

POLITECHNIKA BIAŁOSTOCKA

WYDZIAŁ INFORMATYKI

**PRACA DYPLOMOWA
INŻYNIERSKA**

TEMAT: Implementacja wokselowego silnika graficznego z wykorzystaniem OpenGL

WYKONAWCA: Piotr Zalewski

OPIEKUN PRACY DYPLOMOWEJ : dr inż. Marcin Skoczylas

BIAŁYSTOK 2024 ROK

Opis

Głównym zadaniem tradycyjnych silników graficznych, jest rendering obiektów trójwymiarowych opisanych siatkami wielokątów. Jednakże istnieje także inny sposób opisu danych 3D, stosowany między innymi w tomografii komputerowej, oparty na tzw. wokselach (ang. voxels). Termin woksel oznacza dosłownie wolumetryczny element obrazu, czyli trójwymiarowy piksel. Główną zaletą takiej formy opisu jest dużo łatwiejsza edycja obiektów uwzględniająca także ich wnętrze. Do wad należy zaliczyć specyficzny wygląd renderowanych modeli, problematyczną animację oraz wysokie wymagania pamięciowe. Obiekty opisane wokselami mogą być renderowane przez tradycyjne silniki graficzne, wymaga to jednak wcześniejszej ich konwersji do siatki wielokątów np. za pomocą algorytmu maszerujących kostek (ang. marching cubes). Dużo wygodniejszym rozwiązaniem jest użycie wokselowego silnika graficznego, który renderuje takie dane wprost np. w postaci kostek sześciennych. Podejście takie stosowane jest m.in. w popularnej grze Minecraft do reprezentacji terenu. Silnik wokselowy powinien charakteryzować się wysoką wydajnością umożliwiając rysowanie milionów kostek w ciągu sekundy. Kluczowym czynnikiem jest tutaj rozmiar zbioru danych oraz sposób jego reprezentacji w pamięci. W celu przyśpieszenia renderingu często wykorzystuje się instancjonowanie GPU lub specjalne struktury danych takie jak rzadkie drzewa ósemkowe.

Spis treści

1 Wstęp	1
2 Omówienie technik renderingu grafiki 3D w oparciu o woksele	2
2.1 Reprezentacja bokselowa	2
2.1.1 Rozwiążanie naiwne	3
2.1.2 Upraszczanie siatki	3
2.2 Śledzenie stożka w wolumenie	6
2.3 Maszerujące kostki	8
3 Omówienie struktur danych, metod zarządzania pamięcią wykorzystywanych w opisie wokselowym obiektów trójwymiarowych	10
3.1 Rzadkie N3 drzewa	10
3.2 Podział wolumenu na kawałki	14
4 Zastosowane rozwiązanie	16
4.1 Środowisko implementacji	16
4.2 Struktura silnika	16
4.3 <i>ChunkPool</i> - szczegóły implementacji	18
4.3.1 Dane kawałków wolumenu	18
4.3.2 Dane siatek	19
4.3.3 Renderowanie siatek	23

4.3.4	Alokacja i dealokacja kawałków	27
4.3.5	Dynamiczne poszerzanie bufora <i>VBO</i>	29
4.4	<i>MeshingEngine</i> - szczegóły implementacji	30
4.4.1	Synchronizacja zadań siatkowania	30
4.4.2	Algorytm chciwego siatkowania w GLSL	31
4.4.3	Rezultat zadania siatkującego	34
4.5	<i>WorldGrid</i> - szczegóły implementacji	35
4.5.1	Struktura danych	36
4.5.2	Inicjalizacja struktury danych	39
4.5.3	Działanie struktury danych	40
4.5.4	Ograniczenia struktury danych	41
4.6	<i>ChunkDataStreamer</i> - szczegóły implementacji	42
4.7	<i>ChunkGenerator</i> - szczegóły implementacji	43
4.7.1	przykład z <i>NoiseTerrainGenerator</i>	43
4.8	<i>Engine</i> - szczegóły implementacji	45
5	Środowisko demonstracyjne	49
5.1	Użyte generatory danych	50
5.2	Wielkości pojedynczego kawałka	50
5.3	Warunki partycjonowania komend	50
5.4	Wielkości zakresu widoczności	50
5.5	Liczba możliwych stanów wokseli	51
6	Wyniki testów wydajnościowych	52
6.1	Metodologia	52
6.2	Wyniki i analiza	53
6.2.1	Wpływ wybranego generatora danych	53
6.2.2	Wpływ wielkości zakresu widoczności	56

6.2.3	Wpływ wielkości pojedynczego kawałka	58
6.2.4	Wpływ warunków partycjonowania komend renderujących siatki	59
6.2.5	Wpływ liczby stanów wokseli	61
6.3	Wnioski finalne	62
7	Podsumowanie	64
	Bibliografia	66
	A Słowniczek	67

Rozdział 1

Wstęp

Istnieją różne metody opisu obiektów trójwymiarowych w grafice komputerowej. Jedną z nich jest opis wokselowy, gdzie obiekt reprezentowany jest jako wolumen danych w przestrzeni 3D. Taka reprezentacja obiektu w pamięci stawia wyzwanie odnośnie sposobu jego wizualizacji przy pomocy istniejących narzędzi, subrutyn zapewnianych przez karty graficzne i API graficzne tj. OpenGL oraz dużych wymagań pamięciowych wynikających z natury opisu wokselowego. Z tego względu powstało wiele metod, zarówno renderowania jak i sposobu zarządzania pamięcią, które mają na celu optymalizację obu aspektów, tak aby opis wokselowy, na większą skalę, mógł być wykorzystywany w aplikacjach czasu rzeczywistego.

Celem pracy jest omówienie istniejących technik rozwiązywania wyżej wspomnianych problemów, a następnie opracowanie własnego rozwiązania, które zostanie przetestowane pod względem wydajności.

Praca podzielona jest na siedem rozdziałów. W drugim z rozdziałów omówiono istniejące techniki renderowania w opisie wokselowym. W trzecim rozdziale przyjrzano się wybranym strukturam danych, metodom zarządzania pamięcią wykorzystywanych w opisie wokselowym obiektów trójwymiarowych. W rozdziale czwartym opisano własne rozwiązanie. W rozdziale piątym krótko opisano środowisko demonstracyjne w którym przeprowadzono testy wydajnościowe. W szóstym rozdziale przeanalizowano wyniki testów wydajnościowych. W ostatnim rozdziale znajduje się podsumowanie pracy.

Rozdział 2

Omówienie technik renderingu grafiki 3D w oparciu o woksele

W niniejszym rozdziale omówiono wybrane, istniejące techniki renderowania obiektów w opisie wokselowym. Rozdział składa się z opisu trzech popularnych rozwiązań. W pierwszym podrozdziale opisano rozwiązanie bazujące na reprezentacji wokseli w postaci kostek (*boxel representation*). W drugim podrozdziale omówiono rozwiązanie opierające się na śledzeniu stożka (*cone tracing*) przez wolumen reprezentujący obiekt w trójwymiarze. W trzecim podrozdziale przyjrzano się rozwiązaniu maszerujących kostek (*marching cubes*), w którym konwertuje się wolumen do klasycznej siatki wielokątów.

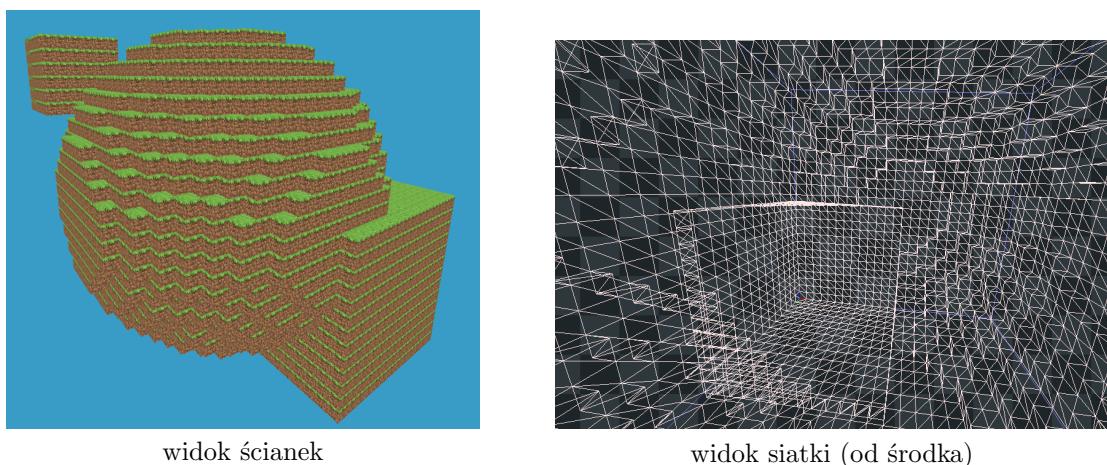
2.1 Reprezentacja bokselowa

Przy reprezentacji bokselowej woksel reprezentowany jest jako kostka składająca się z wielokątów w przestrzeni trójwymiarowej, co określa się mianem boksela. Przy tej reprezentacji wolumeny wokseli są renderowane przy pomocy klasycznej metody rasterowania wielokątów. Problemem jaki stawia ta metoda jest sama generacja siatki z wolumenu i potencjalnie bardzo duży rozmiar wygenerowanej siatki.

2.1.1 Rozwiążanie naiwne

Zakładając, że pojedynczy wierzchołek zawiera pozycje w przestrzeni trójwymiarowej na trzech liczbach zmiennoprzecinkowych, oraz współrzędne tekstury, również na trzech liczbach zmiennoprzecinkowych, i przy każdej widocznej ściance danego woksela liczba wierzchołków rośnie o 6 (ścianka rysowana jest za pomocą dwóch trójkątów i określana jest jako niewidoczna gdy przykrywa ją inny widoczny woksel), to każda kolejna ściana zużywa $6 \cdot 2 \cdot 3 \cdot 4 = 144B$ pamięci serwera.

W poniższym przykładzie (rysunek 1) dla liczby widocznych wokseli równej 9532 liczba wierzchołków wyniosła 20232.



Rys. 1 – rozwiązanie naiwne, przykład

2.1.2 Upraszczanie siatki

Popularnym rozaniem jest zastosowanie algorytmu upraszczającego siatkę reprezentującą wolumen. Wykorzystywany jest do tego algorytm chciwego siatkowania (*greedy meshing*). Pozwala on łączyć mniejsze ścianki, takich samych wokseli, tworzące logicznie większy prostokąt w tej samej płaszczyźnie, w większe ścianki. Wykorzystywana jest tu technika zachlanna.

Działanie algorytmu można opisać pseudokodem z fragmentu kodu numer 1

```

# PROCESSED_PLANE — aktualnie przetwarzany plan w danej orientacji w wolumenie
# POLARITY — 1 siatka planu wskazuje w pozytywna stronę osi, -1 siatka planu wskazuje w negatywna stronę osi
# EDGE_VALUE — wartość brzegowa, (skraj wolumenu dla przetwarzanej osi)

var quads_visibility[W][H] = { false }

any_visible = false
for row in range(H):
    for column in range(W):
        if voxelVisible(row, column, PROCESSED_PLANE):
            if PROCESSED_PLANE == EDGE_VALUE:
                quads_visibility[column][row] = true
                any_visible = true
            continue
        if !voxelVisible(row, column, PROCESSED_PLANE + POLARITY):
            quads_visibility[column][row] = true
            any_visible = true

if any_visible:
    for row in range(H):
        for column in range(W):
            if (quads_visibility[column][row]):
                var voxel_value = fetchVoxel(row, column, PROCESSED_PLANE)
                var mesh_region = (1, 1)
                while (column + mesh_region[0] < W):
                    if (quads_visibility[column + mesh_region[0]][row]):
                        if (fetchVoxel(row, column + mesh_region[0], PROCESSED_PLANE) != voxel_value):
                            break
                        else:
                            break
                    ++mesh_region[0]

                while (row + mesh_region[1] < H):
                    var tmp_mesh_region = 0
                    while (column + tmp_mesh_region < column + mesh_region[0]):
                        if (quads_visibility[column + tmp_mesh_region][row + mesh_region[1]]):
                            if (fetchVoxel(row + mesh_region[1], column + tmp_mesh_region, PROCESSED_PLANE) != voxel_value):
                                goto _exit
                            else:
                                goto _exit
                            ++tmp_mesh_region
                        ++mesh_region[1]
                    _exit:
                    writeQuad(column, row, mesh_region, PROCESSED_PLANE)

                for _row in range(row, row + mesh_region[1]):
                    for _column in range(column, column + mesh_region[0]):
                        quads_visibility[_column][_row] = false

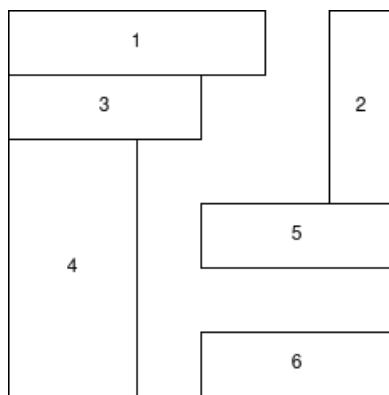
                column += mesh_region[0] - 1

```

Fragment kodu nr. 1 – pseudokod algorytmu chciwego siatkowania

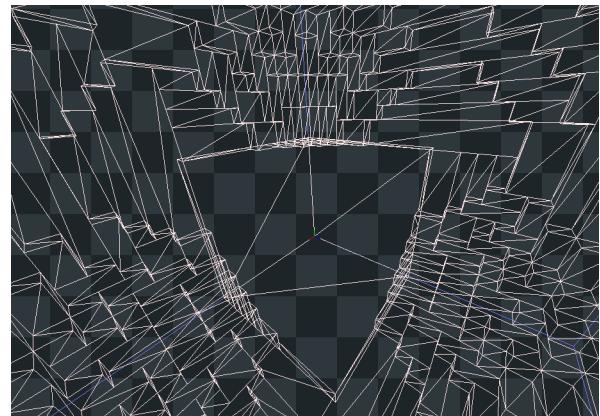
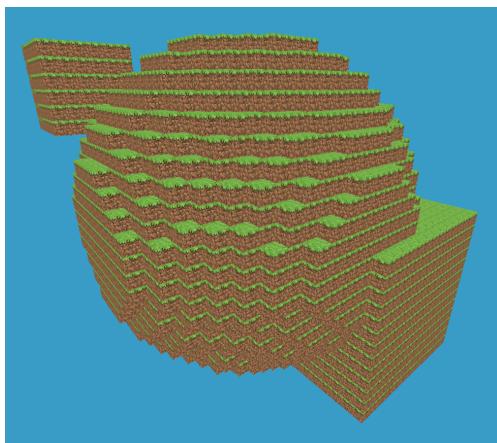
Powyższy algorytm jest przeprowadzany na każdej płaszczyźnie w każdej osi dwukrotnie (jeden raz dla ścianek wskazujących w stronę pozytywnych wartości na danej osi i drugi raz dla ścianek wskazujących w stronę negatywnych wartości na danej osi. Stąd złożoność wyżej przedstawionego algorytmu to $O(2 \cdot ((X+1) \cdot (Y \cdot Z) + (Y+1) \cdot (X \cdot Z) + (Z+1) \cdot (X \cdot Y)))$, gdzie X, Y, Z to wymiary wolumenu na którym przeprowadzany jest algorytm. Jeśli $X = Y = Z = N$ wtedy zapis złożoności można uprościć do $O(6 \cdot N^3 + 6 \cdot N^2)$, co oznacza złożoność sześcienną.

Na rysunku 2 poniżej przedstawiono podział na prostokąty przykładowej płaszczyzny za pomocą wyżej opisanego algorytmu. Numeracja oznacza kolejność dodania prostokątów.



Rys. 2 – wizualne przedstawienie przykładowego wyniku algorytmu chciwego siatkowania (numerki oznaczają kolejność dodania prostokątów)

Na rysunku 3 dla liczby widocznych wokseli równej 9532 liczba wierzchołków wyniosła 7182. Jest to taki sam wolumen jak w przykładzie rozwiązania naiwnego, co oznacza, że liczba wierzchołków zmniejszyła się prawie trzykrotnie.



Rys. 3 – rozwiązywanie z upraszczaniem siatki, przykład

Należy zwrócić uwagę na to, że liczba wierzchołków nie zawsze się zredukuje, np. w przypadku gdy wolumen opisuje trójwymiarową szachownicę, algorytm jest całkowicie nieskuteczny.

Zaletą wyżej wymienionego rozwiązania jest prostota, uniwersalność oraz możliwość nakładania tekstur na boksele. Z kolei wadą, potencjalnie duże wymagania pamięciowe, wynikające z konieczności generacji siatki na podstawie wolumenu i sama konieczność generacji siatki.

2.2 Śledzenie stożka w wolumenie

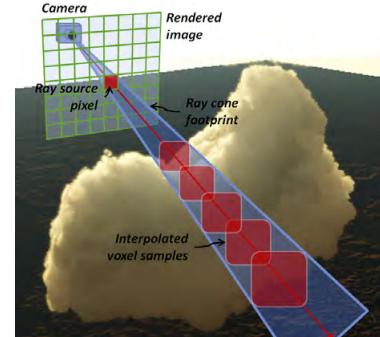
Opisywane w sekcji rozwiązanie oparto na [17].

W podstawowej wersji rozwiązania, rzutowaniu promienia (*ray casting*), na każdy piksel przypada promień potencjalnie przecinający wolumen. Zaczynając od planu bliskiego akumuluje się kolor oraz przeźroczystość, które pobierane są z renderowanego wolumenu. Gdy wartość przeźroczystości wyniesie 1 lub promień opuści wolumen akumulacja jest przerywana. Zakumulowane wartości są kolorem danego piksela.

Opisywane tu rozwiązanie, dla każdego promienia, implementuje algorytm śledzenia stożka (*cone tracing*). Zastosowanie algorytmu redukuje szum (*aliasing*) wynikający z faktu, że im dalej promień znajduje się od źródła (które jest docelowym pikselem na ekranie na który rzutowany jest kolor zakumulowany z tego promienia), tym więcej obszaru wolumenu nań przypada.

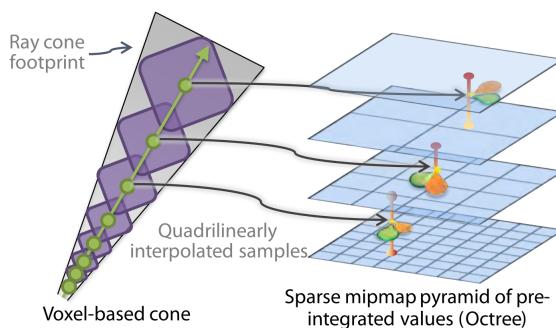
Jak widać na rysunku 4, wielkość obszaru przypadającego na piksel źródłowy (*Ray source pixel*), określa się mianem śladu stożka promienia (*Ray cone footprint*). Aby efektywnie wykorzystać tą informację, dane wolumenu przechowywane są w rzadkim drzewie ósemkowym (ta struktura danych zostanie dokładniej opisane rozdziale 3.1) pozwalającym przechowywać różne rozdzielczości wolumenu na kolejnych poziomach drzewa i optymalnie pomijać obszary puste lub mające tą samą wartość.

