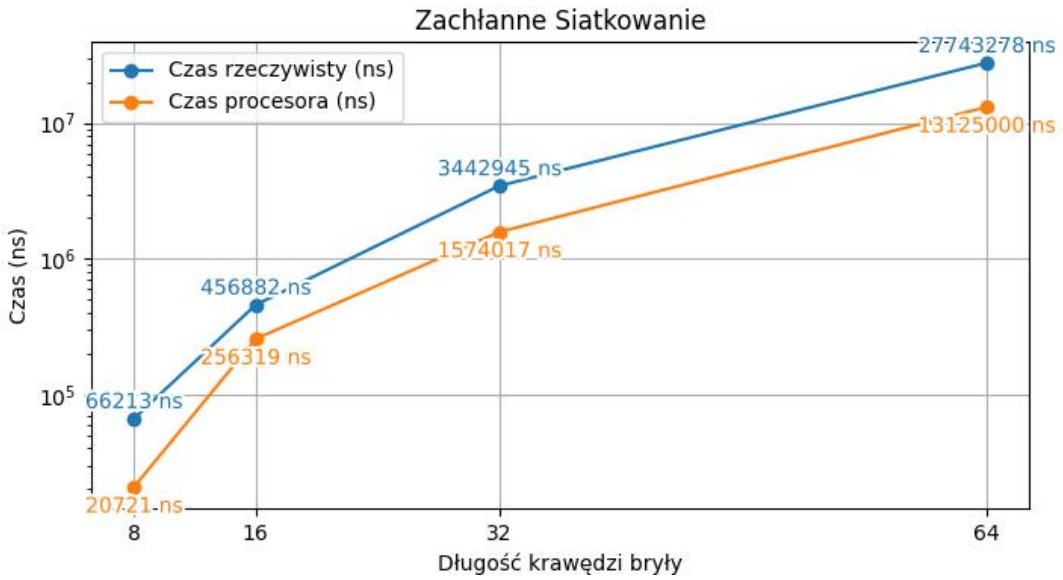
Tabela 4.8: Wyniki pomiaru wydajności dla wielu brył rysowanych metodą ukrywania powierzchni z wsparciem GPU w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Z tabeli 4.8 wynika, że wsparcie GPU w tym przypadku ma bezpośrednie przełożenie na zużycie pamięci, natomiast średnia liczba klatek na sekundę pozostaje niezmieniona. Wynika to z faktu, że algorytm wspiera proces budowania siatki po stronie CPU, przenosząc jego część na GPU. Liczba wynikowych wierzchołków po stronie CPU jest znacznie mniejsza, ale w procesie rysowania pozostaje taka sama.

Liczba brył	Wielkość bryły	Czas tworzenia brył	Czas budowy siatek	Czas przebudowy siatek	Łączny czas budowy sceny
18	16x16x16	4.43 ms (18 brył)	11.81 ms (51 siatek)	7.66ms (33 przebudowy)	20.08ms
18	32x32x32	27.97 ms (18 brył)	79.76 ms (51 siatek)	52.57 ms (33 przebudowy)	174.59 ms
405	16x16x16	233.98 us (405 brył)	176.07 ms (1449 siatek)	633.2 us (1044 przebudowy)	13 ms
405	32x32x32	350.89ms (405 brył)	1.26s (1449 siatek)	926.7 ms (1044 przebudowy)	3.59s
1445	16x16x16	180.78 ms (1445 brył)	634.64 ms (5321 siatek)	473.41ms (3876 przebudowy)	1.27s
1445	32x32x32	1.19s (1445 brył)	4.38 s (5321 siatek)	3.25s (3876 przebudowy)	12.29s

Tabela 4.9: Wyniki pomiaru czasu budowy sceny dla wielu brył rysowanych metodą ukrywania powierzchni z wsparciem GPU w scenie trwającej 1 minutę.

4.7.1.4 Zachłanne siatkowanie



Rysunek 4.7: Wynik mikrotestowania budowy siatki w metodzie zachłannego siatkowania.

Z wykresu mikrotestowania metody zachłannego siatkowania (patrz rysunek 4.7) wynika, że czas procesora stanowi blisko 45% czasu rzeczywistego. W przeciwieństwie do metody ukrywania powierzchni, najmniej czasu procesora zajmuje kostka 8x8x8 (31.29%), a

najwięcej kostka 16x16x16 (56.10%). Metoda zachłannego siatkowania zajmuje od 11.31% do 9.87% czasu rzeczywistego metody naiwnej oraz od 14.09% do 7.01% czasu procesora metody naiwnej. Wynik ten jest lepszy niż w przypadku metody ukrywania powierzchni, jednak zdecydowanie słabszy w porównaniu do ukrywania powierzchni z wsparciem GPU.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy siatki**	**Średnie użycie procesora**	**Zajętość pamięci**
8x8x8	14386 (±4711)	99.95 µs	1.59%	160 wierzchołków 3840 bajtów
16x16x16	13794 (±4915)	521.48 µs	3.06%	564 wierzchołki 13536 bajtów (0.01 MB)
32x32x32	13826 (±4691)	3.54 ms	2.34%	1940 wierzchołki 46560 bajtów (0.04 MB)
64x64x64	13427 (±4594)	30.95 ms	3.45%	14628 wierzchołków 351072 bajty (0.35 MB)

Tabela 4.10: Wyniki pomiaru wydajności dla jednej bryły rysowanej metodą zachłannego siatkowania w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Zgodnie z informacjami zawartymi w tabeli 4.10 można zauważyć, że średnia liczba klatek na sekundę wraz ze wzrostem bryły prawie się nie zmienia. Wręcz jest widoczna tendencja rosnąca wynikająca z niedoskonałości pomiaru.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało spadkiem średniej liczby klatek na sekundę o 4.12%.
- Przejście z bryły 16x16x16 do 32x32x32 skutkowało wzrostem średniej liczby klatek na sekundę o 0.23%.
- Przejście z bryły 32x32x32 do 64x64x64 skutkowało spadkiem średniej liczby klatek na sekundę o 2.89%.

Czas budowy siatki jest momentami nimalże trzykrotnie wyższy niż w przypadku metody ukrywania powierzchni z wsparciem GPU, ale za to dodatkowo zmniejsza zajętość pamięci niemalże o połowę względem metody ukrywania powierzchni z wsparciem GPU.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało wzrostem czasu budowy o 421.74% oraz wzrostem zajętości pamięci o 252.50%.

- Przejście z bryły 16x16x16 do 32x32x32 spowodowało wzrost czasu budowy o 578.84% oraz wzrost zajętości pamięci o 243.97%.
- Przejście z bryły 32x32x32 do 64x64x64 przyniosło wzrost czasu budowy o 774.29% oraz wzrost zajętości pamięci o 654.02%.

Oznacza to, że przejście z bryły 8x8x8 na bryłę 64x64x64 poskutkowało wzrostem średniej liczby klatek na sekundę o 6.66%, wzrostem czasu budowy siatki o 30865.48%, oraz wzrostem zajętości pamięci o 9042.5%.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Liczba wierzchołków**	**Zajętość pamięci**
18	16x16x16	13950 (±4056)	3380	81120 bajtów (0.08 MB)
18	32x32x32	13361 (±3847)	17912	429888 bajtów (0.42 MB)
405	16x16x16	3723 (±914)	23564	565536 bajtów (0.56 MB)
405	32x32x32	3945 (±780)	145200	3484800 bajtów (3.48 MB)
1445	16x16x16	1118 (±125)	76252	1830048 bajtów (1.83 MB)
1445	32x32x32	1121 (±94)	504532	12108768 bajtów (12.1 MB)

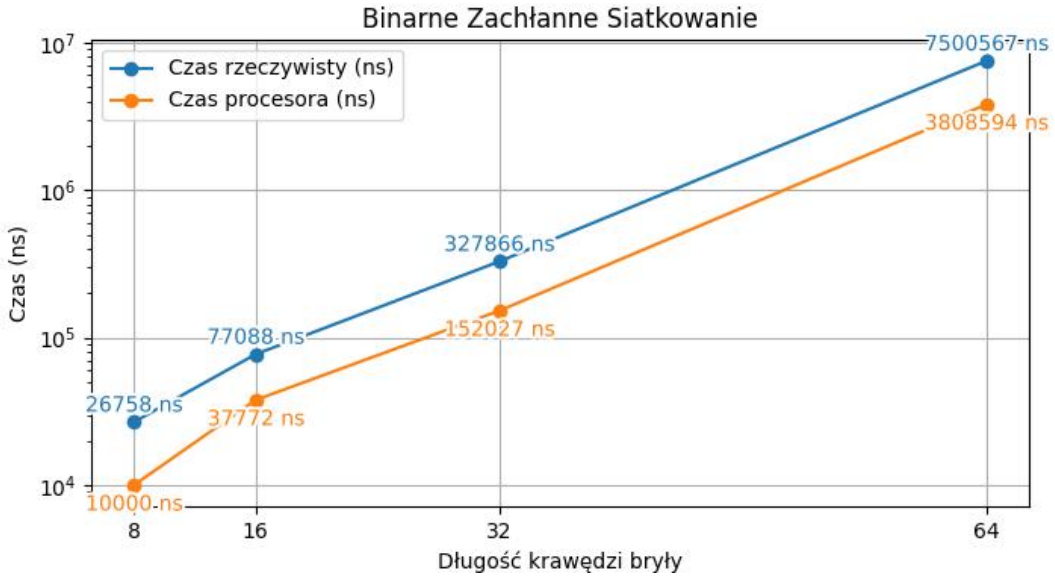
Tabela 4.11: Wyniki pomiaru wydajności dla wielu brył rysowanych metodą zachłannego siatkowania w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Tabela 4.11 pokazuje, że w pomiarze dla wielu brył zachłanne siatkowanie wykazuje przewagę w kwestii zajętości pamięci nad innymi technikami. Porównanie średniej liczby klatek na sekundę dla zachłannego siatkowania i technik ukrywania powierzchni nie wykazuje znaczącej poprawy, mimo zmniejszenia liczby wierzchołków. Ponadto, zwiększanie wielkości bryły nie wpływa znacząco na średnią liczbę klatek na sekundę. Sugeruje to, że głównym ograniczeniem nie jest bezpośrednio liczba wierzchołków, lecz liczba wywołań rysowania.

Liczba brył	Wielkość bryły	Czas tworzenia brył	Czas budowy siatek	Czas przebudowy siatek	Łączny czas budowy sceny
18	16x16x16	8.79ms (18 brył)	24.12 ms (51 siatek)	15.59ms (33 przebudowy)	31.99ms
18	32x32x32	67.66ms (18 brył)	185.2ms (51 siatek)	118.35ms (33 przebudowy)	271.68ms
405	16x16x16	229.13 ms (405 brył)	792.94ms (1449 siatek)	573.98ms (1044 przebudowy)	1.02s
405	32x32x32	1.49s (405 brył)	5.34s (1449 siatek)	3.87s (1044 przebudowy)	7.52s
1445	16x16x16	802.48 ms (1445 brył)	2.84s (5321 siatek)	2.08s (3876 przebudowy)	3.61s
1445	32x32x32	5.18s (1445 brył)	19.08s (5321 siatek)	13.94s (3876 przebudowy)	26.9 s

Tabela 4.12: Wyniki pomiaru czasu budowy sceny dla wielu brył rysowanych metodą zachłannego siatkowania w scenie trwającej 1 minutę.

4.7.1.5 Binarne zachłanne siatkowanie



Rysunek 4.8: Wynik mikrotestowania budowy siatki w metodzie binarnego zachłannego siatkowania.

Z wykresu mikrotestowania metody binarnego zachłannego siatkowania (patrz rysunek 4.8) wynika, że procent czasu procesora jest zbliżony do jego podstawowej wersji. Bryła o rozmiarze 8x8x8 zajmuje 37.37% czasu procesora względem czasu rzeczywistego, przy

średniej wynoszącej 45%. Metoda binarnego zachłannego siatkowania zajmuje jedynie od 4.57% do 0.94% czasu rzeczywistego metody naiwnej oraz od 3.38% do 0.85% czasu procesora metody naiwnej. Jest to zdecydowanie najszybsza metoda spośród wszystkich testowanych.

Pomiar dla bryły 64x64x64 nie został wykonany. Chociaż teoretycznie pomiar ten powinien być poprawny pod względem czasu wykonania, tak graficznie przedstawiłby jedynie artefakty. Algorytm binarnego zachłannego siatkowania jest zaimplementowany z użyciem zmiennej `uint64`, która jest niewystarczająca dla bryły o rozmiarze 64x64x64. Aktualnie użycie zmiennej `uint128` nie jest możliwe, ponieważ nie jest dostępna w standardzie *C++20*. Istnieje możliwość, że zmiana zmiennej w implementacji skutkowałaby zwiększonym czasem działania algorytmu.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy siatki**	**Średnie użycie procesora**	**Zajętość pamięci**
8x8x8	14331 (±4648)	50.57 µs	0.86%	160 wierzchołków 3840 bajtów
16x16x16	13776 (±4753)	134.72 µs	3.51%	572 wierzchołki 13728 bajtów (0.01 MB)
32x32x32	13838 (±4731)	508.48 µs	2.37%	1916 wierzchołków 45984 bajtów (0.04 MB)
64x64x64	—-	—-	—-	—-

Tabela 4.13: Wyniki pomiaru wydajności dla jednej bryły rysowanej metodą binarnego zachłannego siatkowania w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Zgodnie z informacjami zawartymi w tabeli 4.13 średnia liczba klatek na sekundę pozostaje zgodna z oryginalną wersją algorytmu zachłannego siatkowania.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało spadkiem średniej liczby klatek na sekundę o 3.87%.
- Przejście z bryły 16x16x16 do 32x32x32 skutkowało wzrostem średniej liczby klatek na sekundę o 0.45%.

Choć zajętość pamięci również jest identyczna jak w bazowej wersji tego algorytmu tak czas budowania siatki w przypadku bryły 32x32x32 jest aż o 85.63% niższy.

- Przejście z bryły 8x8x8 do 16x16x16 skutkowało wzrostem czasu budowy o 166.40% oraz wzrostem zajętości pamięci o 257.50%.

- Przejście z bryły 16x16x16 do 32x32x32 spowodowało wzrost czasu budowy o 277.43% oraz wzrost zajętości pamięci o 234.97%.

Oznacza to, że przejście z bryły 8x8x8 na bryłę 32x32x32 poskutkowało spadkiem średniej liczby klatek na sekundę o 3.44%, wzrostem czasu budowy siatki o 905.3%, oraz wzrostem zajętości pamięci o 1097.5%.

Binarne zachłanne siatkowanie nie zostało przetestowane dla bryły 64x64x64 z uwagi na ograniczenia algorytmu. W tym przypadku wymagane byłoby użycie zmiennej `uint128`, która nie jest dostępna w standardzie *C++20*. Implementacja oparta na `uint64` nie pozwala na obsługę bryły 64x64x64, co wykracza poza możliwości tej metody.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Liczba wierzchołków**	**Zajętość pamięci**
18	16x16x16	13520 (±4128)	3360	80640 bajtów (0.08 MB)
18	32x32x32	13380 (±3975)	17760	426240 bajtów (0.42 MB)
405	16x16x16	3783 (±897)	23548	565152 bajtów (0.56 MB)
405	32x32x32	3824 (±855)	145500	3492000 bajtów (3.49 MB)
1445	16x16x16	1076 (±157)	76052	1825248 bajtów (1.82 MB)
1445	32x32x32	1142 (±96)	505804	12139296 bajtów (12.13 MB)

Tabela 4.14: Wyniki pomiaru wydajności dla wielu brył rysowanych metodą binarnego zachłannego siatkowania w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Chociaż binarne zachłanne siatkowanie poprawia szybkość budowania siatek, zgodnie z tabelą 4.14 nie ma ono żadnego wpływu na średnią liczbę klatek na sekundę, liczbę wierzchołków ani zajętość pamięci w porównaniu z podstawową techniką zachłannego siatkowania.

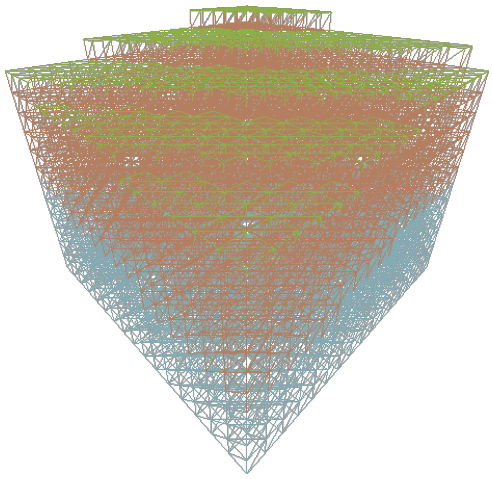
Liczba brył	Wielkość bryły	Czas tworzenia brył	Czas budowy siatek	Czas przebudowy siatek	Łączny czas budowy sceny
18	16x16x16	2.03 ms (18 brył)	4.93 ms (51 siatek)	3.14 ms (33 przebudowy)	13.14ms
18	32x32x32	10.21 ms (18 brył)	22.99 ms (51 siatek)	13.42 ms (33 przebudowy)	108.64 ms
405	16x16x16	50.28ms (405 brył)	157.56 ms (1449 siatek)	114.46 ms (1044 przebudowy)	378.05 ms
405	32x32x32	192.4 ms (405 brył)	683.47 ms (1499 siatek)	508.41 ms (1044 przebudowy)	3.14 s
1445	16x16x16	779.88 us (1445 brył)	559.77 ms (5321 siatek)	2.19 ms (3876 przebudowy)	50.68 ms
1445	32x32x32	612.02 ms (1445 brył)	2.21 s (5321 siatek)	1.65 s (3876 przebudowy)	9.82 s

Tabela 4.15: Wyniki pomiaru czasu budowy sceny dla wielu brył rysowanych metodą binarnego zachłannego siatkowania w scenie trwającej 1 minutę.