W zależności od grubości stożka w danym punkcie, aproksymowane jest z których rozdzielczości wolumenu pobierać wartość do akumulacji, odpowiada to czerwonym kwadratom na rysunku. Interpolacja następuje wewnętrz subwolumenów odpowiadających wyższemu poziomowi detali i niższemu poziomowi detali. Wyższy i niższy poziom detali



Rys. 4 – śledzenie stożka
wizualizacja [17]

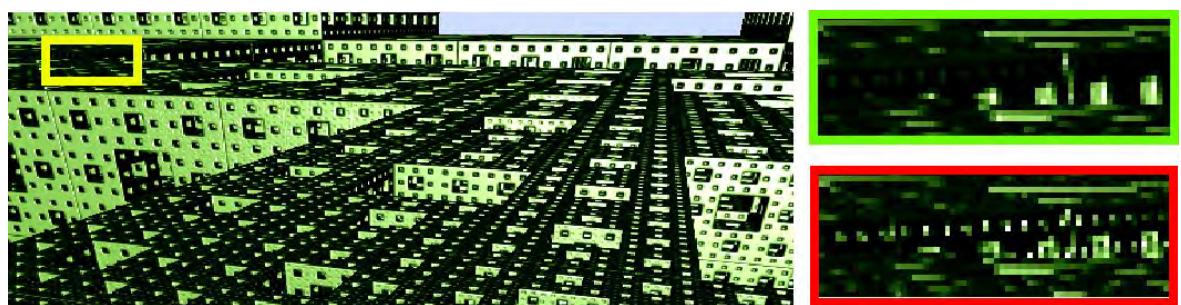
jest wybierany na podstawie śladu stożka w danym punkcie próbkowania na promieniu. Interpolacje wewnętrz tych subwolumenów następują względem punktu próbkowania na promieniu, pozycji próbkowania. Następnie następuje interpolacja również pomiędzy obliczonymi wartościami interpolacji z subwolumenów dwóch rozdzielczości. Wynika to z tego że ślad stożka promienia w danym punkcie jest ciągły, z kolei przechowywane rozdzielczości wolumenu są dyskretne.



Rys. 5 – zależność między śladem stożka a próbowanymi rozdzielczościami [17] (quadrilinear).

Obrazuje to rysunek 5. Interpolacja pomiędzy rozdzielczościami zapewnia lepsze rezultaty pod względem aproksymacji subwolumenu w danym punkcie próbkowania na promieniu. Ze względu na fakt że najpierw następuje interpolacja wewnętrz subwolumenów różnej rozdzielczości, a następnie interpolacja pomiędzy tymi subwolumenami, jest to interpolacja czworoliniowa

Na rysunku 6 przedstawione jest wizualne porównanie metody używającej algorytmu śledzenia stożka (*cone tracing*), zaznaczony obramówką zieloną, i metodę używającą zwykłe rzutowania promienia (*ray casting*), zaznaczony obramówką czerwoną. Widoczny jest dużo mniejszy aliasing w przypadku metody korzystającej z algorytmu śledzenia stożka.



Rys. 6 – porównanie wyniku metod, rzutowanie promienia w czerwonej obramówce, śledzenie stożka w zielonej obramówce [17]

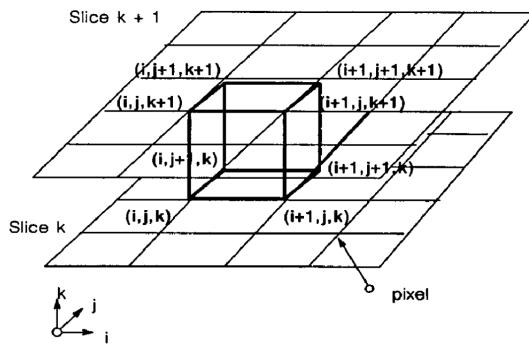
Zalną rozwiązań jest możliwość optymalnego renderowania bardzo dużych wolumenów w wysokiej rozdzielczości. Wadą jest złożoność rozwiązania i niejako

narzucenie metody przechowywania wolumenu (rzadkie drzewo ósemkowe), do którego wad zalicza między innymi duży koszt modyfikacji drzewa, przy modyfikacji wolumenu.

2.3 Maszerujące kostki

Metoda maszerujących kostek (*Marching cubes*) została po raz pierwszy zaprezentowana w [21]. Opiera się ona, podobnie jak wcześniej opisana metoda reprezentacji bokselowej, na generacji siatki z wolumenu. W tym wypadku jednak generowana siatka ma na celu lepiej wizualizować krzywiznę powierzchni.

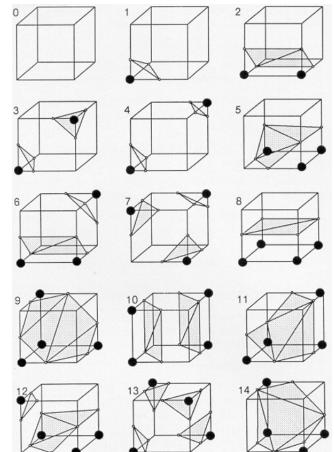
Algorytm maszerujących kostek traktuje grupy po osiem wokseli (w wolemenie), jako punkty tworzące kostkę. Przedstawiono to na rysunku 7.



Rys. 7 – woksele tworzące kostkę, wizualizacja [21]

Następnie na podstawie stanów wokseli, widoczny lub niewidoczny, i w jakim są położeniu względem siebie, determinuje jaka powierzchnia powstaje z tych wokseli, gdzie powierzchnia, to trójkąty tworzone z wierzchołków znajdujących się na środkach krawędzi tworzących kostkę.

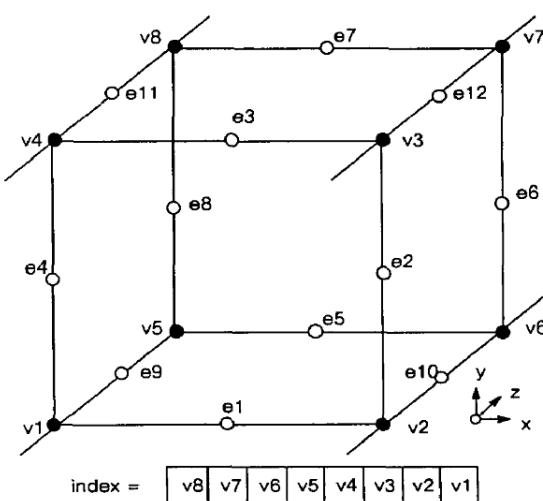
Z racji że kostka ma osiem wierzchołków (które są wokselami) i każdy woksel może mieć dwa stany, widoczny lub niewidoczny, daje to 2^8 , czyli 256 różnych konfiguracji ścianek w kostce. Zbiór ten można zredukować do zaledwie 14 unikalnych konfiguracji [21], które widać na rysunku numer 8. Najpierw redukuje się liczbę konfiguracji do 128. Wynika to z tego, że topologia konfiguracji nie zmieni się jeśli brać pod uwagę konfiguracje o danej liczbie wokseli



Rys. 8 – 14 unikalnych konfiguracji siatek [21]

widocznych i niewidocznych, i tą po odwróceniu wartości każdego woksela na wartość przeciwną w tej konfiguracji. Następnie wartość redukuje się do 14 po wykluczeniu symetrycznych konfiguracji.

Taką 256 elementową tablicę konfiguracji indeksuje się za pomocą liczby ośmio bitowej, tworzonej ze stanów wokseli tworzących kostkę, jak przedstawiono na rysunku numer 9. Przy przechodzeniu przez wolumen po każdej grupie ośmiu wokseli tworzących kostkę, dodaje się odpowiednie płaszczyzny (konfiguracji) do siatki mającej ten wolumen reprezentować.



Rys. 9 – 14 unikalnych konfiguracji siatek [21]

Zaletą metody maszerujących kostek jest konwersja wolumenów do klasycznej reprezentacji w postaci siatki, co zapewnia kompatybilność z klasycznymi systemami do renderowania obiektów 3D. Wadą jest niska jakość wygenerowanej siatki w niektórych przypadkach i sama konieczność konwersji wolumenu do siatki.

Rozdział 3

Omówienie struktur danych, metod zarządzania pamięcią wykorzystywanych w opisie wokselowym obiektów trójwymiarowych

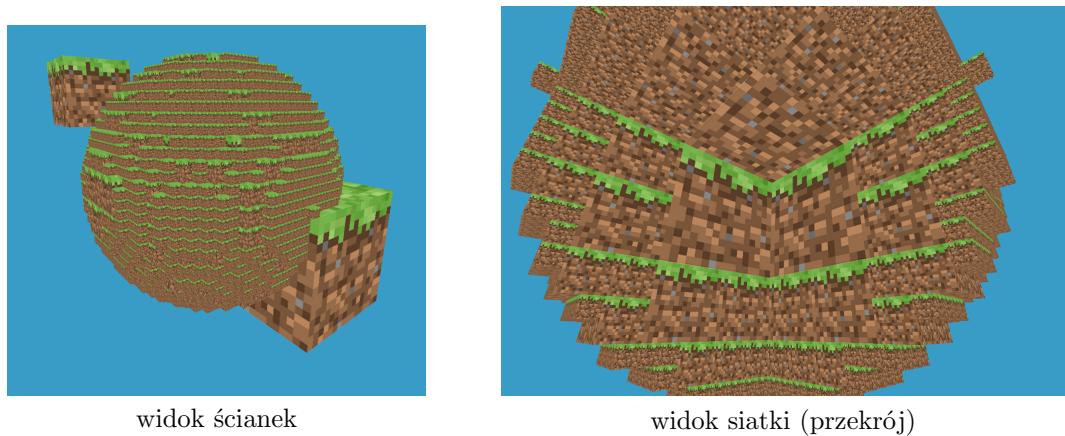
Głównym problemem opisu wokselowego obiektów trójwymiarowych są duże wymagania pamięciowe. W tym rozdziale zostaną omówione dwie popularne metody zarządzania pamięcią obiektów w opisie wokselowym, które starają się rozwiązać ten problem. Pierwszym z rozwiązań jest rozwiązanie oparte na rzadkich N^3 drzewach (*sparse N^3 trees*). Drugim z rozwiązań jest to wykorzystywane przez popularną grę Minecraft czyli podział wolumenu na mniejsze kawałki.

3.1 Rzadkie N^3 drzewa

Rzadkie N^3 drzewa są generalizacją rzadkich drzew ósemkowych. Rzadkie drzewo ósemkowe dzieli wolumen rekurencyjnie na 8 jednakowych obszarów, każdy z takich obszarów to węzeł drzewa. Dodatkowo, jeśli na danym poziomie drzewa dany węzeł stanowiący daną część wolumenu jest pusty lub wszystkie wartości wewnętrz obszaru opisywanego przez ten węzeł są jednakowe, drzewo może zostać ucięte na

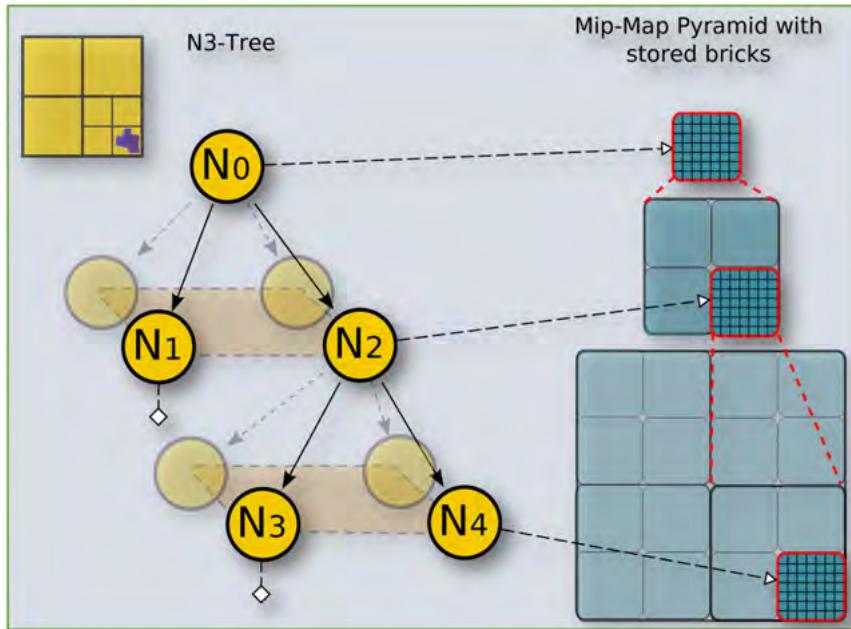
Rozdział 3: Omówienie struktur danych, metod zarządzania pamięcią wykorzystywanych w opisie wokselowym obiektów trójwymiarowych

poziomie tego węzła, co redukuje zużycie pamięci. Oprócz optymalizacji pamięciowej drzewo ósemkowe daje możliwość przechowywania różnych rozdzielczości opisywanego przez nań wolumenu na kolejnych poziomach drzewa. Użyteczność przechowywania rozdzielczości opisano dokładniej w rozdziale 2.2. Na rysunku 10 zobrazowano koncepcję drzewa ósemkowego. Potencjalna liczba wokseli w poniższym wolumenie to 32768 (rozdzielcość wolumenu to 32^3). Rzadkie drzewo ósemkowe powstałe na podstawie wolumenu poniżej ma 6 poziomów, 20466 węzłów i 2580 liści. Można więc powiedzieć że rzadkie drzewo ósemkowe skompresowało wolumen o 92% (brano pod uwagę liczbę liści drzewa w stosunku do potencjalnej liczby wokseli).



Rys. 10 – wizualne przedstawienie konceptu drzewa ósemkowego, im większy boksel tym wcześniejsza gałąź drzewa została ucięta

Drzewo ósemkowe jest N^3 drzewem z N równym 2. Manipulacja N zwiększa liczbę dzieci węzła na każdym poziomie. Na przykład, przy N równym 3, liczba dzieci węzła to 27. Manipulacja N kontroluje kompromis między optymalizacją zużycia pamięci (niskie N , więcej poziomów), a optymalizacją szybkości przechodzenia po drzewie (wysokie N , mniej poziomów) [17]. Rozwiązań zaprezentowane w [17] na każdym węźle przechowuje tzw. cegły (*bricks*), są to małe (zazwyczaj o rozdzielcości 8^3) subwolumeny aproksymujące część wolumenu przypadającą na dany węzeł drzewa [17]. Takie rozwiązanie umożliwia przechowywanie danych w teksturach 3D na GPU i jednocześnie korzystanie z kompresji danych jaką zapewnia rzadkie N^3 drzewo [17]. Poniżej na rysunku 11 zobrazowana jest opisana wyżej struktura danych.

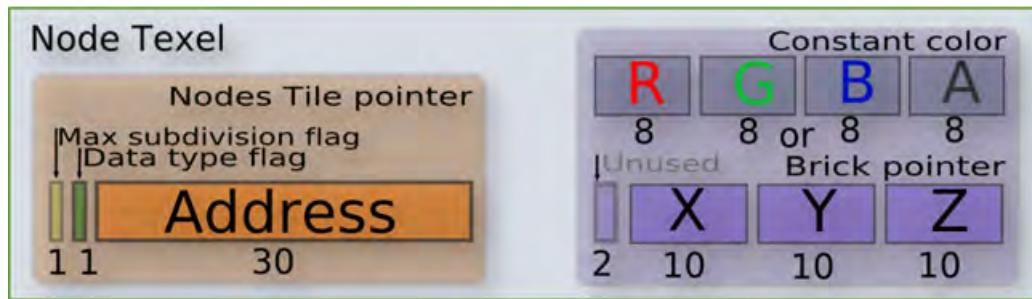


Rys. 11 – struktura N^3 drzewa, żółte kółka z N_x to węzły natomiast niebieskie kratki to cegły, węzły wskazują na cegły co jest zobrazowane przez strzałki przerywane [17]

Nie wszystkie węzły wskazują na cegły. Jeśli węzeł jest pusty lub wypełniony tą samą wartością, w węźle jest przechowywana tylko ta informacja [17].

Wartości wokeli w cegłach przechowują różne kanały, tj. kolor i przeźroczystość. Mogą być rozszerzane o inne kanały np. informacje dotyczące nałożonego materiału [17].

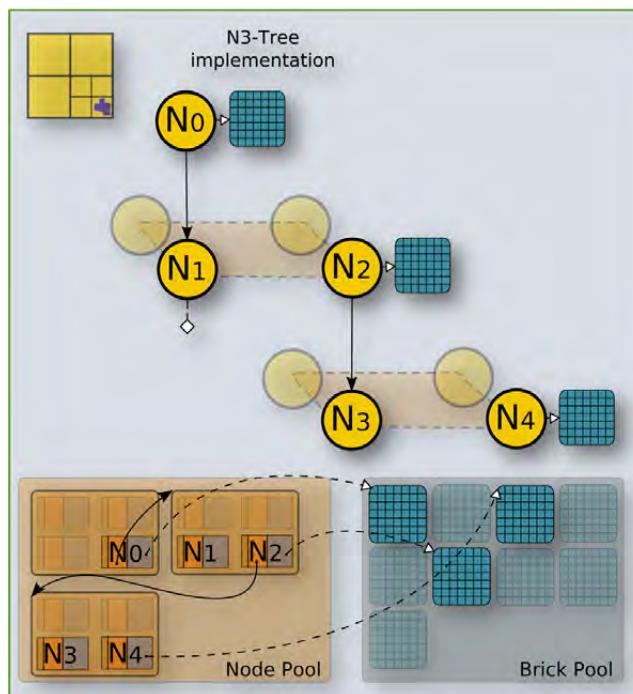
Węzły oraz cegły przechowywane są w prealokowanych pulach pamięci. Pule pamięci alokowane są raz przy inicjalizacji i zarządzane w zakresie implementacji [17]. Węzły przechowują wskaźniki do początku obszaru puli węzłów gdzie znajdują się ich dzieci (pula węzłów ułożona jest w taki sposób, że dzieci danego węzła są posortowane w jednym ciągłym obszarze puli węzłów co zmniejsza wymagania pamięciowe) na 32 bitach, gdzie 31 bit oznacza, czy węzeł może być dalej dzielony (mieć dzieci), oraz adres cegły na 30 bitach, który może być również stałym kolorem na 32 bitach. To, czy węzeł jest adresem cegły czy stałym kolorem jest sygnalizowane przez 30 bit w adresie obszaru dzieci węzła. Przedstawiono to na rysunku 12.



Rys. 12 – struktura przechowywana w każdym węźle drzewa [17]

Należy zaznaczyć że wskaźnik do dzieci węzła i kolor, wskaźnik do cegły węzła, są przechowywane w oddzielnych ciągłych obszarach pamięci, nie są przeplatane (*interleaved*) czyli przechowywane jako tablica struktur (*AOS*, *Array Of Structures*), a jako struktura tablic (*SOA*, *Structure Of Arrays*). Zwiększa to szybkość przechodzenia po drzewie [17].

Tak samo przechowywane są opisane kanały wokseli cegieł. W tym wypadku umożliwia to użycie wbudowanych subrutyn GPU tj. interpolacja tekstur, adresowanie 3D i optymalizacja dostępów do pamięci. Strukturę systemu przechowywania węzłów obrazuje rysunek 13.



Rys. 13 – zobrazowanie przechowywania węzłów i cegieł [17]

Zaletą powyższego rozwiązania jest duży stopień kompresji danych przy jednoczesnej lokalności danych w pamięci, wadą duży koszt modyfikacji wolumenu.

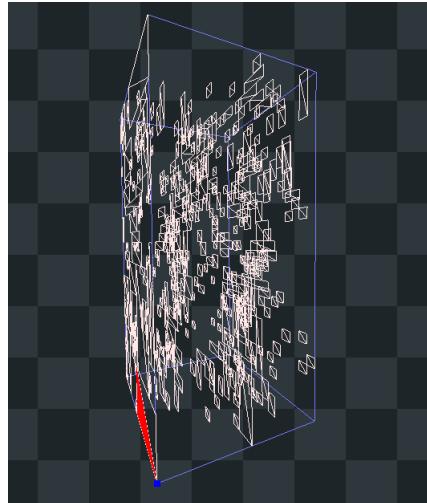
3.2 Podział wolumenu na kawałki

Innym popularnym rozwiązaniem jest podział wolumenu na kawałki (*chunks*). Wykorzystuje to popularna gra Minecraft i wiele innych rozwiązań bazujących na reprezentacji bokselowej opisanej w rozdziale 2.1. Ideą rozwiązania jest wybranie stałego wymiaru kawałka i zarządzanie pamięcią w takich kawałkach.

Rozwiązanie zaprezentowane w [9] proponuje utworzenie puli pamięci siatek kawałków. Póła, to bufor GPU przechowujący wierzchołki siatek kawałków. Dodatkowo w obszarze puli danego kawałka utworzonych jest 6 subobszarów, każdy na inną orientację siatki. Gdzie orientacje to $+X$, $-X$, $+Y$, $-Y$, $+Z$, $-Z$. Na rysunku 14 widać przykład takiej siatki.

Następnie tworzona jest kolejka przechowująca informacje o wolnych obszarach puli. Z kolejki zdejmowany jest element przy alokacji obszaru i dodawany jest element przy dealokacji obszaru. Przy alokacji obszaru, do obszaru pisana jest siatka generowana za pomocą algorytmu chciwego siatkowania dla każdej orientacji siatki alokowanego kawałka. Algorytm chciwego siatkowania opisano w rozdziale 2.1.

Dla każdej siatki przyporządkowana jest komenda ją renderującą. Komendy przechowywane są w ciągłym buforze w pamięci GPU. Takie rozwiązanie umożliwia łatwy podział binarny zbioru komend rysujących kawałki względem jakiegoś kryterium. Kryterium może być np. warunek, czy dany kawałek znajduje się w ścieżym stożku kamery (*camera frustum*) oraz, dzięki dodatkowemu podziałowi obszaru kawałka na subobszary dla każdej orientacji siatki, również, czy dana orientacja siatki kawałków jest odwrócona tyłem do kamery (*back face culling*). Po takim podziale zbioru możliwe jest renderowanie tylko tych kawałków które spełniają dane warunki.



Rys. 14 – przykład siatki kawałka dla orientacji $+X$

Rozdział 3: Omówienie struktur danych, metod zarządzania pamięcią wykorzystywanych w opisie wokselowym obiektów trójwymiarowych

Zaletą rozwiązania jest łatwa rozszerzalność renderowanego wolumenu o nowe kawałki i usuwanie kawałków które np. nie są już widoczne, co idealnie nadaje się do renderowania terenu. Wadą, mała kompresja danych w porównaniu z rozwiązaniem opartym o rzadkie N^3 drzewa.

Rozdział 4

Zastosowane rozwiązanie

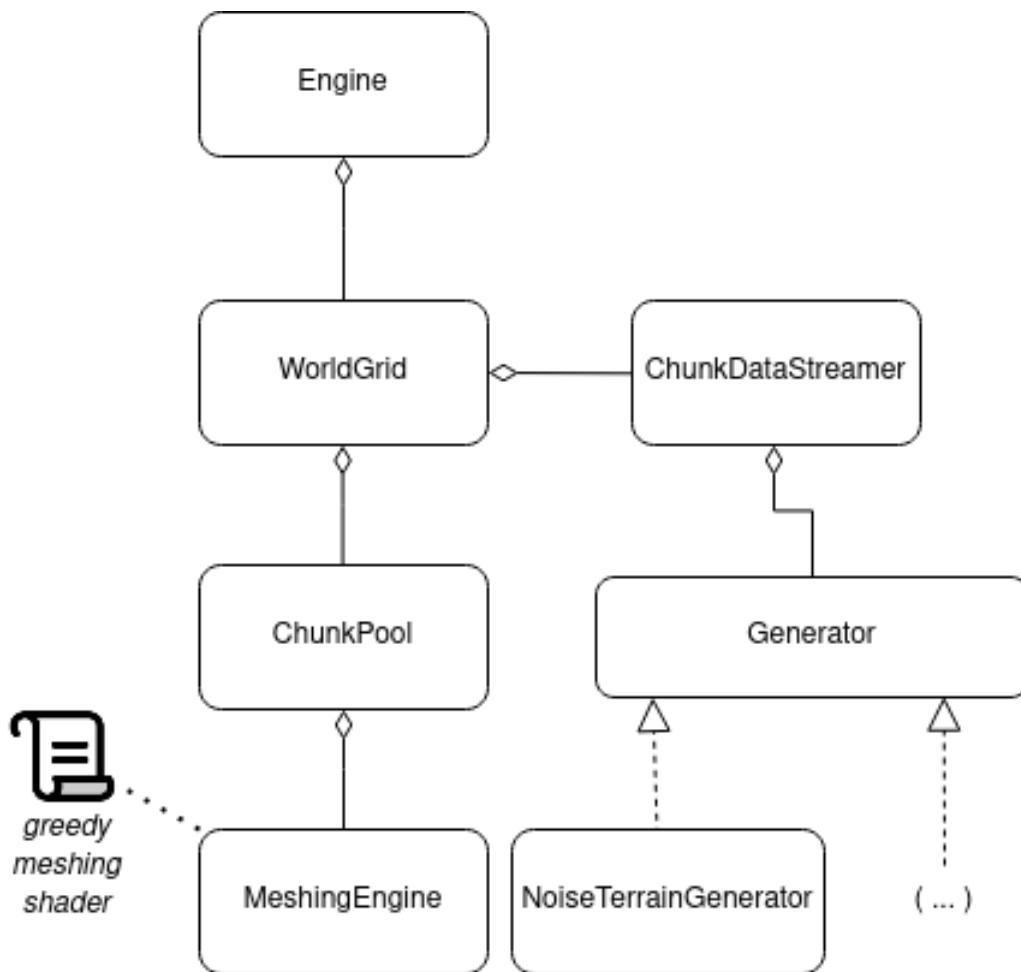
Własne rozwiązanie oparto na metodach reprezentacji bokselowej opisanej w rozdziale 2.1 oraz podziału wolumenu na kawałki opisanego w rozdziale 3.2. Niniejszy rozdział posiada kilka podrozdziałów. W pierwszym podrozdziale opisano środowisko implementacji wokselowego silnika graficznego. Następnie ogólnie opisano strukturę silnika. W kolejnych podrozdziałach opisano szczegółowo moduły wymienione w ogólnym opisie.

4.1 Środowisko implementacji

Silnik zaimplementowano jako statyczną bibliotekę w języku C++20. W implementacji użyto wersji *core 4.5* API OpenGL [18]. Jest to wersja najniższa wymagana aby bez przeszkodek zintegrować bibliotekę w przykładowej aplikacji. Biblioteka korzysta z rozszerzenia *GL_ARB_gl_spirv* [2], jeśli wsparcie rozszerzenia zostanie wykryte. Moduły cieniujące napisano w języku *GLSL*. Do wczytywania funkcji OpenGL API użyto *glad loader* [7].

4.2 Struktura silnika

Silnik składa się z kilku modułów, które w implementacji przekładają się na klasy w języku C++. Strukturę silnika przedstawiono na rysunku 15.



Rys. 15 – struktura silnika przedstawiona za pomocą diagramu klas

Funkcjonalności elementów diagramu z rysunku 15 można streszczyć następująco

- *Engine* - stanowi interfejs dla aplikacji korzystającej z silnika, agreguje wszystkie funkcjonalności.
- *WorldGrid* - zarządza stanem widoczności kawałków wolumenu w stosunku do przekazanej pozycji kamery.
- *ChunkPool* - zarządza pamięcią w której przechowuje się siatki, dane kawałków wolumenu oraz komendy renderujące siatki.
- *MeshingEngine* - kolejkuje zadania siatkowania kawałków wolumenu i wykonuje je na serwerze.
- *greedy meshing shader* - kod algorytmu chciwego siatkowania w języku GLSL.

- *ChunkDataStreamer* - implementuje pule wątków i kolejkuje zadania generacji danych zadanych kawałków wolumenu.
- *Generator/ChunkGenerator* - abstrakcyjna klasa stanowiąca interfejs dla generatorów danych kawałków wolumenu.

4.3 *ChunkPool* - szczegóły implementacji

Ideę działania modułu oparto na rozwiązaniu opisanym w rozdziale 3.2.

ChunkPool przechowuje dane kawałków wolumenu i siatek wygenerowanych na ich podstawie w regionach prealokowanych pul pamięci. Alokuje również komendy renderujące zaalokowane siatki.

4.3.1 Dane kawałków wolumenu

Dane kawałków wolumenu przechowywane są w regionach tablicy alokowanej z pamięci klienta, składającej się z nieujemnych liczb całkowitych wielkości 16 bitów. Pojedyncza liczba 16 bitowa reprezentuje wartość woksela. Taka reprezentacja nakłada ograniczenie odnośnie liczby unikalnych wokseli w wolumenie na 2^{16} oraz wymaga aby dane były dyskretne. Dane te służą do generacji siatki opisywanego przez nań kawałka wolumenu. Przechowuje się je ponieważ wielkość bufora przechowującego siatki może ulec zmianie. Zmiana tego bufora powoduje ponowne siatkowanie wszystkich zaalokowanych kawałków wolumenów. Wielkość tablicy obliczana jest na podstawie maksymalnej liczby kawałków puli i rozmiarze pojedynczego kawałka w następujący sposób: $C_{max} \cdot X \cdot Y \cdot Z$. Rozmiar tablicy jest stały i nie zmienia się po inicjalizacji silnika. Liczba regionów bufora również jest stała i nie zmienia się. Podział tablicy na regiony przedstawiono na rysunku 16.

R0	R1	R2	R3	R4	R4	R5	(...)	RC-2	RC-1
----	----	----	----	----	----	----	-------	------	------

Rys. 16 – regiony przechowującego dane kawałków wolumenu (R_x), C równe jest C_{max}

4.3.2 Dane siatek

Dane siatek wygenerowanych na podstawie danych kawałków wolumenu przechowuje się w regionach bufora alokowanego na serwerze. Bufor przekłada się w implementacji na *Vertex Buffer Object (VBO)* służący do przechowywania danych wierzchołkowych w API OpenGL. Liczba regionów bufora jest stała i nie zmienia się po inicjalizacji. Wielkość regionu, a za tym wielkość bufora w zależności od konfiguracji silnika, może się dynamicznie zwiększyć w trakcie działania. Każdy region *VBO* powiązany jest z regionem danych kawałka wolumenu. Podział na regiony obrazuje rysunek 16.

Wierzchołek

Wierzchołek opisany jest za pomocą pozycji w przestrzeni 3D na trzech 32 bitowych liczbach zmiennoprzecinkowych oraz koordynatu tekstury w takim samym formacie. Obrazuje to fragment kodu numer 2

```
struct Vertex {  
    vmath::Vec3f32 position;  
    vmath::Vec3f32 texcoord;  
};
```

Fragment kodu nr. 2 – wierzchołek pokazany za pomocą struktury z języka C++

Dwie pierwsze liczby z koordynatu tekstury opisują standardową projekcję tekstury na dany kawałek siatki. Trzecia liczba koduje dwie informacje. Pierwsza informacja to wartość wokselu którą reprezentuje dany wierzchołek, pozyskuje się ją poprzez podzielenie liczby przez 6. Drugą informacją jest wektor normalny siatki tworzonej przez dany wierzchołek. Z racji, że przechowywane siatki mogą mieć tylko 6 orientacji (jak opisano w sekcji "Podregiony regionów VBO" w tym rozdziale), pozyskuje się ją przy przeprowadzeniu na liczbie rachunku modulo 6. Kolejne liczby całkowite 0, 1, 2, 3, 4, 5 z ciała modulo 6 odpowiadają wektorom normalnym [1, 0, 0], [-1, 0, 0], [0, 1, 0], [0, -1, 0], [0, 0, 1], [0, 0, -1]. Takie kodowanie wektorów normalnych umożliwia zmniejszenie wymagań pamięciowych poprzez nie przechowywanie wektora normalnego dla każdego wierzchołka. Wielkość pojedynczego wierzchołka to $2 \cdot 3 \cdot 4 = 24$ bajty. W przypadku gdy przechowywano by wektor normalny w postaci nieskompresowanej, wielkość wierzchołka zwiększyła by się o 12 bajtów, i wynosiłaby $3 \cdot 3 \cdot 4 = 36$.

Układ wierzchołków w buforze

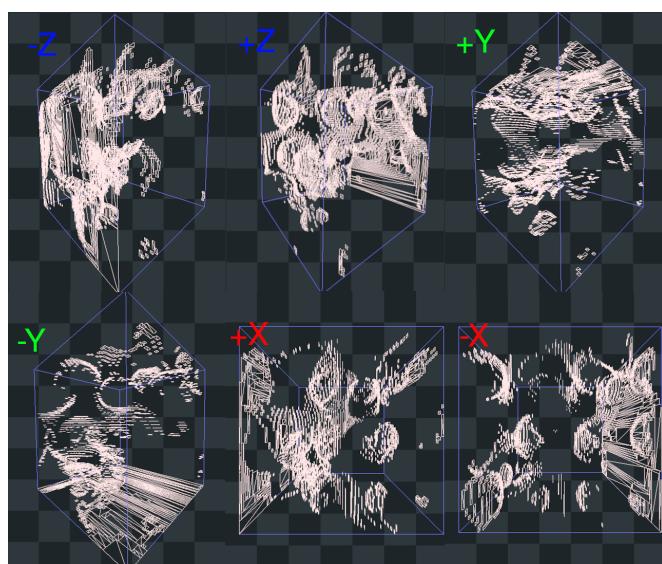
ChunkPool inicjalizuje *Vertex Array Object* (*VAO*) definiujący układ danych wierzchołków w buforze *VBO*. Dane ułożone są w sposób przeplatany. Oznacza to, że naprzemiennie w buforze występuje najpierw pozycja w przestrzeni 3D a następnie koordynat tekstuury.

Pojedynczy kawałek siatki

Wierzchołki w danym regionie bufora tworzą prostokątne kawałki siatki, także każde 4 kolejne wierzchołki tworzą prostokąt. Stąd wynika, że pojedynczy prostokątny kawałek siatki zajmuje $4 \cdot 24 = 96$ bajtów.

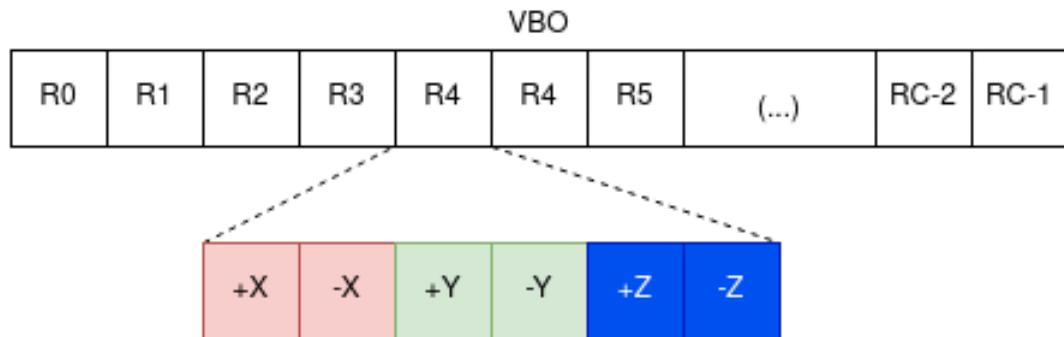
Podregiony regionów *VBO*

Pojedynczy region *VBO* dzielony jest dodatkowo na 6 podregionów. Każdy podregion przechowuje siatkę kawałka dla danej orientacji. Podobnie jak opisano to w rozdziale 3.2, orientacje to $+X$, $-X$, $+Y$, $-Y$, $+Z$, $-Z$. Przykładową siatkę dla każdej orientacji przedstawiono na rysunku 17.



Rys. 17 – pokazanie siatek dla każdej orientacji jednego kawałka wolumenu

Podział na regionów bufora *VBO* na podregiony obrazuje rysunek 18.



Rys. 18 – podział regionu VBO na podregiony dla każdej orientacji siatki

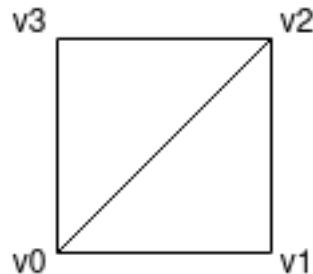
Taki podział zapewnia większą swobodę definiowania filtrów siatek (co opisane jest dokładniej w rozdziale 4.3.3) oraz większą paralelizację procesu generacji siatki (co jest opisane w rozdziale 4.4).

Maksymalna wielkość bufora *VBO*

Maksymalna logiczna wielkość bufora wynika z najgorszego możliwego wyniku algorytmu chciwego siatkowania. Najgorszy przypadek nastąpi gdy dane kawałka wolumenu reprezentują trójwymiarową szachownicę, gdzie połowa wartości to woksele o stanie 0 (niewidoczny) i druga połowa to woksele o stanie większym niż 0 (widoczny), przy takich danych algorytm chciwego siatkowania jest nieskuteczny (żadne siatki wokseli nie zostaną scalone). W tym przypadku na jeden woksel przypada 6 prostokątnych kawałków (jeden na każdą stronę kostki) siatki, stąd każdy następny widoczny woksel dodaje $6 \cdot 96 = 576$ bajtów. Stąd, maksymalna możliwa wielkość bufora to: $C_{max} \cdot X \cdot Y \cdot Z \cdot 576 \cdot \frac{1}{2}$ bajtów. Gdzie $\frac{1}{2}$ wynika z faktu, że przy trójwymiarowej szachownicy połowa wokesli ma wartość 0 (jest niewidoczna).

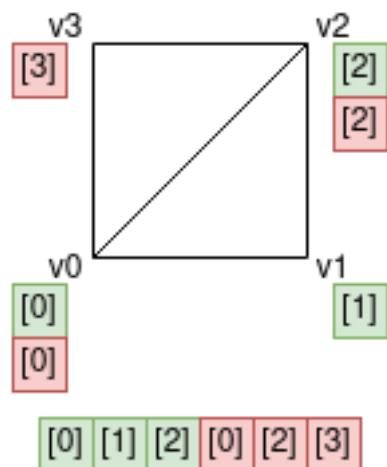
Dane indeksacyjne siatki

Pojedynczy kawałek siatki tworzony jest z 4 wierzchołków. Kawałek siatki renderowany jest jako dwa trójkąty prostokątne takiej samej wielkości, dzielące ze sobą przeciww prostokątną. Widać to na rysunku 19.



Rys. 19 – pojedynczy kawałek siatki

Ze względu na fakt, że kolejno przechowuje się tylko 4 wierzchołki, a nie 6 (z redundancją wierzchołków v_0 i v_2), wymagane jest ich indeksowanie. Sposób indeksowania pokazano na rysunku 20.



Rys. 20 – indeksowanie wierzchołków, zielony kolor - indeksy wierzchołków pierwszego trójkąta, czerwony kolor - indeksy wierzchołków drugiego trójkąta, na dole - indeksy uporządkowane w kolejności

Oznacza to, że każdy pojedynczy kawałek siatki wymaga 6 indeksów, gdzie każdy indeks to liczba całkowita nieujemna reprezentowana na 32 bitach. Z racji, że siatki składają się tylko z takich prostokątnych kawałków, dane indeksów można wykorzystać dla siatki z dowolnego innego podregionu.

Alokuje się pojedynczy bufor *Index Buffer Object (IBO)* zawierający indeksy po stronie serwera. Bufor *IBO*, służy do przechowywania indeksów wierzchołków siatki w API OpenGL. Wielkość bufora podykutowana jest najgorszym możliwym wynikiem algorytmu chciwego siatkowania opisany w sekcji "Maksymalna wielkość bufora

"VBO" powyżej i zdefiniowana jest następująco: $X \cdot Y \cdot Z \cdot 36 \cdot \frac{1}{2} \cdot \frac{1}{6} \cdot 4$ bajtów. Gdzie 36 oznacza liczbę indeksów na kostkę, mnożenie przez $\frac{1}{6}$ wynika z tego, że indeksy są zdefiniowane dla siatki z podregionu, 4 wynika z wielkością indeksu czyli 4 bajty.