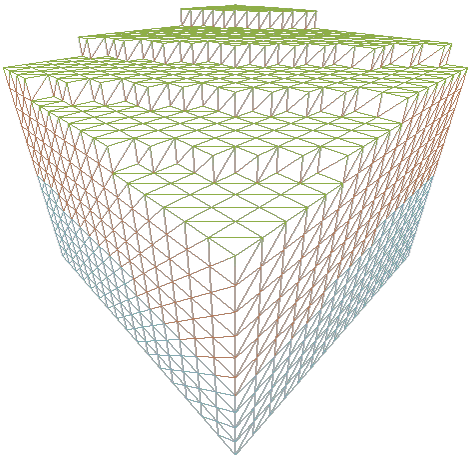
Zgodnie z tabelą 4.15, binarne zachłanne siatkowanie wykazuje najlepszy łączny czas budowania sceny na dużą skalę spośród wszystkich zbadanych technik. Dla najcięższego przypadku, obejmującego 1445 brył o wielkości 32x32x32, metoda ta wykazuje poprawę rzędu 88.18% względem zachłannego siatkowania, 48.57% względem ukrywania powierzchni ze wsparciem GPU, 65.03% względem ukrywania powierzchni oraz 98.19% względem metody naiwnej.

4.7.1.6 Porównanie jakości grafiki

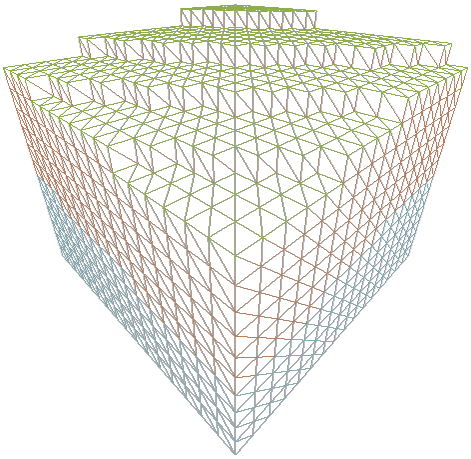
Na rysunkach 4.9, 4.10, 4.11, 4.12 i 4.13 przedstawiono wynikowe siatki brył wokseli dla badanych metod generowania siatki. Wyniki pokazują, że siatki uzyskane za pomocą metod ukrywania powierzchni i ukrywania powierzchni z wsparciem GPU charakteryzują się różną kolejnością wierzchołków. Natomiast w przypadku metod zachłannego siatkowania i binarnego zachłannego siatkowania, siatki są łączone w odmienny sposób, z zamienioną kolejnością łączenia w pionie i poziomie. Jest to wynik różnic w implementacji tych metod.



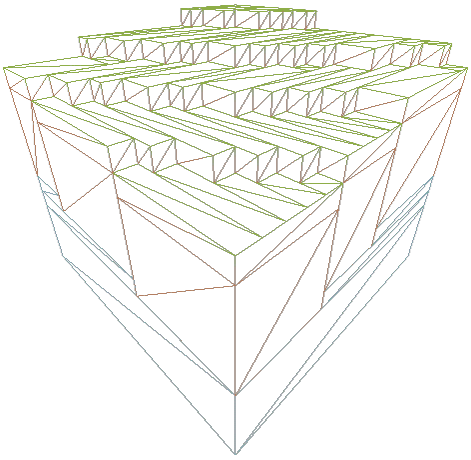
Rysunek 4.9: Siatka bryły wokseli wygenerowana przez metodę naiwną.



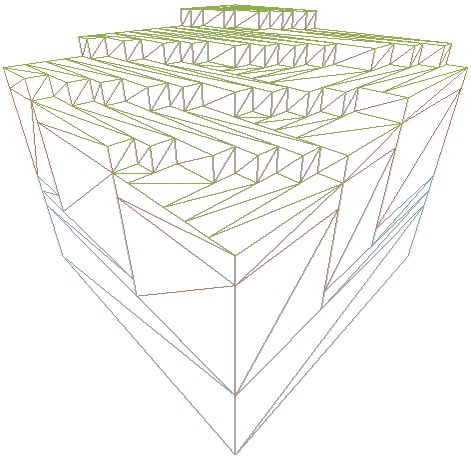
Rysunek 4.10: Siatka bryły wokseli wygenerowana przez metodę ukrywania powierzchni.



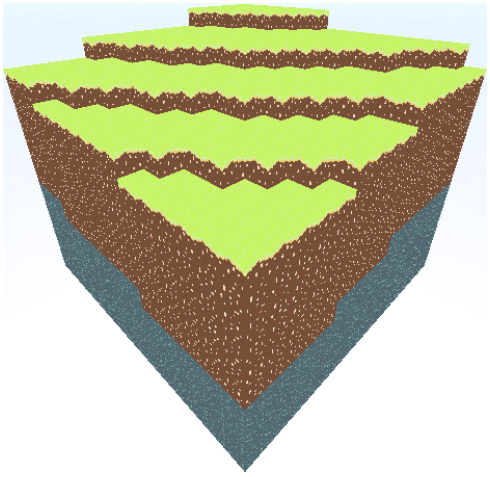
Rysunek 4.11: Siatka bryły wokseli wygenerowana przez metodę ukrywania powierzchni z wsparciem GPU.



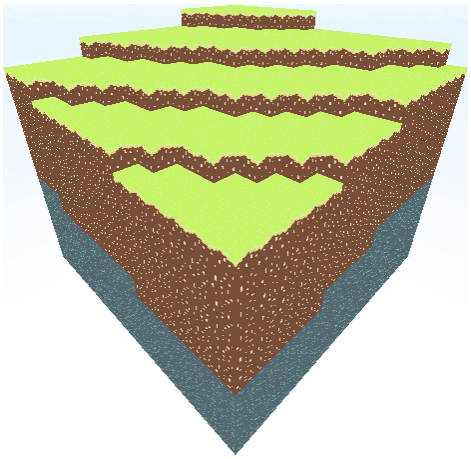
Rysunek 4.12: Siatka bryły wokseli wygenerowana przez metodę zachłannego siatkowania.



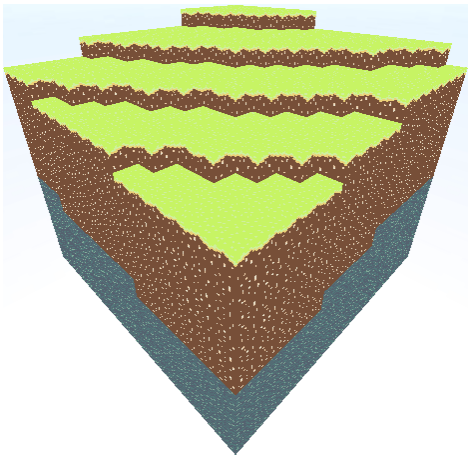
Rysunek 4.13: Siatka bryły wokseli wygenerowana przez metodę binarnego zachłannego siatkowania.



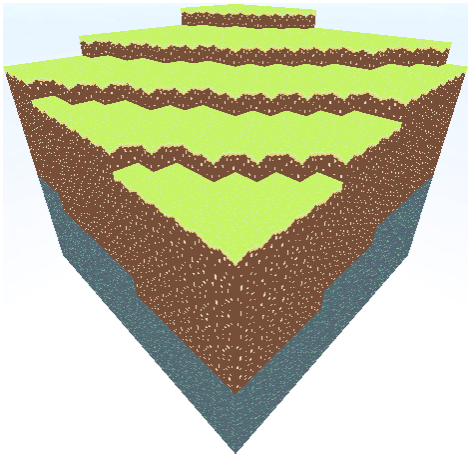
Rysunek 4.14: Bryła wokseli wygenerowana przez metodę naiwną.



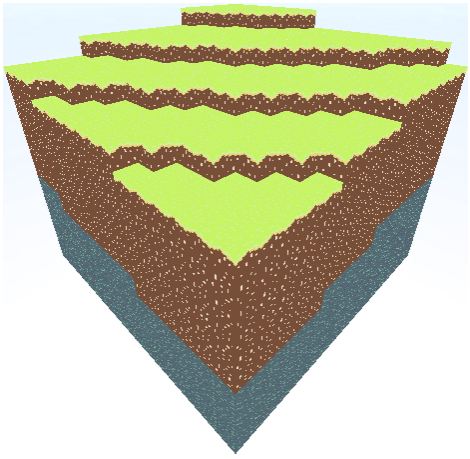
Rysunek 4.15: Bryła wokseli wygenerowana przez metodę ukrywania powierzchni.



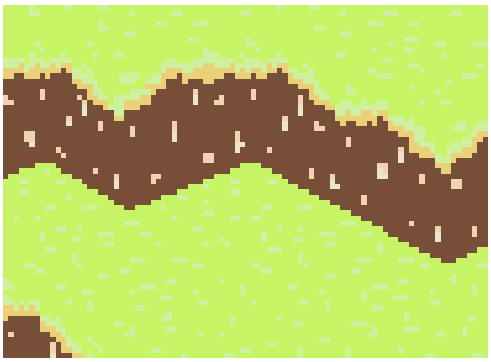
Rysunek 4.16: Bryła wokseli wygenerowana przez metodę ukrywania powierzchni z wsparciem GPU.



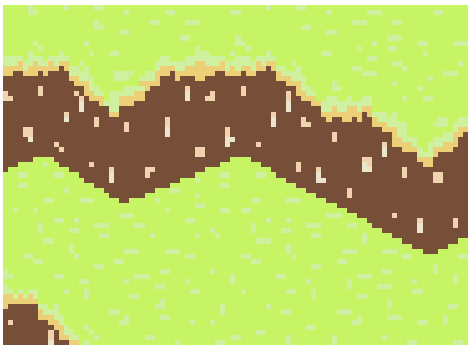
Rysunek 4.17: Bryła wokseli wygenerowana przez metodę zachłannego siatkowania.



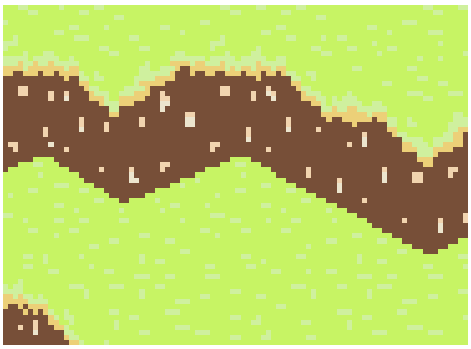
Rysunek 4.18: Bryła wokseli wygenerowana przez metodę binarnego zachłannego siatkowania.



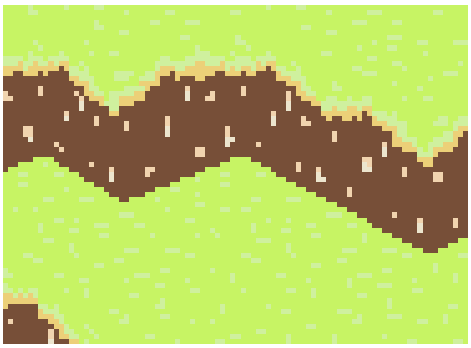
Rysunek 4.19: Przybliżenie na bryłę wokseli wygenerowaną przez metodę naiwną.



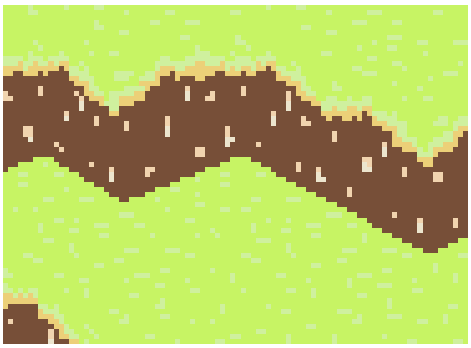
Rysunek 4.20: Przybliżenie na bryłę wokseli wygenerowaną przez metodę ukrywania powierzchni.



Rysunek 4.21: Przybliżenie na bryłę wokseli wygenerowaną przez metodę ukrywania powierzchni z wsparciem GPU.



Rysunek 4.22: Przybliżenie na bryłę wokseli wygenerowaną przez metodę zachłannego siatkowania.



Rysunek 4.23: Przybliżenie na bryłę wokseli wygenerowaną przez metodę binarnego zachłannego siatkowania.

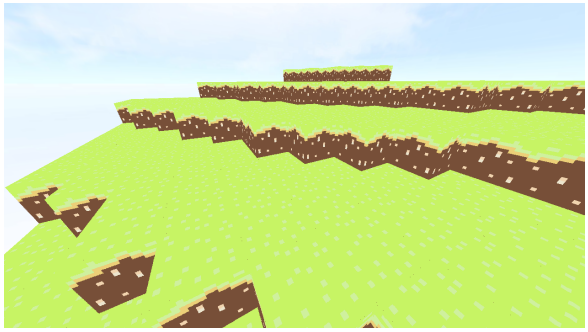
Na rysunkach 4.19, 4.20, 4.21, 4.22 i 4.23 przedstawiono wynikowe, powiększone obrazy brył wokseli dla badanych metod generowania siatki. Największe różnice stwierdzono w przypadku metody ukrywania powierzchni z wsparciem GPU, przedstawionej na rysunku 4.21. Metoda ta w tym przypadku charakteryzuje się nieznacznie odmiennym rysowaniem tekstury ziemi na bokach wokseli.

4.7.2 Śledzenie promieni

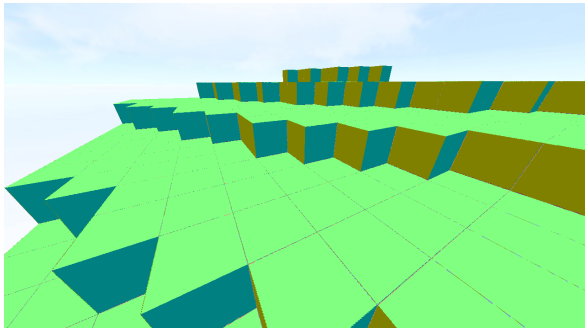
Technika śledzenia promieni została zbadana głównie pod kątem liczby iteracji promienia, zanim ten trafi w pełen woksel. Dodatkowo zebrano czasy budowania scen, średnią liczbę klatek na sekundę oraz zajętość pamięci. W porównaniu do triangulacji, porównanie wydajności tych technik przy użyciu klatek na sekundę czy czasu budowania jest trudne, ponieważ wyniki są zależne od implementacji i sprzętu. Mimo to, wyniki zostaną również ocenione pod kątem liczby klatek na sekundę, gdyż szybkość renderowania zawsze będzie zależna od tych czynników. Implementacja tych technik w niniejszej pracy mogłaby być bardziej efektywna i opierać się na bardziej wydajnym podejściu, co może wpływać na jakość pomiarów średniej liczby klatek na sekundę. Liczba iteracji promienia jest mniej zależna od tych zmiennych, co czyni ją bardziej stabilną miarą wydajności.

4.7.2.1 Stały promień

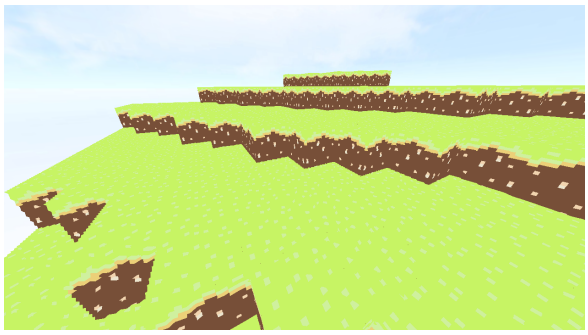
Chociaż zastosowanie stałego promienia jest najprostszym rozwiązaniem w implementacji, niestety prowadzi do problemów graficznych wynikających z pomijania wokseli. Problemy te są widoczne dla każdego rozsądnie dobranego kroku, a nawet dla bardzo małych kroków, takich jak 0.05 (patrz rysunek 4.27 i rysunek 4.26), co skutkuje ogromną liczbą iteracji promienia przy jednoczesnym braku idealnej jakości obrazu. Małe kroki również mają ogromny wpływ na wynikową liczbę klatek na sekundę (patrz tabele 4.16, 4.17, 4.18).



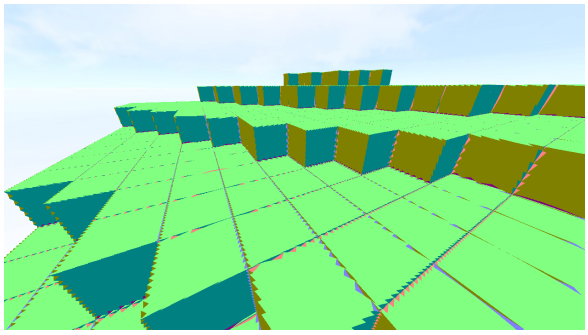
Rysunek 4.24: Scena z teksturami rysowana stałym promieniem o długości 0.01. Łączna liczba iteracji promieni wyniosła 658317322.



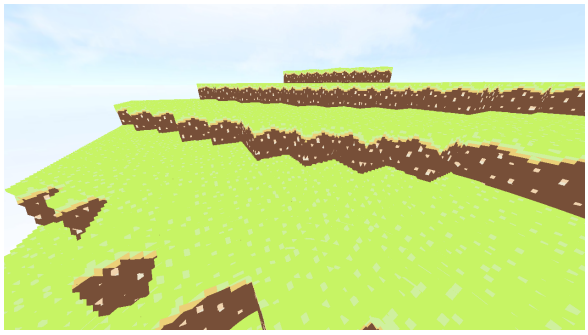
Rysunek 4.25: Scena z normalnymi rysowana stałym promieniem o długości 0.01. Łączna liczba iteracji promieni wyniosła 658317322.



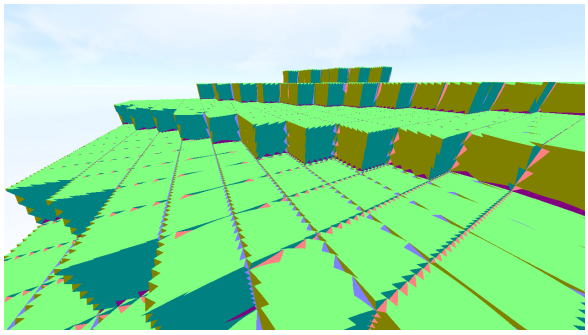
Rysunek 4.26: Scena z teksturami rysowana stałym promieniem o długości 0.05. Łączna liczba iteracji promieni wyniosła 130390387.



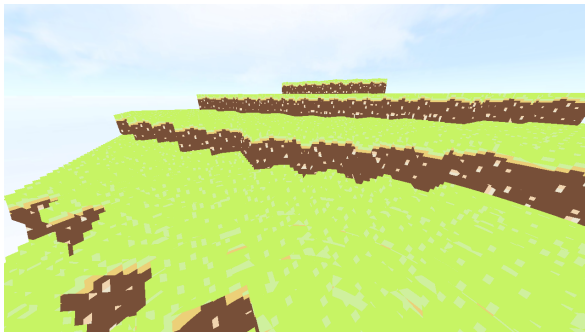
Rysunek 4.27: Scena z normalnymi rysowana stałym promieniem o długości 0.05. Łączna liczba iteracji promieni wyniosła 130390387.