Taki bufor wypełnia się jak pokazano we fragmencie kodu numer 3.

```
static constexpr u32 INDICES_PATTERN[6] = { 0U, 1U, 2U, 0U, 2U, 3U };
std::vector<u32> indices(_engine_context.chunk.max_possible_submesh.indices_size/sizeof(u32));
for (std::size_t i{ 0UL }, q{ 0UL }; i < indices.size(); i += 6UL, q += 4UL) {
    indices[i + 0] = INDICES_PATTERN[0] + static_cast<u32>(q);
    indices[i + 1] = INDICES_PATTERN[1] + static_cast<u32>(q);
    indices[i + 2] = INDICES_PATTERN[2] + static_cast<u32>(q);
    indices[i + 3] = INDICES_PATTERN[3] + static_cast<u32>(q);
    indices[i + 4] = INDICES_PATTERN[4] + static_cast<u32>(q);
    indices[i + 5] = INDICES_PATTERN[5] + static_cast<u32>(q);
}
glNamedBufferStorage(
    _ibo_id,
    static_cast<i64>(_engine_context.chunk.max_possible_submesh.indices_size),
    static_cast<const void*>(indices.data()),
    0
);
```

Fragment kodu nr. 3 – wypełnienie IBO indeksami, język C++

Po zaalokowaniu i wypełnieniu jak we fragmencie kodu numer 3 zawartość bufora i jego rozmiar nie zmienia się.

4.3.3 Renderowanie siatek

Każda siatka z zaalokowanego podregionu ma przypisaną komendę ją renderującą. Komendy przechowywane są po stronie klienta oraz stronie serwera. Po stronie serwera do przechowywania komend użyto bufora *Draw Indirect Buffer Object (DIBO)* z OpenGL API. Bufor *DIBO* aktualizowany jest tylko gdy bufor komend po stronie aplikacji ulegnie zmianie. Bufor *DIBO* tworzony jest w sposób pokazany we fragmencie kodu numer 4.

```
glNamedBufferStorage(
    _dibo_id,
    _chunks_count * sizeof(DrawElementsIndirectCmd),
    nullptr,
    GL_MAP_WRITE_BIT|GL_MAP_PERSISTENT_BIT|GL_MAP_COHERENT_BIT
);
```

Fragment kodu nr. 4 – alokacja bufora DIBO, język C++

Wielkość bufora to C_{max} , tu reprezentowane jako *_chunks_count*, pomnożone przez *sizeof(DrawElementsIndirectCmd)*, co jest wielkością w bajtach pojedyńczej komendy

renderującej siatkę, i pomnożoną przez 6, ze względu na fakt, że każdy kawałek ma 6 stron siatki, co opisano w rozdziale 4.3.2. Jak widać we fragmencie kodu numer 4 bufor tworzony jest z flagami *GL_MAP_WRITE_BIT*, *GL_MAP_PERSISTENT_BIT* oraz *GL_MAP_COHERENT_BIT*. Flagi te umożliwiają zmapowanie bufora na wskaźnik po stronie klienta z możliwością zapisu do tej pamięci (flaga *GL_MAP_WRITE_BIT*). Dodatkowo flaga *GL_MAP_PERSISTENT_BIT* zwalnia z wymogu odmapowywania przed każdym wywołaniem komendy renderującej. Flaga *GL_MAP_COHERENT_BIT* zwalnia z konieczności manualnej synchronizacji zapisów do bufora. Bufor mapowany jest w sposób pokazany we fragmencie kodu numer 5.

```
_dibo_mapped_ptr = glMapNamedBufferRange(
    _dibo_id,
    0, 6 * _chunks_count * sizeof(DrawElementsIndirectCmd),
    GL_MAP_WRITE_BIT|GL_MAP_PERSISTENT_BIT|GL_MAP_COHERENT_BIT
);
```

Fragment kodu nr. 5 – mapowanie bufora DIBO na wskaźnik po stronie aplikacji, język C++

Bufor zmapowany jest stałe, aż do deinicjalizacji silnika. Poprzez tak zmapowany bufor, transfer danych następuje szybciej niż w przypadku zwykłego niezmapowanego bufora z dostępem zapisu i odczytu [1].

Pojedyncza komenda reprezentowana jest przez strukturę pokazaną we fragmencie kodu numer 6.

```
/// @brief structure stores drawing command data
/// which is then stored in draw command buffer
struct DrawElementsIndirectCmd {
    /// @brief number of indices
    vmath::u32 count;
    /// @brief number of instances
    vmath::u32 instance_count;
    /// @brief first index from which to start
    vmath::u32 first_index;
    /// @brief base vertex from which indices start to apply
    vmath::i32 base_vertex;
    /// @brief base instance from which to start
    vmath::u32 base_instance;

    /// @brief what is orientation of submesh drawn by this command
    Face orientation;
    /// @brief what is the chunk id to which the drawn by this command
    /// submesh belongs
    ChunkId chunk_id{ 0U };
};
```

Fragment kodu nr. 6 – struktura komendy renderującej siatkę z podregionu, język C++

Wartości parametrów *instance_count*, *first_index* i *base_instance*, są zawsze takie same. Renderuje się zawsze jedną instancję danej siatki, więc *instance_count* wynosi 1. Bazowa instancja, to instancja pierwsza więc *base_instance* zawsze wynosi 0. Parametr *first_index* zawsze wynosi 0, bo renderuje się daną siatkę w całości. Parametr *count* zmienia się w zależności od wielkości siatki i oznacza liczbę indeksów. Parametr *base_vertex* stanowi przesunięcie w buforze *VBO* w wierzchołkach, zależy on od tego z którego regionu i podregionu bufora dana komenda renderuje siatkę. Dwa dodatkowe parametry nie będące interfejsem do API OpenGL to *orientation* oraz *chunk_id*. Parametr *orientation* zawiera informacje o tym, jaką orientację ma dana siatka. Czym jest orientacja siatki opisano w rozdziale 4.3.2. Parametr *chunk_id* to identyfikator zaallokowanego kawałka do którego przynależy renderowana przez komendę siatka. Funkcjonalności tych parametrów opisano w rozdziale 4.3.4.

Komendy wywoływanie są tak jak pokazano we fragmencie kodu numer 7.

```
glMultiDrawElementsIndirect(  
    GL_TRIANGLES,  
    GL_UNSIGNED_INT,  
    nullptr,  
    use_partition ? _draw_cmds.partition_size : _draw_cmds.size(),  
    sizeof(DrawElementsIndirectCmd)  
)
```

Fragment kodu nr. 7 – funkcja wywołująca komendy renderujące siatki z zaallokowanych podregionów, język C++

Fragment kodu numer 7 wywoływany jest raz na klatkę. Taka organizacja komend renderujących siatki zapewnia ograniczenie liczby przesyłanych danych z klienta do serwera w każdej klatce [1].

Partycjonowanie komend

Komendy mogą zostać podzielone względem zadanego kryterium. Takim kryterium może być np., czy dana siatka renderowana przez daną komendę znajduje się we frustum kamery lub, czy dana siatka odwrócona jest tyłem do kamery (*back face culling*). Warunek może być dowolny. Algorytm partycjonowania komend pokazano we fragmencie kodu numer 8, bazuje on na kroku partycjonowania zbioru w algorytmie *quick sort*.

```

/// @brief function partitions the _draw_cmds based on <unary_op> setting the <_draw_cmds_partition_size> member
/// @tparam ...Args types of aux arguments to pass to unary_op function
/// @param unary_op unary operation which is criterion based on which the commands are partitioned
/// @param use_last_partition If true then the previous partition will be partitioned again
/// @param args Aux arguments to pass to unary_op function
template<typename ...Args>
void partitionDrawCommands(
    bool(*unary_op)(Face orientation, vmath::Vec3f32 position, Args... args),
    bool use_last_partition,
    Args... args
) noexcept {
    std::size_t begin{ 0UL };
    std::size_t end{ use_last_partition ? _draw_cmds_partition_size - 1UL : _draw_cmds.size() - 1UL };
    while(true) {
        while(begin < _draw_cmds.size() &&
              unary_op(
                  _draw_cmds[begin].orientation, _chunks[_chunk_id_to_index[_draw_cmds[begin].chunk_id]].position,
                  args...
              )
        ) {
            ++begin;
        }
        while(end != std::numeric_limits<std::size_t>::max() &&
              !unary_op(
                  _draw_cmds[end].orientation, _chunks[_chunk_id_to_index[_draw_cmds[end].chunk_id]].position,
                  args...
              )
        ) {
            --end;
        }
        if (end == std::numeric_limits<std::size_t>::max() || begin >= end) {
            break;
        }
        _chunks[_chunk_id_to_index[_draw_cmds[begin].chunk_id]].draw_cmd_indices[_draw_cmds[begin].orientation] = end;
        _chunks[_chunk_id_to_index[_draw_cmds[end].chunk_id]].draw_cmd_indices[_draw_cmds[end].orientation] = begin;
        std::swap(_draw_cmds[end], _draw_cmds[begin]);
        --end;
        ++begin;
    }
    _draw_cmds_partition_size = begin;
    _draw_cmds_dirty = true;
}

```

Fragment kodu nr. 8 – funkcja dzieląca komendy według zadanego kryterium, język C++

Po takim podziale na początku bufora znajdują się komendy spełniające zadany warunek. Komendy są następnie przekazywane do bufora *DIBO*. W efekcie renderowane są tylko te siatki, które spełniły warunek, co znacznie optymalizuje czas renderowania zaalokowanych kawałków (np. w przypadku gdy zastosuje się *frustum culling*).

4.3.4 Alokacja i dealokacja kawałków

Dane zaalokowanego kawałka przechowywane są strukturze przedstawionej we fragmencie kodu numer 9.

```
/// @brief structure stores chunk metadata information
struct Chunk {
    /// @brief chunk world space position
    vmath::Vec3f32 position;
    /// @brief indices of draw commands belonging to that chunk
    vmath::u32 draw_cmd_indices[6];
    /// @brief pointer to allocated cpu region for this chunk
    std::span<vmath::u16> cpu_region;
    /// @brief unique id of this chunk
    ChunkId chunk_id{ 0U };
    /// @brief indicates if chunk is meshed and actively drawn
    bool complete;
};
```

Fragment kodu nr. 9 – struktura przechowująca dane zaalokowanego kawałka, język C++

Parametr *position* oznacza pozycje środka kawałka w przestrzeni 3D. Parametr *draw_cmd_indices* to tablica przechowująca indeksy komend renderujących kolejne orientacje siatki danego kawałka ($+X, -X, +Y, -Y, +Z, -Z$). Parametr *cpu_region* to wskaźnik na region bufora przechowującego dane kawałków wolumenu. Parametr *chunk_id* to identyfikator kawałka. Parametr *complete* oznacza, czy dany zaalokowany kawałek jest kompletny czyli, czy posiada już siatkę i zaalokowane komendy renderujące tą siatkę.

Kawałki alokowane są z tablicy wolnych, możliwych do zaalokowania kawałków. Dane wolnego kawałka przechowuje struktura pokazana we fragmencie kodu numer 10.

```
/// @brief structure stores free chunk metadata
struct FreeChunk {
    /// @brief free chunk id
    ChunkId chunk_id{ 0U };
    /// @brief free cpu region to allocate (derived from <_voxel_data>)
    std::span<vmath::u16> cpu_region;
};
```

Fragment kodu nr. 10 – struktura przechowująca dane wolnego kawałka, język C++

Parametry oznaczają to samo co te o tej samej nazwie w strukturze przedstawionej we fragmencie kodu numer 9.

Wolne kawałki przechowywane są w buforze cyklicznym [3] będącym kolejką FIFO, o stałym rozmiarze C_{max} . Przy inicjalizacji bufor cykliczny wypełniany jest w sposób pokazany we fragmencie kodu numer 11

```
_free_chunks = RingBuffer<FreeChunk>(_chunks_count, {});
for (std::size_t i{ 0U }; i < _chunks_count; ++i) {
    _free_chunks.write({
        .chunk_id = static_cast<u32>(i),
        .cpu_region = std::span<u16>(_voxel_data.data() + i * _engine_context.chunk_size_1D, _engine_context.chunk_size_1D)
    });
}
```

Fragment kodu nr. 11 – bufor cykliczny `_free_chunks` wypełniany wolnymi kawałkami, `_chunks_count` odpowiada C_{max} , `_voxel_data` odpowiada buforowi przechowującemu dane kawałków wolumenu, język C++

Przy alokacji kawałka, wolny kawałek odczytywany jest z bufora wolnych kawałków i tworzony jest na jego podstawie zaalokowany kawałek z fragmentu kodu numer 9. Gdy nie ma wolnych kawałków, zwracany jest identyfikator o wartości $2^{32} - 1$. Przy dealokacji kawałka podaje się identyfikator kawałka do dealokacji, zwolniony kawałek oddawany jest do bufora wolnych kawałków.

Zaalokowany kawałek ma 4 reprezentacje w pamięci.

- metadane kawałka (klient) - struktura pokazana we fragmencie kodu numer 9
- region danych kawałka (klient) - pamięć wskazywana przez parametr `cpu_region` w metadanych
- region siatki kawałka (serwer) - region pamięci *VBO* który jest obliczany na podstawie identyfikatora kawałka
- komendy renderujące siatkę kawałka (klient i serwer) - jedna na każdą orientację

Z racji, że przy inicjalizacji zdefiniowana zostaje maksymalna możliwa liczba kawałków zaalokowana w jednym momencie C_{max} , cała pamięć alokowana jest raz przy inicjalizacji (opcjonalnie wyjątkiem jest pamięć *VBO*, co zostanie dokładniej opisane w rozdziale 4.3.5).

Pamięć kawałków zarządzana jest przy użyciu wzorca puli obiektów [20]. Oznacza to, że przechowuje się informacje o tym, czy kawałek jest zaalokowany (w użyciu), czy nie. Stany kawałków przechowywane są w tablicy, która tłumaczy identyfikatory kawałków na indeksy w tablicy zaalokowanych kawałków. W przypadku gdy kawałek

o zadanym identyfikatorze nie jest zaalokowany, indeks wynosi $2^{32} - 1$. Przechowywanie takiej tablicy istotne jest, gdy kawałki są dealokowane. W przypadku usunięcia kawałka ze środka tablicy zaalokowanych kawałków, na jego miejsce wpisywany jest kawałek z końca tablicy. Wtedy indeks kawałka dealokowanego w tablicy tłumaczącej identyfikatory na indeksy jest ustawiany na $2^{32} - 1$, z kolei indeks kawałka przesuniętego z końca na miejsce kawałka dealokowanego ustawiany na indeks dealokowanego kawałka przed jego invalidacją. Podobnie wygląda to w przypadku komend renderujących, które są dealokowane wraz z dealokowanym kawałkiem w przypadku, gdy dany kawałek jest kompletny (parametr *complete* w strukturze z fragmentu kodu numer 9 ustawiony jest na wartość *true*). Po kolei dealokowane komendy są usuwane i na ich miejsce wstawiane są komendy z końca tablicy. Stąd istotne jest przechowywanie identyfikatora kawałka oraz orientacji siatki w komendzie renderującej jak pokazano we fragmencie kodu numer 6. Dzięki temu, gdy komenda z końca bufora komend przesuwana jest na miejsce dealokowanej komendy, jej indeks w strukturze kawałka do którego należy może być zaktualizowany (parametr *draw_cmd_indices* w strukturze z fragmentu kodu numer 9 indeksowany jest orientacją siatki).

Gdy kawałek jest alokowany, na kolejkę zadań siatkowania dodawane jest zadanie siatkowania tego kawałka (mechanizm działania jest dokładniej opisany w rozdziale 4.4). Następnie *ChunkPool* jest cyklicznie odpytywany (raz na klatkę), czy siatkowanie któregoś z kawałków zostało ukończone. Jeśli tak, kawałek którego siatka jest gotowa zostaje oznaczony jako kompletny (parametr *complete* w strukturze z fragmentu kodu numer 9 ustawiany jest na wartość *true*) i zostają zaalokowane komendy renderujące wygenerowane siatki. Kawałek może zostać zdealokowany zanim jego siatka zostanie wygenerowana, w takim wypadku wygenerowana siatka zostaje odrzucona.

4.3.5 Dynamiczne poszerzanie bufora *VBO*

Bufor *VBO* poszerza się, gdy wynik zadania siatkującego zwróci informację o tym, że siatka danego kawałka nie zmieściła się w podregionie. Gdy siatka nie mieści się, algorytm chciwego siatkowania wykonuje się do końca, generując informacje o tym, jak duża jest siatka która się nie mieści ale nie wpisuje kompletnych danych samej siatki do bufora. Z informacji zwrotnej o wielkości siatek dla każdej orientacji, wybierana jest maksymalna wartość. Następnie wartość ta, mnożona jest przez zmienną konfiguracyjną silnika *chunk_pool_growth_coefficient* opisaną rozdziale 4.8 i na tej podstawie ustalana jest nowa wielkość bufora *VBO*.

Po realokacji bufora VBO , wszystkie kompletne kawałki odznaczane są jako niekompletne (łącznie z kawałkiem którego siatka spowodowała przepełnienie bufora) i dealokowane są ich komendy renderujące. Następnie ponownie dodawane są dla każdego z nich zadania siatkujące.

4.4 *MeshingEngine* - szczegóły implementacji

MeshingEngine odpowiada za kolejkowanie i uruchamianie zadań siatkowania kawałków na serwerze. Zadania kolejkowane są na buforze cyklicznym [3] o stałym rozmiarze C_{max} . *MeshingEngine* musi być cyklicznie odpytywany (np. raz na klatkę), czy dana siatka jest gotowa. Odpytywanie w tym wypadku przekłada się na odpytywanie *ChunkPool*, wspomniane w rozdziale 4.3.4.

4.4.1 Synchronizacja zadań siatkowania

Struktura przechowująca dane zadania, które przechowywane są na buforze cyklicznym, pokazana jest we fragmencie kodu numer 12.

```
/// @brief command describing meshing execution of a single chunk
struct Command {
    /// @brief id of chunk meshed by command
    ChunkId chunk_id;
    /// @brief position of the chunk
    vmath::Vec3f32 chunk_position;
    /// @brief pointer to voxel data to be issued before meshing starts
    std::span<const vmath::u16> voxel_data;
    /// @brief gl fence for which to wait in case the command is active one
    /// also indicates !!!IF COMMAND IS INITIALIZED (nullptr here if not)!!!
    void* fence{ nullptr };
    /// @brief axis progress keeps track of how many planes on each axes were meshed
    vmath::i32 axis_progress;
};
```

Fragment kodu nr. 12 – struktura zadania siatkowania kawałka, język C++

Parametr *chunk_id* jest identyfikatorem kawałka do którego odnosi się dane zadanie. Parametr *chunk_position* to pozycja kawałka w przestrzeni 3D. Parametr *voxel_data* to dane kawałka z których generowana będzie siatka. Parametr *fence* to zmienna typu *GLsync*, jeśli nie ma wartości *nullptr*. Parametr *axis_progress* oznacza postęp algorytmu chciwego siatkowania dla danego zadania.

Gdy *ChunkPool* zleca zadanie siatkowania danego kawałka wolumenu w momencie gdy bufor cykliczny jest pusty, zadanie wykonywane jest od razu. Gdy inne zadanie jest w trakcie wykonania, zadanie wpisywane jest do bufora cyklicznego.

Wykonywane zadanie odczytywane jest z bufora cyklicznego i wpisywane jest jako aktywne zadanie. Aktywne zadanie ma wartość parametru *fence* inną niż *nullptr*. Wynika to z tego, że po wywołaniu algorytmu chciwego siatkowania w sposób pokazany we fragmencie kodu numer 13, tworzony jest obiekt *GLsync*, który jest przypisywany do zmiennej *fence*, jak pokazano we fragmencie kodu numer 14.

```
glDispatchCompute(6, 1, 1);
```

Fragment kodu nr. 13 – wywołanie algorytmu chciwego siatkowania po stronie klienta, język C++

```
command.fence = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
```

Fragment kodu nr. 14 – stworzenie obiektu GLsync po wywołaniu algorytmu chciwego siatkowania, język C++

Stworzenie obiektu *GLsync* tuż po wywołaniu algorytmu chciwego siatkowania (fragment kodu numer 13), oznacza że obiekt nie zostanie zasygnalizowany dopóki wywołanie algorytmu przed jego stworzeniem nie zostanie zakończone. Umożliwia to synchronizacje pomiędzy *MeshingEngine* oraz *ChunkPool*. Wspomniane wcześniej odpytywanie opiera się na sprawdzeniu stanu obiektu *GLsync* wykonywanej komendy. Jeśli obiekt jest w stanie zasygnalizowanym, oznacza to że siatka została wygenerowana.