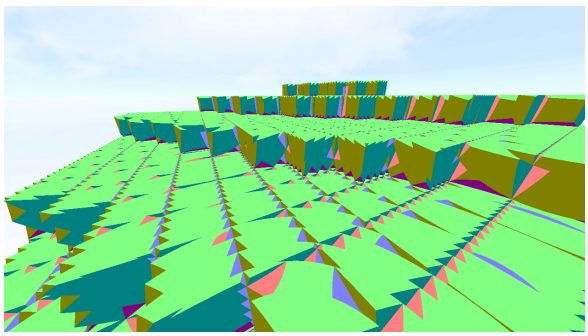
Rysunek 4.28: Scena z teksturami rysowana stałym promieniem o długości 0.1. Łączna liczba iteracji promieni wyniosła 66209105.



Rysunek 4.29: Scena z normalnymi rysowana stałym promieniem o długości 0.1. Łączna liczba iteracji promieni wyniosła 66209105.



Rysunek 4.30: Scena z teksturami rysowana stałym promieniem o długości 0.2. Łączna liczba iteracji promieni wyniosła 33831991.



Rysunek 4.31: Scena z normalnymi rysowana stałym promieniem o długości 0.2. Łączna liczba iteracji promieni wyniosła 33831991.

Wielkość bryły	Średnia klatek na sekunde (± SD)	Czas budowy struktury	Średnia liczba iteracji promieni	Zajętość pamięci
8x8x8	3364 (±499)	12 µs	2316728 (±263732)	2048 bajtów
16x16x16	2982 (±410)	269 µs	6505990 (±730598)	16384 bajtów (0.01 MB)
32x32x32	2387 (±321)	789 µs	15109949 (±1044053)	131072 bajtów (0.13 MB)
64x64x64	1195 (±143)	3.595 ms	43138504 (±1288777)	1048576 bajtów (1.04 MB)

Tabela 4.16: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu stałego kroku promienia o długości 0.1 w scenie trwającej 1 minutę.

Wielkość bryły	Średnia klatek na sekunde (± SD)	Czas budowy struktury	Średnia liczba iteracji promieni	Zajętość pamięci
8x8x8	3149 (±399)	12 µs	4493668 (±521649)	2048 bajtów
16x16x16	2419 (±335)	58 µs	12800911 (±1465517)	16384 bajtów (0.01 MB)
32x32x32	1670 (±231)	694 µs	29954402 (±2095507)	131072 bajtów (0.13 MB)
64x64x64	702 (±68)	3.469 ms	86059309 (±2571503)	1048576 bajtów (1.04 MB)

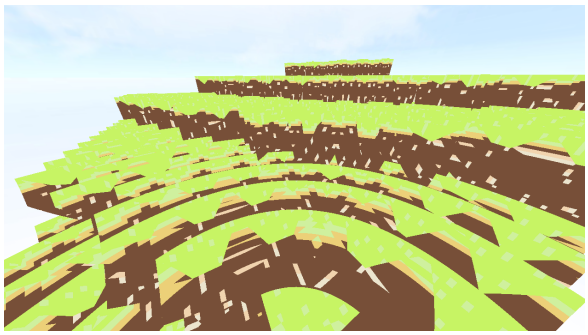
Tabela 4.17: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu stałego kroku promienia o długości 0.05 w scenie trwającej 1 minutę.

Wielkość bryły	Średnia klatek na sekunde (± SD)	Czas budowy struktury	Średnia liczba iteracji promieni	Zajętość pamięci
8x8x8	1696 (±230)	11 µs	21772413 (±2677311)	2048 bajtów
16x16x16	942 (±143)	58 µs	62816572 (±7478404)	16384 bajtów (0.01 MB)
32x32x32	487 (±58)	432 µs	148828049 (±10376893)	131072 bajtów (0.13 MB)
64x64x64	164 (±9.45)	5.008 ms	429281895 (±12881198)	1048576 bajtów (1.04 MB)

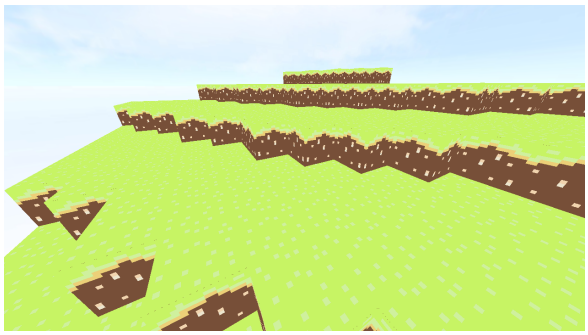
Tabela 4.18: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu stałego kroku promienia o długości 0.01 w scenie trwającej 1 minutę.

4.7.2.2 Algorytm szybkiego przechodzenia przez woksele

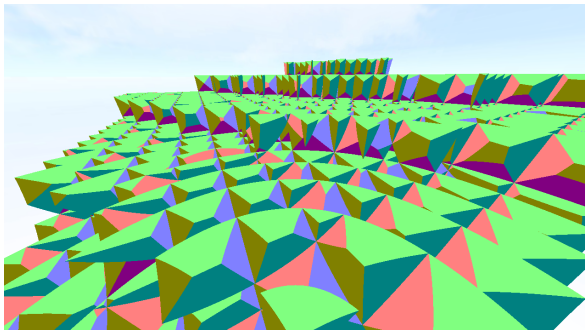
Algorytm szybkiego przechodzenia przez woksele osiągnął doskonałe wyniki zarówno pod względem wizualnym, jak i wydajnościowym. Aby narysować scenę widoczną na rysunku 4.33 przy użyciu stałego kroku, zachowując podobną liczbę iteracji promienia dla tej konkretnej sceny, dobrano krok o długości 0.707 (wynikowa liczba iteracji promienia to około 10476236 iteracji). Efekty oraz problemy związane z stałym krokiem dla tej ilości iteracji są widoczne na rysunku 4.32 szczególnie w przypadku wokseli znajdujących się blisko kamery.



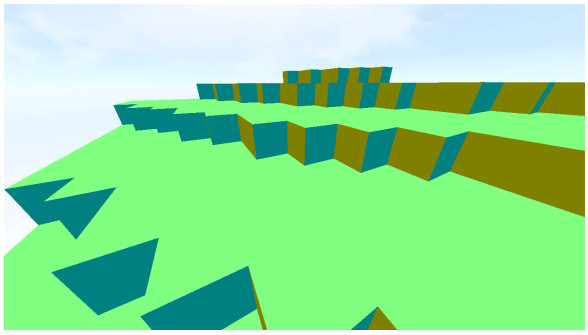
Rysunek 4.32: Scena z teksturami rysowana stałym promieniem o długości 0.707



Rysunek 4.33: Scena z teksturami rysowana przy użyciu algorytmu szybkiego przemierzania wokseli.



Rysunek 4.34: Scena z normalnymi rysowana stałym promieniem o długości 0.707



Rysunek 4.35: Scena z normalnymi rysowana przy użyciu algorytmu szybkiego przemierzania wokseli.

Wielkość bryły	Średnia klatek na sekunde (± SD)	Czas budowy struktury	Średnia liczba iteracji promieni	Zajętość pamięci
8x8x8	3456 (±586)	20 µs	406024 (±39170)	2048 bajtów
16x16x16	3362 (±575)	58 µs	1000743 (±114928)	16384 bajtów (0.01 MB)
32x32x32	3300 (±568)	518 µs	2195942 (±174189)	131072 bajtów (0.13 MB)
64x64x64	2839 (±414)	3.634 ms	6208756 (±259184)	1048576 bajtów (1.04 MB)

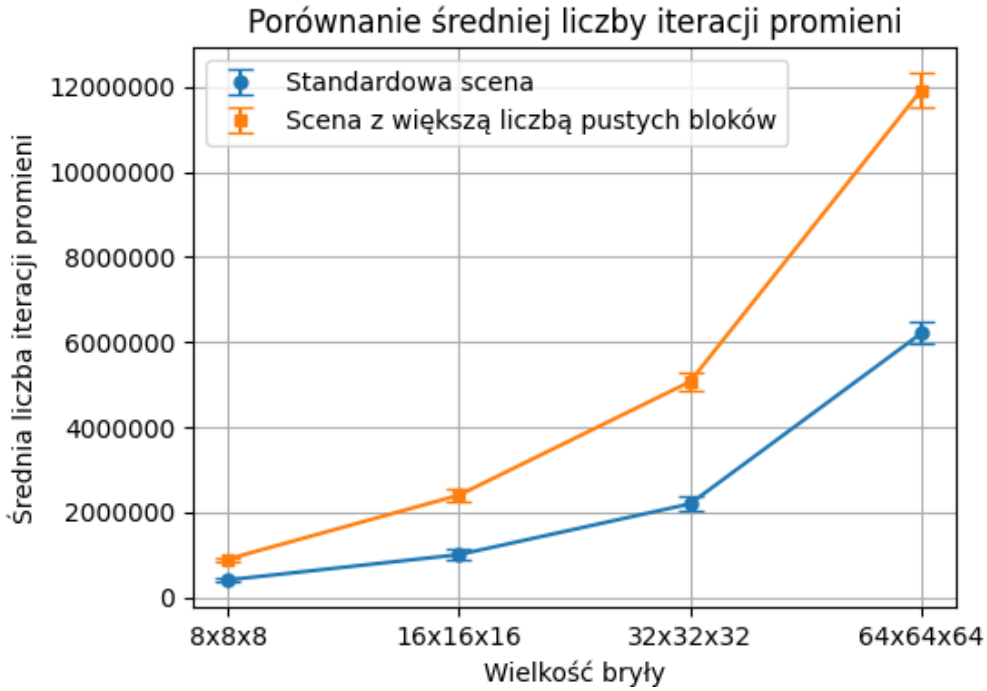
Tabela 4.19: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu algorytmu szybkiego przechodzenia przez woksele w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Wielkość bryły	Średnia klatek na sekunde (± SD)	Czas budowy struktury	Średnia liczba iteracji promieni	Zajętość pamięci
8x8x8	3659 (±452)	13 µs	893516 (±48084)	2048 bajtów
16x16x16	3412 (±491)	75 µs	2400339 (±139968)	16384 bajtów (0.01 MB)
32x32x32	3154 (±451)	486 µs	5071183 (±229623)	131072 bajtów (0.13 MB)
64x64x64	2570 (±349)	4.343 ms	11933920 (±407905)	1048576 bajtów (1.04 MB)

Tabela 4.20: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu algorytmu szybkiego przechodzenia przez woksele w scenie trwającej 1 minutę z większą ilością pustych bloków, gdzie SD oznacza odchylenie standardowe.

Porównując tabele 4.20 oraz 4.19 można zauważyć, że puste obszary niekorzystnie wpływają na średnią liczbę iteracji promieni, co prowadzi do obniżenia wydajności. W przypadku bardziej zapełnionego terenu, promień średnio pokonuje krótszą trasę, co przekłada się na lepszą efektywność. Tą zależnośc można również zobaczyć na rysunku 4.36.

Tabela 4.21 porównująca wydajność dla wielu brył wskazuje, że technika ta słabo radzi sobie w reprezentacji dużych światów, zarówno pod względem średniej liczby klatek na sekundę, jak i zajętości pamięci.



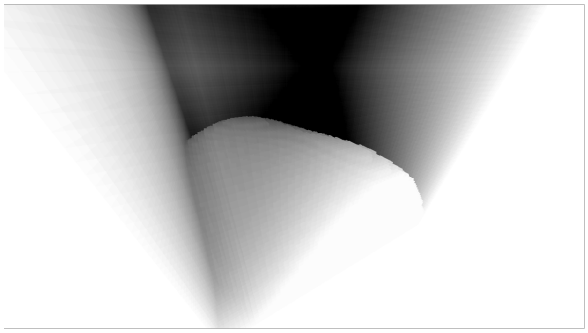
Rysunek 4.36: Porównanie średniej liczby iteracji promieni w zależności od wielkości bryły dla algorytmu szybkiego przechodzenia przez woksele.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
18	16x16x16	305 (±33)	1449854 (±209563)	294912 bajtów (0.29 MB)
18	32x32x32	300 (±32)	3729011 (±279686)	2359296 bajtów (2.35 MB)
405	16x16x16	15.01 (±1.36)	14873756 (±536618)	6635520 bajtów (6.63 MB)
405	32x32x32	15.15 (±0.84)	30082315 (±1082755)	53084160 bajtów (53.08 MB)
1445	16x16x16	4.40 (±0.03)	13978692 (±518482)	23674880 bajtów (23.67 MB)
1445	32x32x32	4.41 (±0.08)	28183415 (±1034064)	189399040 bajtów (189.39 MB)

Tabela 4.21: Wyniki pomiaru wydajności dla wielu brył rysowanych przy użyciu algorytmu szybkiego przechodzenia przez woksele w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

4.7.2.3 Mapa cegiełkowa

Mapa cegiełkowa jest prosta w implementacji, jednak jej potencjalnym minusem jest fakt, że jeden woksel wymaga przechowywania i renderowania całej mapy cegiełkowej, w której się znajduje. Aby prześledzić ścieżkę promienia, można zbierać informacje o iteracjach promienia i na tej podstawie, w jednostce cieniującej fragmentów, rysować piksel bardziej zacieniony, gdy liczba iteracji jest duża, oraz jaśniejszy, gdy jest ich mało. Na rysunkach 4.37 oraz 4.38 przedstawiono porównanie liczby iteracji w mapie cegiełkowej względem surowego formatu przetwarzanego przez algorytm szybkiego przechodzenia przez woksele.



Rysunek 4.37: Mapa iteracji promieni dla surowego formatu wokseli.

Rysunek 4.38: Mapa iteracji promieni dla map cegiełkowych.

Dla surowego formatu widać znaczne ilości iteracji promienia w pustych przestrzeniach nad terenem. Jest to spowodowane tym, że promień pokonuje całą bryłę, nie trafiając w żaden woksel. Liczba iteracji w takich miejscach jest znacznie większa niż w przypadku przedwczesnego trafienia w woksel. Z drugiej strony, mapa cegiełkowa skutecznie eliminuje tę wadę. W takim przypadku promień pokonuje pustą mapę cegiełkową o wielkości 8x8x8 w jednym kroku w efekcie przemieszczając się przez puste przestrzenie efektywnie. Kiedy jednak mapa cegiełkowa zawiera choćby jeden woksel, promień iteruje woksel po wokselu przy użyciu algorytmu szybkiego przechodzenia przez woksele. Wynikiem tego zachowania są charakterystyczne bryły o wielkości 8x8x8 w okolicach terenu. Na rysunku 4.38 można również zauważyć charakterystyczną czarną poświatę w pobliżu terenu. Są to miejsca, po których promień iteruje najwięcej, ponieważ znajdują się wystarczająco blisko terenu, aby utworzyć mapę cegiełkową, ale promień przemierza puste woksele w jej strukturze.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy struktury**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
8x8x8	3448 (±507)	10 µs	406612 (±38657)	2052 bajtów
16x16x16	3264 (±471)	74 µs	1000821 (±114074)	16388 bajtów (0.01 MB)
32x32x32	2917 (±408)	575 µs	2046083 (±95108)	129032 bajtów (0.12 MB)
64x64x64	1591 (±259)	5.061 ms	1717605 (±76601)	806976 bajtów (0.80 MB)

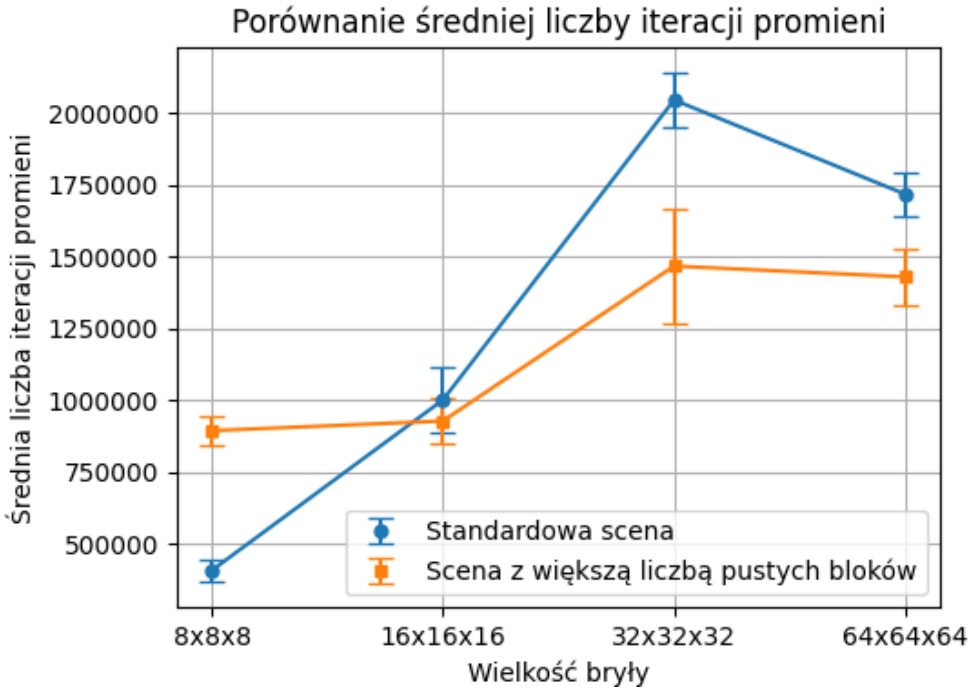
Tabela 4.22: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu struktury map cegiełkowych w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy struktury**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
8x8x8	3389 (±487)	9 µs	893848 (±47931)	2052 bajtów
16x16x16	2966 (±436)	59 µs	927532 (±78856)	10244 bajtów (0.01 MB)
32x32x32	2222 (±324)	431 µs	1467372 (±197930)	77832 bajtów (0.07 MB)
64x64x64	1437 (±237)	3.262 ms	1429702 (±96231)	440384 bajtów (0.44 MB)

Tabela 4.23: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu struktury map cegiełkowych w scenie trwającej 1 minutę z większą ilością pustych bloków, gdzie SD oznacza odchylenie standardowe.