Gdy zadanie się wykona i w buforze cyklicznym są inne zadania, następne zadanie zostaje wybrane jako aktywne i jest uruchamiane.

4.4.2 Algorytm chciwego siatkowania w GLSL

Algorytm chciwego siatkowania na serwerze działa na tej samej zasadzie jak przedstawiono to w rozdziale 2.1.2.

Algorytm zaimplementowano w języku GLSL jako *compute shader* [18] i jest on uruchamiany po stronie serwera. Moduł cieniujący typu *compute* uruchamiany jest z 6 grupami o wielkości domyślnej 64 wątków. Wielkość grupy można łatwo zmienić za

pomocą zmiennej *local_size_x* w kodzie modułu cieniącego i zmiennej konfiguracyjnej silnika *meshing_shader_local_group_size* opisanej w rozdziale 4.8. Każda grupa siatkuje inną stronę, orientację kawałka. Każdy wątek w grupie siatkuje inny "plaster" siatki wzdłuż osi orientacji.

Każda grupa wpisuje dane siatek do innych podregionów, więc nie wymaga się pomiędzy nimi synchronizacji. Z kolei w przypadku wątków z danej grupy, synchronizacja jest wymagana, bo wątki piszą do tego samego podregionu. Synchronizacja osiągnięta jest za pomocą atomowo inkrementowanego licznika, dzielonego w zakresie danej grupy. Za każdym razem gdy dany wątek ma dodać nowy element siatki, najpierw inkrementuje licznik. Inkrementacja zwraca wartość licznika przed inkrementacją, na podstawie której wyliczane jest miejsce podregionu do którego wpisać siatkę. Deklarację licznika w module cieniąjącym pokazano we fragmencie kodu numer 15

```
shared uint local_mesh_quads_count;
```

Fragment kodu nr. 15 – atomowy licznik kawałków siatek w danej grupie, język GLSL

Do poprawnego działania algorytmu muszą zostać zdefiniowane stałe pokazane we fragmencie kodu numer 16.

```
layout(std140, binding = 2) uniform MeshingDescriptor {
    uint vbo_offsets[6];
    uint max_submesh_size_in_quads;
    vec3 chunk_position;
    ivec3 chunk_size;
};
```

Fragment kodu nr. 16 – deskryptor algorytmu siatkowania jako uniform buffer w kodzie algorytmu chciwego siatkowania, język GLSL

Parametr *vbo_offsets*, oznacza początki podregionów *VBO* dla każdej grupy. Parametr *max_submesh_size_in_quads* oznacza maksymalną możliwą liczbę kawałków siatek w podregionie. Parametr *chunk_position*, oznacza to samo co parametr strukturze z fragmentu kodu numer 12 o tej samej nazwie. Parametr *chunk_size* oznacza wymiary *X, Y, Z* kawałka. Struktura z fragmentu kodu numer 16 przekłada się na strukturę z fragmentu kodu numer 17 po stronie klienta.

```
/// @brief Descriptor of meshing, it maps to the ubo of binding id 2 in
/// greedy meshing shader
struct Descriptor {
    /// @brief offsets of submeshes (constant)
    alignas(16) vmath::u32 vbo_offsets[6][4];
    /// @brief max submesh size in quads
    alignas(16) vmath::u32 max_submesh_size_in_quads;
    /// @brief position changes per meshing command (mutable)
    alignas(16) vmath::Vec3f32 chunk_position;
    /// @brief size of a~chunk (constant)
    alignas(16) vmath::Vec3i32 chunk_size;
};
```

Fragment kodu nr. 17 – deskryptor algorytmu siatkowania jako struktura w MeshingEngine, język C++

Dane struktury z fragmentu kodu numer 17 alokowane są jako *Uniform Buffer Object (UBO)* [18] po stronie serwera.

Dane kawałka z których generowana jest siatka w danym wykonaniu, zdefiniowana jest jako *Shader Storage Buffer Object (SSBO)* do którego przekazywane są dane siatkowanego kawałka przed wywołaniem algorytmu. Bufor jest zmapowany podobnie jak bufor *DIBO*, co jest opisane w rozdziale 4.3.3. Ma on swoją reprezentację w algorytmie jak pokazano we fragmencie kodu numer 18.

```
layout(std430, binding = 5) readonly buffer VoxelData {
    uint voxel_data[];
};
```

Fragment kodu nr. 18 – bufor z danymi kawałka siatkowanego jako bufor w kodzie algorytmu chciwego siatkowania, język GLSL

We fragmencie kodu numer 18 widać, że dane są typu *uint*, którego wielkość to 32 bity. Z tego względu dostęp do danych po indeksie musi odbywać się z ekstrakcją bitów (woksel zdefiniowany jest jako liczba 16 bitowa).

Dane siatki to region *VBO* powiązany z buforem *SSBO*. Jego reprezentację w algorytmie pokazano we fragmencie kodu numer 19

```
layout(std430, binding = 7) writeonly buffer MeshData {
    float vbo[];
};
```

Fragment kodu nr. 19 – zmapowany region VBO do bufora SSBO w module cieniującym, język GLSL

Jest on tablicą typów *float*, ponieważ taka reprezentacja zapewnia największą kompakcję danych. Wynika to z wymagań jakie nakłada *std430* [19].

Bardzo istotny jest bufor typu *SSBO* przechowujący dane tymczasowe. Stanowi on interfejs pomiędzy klientem, a serwerem. Jego reprezentację w module cieniującym pokazano we fragmencie kodu numer 20.

```
layout(std430, binding = 6) buffer MeshingTemp {
    uint written_quads[6]; // = { +x, -x, +y, -y, +z, -z }
    uint axes_steps[6];
    uint overflow_flag;
};
```

Fragment kodu nr. 20 – struktura tymczasowych danych jako SSBO w module cieniującym, język GLSL

Do elementu tablicy *written_quads*, dana grupa zapisuje liczbę wpisanych kawałków siatki. Na jej podstawie wiadomo jak duża jest wpisana siatka. Parametr *axes_steps* przechowuje postęp algorytmu w każdej orientacji. Po wykonaniu się algorytmu jest inkrementowana o 64. Parametr *overflow_flag* jest ustawiany w algorytmie na 1, gdy któraś z siatek nie mieści się w podregionie. Po wykonaniu się algorytmu dla danego kawałka i odczytaniu przez klienta wartości flagi równej 1, bufor *VBO* zostaje poszerzony jak opisano w rozdziale 4.3.5.

4.4.3 Rezultat zadania siatkującego

Jako rezultat algorytmu siatkowania przeprowadzonego na zadany kawałku, do *ChunkPool* przekazywany jest obiekt struktury pokazanej we fragmencie kodu numer 21.

```
/// @brief is an interface and holds completed command data
struct Result {
    /// @brief identifier of a ~successfully meshed chunk (maps to
    /// chunk id in ChunkPool)
    ChunkId chunk_id;
    /// @brief written indices count *per face* indexed with ve001::Face enum.
    /// determines how many indices to render
    std::array<vmath::u32, 6> written_indices;
    /// @brief if true then number of potentially written vertices is
    /// bigger than current chunk region size and pool needs to be extended
    bool overflow_flag{ false };
};
```

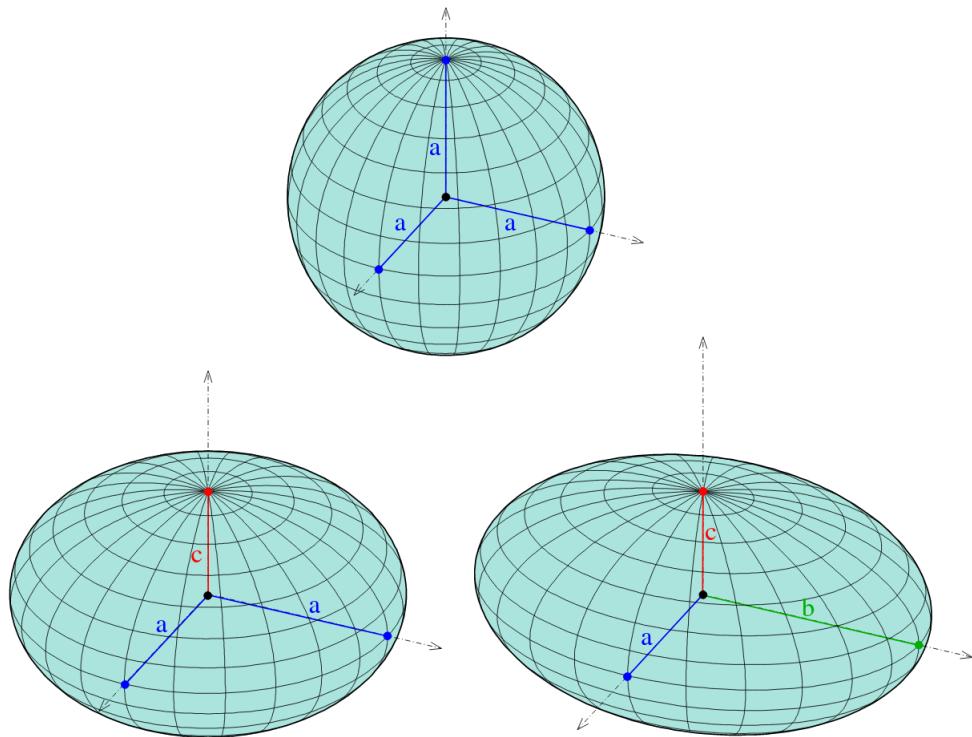
Fragment kodu nr. 21 – struktura wyniku zadania siatkującego, język C++

Zmienna *chunk_id* oznacza to samo, co parametr o tej samej nazwie we fragmencie kodu numer 12. Parametr *written_indices* oznacza liczbę wpisanych indeksów siatki dla każdej z jej orientacji. Parametr *overflow_flag* oznacza to samo co parametr o tej samej nazwie we fragmencie kodu numer 20.

Na podstawie tych danych *ChunkPool* może zaalokować komendy renderujące siatki dla kawałka którego siatka została wygenerowana lub zwiększyć wielkość *VBO* (jeśli parametr *overflow_flag* wynosi *true*).

4.5 *WorldGrid* - szczegóły implementacji

WorldGrid zarządza widocznością kawałków wolumenu. Zakres widoczności kawałków definiowany jest przez półosie elipsoidy i punkt środkowy elipsoidy. Gdy półosie elipsoidy są takie same, definiuje on sferę, z kolei gdy dwie z półosi są takie same, definiuje on sferoidę. Pokazano to na rysunku 21.



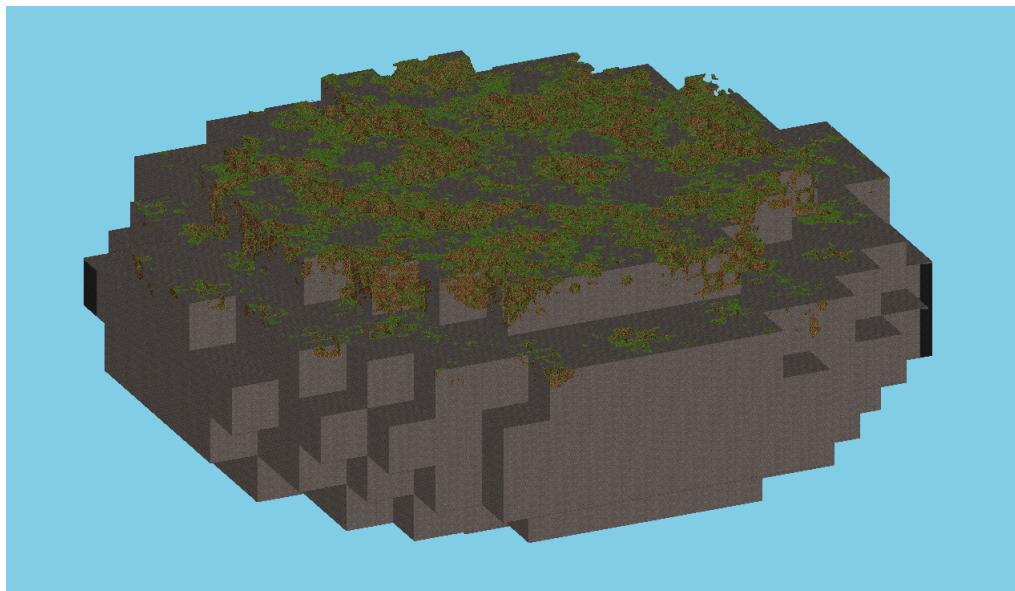
Rys. 21 – Na górze rysunku - sfera, po lewej na dole - sferoida, po prawej na dole - elipsoida [5]

Elipsoida definiowana jest przez równanie nr. 1

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

Równanie nr. 1 – zmienne x , y i z - koordynaty środka elipsoidy, zmienne a , b i c - wielkości półosi elipsoidy

Środkiem elipsoidy jest pozycja kamery. Wszystkie kawałki, które mieszczą się w obszarze tak zdefiniowanej elipsoidy traktowane są jako widoczne. Z kolei kawałki z poza jako niewidoczne. Gdy środek elipsoidy wraz z pozycją kamery zmienia się o zadany wektor, nowe kawałki które po przesunięciu wchodzą w obszar elipsoidy są alokowane, z kolei zaalokowane kawałki które po przesunięciu przestają wchodzić do obszaru elipsoidy, dealokowane. Na rysunku 22 pokazano jak w praktyce wygląda obszar elipsoidy wypełniony kawałkami volumenu.



Rys. 22 – kawałki o wymiarach $64x64x64$ wchodzące w elipsoidę o półosiach $a = 500$, $b = 200$, $c = 500$

4.5.1 Struktura danych

Aby możliwie jak najoptymalniej zarządzać alokacją i dealokacją kawałków stworzono strukturę danych, która pozwala na przeprowadzanie testów, czy niezaalokowany kawałek jest, lub zaalokowany nie jest wewnątrz elipsoidy jedynie dla tych kawałków,

które są na brzegach elipsoidy. Gdzie brzegiem elipsoidy, nazywa się kawałki, które sąsiadują z którykolwiek strony z wolną przestrzenią (liczba ich sąsiadów jest mniejsza niż 6).

Struktura danych składa się z tablicy widocznych kawałków, gdzie widoczny kawałek przechowuje liczbę obecnych sąsiadów oraz indeksy do tych sąsiadów w tej samej tablicy. Tablica zarządzana jest podobnie jak tablica kawałków opisana w rozdziale 4.3.4, za pomocą wzorca puli obiektów [20]. Strukturę widocznego kawałka pokazano we fragmencie kodu numer 22.

```
/// @brief id of visible chunk
using VisibleChunkId = vmath::u32;
/// @brief visible chunk metadata structure
struct VisibleChunk {
    static constexpr vmath::u32 INVALID_NEIGHBOUR_INDEX{
        std::numeric_limits<vmath::u32>::max()
    };
    /// @brief id of a~chunk in the context of world grid
    VisibleChunkId visible_chunk_id;
    /// @brief id of a~chunk in the context of chunk pool
    ChunkId chunk_id;
    /// @brief position of visible chunks in chunk size units
    vmath::Vec3i32 position_in_chunks;
    /// @brief number of visible chunks's neighbours
    vmath::u32 neighbours_count{ 0U };
    /// @brief contains indices of neighbours in <-visible_chunks> array
    std::array<vmath::u32, 6> neighbours_indices{
        INVALID_NEIGHBOUR_INDEX,
        INVALID_NEIGHBOUR_INDEX,
        INVALID_NEIGHBOUR_INDEX,
        INVALID_NEIGHBOUR_INDEX,
        INVALID_NEIGHBOUR_INDEX,
        INVALID_NEIGHBOUR_INDEX
    };
};
```

Fragment kodu nr. 22 – struktura widocznego kawałka, język C++

Parametr *position_in_chunks* oznacza pozycje widocznego kawałka w jednostkach wielkości pojedynczego kawałka, z tego względu jest reprezentowana za pomocą liczb całkowitych. Parametr *neighbours_count* oznacza liczbę sąsiadów, gdy kawałek ma wszystkich sąsiadów jest ona równa 6. Parametr *neighbours_indices* przechowuje indeksy w tablicy widocznych kawałków do sąsiadów tego kawałka (stanowi wskaźnik na sąsiada widocznego kawałka), dla każdej strony, gdzie strony po kolej to $+X$, $-X$, $+Y$, $-Y$, $+Z$, $-Z$. Gdy wartość indeksu któregokolwiek z sąsiadów to $2^{32} - 1$, znaczy że sąsiad nie istnieje z danej strony widocznego kawałka. Parametr *visible_chunk_id* jest identyfikatorem widocznego kawałka i działa analogicznie jak identyfikator kawałka opisany w rozdziale 4.3.4 (stanowi indeks do tablicy tłumaczącej identyfikator widocznego kawałka na indeks w tablicy widocznych kawałków). Parametr *chunk_id*

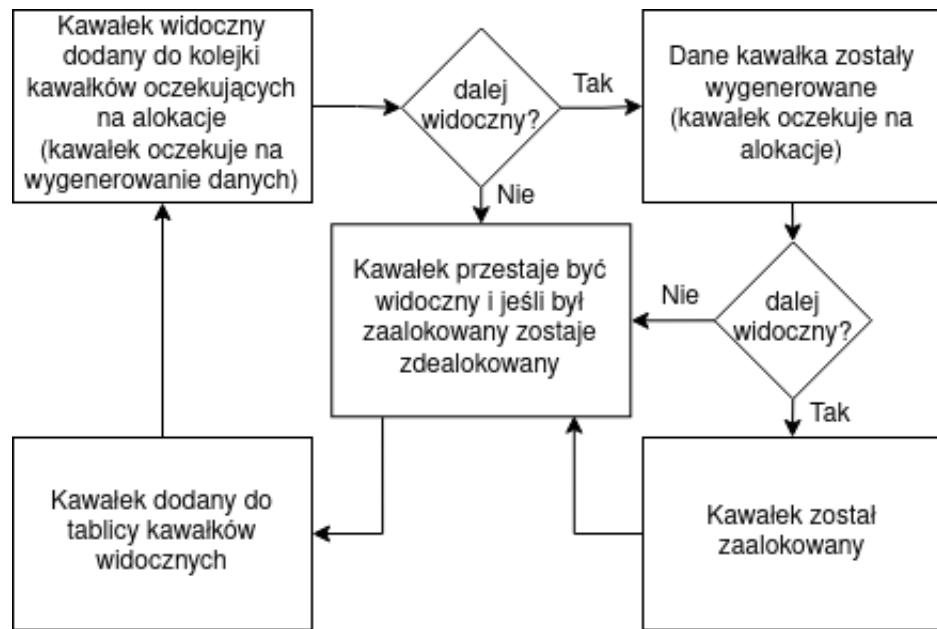
to identyfikator zaalokowanego kawałka w *ChunkPool* powiązanego z widocznym kawałkiem.

Istnieje podział na widoczne kawałki i zaalokowane kawałki (dwa różne identyfikatory) z trzech powodów. Pierwszym powodem jest to, że mogą istnieć kawałki puste, a takie nie wymagają alokacji pamięci i istnieją tylko wirtualnie w tablicy widocznych kawałków aby podtrzymać spójność opisywanej struktury danych. Drugim powodem jest to, że nie zawsze uda się zaalokować kawałek i musi on poczekać aż jakiś kawałek zostanie zdealokowany (pula pamięci *ChunkPool* na daną chwilę jest wyczerpana). Trzeci powód to potrzeba oczekania na wynik generacji danych kawałka przez *ChunkDataStreamer* opisany w rozdziale 4.6. Widoczne kawałki oczekujące na alokacje w *ChunkPool* lub oczekujące na wygenerowanie ich danych przez *ChunkDataStreamer* przechowywane są na buforze cyklicznym będącym kolejką *FIFO* o stałym rozmiarze C_{max} . Strukturę widocznego kawałka oczekującego na zaalokowanie lub wygenerowanie pokazano we fragmencie kodu numer 23.

```
/// @brief handle to chunk which is to be generated and than allocated
struct ToAllocateChunk {
    /// @brief handle to data which will be generated in the future by
    /// chunk data streamer
    std::future<std::optional<std::span<const vmath::u16>>> data;
    /// @brief handle to visible chunk
    VisibleChunkId visible_chunk_id;
    /// @brief handle to generated data
    std::optional<std::span<const vmath::u16>> ready_data{ std::nullopt };
};
```

*Fragment kodu nr. 23 – struktura widocznego kawałka oczekującego na alokacje w *ChunkPool* lub wygenerowanie danych przez *ChunkDataStreamer*, język C++*

Parametr *data* przechowuje uchwyt (*handle*) w postaci *std::future* [14] do wyniku generacji danych, zwraca ją *ChunkDataStreamer* opisany w rozdziale 4.6. W nim zostaje ustawiony wynik gdy dane są gotowe. Jak widać wynik może być opcjonalny, dzieje się tak gdy wygenerowane dane kawałka są puste (wszystkie woksele mają wartość 0) lub wystąpił błąd generacji. Gdy wynik generacji jest gotowy, wynik ze zmiennej *data* przypisywany jest do zmiennej *ready_data* i wygenerowany już kawałek oczekuje na alokacje w *ChunkPool*. Jeśli kawałek przestaje być widoczny zanim zostanie wygenerowany i zaalokowany, wygenerowane dane zostają odrzucone i kawałek nie zostaje zaalokowany. Z tąt cykl życia widocznego kawałka można opisać maszyną stanów z rysunku 23.



Rys. 23 – cykl życia widocznego kawałka

4.5.2 Inicjalizacja struktury danych

Na początku oblicza się maksymalną możliwą liczbę kawałków w elipsoidzie z równania nr. 2

$$C_{max} = \left\lfloor \frac{2a}{X} \right\rfloor \cdot \left\lfloor \frac{2b}{Y} \right\rfloor \cdot \left\lfloor \frac{2c}{Z} \right\rfloor$$

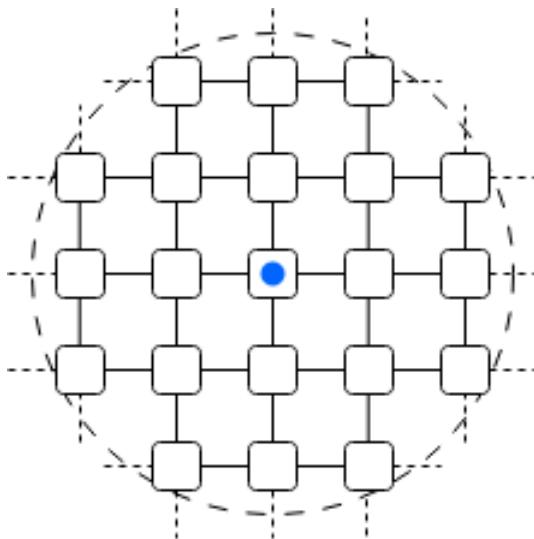
Równanie nr. 2 – zmienne X, Y, Z - wymiary pojedyńczego kawałka, zmienne a, b, c - półosie elipsoidy

Na podstawie wyniku z równania nr. 2 alokuje się odpowiednią ilość pamięci zarówno w *WorldGrid* jak również w *ChunkPool*. Należy zaznaczyć, że równanie nr. 2 w praktyce powoduje alokacje większej ilości pamięci niż potrzeba. Następnie iteruje się po dyskretnych punktach prostopadłościanu o wymiarach wyliczonych tak jak pokazano to w równaniu nr. 3.

$$\vec{G} = 2 \left[\frac{a}{X} \frac{b}{Y} \frac{c}{Z} \right] + 1$$

Równanie nr. 3 – wyliczenie wymiarów G prostopadłościanu w który można wpisać elipsoidę

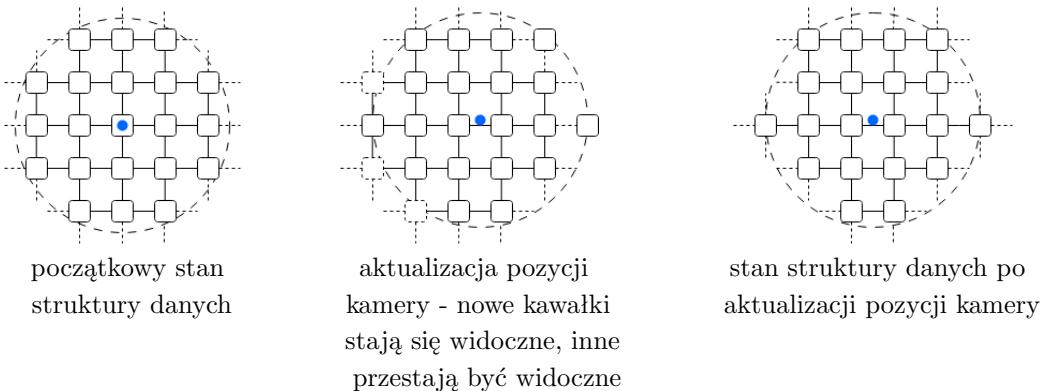
Najpierw iteruje się po punktach takiego prostopadłościanu względem pozycji początkowej kamery i dodaje się punkty, które wchodzą w elipsoidę. Potem dla każdego dodanego punktu generuje się dane używając *ChunkDataStreamer*, a następnie uzupełnia się jego sąsiadów. Po takiej inicjalizacji strukturę można zobrazować w sposób pokazany na rysunku 24.



Rys. 24 – struktura danych WorldGrid, reprezentacja w 2D, niebieska kropka - pozycja kamery, białe kwadraty - widoczne kawałki, czarne linie - wskaźniki na sąsiadów (w dwie strony), przerywane czarne linie - brak sąsiadów, przerywany okrąg - wielkość elipsoidy

4.5.3 Działanie struktury danych

Struktura danych jest aktualizowana poprzez podanie nowej pozycji kamery. Jak przebiega taka aktualizacja pokazano wizualnie na rysunku 25.



Rys. 25 – przebieg aktualizacji struktury danych WorldGrid przed, po i w trakcie aktualizacji pozycji kamery zobrazowany w 2D, niebieska kropka - pozycja kamery, białe kwadraty - widoczne kawałki, białe kwadraty z przerywaną linią - dealokowane widoczne kawałki, czarne linie - wskaźniki na sąsiadów (w dwie strony), przerywane czarne linie - brak sąsiadów, przerywany okrąg - wielkość elipsoidy

Gdy struktura danych się aktualizuje, sprawdzane są widoczne kawałki o liczbie sąsiadów mniejszej niż 6. Najpierw iteruje się przez wszystkich nieobecnych sąsiadów tego kawałka i sprawdza się, czy jest on widoczny. Jeśli jest widoczny, odznacza się to w tablicy tymczasowej wielkości \vec{G} indeksem tego sąsiada w tablicy widocznych kawałków (której celem jest możliwość szybkiego sprawdzenia czy dany sąsiad został już dodany w tym wywołaniu ponieważ kawałki mogą dzielić ze sobą tych samych sąsiadów), dodaje się kawałek do tablicy widocznych kawałków oraz dodaje się kawałek do tablicy kawałków widocznych oczekujących na alokacje, jednocześnie wywołując generacje danych dla kawałka przez *ChunkDataStream*. Gdy przeiteruje się przez wszystkich niewidocznych sąsiadów, sprawdza się czy przetwarzany kawałek jest dalej widoczny, jeśli nie, dealokuje się go. Następnie sprawdzane jest czy któryś z kawałków został wygenerowany, zaalokowany i można zdjąć go z kolejki FIFO kawałków oczekujących na alokacje.

Po iteracji przez wszystkie widoczne kawałki oczekuje się na wszystkie pozostałe kawałki oczekujące na wygenerowanie i zaalokowanie. Na koniec tablica tymczasowa przechowująca indeksy widocznych kawałków jest inwalidowana wartością $2^{32} - 1$.

4.5.4 Ograniczenia struktury danych

Aktualizacja opisywanej struktury danych jest niepełna, gdy zamiana pozycji kamery jest za duża. Gdzie za duża zmiana, występuje wtedy gdy w danej iteracji

widoczny zostaje kawałek, który nie ma żadnych sąsiadów lub niewidoczny zostaje kawałek, który ma wszystkich sąsiadów.

4.6 *ChunkDataStreamer* - szczegóły implementacji

ChunkDataStreamer implementuje pule wątków równą wynikowi pokazanemu we fragmencie kodu numer 24.

```
auto final_threads_count = threads_count;
if (final_threads_count == 0U) {
    final_threads_count = std::thread::hardware_concurrency() == 0 ? 1 : std::thread::hardware_concurrency();
}
for (std::uint32_t i{ 0U }; i < final_threads_count; ++i) {
    _threads.push_back(std::jthread(&ChunkDataStreamer::thread, this));
}
```

Fragment kodu nr. 24 – określanie liczby wątków w puli *ChunkDataStreamer*, język C++

We fragmencie kodu numer 24 parametr *threads_count* jest parametrem konfigurowalnym silnika. Opisano go w rozdziale 4.8.

Każdy wątek wchodzi do funkcji i wykonujeinicjalizacje aktualnie ustawionego generatora typu *ChunkGenerator*, który jest opisany w rozdziale 4.7. Następnie w pętli próbuje odczytać z kolejki FIFO, będącej buforem cyklicznym (jest on bezpieczny wątkowo (*thread safe*)), o stałym rozmiarze strukturę opisującą zadanie generacji danych, aż do zakończenia działania programu. Strukturę pokazano we fragmencie kodu numer 25.

```
struct Promise {
    std::promise<std::optional<std::span<const vmath::u16>>> value;
    vmath::Vec3i32 position;
};
```

Fragment kodu nr. 25 – struktura opisująca zadanie generacji danych, język C++

Gdzie parametr *value* typu *std::promise* [15], to wartość z której pobrano *std::future* pokazany we fragmencie kodu numer 23. W parametrze *value*, gdy wątek wygeneruje dane, ustawia się wynik generacji (wygenerowane dane kawałka), co jest widoczne w pobranej z nań zmiennej typu *std::future*. Parametr *position* oznacza logiczną pozycję generowanego kawałka w przestrzeni (po pomnożeniu przez wektor $[X, Y, Z]$ i odjęciu wektora $[\frac{X}{2}, \frac{Y}{2}, \frac{Z}{2}]$ daje rzeczywistą pozycję środka kawałka w przestrzeni).

4.7 *ChunkGenerator* - szczegóły implementacji

ChunkGenerator to prosta klasa abstrakcyjna implementująca interfejs generatora danych kawałków. Przedstawiono ją we fragmencie kodu numer 26

```
struct ChunkGenerator {
    /// @brief initializes data of the generator thread-wise
    /// @return true if initialization failed
    virtual bool threadInit() noexcept = 0;
    /// @brief generates data at <chunk_position>
    /// @return generated data, if fails just return std::nullopt
    virtual std::optional<std::span<const vmath::u16>> gen(vmath::Vec3i32 chunk_position) noexcept = 0;
};
```

Fragment kodu nr. 26 – klasa abstrakcyjna implementująca interfejs generatora danych, język C++

Metoda *threadInit* jest uruchamiana przez każdy wątek i ma za zadanie inicjalizować generator w kontekście pojedynczego wątku, zwraca wartość *true* jeśli wystąpił błąd inicjalizacji. Metoda *gen* to funkcja generująca kawałek o danych współrzędnych.

4.7.1 przykład z *NoiseTerrainGenerator*

Jako przykład napisano generator implementujący generacje kawałków z szumu, dziedziczący z klasy *ChunkGenerator*. Do generacji szumu użyto zewnętrznej biblioteki *FastNoise2* [6]. Zasoby potrzebne do generacji zadeklarowane są jako lokalne dla wątku, jak pokazano we fragmencie kodu numer 27.

```
static constexpr std::size_t BUFFERS_COUNT{ 2UL };

(...)

thread_local std::vector<vmath::f32> NoiseTerrainGenerator::_tmp_noise;
thread_local std::array<std::vector<vmath::u16>, NoiseTerrainGenerator::BUFFERS_COUNT> NoiseTerrainGenerator::_noise_buffers;
thread_local vmath::u32 NoiseTerrainGenerator::_current_buffer;
thread_local FastNoise::SmartNode<> NoiseTerrainGenerator::_smart_node;
```

Fragment kodu nr. 27 – zasoby do generacji szumu, język C++

W metodzie *threadInit* alokowana jest pamięć. Jak pokazano we fragmencie kodu numer 28.

```

_current_buffer = 0U;
try {
    _tmp_noise.resize(_config.terrain_size[0] * _config.terrain_size[1] * _config.terrain_size[2], 0.F);
    for (auto& buffer : _noise_buffers) {
        buffer.resize(_config.terrain_size[0] * _config.terrain_size[1] * _config.terrain_size[2], 0U);
    }
    _smart_node = FastNoise::NewFromEncodedNodeTree(
        "IQAZABAAExQoQA0AAwAAAAAAEAIAAAAAA/AAAAAAAABAwCPwnU9AQQA"
        "AAAAANejCMEAAAAAAAAAAAAAAAD//wEAAAClcjz4="
    );
} catch(const std::exception&) {
    return true;
}
return false

```

*Fragment kodu nr. 28 – inicjalizacja zasobów do generacji w metodzie threadInit,
język C++*

W metodzie *gen* szum generowany jest jak pokazano we fragmencie kodu numer 29.

```

std::optional<std::span<const vmath::u16>> NoiseTerrainGenerator::gen(vmath::Vec3i32 chunk_position) noexcept {
    const auto p0 = Vec3i32::mul(chunk_position, _config.terrain_size);
    const auto p1 = _config.terrain_size;

    _smart_node->GenUniformGrid3D(
        _tmp_noise.data(),
        p0[0], p0[1], p0[2], p1[0], p1[1], p1[2],
        _config.noise_frequency, _config.seed
    );

    auto& buffer = _noise_buffers[_current_buffer];

    std::size_t i{ 0UL };
    u32 not_empty{ 0UL };
    for (const auto noise_value : _tmp_noise) {
        auto fvalue = ((noise_value + 1.F)/2.F);

        u16 value{ 0U };
        if (fvalue >= _config.visibility_threshold) {
            const auto stretched_upper_fvalue = ((1.F - fvalue) + _config.visibility_threshold);
            value = std::clamp(static_cast<u32>(
                stretched_upper_fvalue * static_cast<f32>(_config.quantize_values)), 0U, _config.quantize_values
            );
        }

        not_empty += static_cast<u32>(value > 0);
        buffer[i++] = value;
    }
    if (not_empty) {
        _current_buffer = (_current_buffer + 1) % _noise_buffers.size();
    }
}

return not_empty ? std::optional(std::span<const u16>(buffer)) : std::nullopt;
}

```

Fragment kodu nr. 29 – generacja kawałka w metodzie gen, język C++

Jak widać we fragmencie kodu numer 29, szum należy przekonwertować do wartości dyskretnych ponieważ biblioteka generuje wartości w formacie zmienoprzecinkowym. Nakładany jest również próg widoczności reprezentowany przez parametr

_visibility_threshold. Dodatkowo należy zwrócić uwagę na użycie wzorca *double buffer* [20] do przechowywania danych wygenerowanych przez wątek. Wynika to z tego, że zwracany wynik to wskaźnik na bufor z danymi. Wynik może zostać nadpisany, gdy wątek przejmie inne zadanie, co w efekcie może nadpisać dane kawałka, który nie został jeszcze zaalokowany. Gdy wątek wygeneruje dane do jednego bufora, następne dane będą generowane do następnego itd. Zmniejsza to ryzyko wyścigu ale nie daje pewności, że dane na pewno nie zostaną nadpisane. W praktyce jest to wystarczające.

4.8 *Engine* - szczegóły implementacji

Engine agreguje wszystkie funkcjonalności i stanowi interfejs do biblioteki dla aplikacji z niej korzystającej.

Do konstruktora klasy *Engine* przekazuje się strukturę konfiguracyjną silnika pokazaną we fragmencie kodu numer 30.

```
/// @brief configuration structuer
struct Config {
    /// @brief world size as semi axes of an ellipsoid
    vmath::Vec3f32 world_size;
    /// @brief initial camera position
    vmath::Vec3f32 initial_position;
    /// @brief resolution in voxels of the single chunk
    vmath::Vec3i32 chunk_size;
    /// @brief data generator called to generate data
    std::unique_ptr<ChunkGenerator> chunk_data_generator;
    /// @brief configures number of threads used by ChunkDataStreamer
    /// if 0 then the number of threads will that of the number of CPU threads
    vmath::u32 chunk_data_streamer_threads_count;
    /// @brief growth coefficient of the pool. Value of overflow is multiplied
    /// by this coefficient deciding new max size of the chunk pool
    vmath::f32 chunk_pool_growth_coefficient;
    /// @brief meshing shader step/local group size. It should be the same
    /// as local_size_x in greedy meshing shader
    vmath::i32 meshing_shader_local_group_size{ 64 };
    /// @brief path to meshing shader src
    std::filesystem::path meshing_shader_src_path{"shaders/src/greedy_meshing_shader/shader.comp"};
    /// @brief path to meshing shader bin in spirv (optional)
    std::optional<std::filesystem::path> meshing_shader_bin_path{"shaders/bin/greedy_meshing_shader/comp.spv"};
};
```

Fragment kodu nr. 30 – struktura konfigurująca silnik, język C++

Parametr *world_size* oznacza półosie elipsoidu opisanego w rozdziale 4.5. Parametr *initial_position* oznacza inicjalizacyjną pozycję kamery, funkcjonalność której opisano w rozdziale 4.5. Parametr *chunk_size* oznacza wymiary kawałka wolumenu. Parametr *chunk_data_streamer_threads_count* konfiguruje liczbę wątków w puli *ChunkDataStreamer* (co opisano w rozdziale 4.6). Jeśli parametr ma wartość 0, liczba wątków równa

jest tej zwracanej przez funkcje `std::thread::hardware_concurrency()` [16]. Parametr `chunk_data_generator` to wskaźnik na obiekt klasy `ChunkGenerator`, który jest opisany w rozdziale 4.7. Parametr `chunk_pool_growth_coefficient` to współczynnik wzrostu bufora `VBO`, którego mechanizm poszerzenia opisano w rozdziale 4.3.5. Parametr `meshing_shader_local_group_size` jest tożsamy z liczbą wątków w grupie w module cieniującym implementującym algorytm chciwego siatkowania, co opisano w rozdziale 4.4.2. Parametr `meshing_shader_src_path` oznacza ścieżkę do kodu źródłowego modułu cieniującego zawierającego algorytm chciwego siatkowania w GLSL, a parametr `meshing_shader_bin_path` oznacza ścieżkę do skompilowanego do kodu `SPIRV` modułu cieniującego zawierającego algorytm chciwego siatkowania (parametr jest opcjonalny).