Tabele 4.23 oraz 4.22 wskazują, że sceny zawierające wiele pustych przestrzeni korzystnie wpływają na wydajność poprzez zmniejszenie średniej liczby iteracji promieni. Dodatkowo, takie sceny znacząco redukują zajętość pamięci, gdyż puste przestrzenie o rozmiarach cegiełki nie są przechowywane w pamięci. Zależność tę można zaobserwować na rysunku 4.39.

Tabela 4.24 wskazuje, że mimo zmniejszenia zajętości pamięci o połowę w porównaniu do algorytmu szybkiego przechodzenia przez woksele, rysowanie dużych światów w czasie rzeczywistym wciąż nie jest możliwe bez zastosowania dodatkowych optymalizacji.



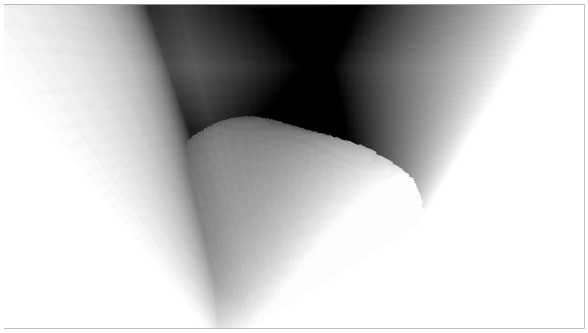
Rysunek 4.39: Porównanie średniej liczby iteracji promieni w zależności od wielkości bryły dla map cegiełkowych.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
18	16x16x16	299 (±27)	1290239 (±110909)	288840 bajtów (0.28 MB)
18	32x32x32	251 (±25)	1673396 (±49648)	2046096 bajtów (2.04 MB)
405	16x16x16	14.83 (±1.37)	2346022 (±79567)	2476680 bajtów (2.47 MB)
405	32x32x32	14.21 (±0.81)	2580726 (±81017)	18277648 bajtów (18.27 MB)
1445	16x16x16	4.39 (±0.05)	2264031 (±82018)	8882440 bajtów (8.88 MB)
1445	32x32x32	4.27 (±0.04)	2459154 (±92415)	64885264 bajtów (64.88 MB)

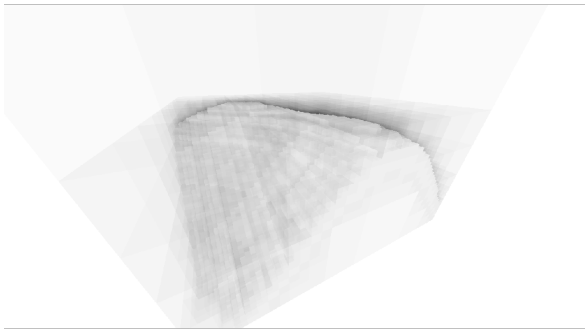
Tabela 4.24: Wyniki pomiaru wydajności dla wielu brył rysowanych przy użyciu map cegiełkowych w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

4.7.2.4 Drzewo ósemkowe

Drzewa ósemkowe są zdecydowanie bardziej złożone w implementacji, a ich wydajność jest silnie uzależniona od jakości tej implementacji. Dzięki temu, że drzewa ósemkowe nie są ograniczone do surowego formatu, jak ma to miejsce w przypadku map cegiełkowych, to mogą znacznie oszczędzać pamięć. W sytuacji, gdy jakieś poddrzewo składa się wyłącznie z jednego typu bloku, to nie są one przechowywane osobno, a jedynie sam liść niesie w sobie informacje jak powinien narysować swoją powierzchnie niezależnie od poziomu poddrzewa. Na rysunkach 4.40 oraz 4.41 przedstawiono porównanie liczby iteracji w drzewie ósemkowym względem surowego formatu przetwarzanego przez algorytm szybkiego przechodzenia przez woksele.



Rysunek 4.40: Mapa iteracji promieni dla surowego formatu wokseli.



Rysunek 4.41: Mapa iteracji promieni dla drzew ósmkowych.

Rysunek 4.41 przedstawia charakterystyczne podziały drzew ósemkowych na osiem równych części, przypominające sześciany w sześcianach, które stopniowo stają się mniejsze i bardziej szczegółowe. Puste przestrzenie, takie jak te nad terenem, zawierają mniej podziałów, natomiast im bliżej terenu, tym podziałów jest więcej. W przeciwieństwie do map cegiełkowych, nie występują tutaj wyróżniające się, agresywne sześciany. Czarna poświata wokół terenu jest również odpowiednio mniej intensywna.

Tabela 4.25 oraz tabela 4.26 wskazują, że średnia liczba promieni dla sceny standardowej i tej z większą liczbą pustych bloków jest zbliżona. Przewaga sceny z większą liczbą pustych bloków staje się widoczna dopiero przy bryle o wielkości 64x64x64. Zajętość pamięci jest przedstawiona na dwa sposoby — wartość oznaczona gwiazdką reprezentuje wartość teoretyczną, natomiast wartość bez gwiazdki odnosi się do wartości uzyskanej w badanej implementacji. Taki podział wynika z faktu, że w aktualnej implementacji struktura węzła posiada dodatkowe wypełnienie (ang. *Padding*). W idealnej wersji, poprzez lepsze przekazanie danych do karty graficznej, możliwe jest wyeliminowanie tego wypełnienia. Dlatego w tabeli przedstawiono również wartość teoretyczną.

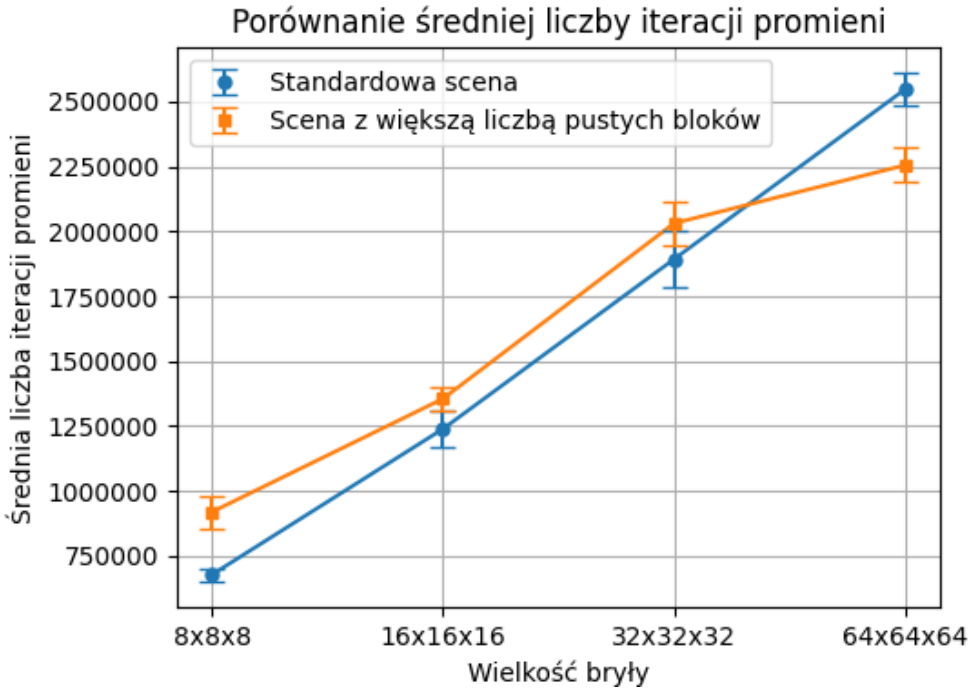
W tabeli 4.27 przedstawiającej pomiary dla wielu brył, można zauważyć, że teoretyczna wartość zajętości pamięci może być momentami lepsza niż w przypadku mapy cegiełkowej.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy struktury**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
8x8x8	2222 (±311)	49 µs	677376 (±26040)	11952 bajtów (*8964 bajtów)
16x16x16	1585 (±207)	475 µs	1239229 (±72332)	62256 bajtów (*46692 bajtów) (0.06 MB)(*0.04 MB)
32x32x32	1105 (±124)	1.47 ms	1893346 (±106942)	251184 bajtów (*188388 bajtów) (0.25 MB)(*0.18 MB)
64x64x64	696 (±90)	11.503 ms	2548140 (±63403)	1285296 bajtów (*963972 bajtów) (1.28 MB)(*0.96 MB)

Tabela 4.25: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu struktury drzew ósemkowych w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

Wielkość bryły	**Średnia klatek na sekunde (± SD)**	**Czas budowy struktury**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
8x8x8	2051 (±322)	43 µs	919147 (±64098)	10032 bajtów (*7524 bajtów)
16x16x16	1521 (±214)	311 µs	1355500 (±47654)	57648 bajtów (*43236 bajtów) (0.05 MB)(*0.04 MB)
32x32x32	1076 (±135)	2.573 ms	2031532 (±83584)	252720 bajtów (*189540 bajtów) (0.25 MB)(*0.18 MB)
64x64x64	738 (±101)	10.681 ms	2255435 (±66933)	1233072 bajtów (*924804 bajtów) (1.23 MB)(*0.92 MB)

Tabela 4.26: Wyniki pomiaru wydajności dla jednej bryły rysowanej przy użyciu struktury drzew ósemkowych w scenie trwającej 1 minutę z większą ilością pustych bloków, gdzie SD oznacza odchylenie standardowe.



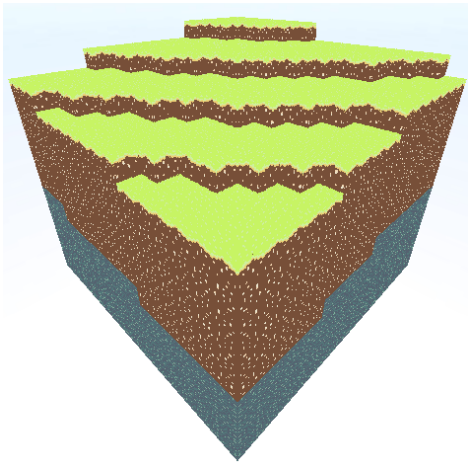
Rysunek 4.42: Porównanie średniej liczby iteracji promieni w zależności od wielkości bryły dla drzew oktalnych.

Liczba brył	**Wielkość bryły**	**Średnia klatek na sekundę (± SD)**	**Średnia liczba iteracji promieni**	**Zajętość pamięci**
18	16x16x16	210 (±19)	1570773 (±100584)	511968 bajtów (*383976 bajtów) (0.51 MB)(*0.38 MB)
18	32x32x32	178 (±16)	2093435 (±54204)	2347104 bajtów (*1760328 bajtów) (2.34 MB)(*1.76 MB)
405	16x16x16	12.09 (±0.89)	2692590 (±85933)	4727664 bajtów (*3545748 bajtów) (4.72 MB)(*3.54 MB)
405	32x32x32	11.68 (±0.51)	3070067 (±95060)	21142512 bajtów (*15856884 bajtów) (21.14 MB)(*15.85 MB)
1445	16x16x16	3.66 (±0.04)	2632766 (±96318)	16831728 bajtów (*12623796 bajtów) (16.83 MB)(*12.62 MB)
1445	32x32x32	3.52 (±0.03)	2981109 (±110025)	75493104 bajtów (*56619828 bajtów) (75.49 MB)(*56.61 MB)

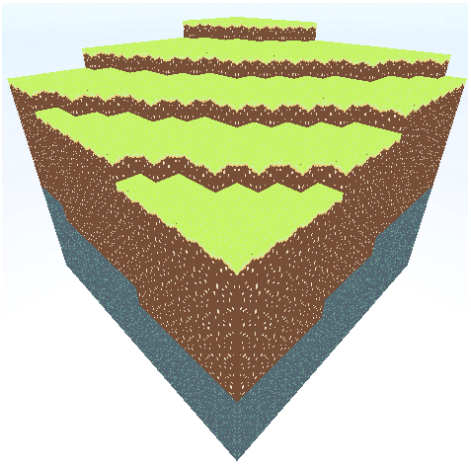
Tabela 4.27: Wyniki pomiaru wydajności dla wielu brył rysowanych przy użyciu drzew oktalnych w scenie trwającej 1 minutę, gdzie SD oznacza odchylenie standardowe.

4.7.2.5 Porównanie jakości grafiki

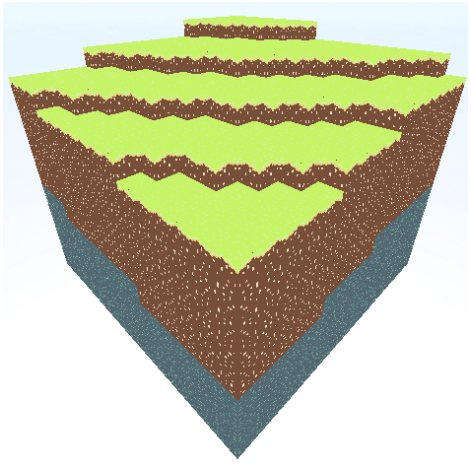
Na rysunkach 4.43, 4.44, 4.45 i 4.46 przedstawiono wynikowe obrazy brył wokseli dla badanych metod śledzenia promieni. Jakość obrazów uzyskanych przy użyciu wszystkich tych metod jest niemal identyczna, z wyjątkiem metody stałego kroku, gdzie zauważalne są nienaturalności w przejściach między wokselami.



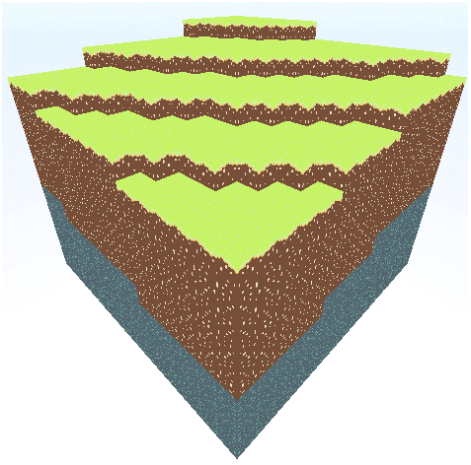
Rysunek 4.43: Bryła wokseli wygenerowana metodą stałego kroku promienia.



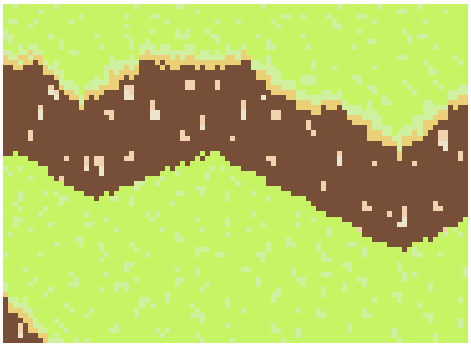
Rysunek 4.44: Bryła wokseli wygenerowana przy pomocy szybkiego algorytmu przechodzenia przez woksele.



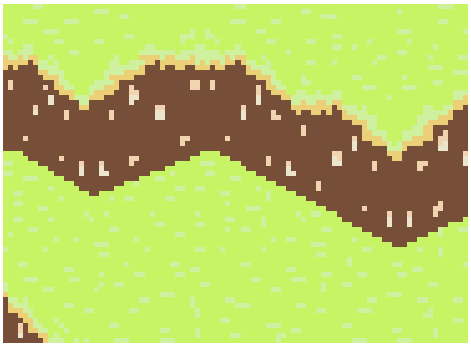
Rysunek 4.45: Bryła wokseli wygenerowana przy pomocy struktury map cegiełkowych.



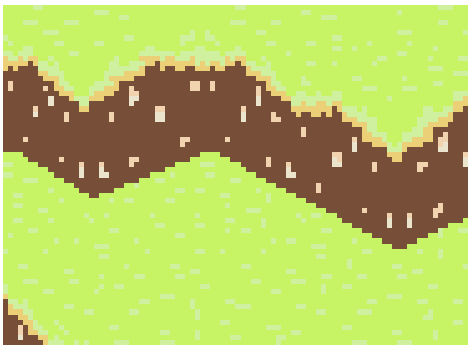
Rysunek 4.46: Bryła wokseli wygenerowana przy pomocy struktury drzew ósemkowych.



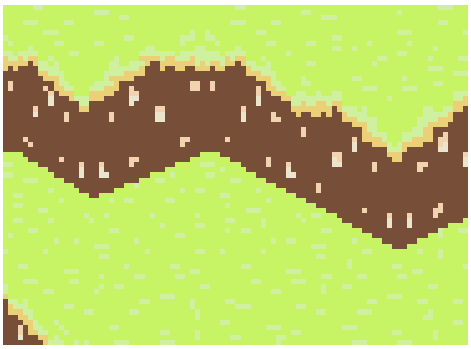
Rysunek 4.47: Przybliżenie na bryłę wokseli wygenerowaną metodą stałego kroku promienia.



Rysunek 4.48: Przybliżenie na bryłę wokseli wygenerowaną przy pomocy szybkiego algorytmu przechodzenia przez woksele.



Rysunek 4.49: Przybliżenie na bryłę wokseli wygenerowaną przy pomocy struktury map cegiełkowych.



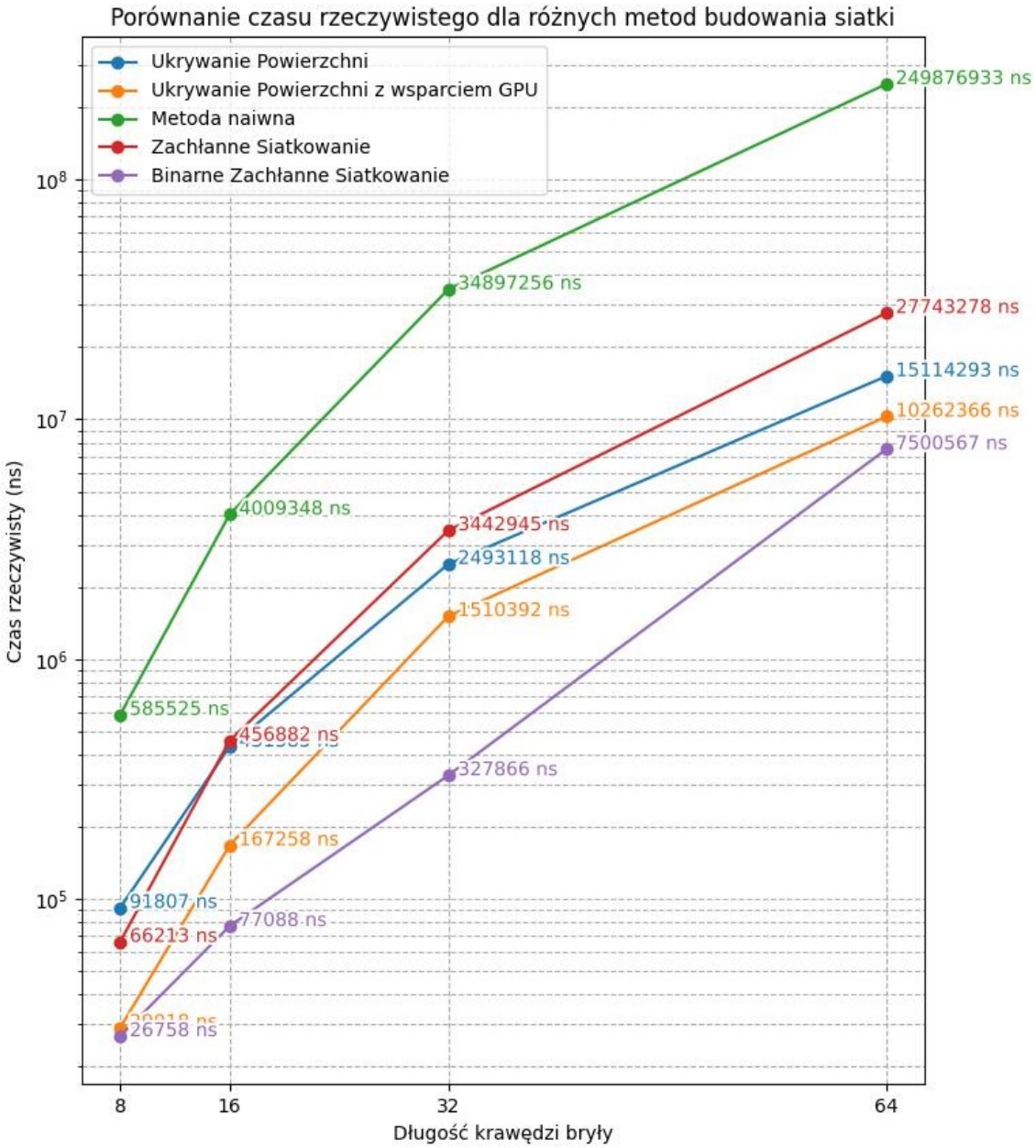
Rysunek 4.50: Przybliżenie na bryłę wokseli wygenerowaną przy pomocy struktury drzew ósemkowych.

Na rysunkach 4.47, 4.48, 4.49 i 4.50 przedstawiono wynikowe, powiększone obrazy brył wokseli dla badanych metod śledzenia promieni. Największe różnice stwierdzono w przypadku metody stałego kroku promienia, przedstawionej na rysunku 4.47. Metoda ta w tym przypadku charakteryzuje się odmiennym rysowaniem tekstury ziemi na bokach wokseli oraz na nierównościach w przejściu między wokselami.

4.8 Interpretacja wyników

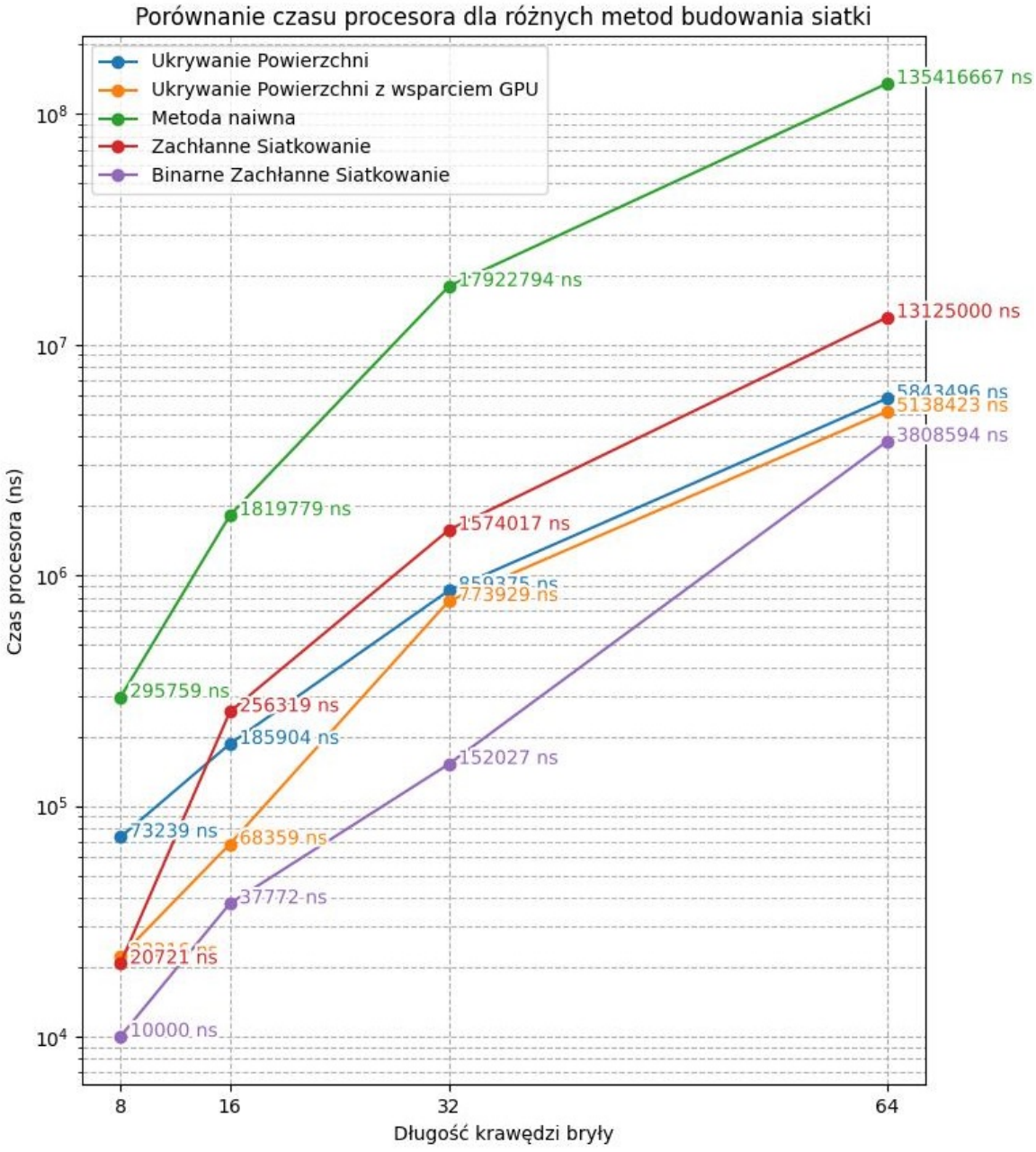
Wyniki są interpretowane oddzielnie dla triangulacji i śledzenia promieni, zestawiając wszystkie przebadane techniki i porównując je w ramach każdej kategorii. Następnie, w sekcji 4.9 przedstawione są wspólne wnioski wynikające z obu technik.

4.8.1 Triangulacja



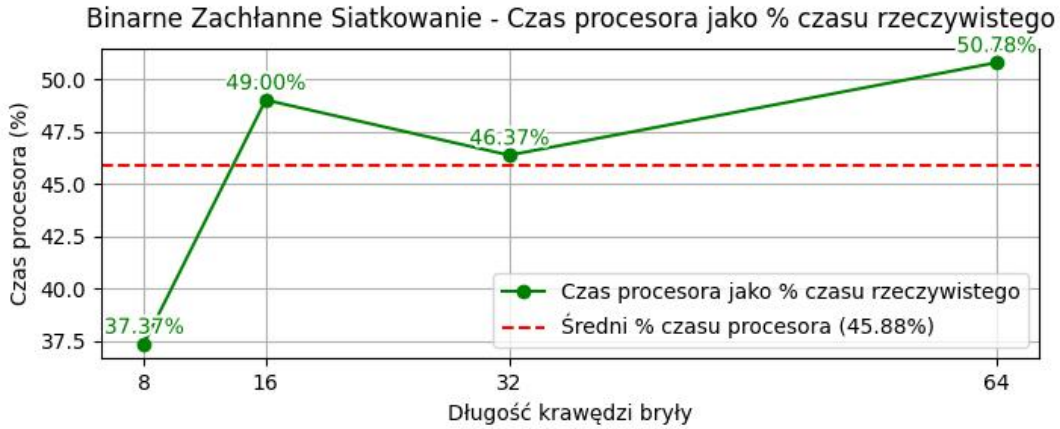
Rysunek 4.51: Porównanie pomiarów czasu rzeczywistego dla różnych metod budowy siatki.

Porównując wszystkie techniki przedstawione na rysunku 4.51, zdecydowanie najlepiej wypada technika binarnego zachłannego siatkowania. Z wcześniejszych badań wynika, że jest to technika, która najszybciej buduje siatkę i charakteryzuje się najmniejszą zajętością pamięci. Standardowa implementacja zachłannego siatkowania oferowała znacznie mniejszą zajętość pamięci i mniejszą liczbę wierzchołków w końcowym procesie rysowania, ale jak widać na rysunku 4.51, osiągała to kosztem dłuższego czasu budowania niż inne techniki. Binarne zachłanne siatkowanie stanowi znaczne rozwinięcie tej techniki, deklasując wszystkie pozostałe dostępne metody.



Rysunek 4.52: Porównanie pomiarów czasu procesora dla różnych metod budowy siatki.

Biorąc pod uwagę jedynie porównanie czasu procesora przedstawione na rysunku 4.52, można zauważyć, że binarne zachłanne siatkowanie zdobywa znaczną przewagę nad metodą ukrywania powierzchni z wsparciem GPU w przypadku małych brył względem czasu rzeczywistego. Wyniki te sugerują, że binarne zachłanne siatkowanie traci znaczne ilości czasu na operacje wejścia/wyjścia oraz inne narzuty przy przetwarzaniu małych brył czego potwierdzenie można zobaczyć na rysunku 4.53. Podobną tendencję zwiększonego narzutu dla małych brył zaobserwowano w przypadku zachłannego siatkowania co również zostało przedstawione na rysunku 4.54.

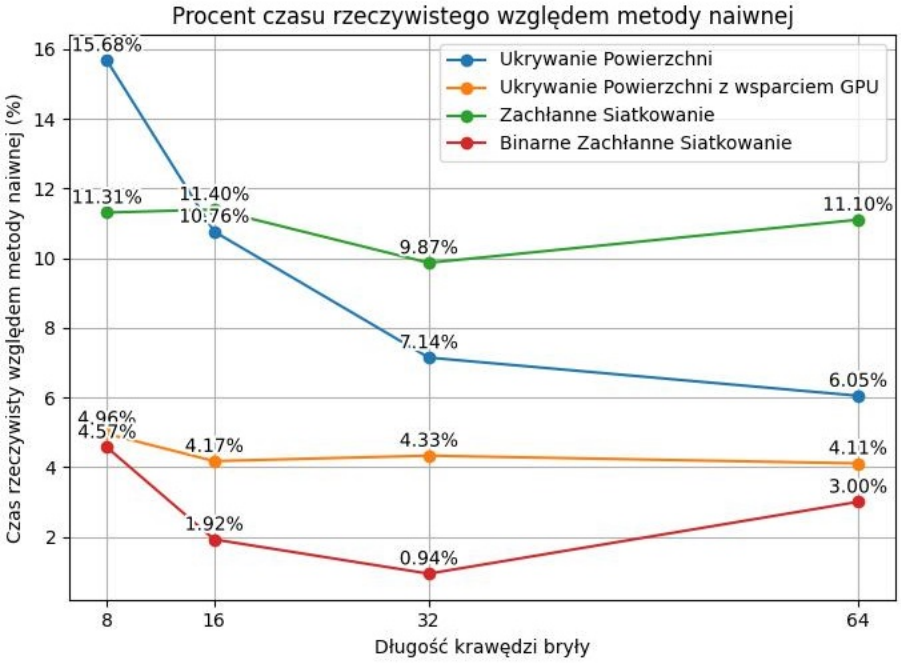


Rysunek 4.53: Porównanie procentu czasu procesora względem czasu rzeczywistego dla techniki binarnego zachłannego siatkowania.

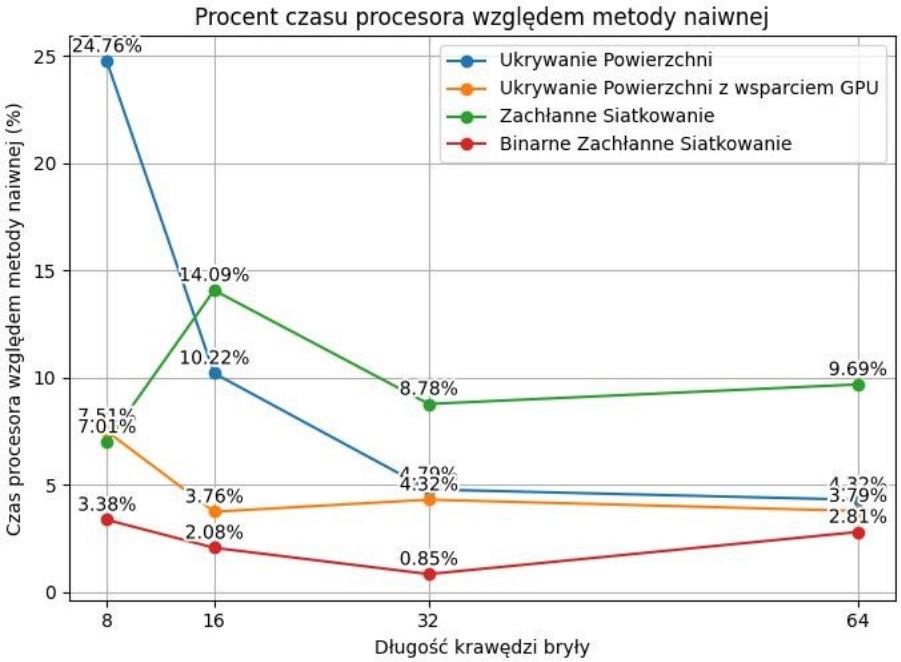
Rysunek 4.54: Porównanie procentu czasu procesora względem czasu rzeczywistego dla techniki zachłannego siatkowania.

Zgodnie z danymi widocznymi na rysunkach 4.56 oraz 4.55 zaobserwowano, że największą poprawę wydajności w stosunku do metody naiwnej uzyskują wszystkie techniki dla brył o wielkości 32x32x32 oraz 64x64x64, szczególnie w kontekście czasu rzeczywistego. Wyjątkiem jest technika ukrywania powierzchni z wsparciem GPU, której procentowy czas rzeczywisty względem metody naiwnej pozostawał na podobnym poziomie już od bryły o

wielkości 16x16x16. Dodatkowo, obie techniki zachłannego siatkowania szczególnie dobrze prezentują się dla bryły o wielkości 32x32x32.

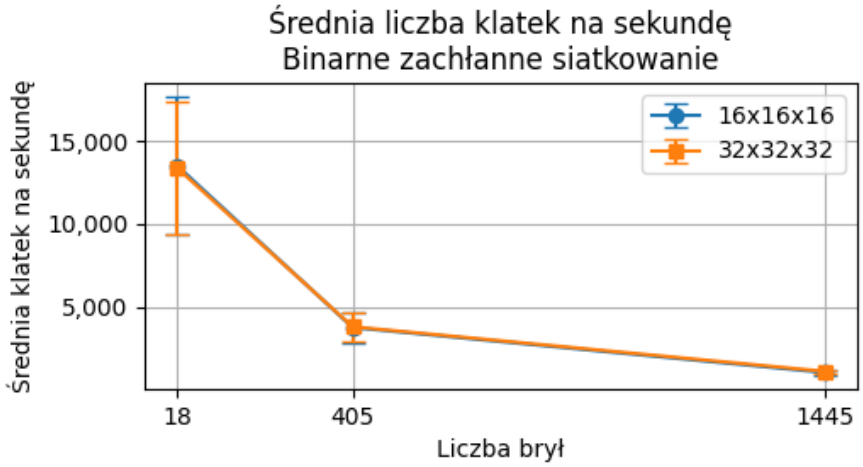


Rysunek 4.55: Porównanie procentu czasu procesora względem metody naiwnej dla różnych metod budowy siatki.

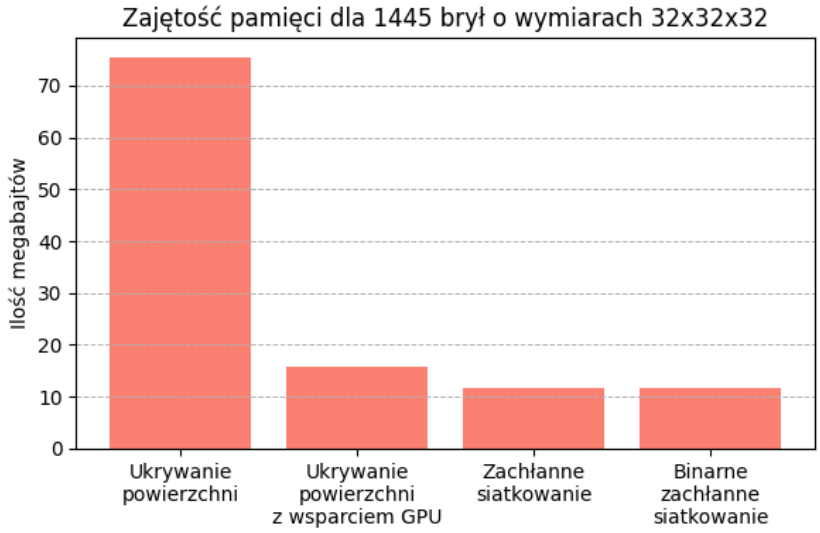


Rysunek 4.56: Porównanie procentu czasu rzeczywistego względem metody naiwnej różnych metod budowy siatki.

Niezwykle ciekawą obserwacją jest fakt, że podwajanie wymiarów brył nie ma dużego wpływu na średnią liczbę klatek na sekundę, natomiast zwiększanie liczby brył ma znaczący wpływ na ten parametr. Tę zależność można zaobserwować dla każdej z metod, a przykładowa metoda przedstawiona jest na rysunku 4.57. Sugeruje to, że liczba klatek na sekundę jest mocno ograniczana przez ilość zawołań rysujących. Dodatkowo, na rysunku 4.58 można zauważyć, że dla dużych światów i dużych brył zarówno ukrywanie powierzchni z wsparciem GPU, jak i zachłanne siatkowanie oraz binarne zachłanne siatkowanie radzą sobie podobnie dobrze.