Interfejs (metody) udostępniany przez klasę `Engine` pokazano we fragmencie kodu numer 31.

```
/// @brief initialization of resources
/// @return true if there were errors during engine's initialization
bool init() noexcept;
/// @brief updates world grid state based on camera position
/// @param position new camera position
void updateCameraPosition(vmath::Vec3f32 position) noexcept;
/// @brief polls for chunks updates (non blocking!)
/// @return true if new chunk was loaded
bool pollChunksUpdates() noexcept;
/// @brief updates draw state, binds vao, vbo. If draw command buffer is dirty
/// supplies draw commands to gpu
void updateDrawState() noexcept;
/// @brief draws a scene
void draw() noexcept;
/// @brief deinitializes opengl resources
void deinit() noexcept;
```

Fragment kodu nr. 31 – metody interfejsu silnika, język C++

Metody `init` i `deinit` odpowiadają za inicjalizację i deinicjalizację. Metoda `init` zwraca wartość `true` jeśli wystąpiły błędy przy inicjalizacji silnika. Należy zaznaczyć, że silnik nie inicjalizuje kontekstu OpenGL. Zakłada się, że aplikacja korzystająca z biblioteki zrobi to we własnym zakresie przed inicjalizacją silnika. Metoda `updateCameraPosition` aktualizuje `WorldGrid` nową pozycją kamery, co opisano w rozdziale 4.5. Metoda `pollChunksUpdate` odpytuje czy dla jakiegoś kawałka została wygenerowana siatka, co opisano dokładniej w rozdziale 4.3 i 4.4. Metoda `updateDrawState` wykonuje potrzebne aktualizacje stanu kontekstu OpenGL, wiąże `VAO`, `IBO`, `VBO` oraz `DIBO`. Jeśli nastąpiła zmiana w strukturze tablicy komend renderujących siatki, przekazuje bufor komend z klienta do serwera. Metoda `draw` wywołuje komendy renderujące siatki, jak opisano w rozdziale 4.3.3.

Dodatkowo klasa *Engine* implementuje przykładowy warunek partycjonowania komend (co opisano w rozdziale 4.3.3), *frustum culling* metodą opartą na twierdzeniu o separacji hiperpłaszczyznowej [10] [11]. Metodę pokazano we fragmencie kodu numer 32.

```
/// @brief applies frustum culling based on separating axis theorem
/// @param use_last_partition wether to use last partitioning
/// @param z_near near plane
/// @param z_far far plane
/// @param x_near half width of the near plane of the frustum = eg.
/// aspect_ratio * tan(view_angle/2) * z_near
/// @param y_near half width of the near plane of the frustum = eg.
/// tan(view_angle/2) * z_near
/// @param view_matrix view matrix of the used camera
void applyFrustumCullingPartition(
    bool use_last_partition,
    vmath::f32 z_near,
    vmath::f32 z_far,
    vmath::f32 x_near,
    vmath::f32 y_near,
    vmath::Mat4f32 view_matrix
);
```

Fragment kodu nr. 32 – metoda partycjonująca komendy względem warunku czy kawałek znajduje się we frustum kamery, język C++

Dodatkowo zapewnia interfejs do generycznej metody partycjonowania komend z klasy *ChunkPool*. Metodę widać we fragmencie kodu numer 33.

```
/// @brief passes the custom partitioning call to internal partition function
/// @tparam ...Args types of aux arguments to pass to unary_op function
/// @param unary_op unary operation which is criterion based on which the commands are partitioned
/// @param use_last_partition If true then the previous partition will be partitioned again
/// @param args Aux arguments to pass to unary_op function
template<typename ...Args>
void applyCustomPartition(
    bool(*unary_op)(Face orientation, vmath::Vec3f32 position, Args... args), bool use_last_partition, Args&&... args
) noexcept {
    _world_grid..chunk_pool.partitionDrawCommands(unary_op, use_last_partition, std::forward<Args>(args)...);
}
```

Fragment kodu nr. 33 – generyczna metoda partycjonująca komendy według zadanego warunku, język C++

Partycjonowanie można włączyć i wyłączyć zmienną logiczną *partitioning* zawartą w klasie *Engine*. Klasa zawiera również zmienną *error* w której ustawiane są flagi błędów. Możliwe do wystąpienia błędy pokazano we fragmencie kodu numer 34.

```
enum Error : vmath::u32 {
    NO_ERROR = 0x0U,
    GPU_ALLOCATION_FAILED = 0x01U,
    CPU_ALLOCATION_FAILED = 0x02U,
    GPU_BUFFER_MAPPING_FAILED = 0x04U,
    FENCE_WAIT_FAILED = 0x08U,
    SHADER_ATTACH_FAILED = 0x10U,
    CHUNK_DATA_STREAMER_THREAD_ALLOCATION_FAILED = 0x20U,
    CHUNK_DATA_STREAMER_THREAD_INITIALIZATION_FAILED = 0x20U,
};
```

Fragment kodu nr. 34 – flagi błędów jakie może wygenerować silnik, język C++

Rozdział 5

Środowisko demonstracyjne

Aplikacja demonstracyjna implementuje renderowanie kawałków za pomocą sinika i kamery, którą można poruszać się za pomocą klawiatury oraz obracać się za pomocą myszy. W module cieniującym modelu kawałków użyto palety kolorów indeksowanej stanem woksela (co to jest stan woksela opisano w rozdziale 4.3). Dodano też jedno światło kierunkowe w modelu oświetlenia *phong* [12]. Aplikacja implementuje interfejs CLI przez który można konfigurować:

- Użyty generator danych.
- Wielkość pojedynczego kawałka.
- Użycie warunki partycjonowania komend.
- Wielkość zakresu widoczności.
- Liczbę możliwych stanów wokseli.

W aplikacji skorzystano z poniżej wymienionych zewnętrznych bibliotek:

- *spdlog* - biblioteka użyta do logowania komunikatów w konsoli [13].
- *glfw* - biblioteka użyta do tworzenia okienek i sczytywania wejścia z klawiatury oraz myszy [8].
- *cli11* - biblioteka ułatwiająca budowanie interfejsów CLI [4].

5.1 Użyte generatory danych

Do testów posłużyono się dwoma różnymi generatorami danych. Pierwszy z generatorów *NoiseTerrainGenerator* opisano w rozdziale 4.7.1. Drugi z generatorów *SimpleTerrainGenerator* jest dużo prostszy, pregeneruje on dane dla kawałka przy inicjalizacji i zawsze zwraca takie same dane niezależnie od pozycji kawałka (nie wykonuje żadnych obliczeń). Użycie takiego bardzo uproszczonego generatora pozwoli wyciągnąć wnioski o tym, jak duży wpływ ma generacja danych na opóźnienie (*latency*) w każdej klatce.

5.2 Wielkości pojedynczego kawałka

W testach użyto dwóch wielkości kawałków, 32x32x32 oraz 64x64x64. W każdym wypadku parametr *local_size_x* w module cieniującym algorytmu chciwego siatkowania jak opisano w rozdziale 4.4.2 oraz parametr *meshing_shader_local_group_size* ustawiano tak, aby wynosiły one odpowiednio 32 i 64.

5.3 Warunki partycjonowania komend

Zaimplementowano dwa warunki partycjonowania komend opisanego w rozdziale 4.3.3. Jeden warunek dzieli komendy na podstawie tego czy znajdują się we frustum kamery (*frustum culling*) drugi na podstawie tego, czy siatka znajduje tyłem do kierunku patrzenia kamery (*back face culling*).

5.4 Wielkości zakresu widoczności

Wielkość zakresu widoczności tłumaczy się na wielkość parametru konfiguracyjnego *world_size* (opisanego w rozdziale 4.8) przekładającego się na wielkość pól elipsoidy definiującej zakres widoczności kawałków (co opisano w rozdziale 4.5).

5.5 Liczba możliwych stanów wokseli

Liczba stanów jakie mogą przybierać woksele jest istotna w przypadku siatkowania kawałków. Im więcej dyskretnych stanów mogą mieć woksele w wolumenie, tym wygenerowana siatka będzie mniej optymalna.

Rozdział 6

Wyniki testów wydajnościowych

6.1 Metodologia

Przy przeprowadzaniu testów były mierzone następujące wartości:

- Czas jednej klatki w milisekundach.
- Czas siatkowania kawałka w milisekundach (mierzony do momentu oznaczenia kawałka jako kompletny).
- Czas siatkowania kawałka w milisekundach (mierzony do momentu wykonania się algorytmu).
- Liczba wygenerowanych prymitywów (wierzchołków) na klatkę.
- Liczba wygenerowanych fragmentów na klatkę.
- Aktywne zużycie pamięci kawałków na klatkę po stronie serwera, biorąc pod uwagę wypełnienie regionów.
- Aktywne zużycie pamięci kawałków na klatkę po stronie serwera, nie biorąc pod uwagę wypełnienia regionów.
- Pasywne zużycie pamięci kawałków na klatkę po stronie serwera, czyli całkowita wielkość puli (na klatkę zawsze taka sama).
- Aktywne zużycie pamięci kawałków na klatkę po stronie klienta.

- Pasywne zużycie pamięci kawałków na klatkę po stronie klienta, czyli całkowita wielkość puli (na klatkę zawsze taka sama).

W testach scenariusz był podobny, po inicjalizacji kamerą przesuwa się w jedną ze stron na czas trwania testu, prowokując alokacje i dealokacje kawałków. Liczba próbek na test to ok. 5000 klatek oraz ok. 3000 próbek dotyczących generacji siatek kawałków.

Tak wygenerowane próbki podzielono na grupy po 60 próbek, następnie dla każdej grupy wyliczono średnią oraz odchylenie standardowe, co pokazano na wykresach.

Należy zwrócić uwagę, że z próbek liczby wygenerowanych prymitywów na klatkę i liczby wygenerowanych fragmentów na klatkę obliczono stosunek. W ten sposób uzyskuje się informacje odnośnie zysków jakie przynosi partycjonowanie komend renderujących siatki opisane w rozdziale 4.3.3.

Testy zostały przeprowadzone dla kilku wariantów (możliwe wartości do konfiguracji opisano w rozdziale 5), aby sprawdzić jaki wpływ ma dany parametr działanie silnika. Rozdział 6.2 został podzielony względem tego jakiego parametru wpływ jest sprawdzany. Przy testach dla każdego wariantu znajdują się również wnioski.

Testy przeprowadzono na maszynie o następujących parametrach:

- OS - GNU/Linux x86_64 6.6.1-arch1-1 z X11.
- Rozdzielcość ekranu - 1920x1080.
- CPU - Intel i5-1035G1 (8) @ 3.600GHz.
- GPU - Intel Iris Plus Graphics G1.
- Ilość pamięci RAM - 16GB.

6.2 Wyniki i analiza

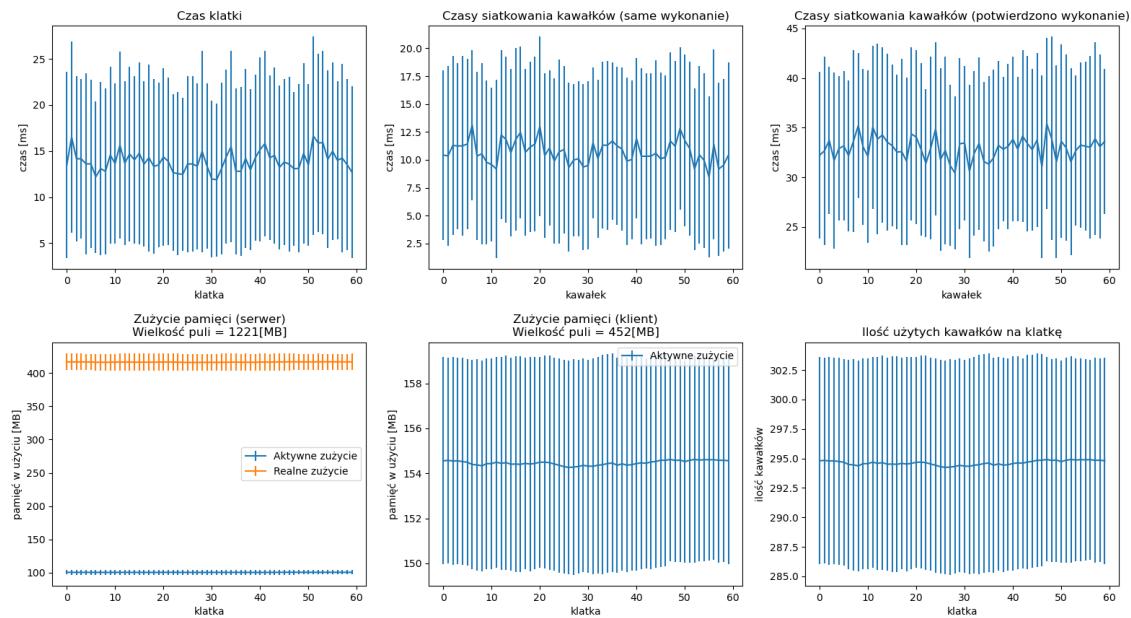
6.2.1 Wpływ wybranego generatora danych

Parametry testu:

- Generator - *NoiseTerrainGenerator* i *SimpleTerrainGenerator*.
- Wielkość zakresu widoczności - 400x200x400.

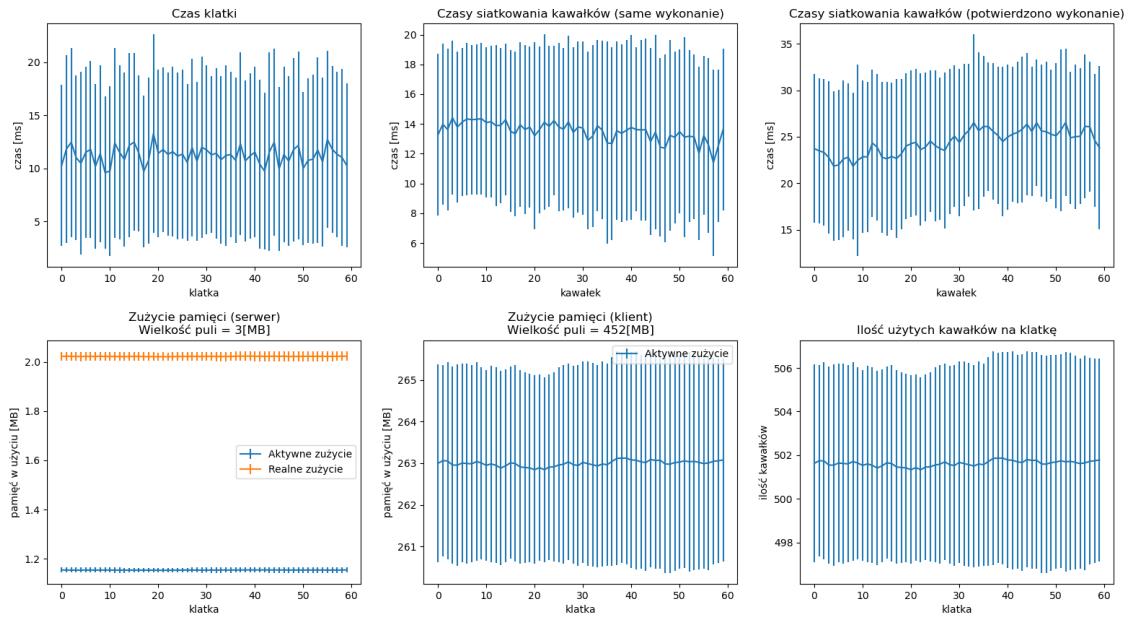
- Liczba stanów wokseli - 2.
- Warunki partycjonowania komend - *frustum culling + back face culling*.
- Wielkość kawałka - 64^3 .

Z użyciem *NoiseTerrainGenerator*



Rys. 26 – test nr. 1

Z użyciem *SimpleTerrainGenerator*



Rys. 27 – test nr. 2

Wnioski

Z racji, że dane generowane przez *SimpleTerrainGenerator* pozwalają na generację bardzo optymalnej siatki, zużycie pamięci serwera jest dużo mniejsze niż w przypadku *NoiseTerrainGenerator*. W przypadku *NoiseTerrainGenerator* należy zwrócić uwagę, że ilość pamięci serwera która faktycznie jest wypełniona siatką jest mniejsza od całej pamięci zaalokowanych kawałków i dużo mniejsza od faktycznej wielkości puli. Można z tego wyciągnąć dwa wnioski. Po pierwsze, przez to że wielkość regionu narzucona jest przez najbardziej nieoptymalny przypadek siatkowania, duża ilość pamięci w typowym przypadku (np. renderowanie terenu jak w przypadku *NoiseTerrainGenerator*) pozostaje niewykorzystana. Drugi wniosek który wyciągnąć można po spojrzeniu również na wykres zużycia pamięci klienta, to że jeśli istnieje dużo kawałków całkowicie pustych (co opisano w rozdziale 4.5), dodaje to do ilości niewykorzystanej pamięci (ilosc zużycia pamięci klienta jest większa w przypadku testu numer 2, jeśli wszystkie kawałki byłyby niepuste w teście numer 1 tak jak w teście numer 2, ilość zużycia pamięci klienta byłaby taka sama w obu przypadkach). Na podstawie wykresów czasów siatkowania również można wyciągnąć dwa wnioski. Pierwszy wniosek obserwuje się w obu testach, istnieje spora różnica między czasem

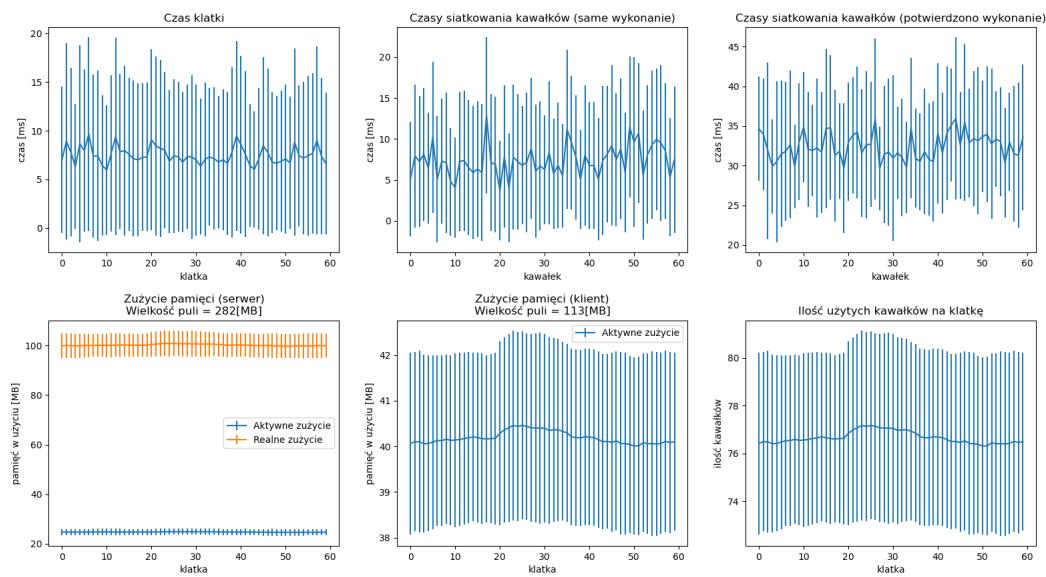
wykonania algorytmu siatkowania dla kawałka i czasem potwierdzenia wykonania (siatka została wygenerowana i kawałek został odznaczony jako kompletny). Może to wynikać z za małej częstotliwości sprawdzania czy aktywne zadanie siatkowania kawałka już się wykonało (w aplikacji jest to robione raz w pętli głównej). Drugi ciekawy wniosek pochodzi z zaobserwowania, że czas siatkowania dla samego wykonania jest szybszy w przypadku testu numer 1. Logicznie wnioskując powinno być na odwrót ponieważ kawałki generowane w teście numer 2, to szum i generują mniej optymalną siatkę. Sugeruje to, że optymalna siatka nie musi przekładać się na szybszy czas siatkowania.

6.2.2 Wpływ wielkości zakresu widoczności

Parametry testu:

- Generator - *NoiseTerrainGenerator*.
- Wielkość zakresu widoczności - 200x200x200, 400x200x400, 500x250x500.
- Liczba stanów wokseli - 2.
- Warunki partycjonowania komend - *frustum culling + back face culling*.
- Wielkość kawałka - 64^3 .