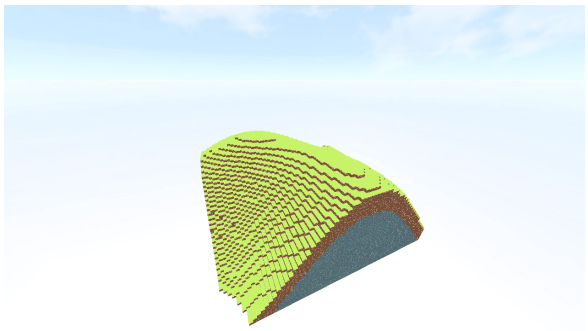
Rysunek 4.57: Średnia liczba klatek na sekundę w zależności od liczby brył dla metody binarnego zachłannego siatkowania.



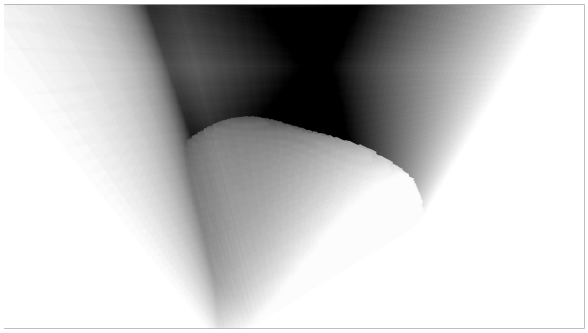
Rysunek 4.58: Zajętość pamięci dla 1447 brył o wymiarach 32x32x32 dla różnych metod budowania siatki.

4.8.2 Śledzenie promieni

Porównując wszystkie cztery techniki widoczne na rysunkach 4.59, 4.60, 4.61, 4.62 można zauważyć, że mapy cegiełkowe (patrz rysunek 4.61), oraz drzewa ósemkowe (patrz rysunek 4.62) wykonują najmniej iteracji promieni. Która z tych technik jest lepsza, zależy od typu przechowywanych danych oraz sposobu ich użycia. Mapy cegiełkowe są bardziej surowe, prostsze w edycji oraz przesyłaniu (ang. *Streaming*), podczas gdy drzewa ósemkowe w niektórych przypadkach potrafią być bardziej efektywne pod względem zużycia pamięci.

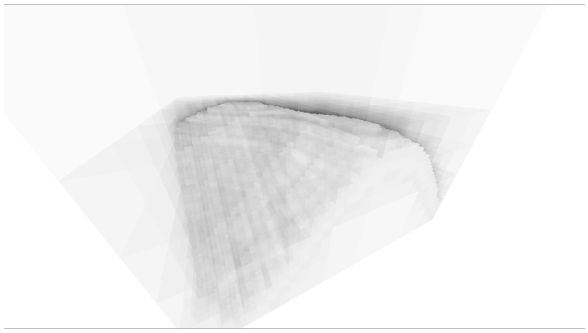


Rysunek 4.59: Bazowy teren będącym referencją dla którego tworzone są mapy iteracji promieni.



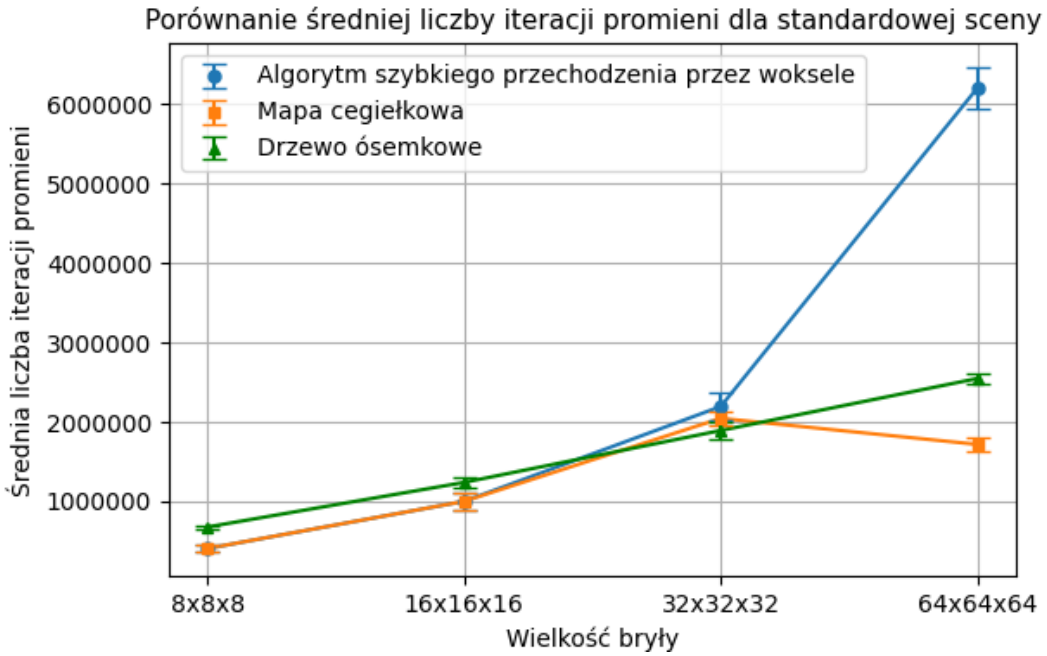
Rysunek 4.60: Mapa iteracji promieni dla surowego formatu wokseli.

Rysunek 4.61: Mapa iteracji promieni dla map cegiełkowych .

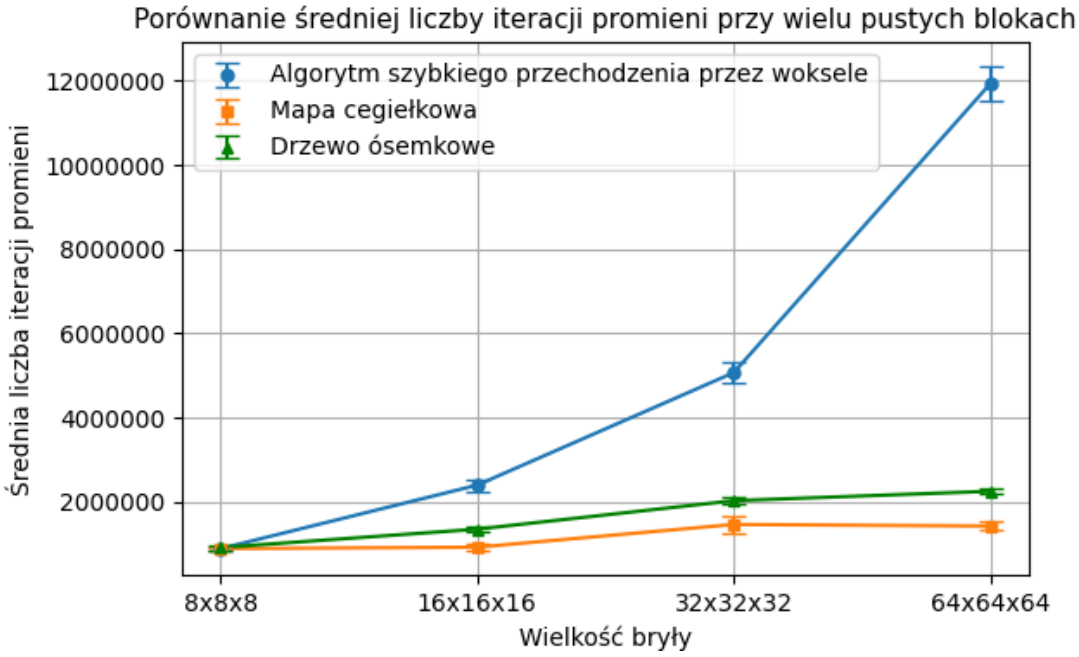


Rysunek 4.62: Mapa iteracji promieni dla drzew ósemkowych.

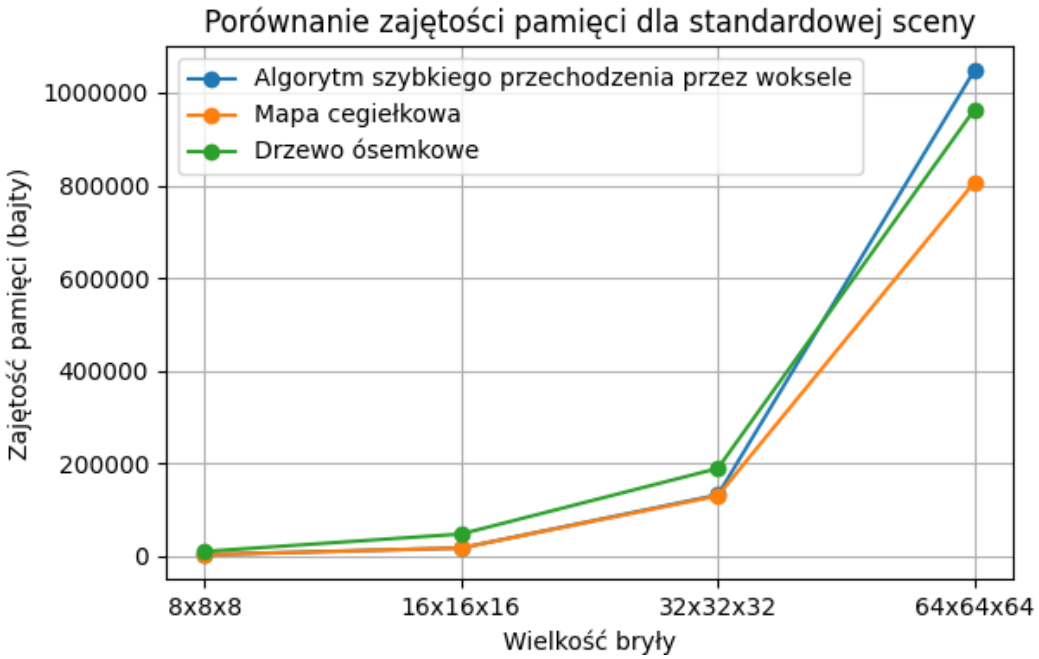
Na rysunkach 4.63 i 4.64 zaobserwowano, że mapa cegiełkowa charakteryzuje się najmniejszą średnią liczbą iteracji promieni spośród wszystkich technik, niezależnie od stopnia zapełnienia struktury terenu. Ponadto, rysunki 4.65 i 4.66 wskazują, że mapa cegiełkowa wykazuje najmniejsze zużycie pamięci dla badanego terenu, bez względu na to, czy w badanym terenie przeważa większa liczba pustych bloków. Podobnie jak w przypadku triangulacji, pomiary wykazały, że średnia liczba klatek na sekundę jest mocno zależna od liczby rysowanych brył. Zmiana wielkości bryły w dużych scenach nie ma znaczącego wpływu.



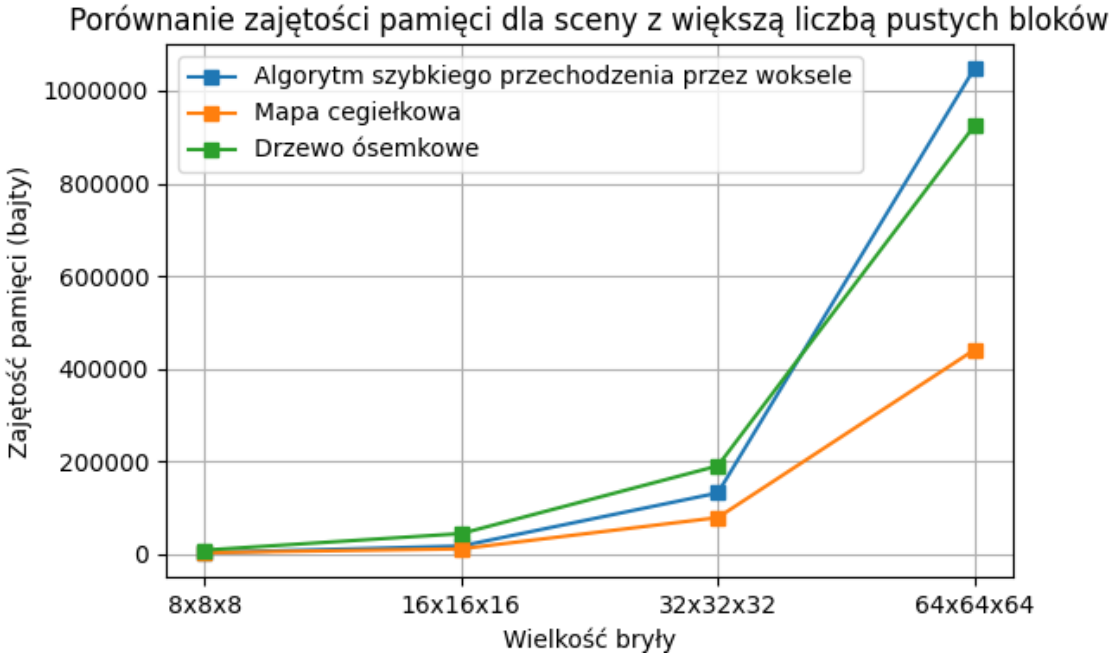
Rysunek 4.63: Porównanie średniej liczby iteracji promieni w zależności od wielkości bryły dla różnych struktur danych.



Rysunek 4.64: Porównanie średniej liczby iteracji promieni dla świata w duzej mierze stworzonego z pustych wokseli w zależności od wielkości bryły dla różnych struktur danych.



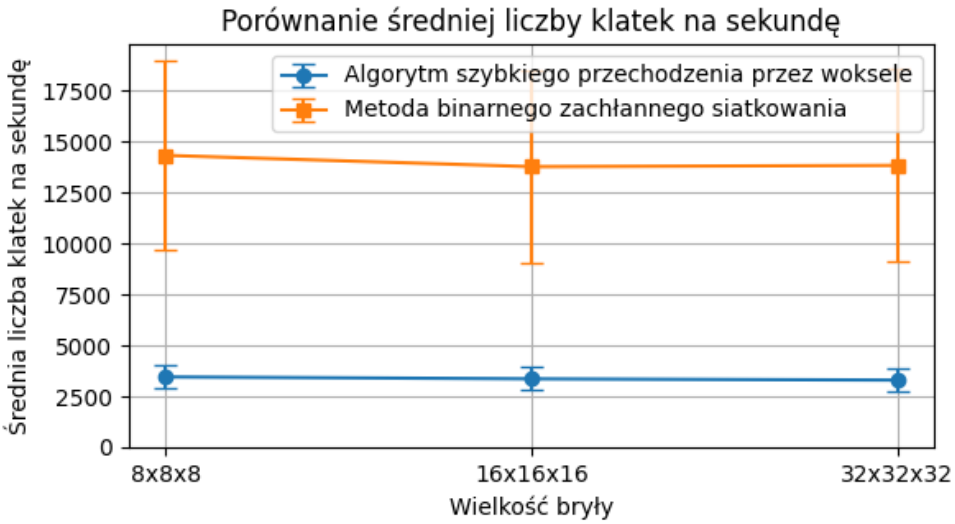
Rysunek 4.65: Porównanie zajętości pamięci w zależności od wielkości bryły dla różnych struktur danych.



Rysunek 4.66: Porównanie zajętości pamięci dla świata w dużej mierze stworzonego z pustych wokseli w zależności od wielkości bryły dla różnych struktur danych.

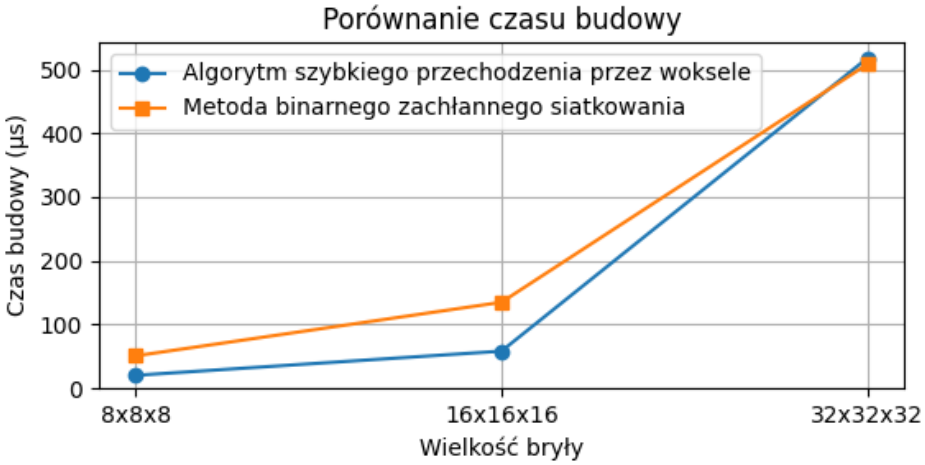
4.8.3 Porównanie dwóch najlepszych technik renderowania

Aby porównać dwie najlepsze metody w swoich kategoriach: binarne zachłanne siatkowanie (triangulacja) oraz szybki algorytm przechodzenia przez woksele (śledzenie promieni), zebrano dane obu technik dotyczące średniej liczby klatek na sekundę, czasu budowy i zajętości pamięci. Na rysunku 4.67 wykazano, że binarne zachłanne siatkowanie w przypadku średniej liczby klatek na sekundę osiąga wyniki ponad 3.5 razy lepsze niż szybki algorytm przechodzenia przez woksele.



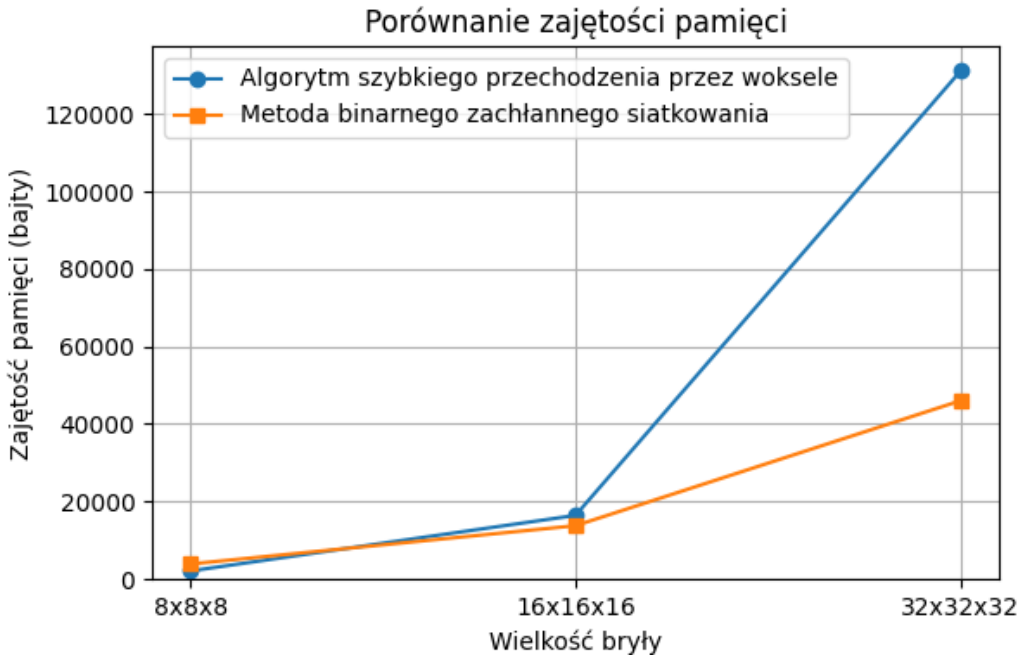
Rysunek 4.67: Porównanie średniej liczby klatek na sekundę w zależności od wielkości bryły dla najlepszych metod triangulacji i śledzenia promieni.

Jednakże porównując czas budowy przedstawiony na rysunku 4.68 algorytm szybkiego przechodzenia przez woksele okazuje się znacznie bardziej efektywny dla brył o rozmiarach 8x8x8 oraz 16x16x16.



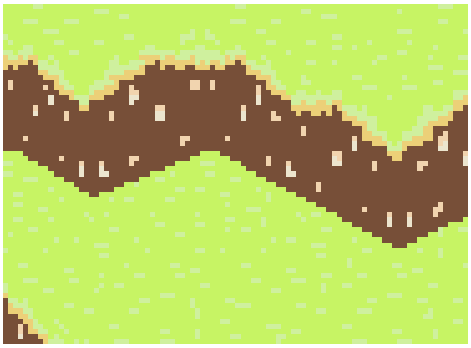
Rysunek 4.68: Porównanie czasu budowy w zależności od wielkości bryły dla najlepszych metod triangulacji i śledzenia promieni.

W przypadku porównania zajętości pamięci, przedstawionego na rysunku 4.69, metoda binarnego zachłannego siatkowania wykazuje znaczną przewagę dla bloku 32x32x32. Jest to zgodne z oczekiwaniami, ponieważ metoda ta przesyła preprocesowane wierzchołki, podczas gdy algorytm szybkiego przechodzenia przez woksele przesyła wszystkie dane terenu.

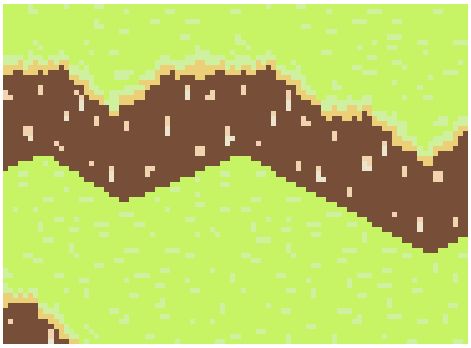


Rysunek 4.69: Porównanie zajętości pamięci w zależności od wielkości bryły dla najlepszych metod triangulacji i śledzenia promieni.

Zarówno zachłanne binarne siatkowanie, jak i algorytm szybkiego przechodzenia przez woksele, zapewniają porównywalną jakość obrazu. Niewielkie różnice można zaobserwować na poziomie poszczególnych pikseli tekstury ziemi (patrz rysunek 4.70 i 4.71).



Rysunek 4.70: Przybliżenie na bryłę wokseli wygenerowaną przy pomocy szybkiego algorytmu przechodzenia przez woksele.



Rysunek 4.71: Przybliżenie na bryłę wokseli wygenerowaną przez metodę binarnego zachłannego siatkowania.

4.9 Wnioski z badań

Porównując obie techniki renderowania, można stwierdzić, że triangulacja jest bardziej wydajną techniką w swojej podstawowej implementacji niż śledzenie promieni. Dotyczy to zarówno średniej liczby klatek na sekundę, jak i zajętości pamięci przekazywanej do karty graficznej. W przypadku śledzenia promieni były to pełne struktury wokseli, natomiast w przypadku triangulacji wcześniej przetworzone struktury w postaci wierzchołków.

Z badań wynika, że zarówno dla śledzenia promieni, jak i triangulacji, zwiększenie wielkości bryły nie miało znaczącego wpływu na średnią liczbę klatek na sekundę, natomiast zwiększenie liczby brył miało już istotny wpływ. Sugeruje to, że w obu przypadkach liczba wykonanych zawołań rysujących skutecznie ogranicza wydajność. Potencjalną poprawę mogłoby zapewnić usuwanie niewidocznych powierzchni, co redukowałoby liczbę zawołań rysujących jedynie do brył widocznych na ekranie. Również dynamiczne grupowanie (ang. *Batching*), które łączy wiele obiektów w jedno wywołanie, mogłoby znacząco zminimalizować liczbę operacji renderowania. Szczególnie obiecujące w kontekście wydajnego dynamicznego grupowania wydaje się binarne zachłanne siatkowanie, które generuje najmniejszą liczbę wierzchołków, których złączenie może być najwydajniejsze.

W przypadku dobudowywania świata, metoda triangulacji spędza większość czasu na przebudowie istniejących siatek w celu uwzględnienia nowego sąsiada. Zmniejszenie czasu przebudowy poprzez modyfikację siatki tylko na krawędziach bryły, zamiast rekonstrukcji całej siatki od podstaw, mogłoby przynieść obiecujące rezultaty w kontekście efektywności budowania świata. Jest to obszar z potencjałem do dalszych usprawnień.

W przypadku śledzenie promieni brak poprawy średniej liczby klatek na sekundę, pomimo zmniejszenia liczby iteracji promieni dla map cegiełkowych i drzew ósemkowych, sugeruje, że implementacja jest niewystarczająco wydajna i nie korzysta w pełni z potencjału tych struktur przyspieszających. Dowodzi to, że efektywne wdrożenie tych technik jest znacznie trudniejsze. Ponadto, debugowanie i profilowanie są znacznie bardziej skomplikowane w przypadku jednostek cieniujących i pracy na karcie graficznej niż na procesorze. To utrudnia identyfikację błędów oraz wąskich gardeł wydajnościowych w kodzie programu.

Rozdział 5

Podsumowanie

Celem niniejszej pracy była analiza i porównanie różnych technik optymalizacji wokselowej reprezentacji świata gry. Główna problematyka tego zagadnienia koncentrowała się na wydajności, zajętości pamięci, czasie budowania scen oraz jakości grafiki generowanej za pomocą technik takich jak triangulacja z różnymi metodami budowania siatek oraz śledzenie promieni, z uwzględnieniem różnorodnych metod i struktur danych, takich jak mapy cegiełkowe i drzewa ósemkowe.

Na potrzeby badań opracowano środowisko pomiarowe oparte na *OpenGL API* oraz *C++*, w którym zaimplementowano i przetestowano różne techniki renderowania, uwzględniając specyficzne metody dla każdej z nich. W przypadku triangulacji eksperymenty miały na celu ocenę efektywności metod takich jak metoda naiwna, ukrywanie powierzchni, ukrywanie powierzchni ze wsparciem GPU, zachłanne siatkowanie oraz binarne zachłanne siatkowanie. Natomiast w kontekście śledzenia promieni, badania skupiły się na analizie różnych struktur danych stosowanych wewnątrz brył wokseli, takich jak mapy cegiełkowe oraz drzewa ósemkowe.

Wyniki badań wykazały, że techniki oparte na triangulacji, zwłaszcza metoda binarnego zachłannego siatkowania, zapewniają znaczną poprawę efektywności renderowania oraz oszczędność zasobów pamięciowych. Chociaż techniki śledzenia promieni wypadły znacznie gorzej w porównaniu do triangulacji, analiza różnych struktur danych w tej kategorii wykazała, że mapy cegiełkowe są najbardziej wydajne zarówno pod względem średniej liczby iteracji promieni, jak i zajętości pamięci spośród testowanych struktur.

Ponadto, wyniki badań wskazały na obszary, w których możliwe są dalsze optymalizacje, takie jak techniki eliminacji niewidocznych powierzchni oraz dynamiczne grupowanie obiektów w jedno wywołanie, ponieważ nadmierna liczba zawołań rysowania znacząco ogranicza wydajność. Perspektywy dalszych badań obejmują eksplorację hybrydowych podejść, które łączą zalety obu technik. Przyszłe prace mogą również skupić się na rozszerzeniu badań o dodatkowe techniki graficzne, takie jak cienie i odbicia, w celu sprawdzenia, która z kategorii renderowania ma mniejszy narzut wydajnościowy pod kątem rozwijania jakości graficznej. Istnieje możliwość, że chociaż triangulacja wypada znacznie lepiej w

swojej podstawowej implementacji, to przy bardziej skomplikowanych efektach graficznych śledzenie promieni może okazać się bardziej wydajne w renderowaniu wokseli.

Bibliografia

[1] John Amanatides i Andrew Woo. „A Fast Voxel Traversal Algorithm for Ray Tracing". W: *Proceedings of EuroGraphics* 87 (sierp. 1987).

[2] Bartosz Taudul. *Tracy Repository*. 2024. (Term. wiz. 27. 05. 2024).

[3] Matthäus G. Chajdas. „A voxel-based visualization pipeline for high-resolution geometry". W: 2015. URL: https://api.semanticscholar.org/CorpusID:10966852.

[4] Daniel Cohen-Or i Arie Kaufman. „Fundamentals of Surface Voxelization". W: *Graphical Models and Image Processing* 57.6 (1995), s. 453–461. ISSN: 1077-3169. DOI: https://doi.org/10.1006/gmip.1995.1039. URL: https://www.sciencedirect.com/science/article/pii/S1077316985710398.

[5] Davis Morley. *Greedy Meshing Voxels Fast - Optimism in Design Handmade Seattle 2022*. 2023. (Term. wiz. 27. 05. 2024).

[6] Gabi Melman. *Spdlog Repository*. 2024. (Term. wiz. 27. 05. 2024).

[7] Google. *Google Benchmark Repository*. 2024. (Term. wiz. 27. 05. 2024).

[8] Henrik Rydgård. *Minitrace Repository*. 2024. (Term. wiz. 27. 05. 2024).

[9] Aksel Hjerpbakk. „Novel Extended Brickmap for Real-time Ray Tracing". Master thesis. Norwegian University of Science i Technology (NTNU), 2022. URL: https://hdl.handle.net/11250/3035458.

[10] Humair Ahmed. *Minecraft for $2.5 Billion - Yes, 'the App' is King*. (Term. wiz. 27. 05. 2024).

[11] Javier Cuervo. *Exploring the fascinating world of voxel: understanding its meaning and applications*. 2024. (Term. wiz. 25. 05. 2024).

[12] Jesse Beder. *Yaml-cpp Repository*. 2024. (Term. wiz. 27. 05. 2024).

[13] Jordan Peck. *FastNoiseLite Repository*. 2024. (Term. wiz. 27. 05. 2024).

[14] Juan Diego Montoya. *Teardown Teardown*. 2022. (Term. wiz. 27. 05. 2024).

[15] Radiology Key. *Static Anatomic Techniques*. URL: https://radiologykey.com/static-anatomic-techniques/ (term. wiz. 21. 05. 2024).

[16] kosshi. *Voxplat*. 2022. (Term. wiz. 27. 05. 2024).

[17] Samuli Laine. „A Topological Approach to Voxelization". W: *Computer Graphics Forum* 32 (lip. 2013). DOI: 10.1111/cgf.12153.

[18] Samuli Laine i Tero Karras. „Efficient Sparse Voxel Octrees". W: *IEEE transactions on visualization and computer graphics* 17 (paź. 2010), s. 1048–59. DOI: 10.1109/TVCG.2010.240.

[19] LedPulse. *From pixel to voxel. A dimensional leap.* URL: https://www.ledpulse.com/blog-posts/from-pixel-to-voxel-a-dimensional-leap (term. wiz. 25. 05. 2024).

[20] Theodor Lundqvist, Jintao Yu i Jiuming Zeng. *Voxel Ray Marcher*. 2023. URL: https://fileadmin.cs.lth.se/cs/Education/EDAN35/projects/2023/VoxelRayMarcher.pdf.

[21] MADISON MACKEY. *Explore Subsurface Geology Using Voxel Layers in Three Easy Steps*. 2021. (Term. wiz. 02. 06. 2024).

[22] Alexander Majercik, Cyril Crassin, Peter Shirley i Morgan McGuire. „A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering". W: *Journal of Computer Graphics Techniques (JCGT)* 7.3 (wrz. 2018), s. 66–81. ISSN: 2331-7418. URL: http://jcgt.org/published/0007/03/04/.

[23] Matej 'Retro' Jan. *Pixels and voxels, the long answer*. 2016. URL: https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190 (term. wiz. 25. 05. 2024).

[24] Matthew Rodusek. *Result Repository*. 2024. (Term. wiz. 27. 05. 2024).

[25] Mikola Lysenko. *Meshing in a Minecraft Game*. 2012. (Term. wiz. 27. 05. 2024).

[26] Nigel Stewart. *Glew Repository*. 2024. (Term. wiz. 27. 05. 2024).

[27] ocornut. *Imgui Repository*. 2024. (Term. wiz. 27. 05. 2024).

[28] *OpenGL — Wikipedia, The Free Encyclopedia*. 2024. (Term. wiz. 27. 05. 2024).

[29] Learn OpenGL. *Hello Triangle — Learn OpenGL*. [Online; accessed 21-May-2024]. URL: https://learnopengl.com/Getting-started/Hello-Triangle.

[30] Jorge Revelles, Carlos Ureña, M. Lastra, Dpt Lenguajes, Sistemas Informaticos i E. Informatica. „An Efficient Parametric Algorithm for Octree Traversal". W: (maj 2000).

[31] Francisco Sans i Rhadamés Carmona. „A Comparison between GPU-based Volume Ray Casting Implementations: Fragment Shader, Compute Shader, OpenCL, and CUDA". W: *CLEI Electronic Journal* 20 (sierp. 2017), s. 7. DOI: 10.19153/cleiej.20.2.7.

[32] Sean Barrett. *STB Repository*. 2024. (Term. wiz. 27. 05. 2024).

[33] Steven Wittens. *Teardown Frame Teardown*. 2023. (Term. wiz. 27. 05. 2024).

[34] TanTanDev. *binary_greedy_mesher_demo*. 2024. (Term. wiz. 27. 05. 2024).

[35] TanTanDev. *Blazingly Fast Greedy Mesher - Voxel Engine Optimizations*. 2024. (Term. wiz. 27. 05. 2024).

[36] Jakub Tyc, Tina Selami, Defne Sunguroglu Hensel i Michael Hensel. „A Scoping Review of Voxel-Model Applications to Enable Multi-Domain Data Integration in Architectural Design and Urban Planning". W: *Architecture* 3.2 (2023), s. 137–174. ISSN: 2673-8945. DOI: 10.3390/architecture3020010. URL: https://www.mdpi.com/2673-8945/3/2/10.

[37] Waleed Kadous. *Does Minecraft use the ordinary Java 3D API for graphics?* (Term. wiz. 27. 05. 2024).

[38] OpenGL Wiki. *Rendering Pipeline Overview — OpenGL Wiki,* [Online; accessed 21-May-2024]. 2022. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Rendering_Pipeline_Overview&oldid=14914.

[39] Wikipedia contributors. *CMake — Wikipedia, The Free Encyclopedia*. 2024. (Term. wiz. 27. 05. 2024).

[40] Wikipedia contributors. *Marching cubes — Wikipedia, The Free Encyclopedia*. 2023. (Term. wiz. 25. 05. 2024).

[41] Wikipedia contributors. *Minecraft — Wikipedia, The Free Encyclopedia*. 2024. (Term. wiz. 27. 05. 2024).

[42] Wikipedia contributors. *Pixel — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Pixel&oldid=1225609472 (term. wiz. 25. 05. 2024).

[43] Wikipedia contributors. *Simple and Fast Multimedia Library — Wikipedia, The Free Encyclopedia*. 2023. (Term. wiz. 27. 05. 2024).

[44] Wikipedia contributors. *Voxel — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Voxel&oldid=1224492564 (term. wiz. 25. 05. 2024).

[45] T. L. van Wingerden. „Real-time Ray tracing and Editing of Large Voxel Scenes". W: 2015. URL: https://api.semanticscholar.org/CorpusID:53850534.

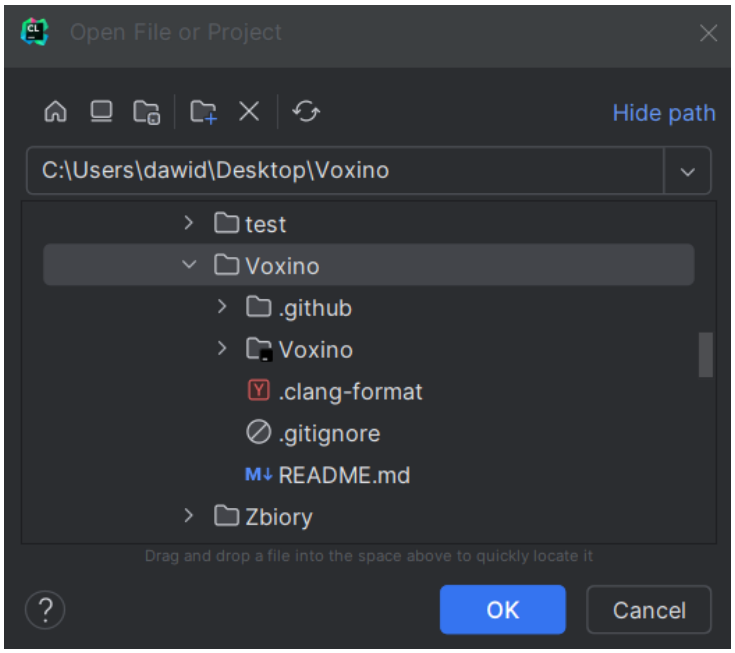
Dodatki

Opis uruchomienia i używania środowiska badawczego

Budowanie projektu za pomocą CLion 2024.1 RC

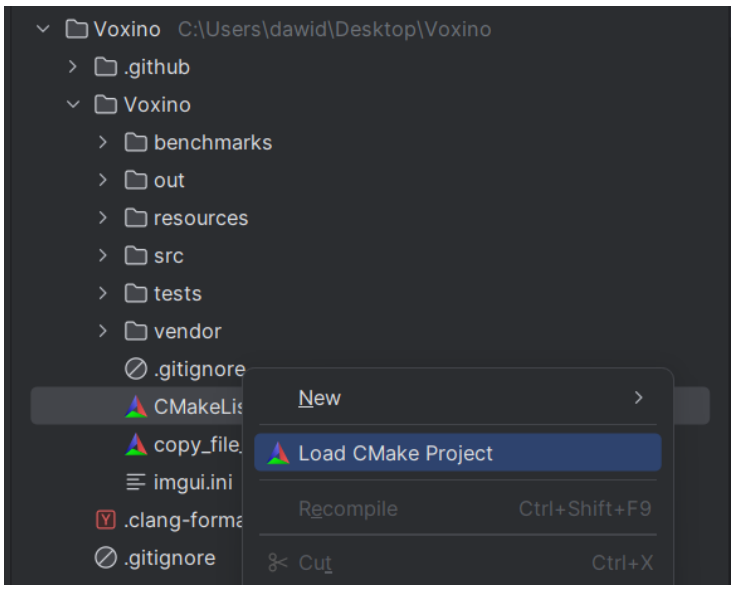
noindent Aby zbudować projekt, konieczne jest posiadanie CMake w wersji co najmniej 3.28.1, połączenia z internetem oraz kompilatora MSVC obsługującego standard C++20.

1. Należy uruchomić CLion i wybrać opcję otwarcia nowego projektu (patrz rysunek 1).



Rysunek 1: Wybór nowego projektu w zakładce „Open File or Project”.

2. W głównym folderze Voxino znajduje się plik `CMakeLists.txt`, na który należy kliknąć prawym przyciskiem myszy i wybrać opcję „Load CMake Project” (patrz rysunek 2).



Rysunek 2: Ładowanie projektu Voxino z głównego pliku CMakeLists.

3. Rozpocznie się pobieranie zewnętrznych bibliotek i ich budowanie (patrz rysunek 3).

```
C:\Users\dawid\AppData\Local\Programs\CLion\bin\cmake\win\x64\bin\cmake.exe ...
-- The CXX compiler identification is MSVC 19.38.33134.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
...
-- Fetching SFML...
-- SFML Fetched!
-- Fetching ImGui...
-- ImGui Fetched!
-- Fetching ImGui-SFML...
-- Found ImGui v1.90.4 in C:/Users/dawid/Desktop/Voxino/Voxino/cmake-build-debug/_deps/imgui-src
-- ImGui-SFML Fetched!
-- Fetching GLM...
...
```

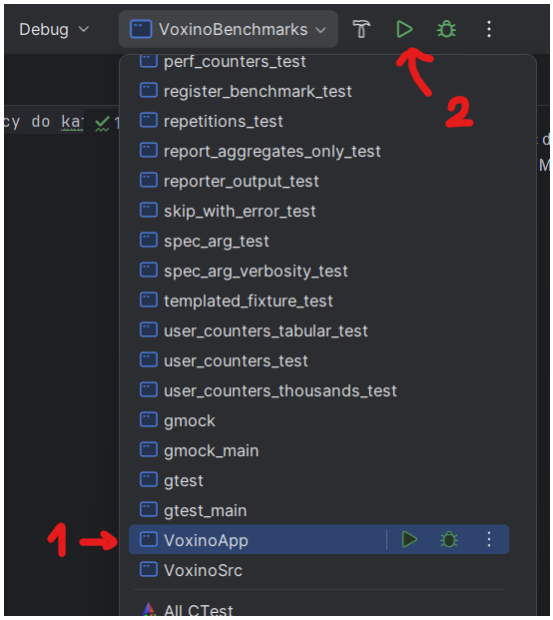
Rysunek 3: Przykładowy tekst wyświetlany w zakładce „CMake” podczas ładowania projektu.

4. Budowa powinna zakończyć się linią: *[Finished]* (patrz rysunek 4).

```
...
-- Configuring done (61.6s)
-- Generating done (0.3s)
-- Build files have been written to: C:/Users/dawid/Desktop/...

[Finished]
```

Rysunek 4: Końcowy tekst w zakładce „CMake" wskazujący na poprawne załadowanie projektu.

5. Następnie należy wybrać *VoxinoApp* z listy dostępnych konfiguracji i kliknąć „Run VoxinoApp" (patrz rysunek 5).



Rysunek 5: Lista dostępnych konfiguracji do zbudowania.

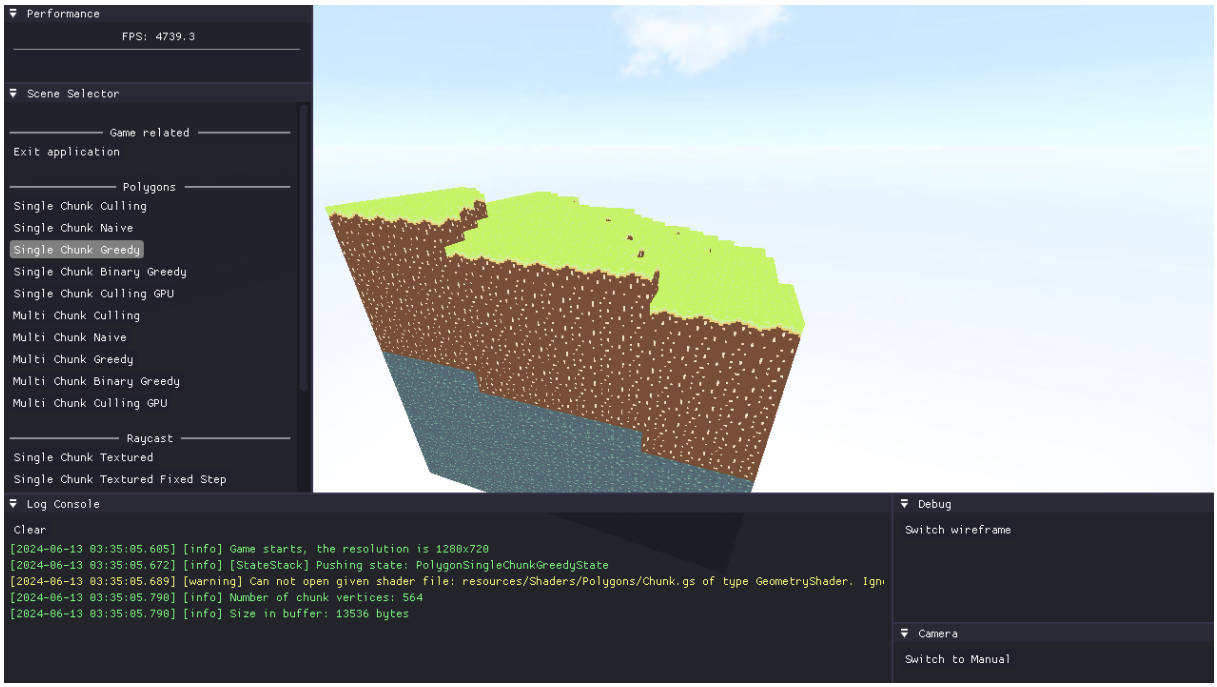
6. Po zakończeniu kompilacji powinno uruchomić się środowisko badawcze. W przypadku wystąpienia jakichkolwiek problemów należy sprawdzić w ustawieniach CLion, czy w sekcji *Build, Execution, Deployment* > *CMake* narzędzie Toolchain jest ustawione na wartość *Visual Studio*.

Opis środowiska

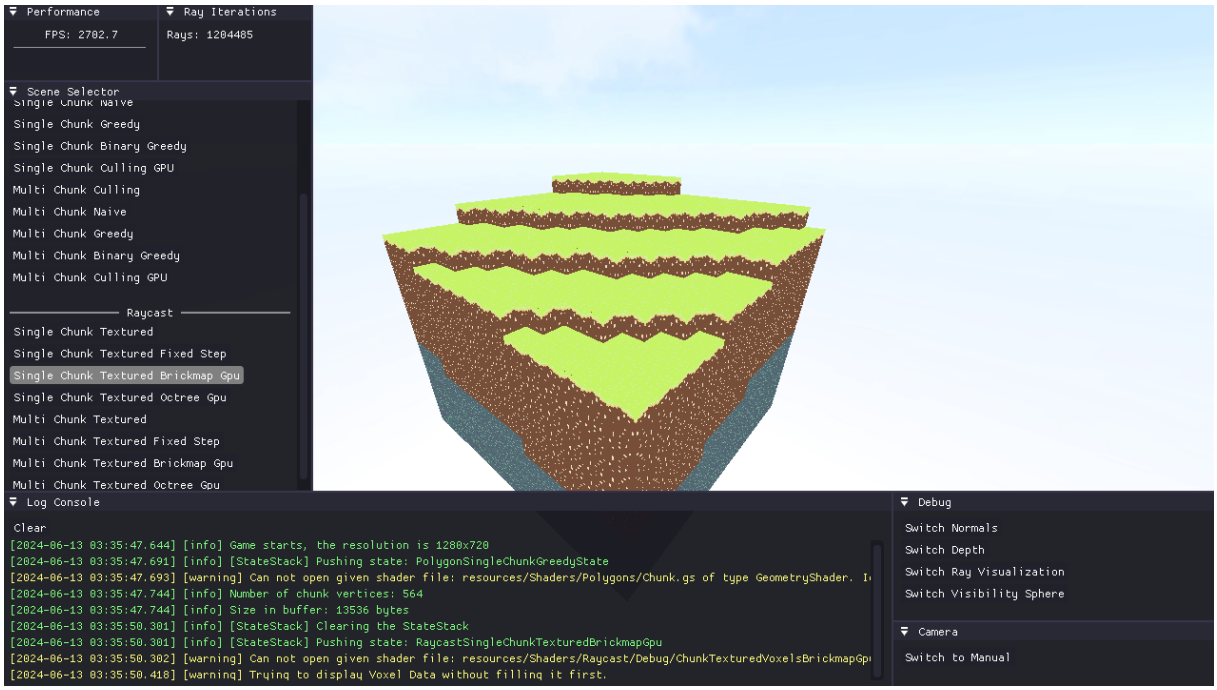
Uruchomione środowisko, widoczne na rysunkach 6 oraz 7, posiada panel wyboru sceny po lewej stronie ekranu. Aktualnie wybrana scena jest oznaczona szarym podświetleniem. Na dole ekranu znajduje się konsola z dodatkowymi informacjami programu. W prawym dolnym rogu umieszczono okno debugowe, które może się różnić w zależności od sceny. Okno to umożliwia przełączenie kamery w tryb manualnego poruszania się oraz uruchomienie kilku przydatnych wizualizacji, takich jak:

- Głębia,
- Normalne,
- Wizualizacja liczby iteracji promieni,
- Sfera usuwająca bloki wokół kamery.

Opcje debugowe mogą nie działać, gdy projekt jest zbudowany w trybie *Release*.



Rysunek 6: Środowisko badawcze przedstawiające scenę chciwego siatkowania.



Rysunek 7: Środowisko badawcze przedstawiające scene map cegiełkowych.

Dodatkowe informacje

W celu uruchomienia testów biblioteki *Google Benchmark* należy uruchomić platformę docelową *VoxinoBenchmarks* (patrz rysunek 8). Testy wydajnościowe znajdują się w pliku `benchmarks/benchmarks/src/ChunkBenchmark.cpp`.

Rysunek 8: Zrzut ekranu przedstawiający testy wydajnościowe biblioteki *Google Benchmark*, oraz odpowiadającej jej platformę docelową.

Główne zmienne programu oraz ustawienia wielkości brył lub ich ilości znajdują się w pliku `src/constants.h`, którego zawartość jest widoczna na rysunku 9.

README.md ChunkBenchmark.cpp constants.h

```
1   #pragma once
2
3   #define _CRTDBG_MAP_ALLOC
4   #include ...
7
8   // #define DISABLE_IMGUI
9   // #define IS_TRACY_ENABLED TRUE
10  #define IS_MINITRACE_ENABLED FALSE
11  // #define TRACY_ON_DEMAND TRUE
12  // #define TRACY_AUTOMATIC_MEMORY_PROFILING
13  // #define TRACY_COLLECT_THUMBNAILS
14  // #define ENABLE_TRACY_MARKERS
15  // #define PLOT_AVERAGE_FPS
16
17  #define CHUNK_CONTAINER_RADIUS 1;
18  #define BLOCK_PER_DIMENSION_IN_CHUNK 16;
19  constexpr static auto IS_MINITRACE_COLLECTING_AT_START = false;
```

Rysunek 9: Zrzut ekranu przedstawiające główne zmienne programu.

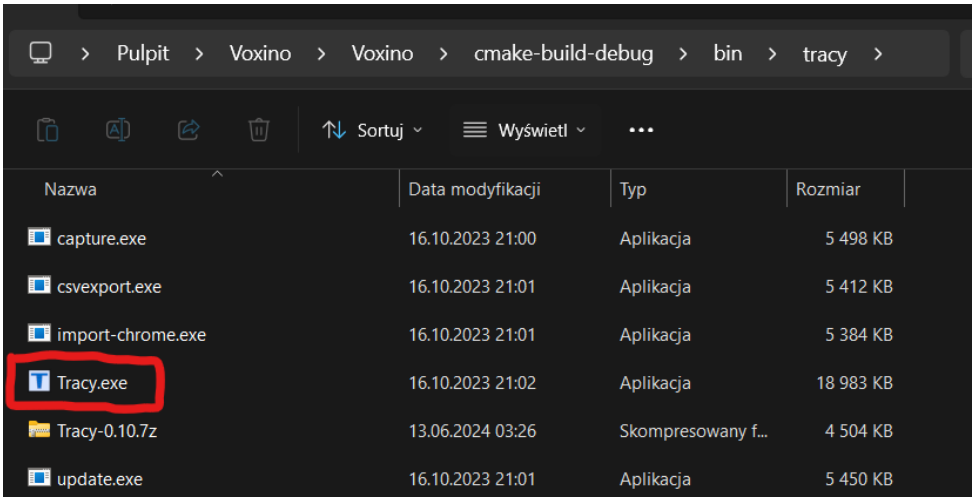
Scenę startową można zmienić w pliku *src/Application.cpp* (patrz rysunek 10).

Application.cpp

```
21  namespace Voxino
123 Application::Application()
145     initializeTracyScreenCapture();
146     setupFlowStates();
147     // mAppStack.push(State_ID::PolygonSingleChunkNaiveState);
148     // mAppStack.push(State_ID::PolygonSingleChunkCullingState);
149     // mAppStack.push(State_ID::PolygonSingleChunkCullingGpuState);
150     mAppStack.push(State_ID::PolygonSingleChunkGreedyState);
151     // mAppStack.push(State_ID::PolygonSingleChunkBinaryGreedyState);
152
153     // mAppStack.push(State_ID::PolygonMultiChunkNaiveState);
154     // mAppStack.push(State_ID::PolygonMultiChunkCullingState);
155     // mAppStack.push(State_ID::PolygonMultiChunkCullingGpuState);
156     // mAppStack.push(State_ID::PolygonMultiChunkGreedyState);
157     // mAppStack.push(State_ID::PolygonMultiChunkBinaryGreedyState);
158     // mAppStack.push(State_ID::RaycastSingleChunkTexturedVoxels);
159     // mAppStack.push(State_ID::RaycastSingleChunkTexturedVoxelsFixedStep);
160     // mAppStack.push(State_ID::RaycastSingleChunkTexturedBrickmapGpu);
161     // mAppStack.push(State_ID::RaycastSingleChunkTexturedOctreeGpu);
162     // mAppStack.push(State_ID::RaycastMultiChunkTexturedVoxels);
163     // mAppStack.push(State_ID::RaycastMultiChunkTexturedVoxelsFixedStep);
164     // mAppStack.push(State_ID::RaycastMultiChunkTexturedBrickmapGpu);
165     // mAppStack.push(State_ID::RaycastMultiChunkTexturedOctreeGpu);
166 }
```

Rysunek 10: Zrzut ekranu przedstawiający kod uruchamiający domyślną scenę aplikacji.

Pobrany kompatybilny klient *Tracy* znajduje się w katalogu `build/bin/tracy` (patrz rysunek 11).



Rysunek 11: Zrzut ekranu przedstawiający katalog zawierający kompatybilny klient narzedzia *Tracy*.

Spis skrótów i symboli

API	Interfejs Programistyczny Aplikacji (ang. *Application Programming Interface*)
BVH	Hierarchia Objętości Ograniczających (ang. *Bounding Volume Hierarchy*)
CMake	Narzędzie do Budowania Wieloplatformowego (ang. *Cross-Platform Make*)
CT	Tomografia Komputerowa (ang. *Computed Tomography*)
CUDA	Zunifikowana Architektura Obliczeniowa Urządzeń (ang. *Compute Unified Device Architecture*)
DAG	Skierowany Graf Acykliczny (ang. *Directed Acyclic Graph*)
GPU	Jednostka Przetwarzania Grafiki (ang. *Graphics Processing Unit*)
CPU	Jednostka Centralna (ang. *Central Processing Unit*)
FVTA	Szybki Algorytm Przechodzenia przez Woksele (ang. *Fast Voxel Traversal Algorithm*)
kD Tree	k-Wymiarowe Drzewo (ang. *k-Dimensional Tree*)
LOD	Poziom Szczegółowości (ang. *Level of Detail*)
MRI	Rezonans Magnetyczny (ang. *Magnetic Resonance Imaging*)
OpenCL	Otwarty Język Obliczeniowy (ang. *Open Computing Language*)
OpenGL	Otwarta Biblioteka Grafiki (ang. *Open Graphics Library*)
SDF	Funkcja Odległości (ang. *Signed Distance Function*)
SFML	Prosta i Szybka Biblioteka Multimedialna (ang. *Simple and Fast Multimedia Library*)
SVO	Rzadkie Drzewo Ósemkowe Wokseli (ang. *Sparse Voxel Octree*)

Lista dodatkowych plików, uzupełniających tekst pracy

W systemie do pracy dołączono dodatkowe pliki zawierające:

- kod źródłowy programu na podstawie którego przeprowadzono badania,
- film pokazujący działanie opracowanego oprogramowania

Spis rysunków

Spis tabel

Spis fragmentów kodu