Zakres widoczności 200x200x200

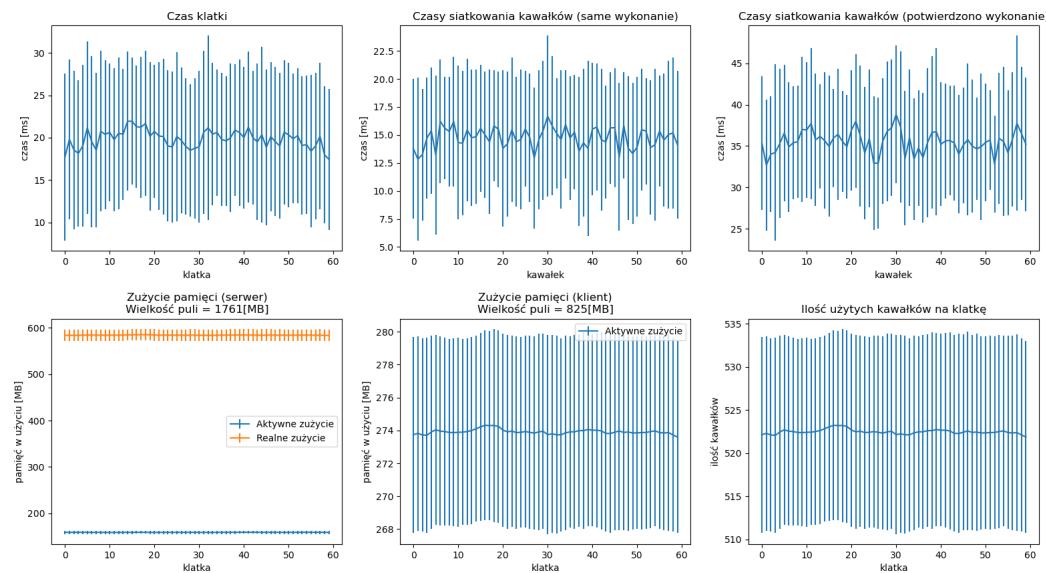


Rys. 28 – test nr. 3

Zakres widoczności 400x200x400

Taki sam jak test numer 1.

Zakres widoczności 500x250x500



Rys. 29 – test nr. 4

Wnioski

Zakres widoczności ma ogromny wpływ zarówno na zużycie pamięci, jak i czas pojedynczej klatki. Uwage przykuwa stopniowe zwiększenie się czasu siatkowania w zależności od wielkości zakresu widoczności. W przypadku wykonania oraz potwierdzenia wykonania jest to logiczne, im większa wielkość zakresu widoczności tym więcej kawałków, co przekłada się na dłuższy czas oczekiwania aż nowe kawałki się wygenerują i zaalokują w każdej klatce. Z kolei powodu coraz dłuższego czasu samego wykonania się algorytmu siatkowania nie udało się skonkludować. Być może ma to związek z tym, że ogólne wykorzystanie GPU zwiększa się.

6.2.3 Wpływ wielkości pojedynczego kawałka

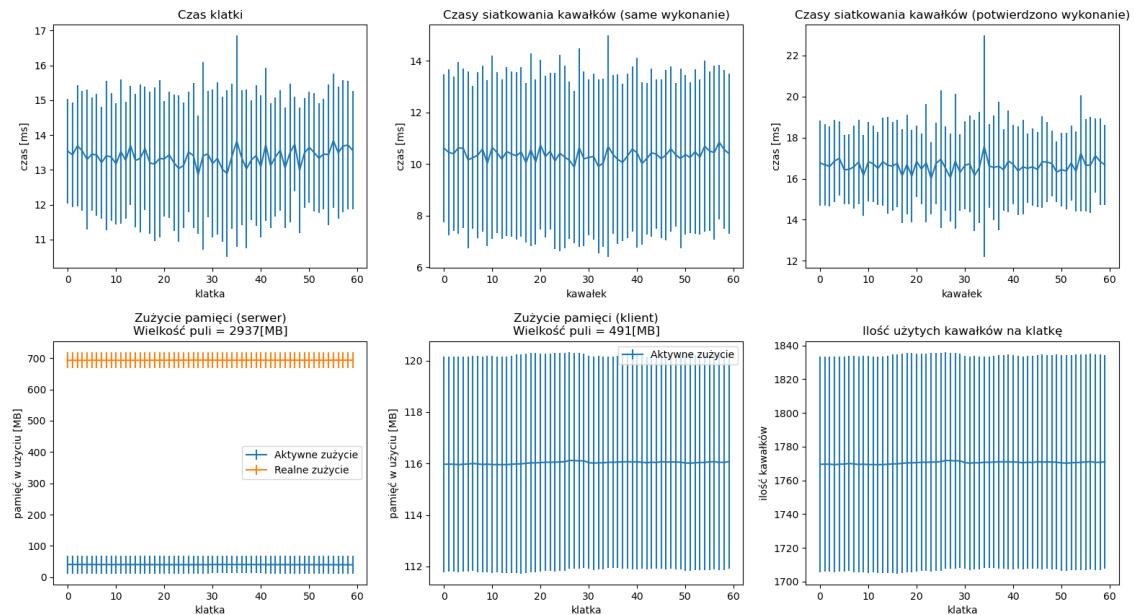
Parametry testu:

- Generator - *NoiseTerrainGenerator*.
- Wielkość zakresu widoczności 400x200x400.
- Liczba stanów wokseli - 2.
- Warunki partycjonowania komend - *frustum culling + back face culling*.
- Wielkość kawałka - 64^3 oraz 32^3 .

Wielkość kawałka 64^3

Taki sam jak test numer 1.

Wielkość kawałka 32^3



Rys. 30 – test nr. 5

Wnioski

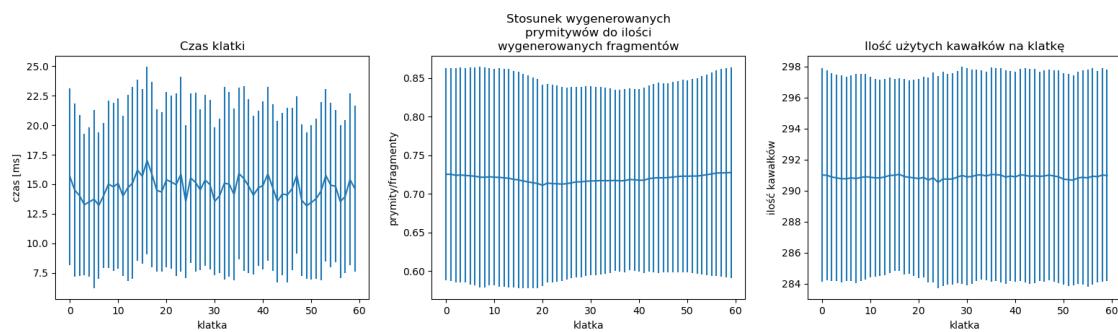
Jak widać w teście numer 5 zużycie pamięci serwera jest znacznie większe. Zwiększenie częstotliwości próbkowania wolumenu kawałkami (czyli zmniejszenie wielkości kawałka), w przypadku gdy wielkość regionu narzucona jest przez najgorszy wynik algorytmu siatkowania i więcej jest dużo więcej optymalnych przypadków siatki od nieoptymalnych przypadków siatki, prowadzi do większej liczby nieużywanej pamięci. Czas wykonania algorytmu siatkowania jest podobny, mimo to że w przypadku testu numer 5 powinien trwać krócej (kawałek jest mniejszy). Czas potwierdzenia i wykonania siatkowania jest większy w przypadku testu numer 1. Wynika to prawdopodobnie ze zredukowanego czasu generacji danych dla jednego kawałka. Niemniej jednak czasy klatek są podobne w obu testach, więc wielkość kawałka 64^3 jest lepszym rozwiązaniem.

6.2.4 Wpływ warunków partycjonowania komend renderujących siatki

Parametry testu:

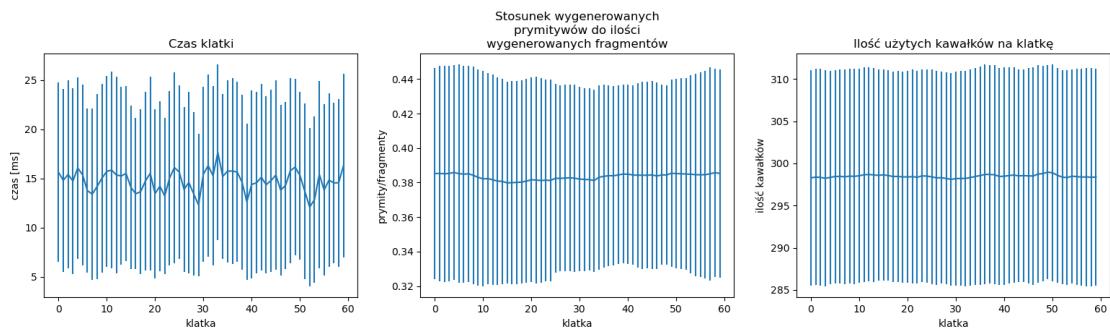
- Generator - *NoiseTerrainGenerator*
- Wielkość zakresu widoczności 400x200x400
- Liczba stanów wokseli - 2
- Warunki partycjonowania komend - brak, sam *frustum culling*, *frustum culling* + *back face culling*
- Wielkość kawałka - 64^3

Warunek: brak



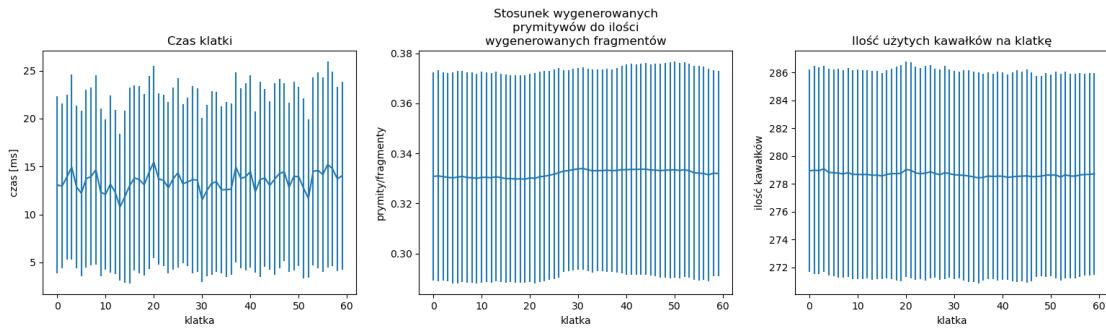
Rys. 31 – test nr. 6

Warunek: *frustum culling*



Rys. 32 – test nr. 7

Warunek: *frustum culling* oraz *back face culling*



Rys. 33 – test nr. 8

Wnioski

Jak widać, stosunek wygenerowanych prymitywów do wygenerowanych fragmentów zmniejsza się. Największy spadek następuje z testu numer 6 do testu numer 7. W przypadku testów numer 7 i numer 8 stosunek zmniejsza się ale nie tak spektakularnie jak pomiędzy testami 6 i 7. Czas klatki zmniejsza się widocznie pomiędzy testami numer 6 i numer 8.

6.2.5 Wpływ liczby stanów wokseli

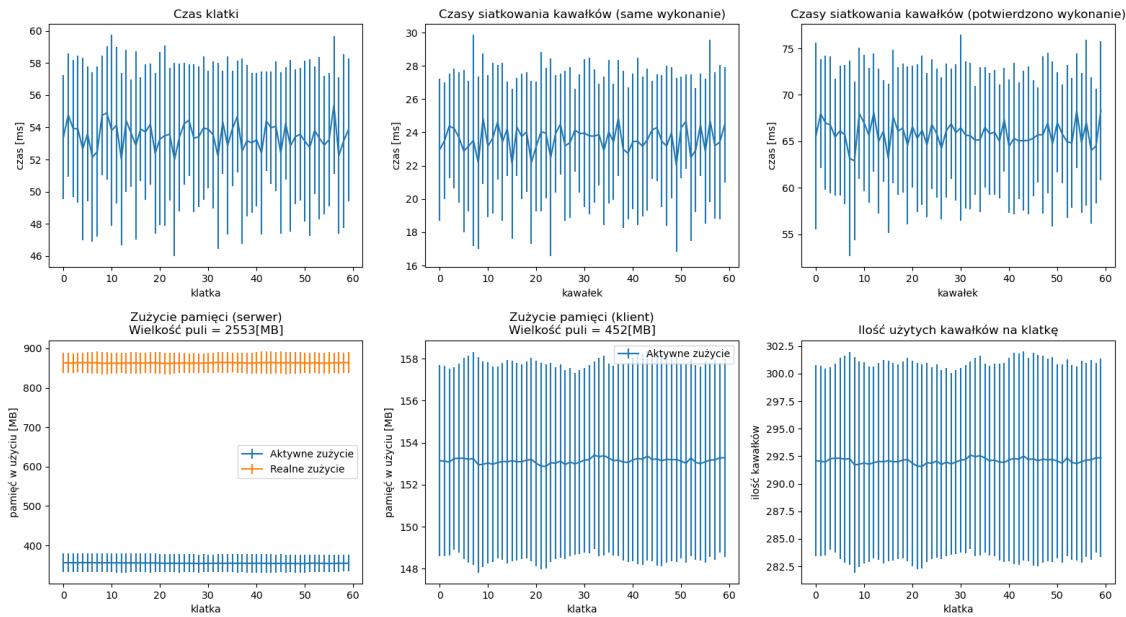
Parametry testu:

- Generator - *NoiseTerrainGenerator*
- Wielkość zakresu widoczności 400x200x400.
- Liczba stanów wokseli - 2 i 1000.
- Warunki partycjonowania komend - *frustum culling* + *back face culling*.
- Wielkość kawałka - 64^3 .

Dwa stany wokseli

Taki sam jak test numer 1.

1000 stanów wokseli



Rys. 34 – test nr. 9

Wnioski

Jak widać w teście numer 9, zwiększenie liczby stanów wokseli powoduje większe zużycie pamięci, dłuższy czas siatkowania oraz ogólnie mniejszą optymalność, co widać po czasie pojedynczej klatki. Większa liczba stanów prowadzi do generacji gorzej zoptymalizowanych siatek, co powoduje większe zużycie pamięci oraz dłuższy średni czas siatkowania kawałka.

6.3 Wnioski finalne

Testy wydajnościowe pokazują, że silnik spełnia wymagania aplikacji czasu rzeczywistego. Długość trwania klatki nie przekracza w żadnym teście 100 milisekund. Jest on w stanie renderować kilka set kawałków o rozdzielczości 64^3 wokseli bez znaczących spadeków wydajności. Nie mniej jednak niektóre z wyników pokazują, że w pewnych aspektach wymaga on optymalizacji. Jak opisano w rozdziale 6.2.1 i co widoczne było w innych testach, czas faktycznego wykonania się algorytmu siatkowania, a wykonania oraz potwierdzenia wykonania się algorytmu dzieli często czas

Rozdział 6: Wyniki testów wydajnościowych

liczony w dziesiątkach milisekund. Oznacza to, że należałoby zwiększyć częstotliwość sprawdzania czy zadanie siatkowania wykonało się dla danego kawałka. We wszystkich testach widać również, że pomimo zastosowania techniki dynamicznego rozszerzania się bufora *VBO*, opisanej w rozdziale 4.3.5, duża część pamięci bufora *VBO* pozostaje niewykorzystana, co jak opisano w rozdziale 6.2.1 wynika z faktu, że w praktyce kawałki mogą się bardzo różnić pod względem optymalności siatki wygenerowanej na ich podstawie. Sugeruje to, że należałoby zmodyfikować strukturę danych tak, aby wielkość regionu nie była narzucona przez najmniej optymalne siatki.

Rozdział 7

Podsumowanie

Celem pracy była implementacja wokselowego silnika graficznego z wykorzystaniem technologii OpenGL.

Cel pracy został osiągnięty. Rezultat bazował na istniejących rozwiązań opisanych w rozdziale 2.1 oraz rozdziale 3.2. Niemniej jednak zawiera on wartość dodaną tj. algorytm chciwego siatkowania zaimplementowany na module cieniującym typu *compute* opisany w rozdziale 4.4, dynamiczne poszerzanie się puli pamięci siatki wolumenu opisane w rozdziale 4.3.3 i struktura danych zarządzająca widocznością kawałków opisana w rozdziale 4.5.

Wyniki testów wydajnościowych zaprezentowanych w rozdziale 6, których środowisko przeprowadzania zostało opisane w rozdziale 5 pokazują, że silnik jest wystarczająco optymalny do wykorzystania w aplikacjach czasu rzeczywistego. Pokazano również, że możliwe są jego optymalizacje w kilku aspektach tj. zwiększenie aktywnego zużycia pamięci serwera, czy zmniejszenie czasu generacji siatki dla danego kawałka jak opisano to w rozdziale 6.3.

Silnik zostanie udostępniony publicznie w postaci repozytorium na platformie github.com z licencją MIT. Repozytorium zawiera *CI/CD pipeline* komplilujący silnik na architekturę *x86_64* dla systemów operacyjnych Windows oraz Linux.

Bibliografia

- [1] Approaching zero driver overhead. <https://www.gdcvault.com/play/1020791>. dostęp: 2023.
- [2] Arb_gl_spirv. https://registry.khronos.org/OpenGL/extensions/ARB/ARB_gl_spirv.txt. dostęp: 2023.
- [3] Circular buffer. https://en.wikipedia.org/wiki/Circular_buffer. dostęp: 2023.
- [4] Cli11. <https://github.com/CLIUtils/CLI11>. dostęp: 2023.
- [5] Ellipsoid. <https://en.wikipedia.org/wiki/Ellipsoid>. dostęp: 2023.
- [6] Fastnoise2. <https://github.com/Auburn/FastNoise2>. dostęp: 2023.
- [7] Glad multi-language gl/gles/egl/glx/wgl loader-generator based on the official specs. <https://glad.dav1d.de/>. dostęp: 2023.
- [8] glfw. <https://github.com/glfw/glfw>. dostęp: 2023.
- [9] High performance voxel engine: Vertex pooling .
<https://nickmcd.me/2021/04/04/high-performance-voxel-engine/>. dostęp: 2023.
- [10] Hyperplane separation theorem. https://en.wikipedia.org/wiki/Hyperplane_separation_theorem. dostęp: 2023.
- [11] More (robust) frustum culling. https://bruop.github.io/improved_frustum_culling/. dostęp: 2023.
- [12] Phong shading. https://en.wikipedia.org/wiki/Phong_shading. dostęp: 2023.

- [13] spdlog. <https://github.com/gabime/spdlog>. dostęp: 2023.
- [14] std::future - cppreference. <https://en.cppreference.com/w/cpp/thread/future>. dostęp: 2023.
- [15] std::promise - cppreference. <https://en.cppreference.com/w/cpp/thread/promise>. dostęp: 2023.
- [16] std::thread::hardware_concurrency - cppreference. https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency. dostęp: 2023.
- [17] Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. Praca doktorska, Grenoble University, Grenoble, lipiec 2011.
- [18] Khronos. *The OpenGL® Graphics System: A Specification*. Khronos, wydanie 4.5 core, czerwiec 2017.
- [19] Khronos. *The OpenGL® Shading Language*. Khronos, wydanie language version: 4.50, maj 2017.
- [20] Robert Nystrom. *Game Programming Patterns* .
<http://gameprogrammingpatterns.com>, -, 2014. dostęp: 2023.
- [21] Harvey E. Cline William E. Lorensen. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH*, 21(4):163–169, lipiec 1987.

Dodatek A

Słowniczek

Wolumen - Obiekt trójwymiarowy w opisie wokselowym. Tablica danych reprezentująca prostopadłościan o dyskretnych wymiarach, gdzie element tablicy to woksel.

Kawałek - Wycinek wolumenu w danym miejscu, o danych wymiarach X, Y, Z .

Woksel - Najmniejszy element wolumenu.

Boksel - Woksel reprezentowany w postaci siatki wielokątów.

Siatkowanie - Proces generacji siatki wielokątów na podstawie wolumenu.

Renderowanie - Proces prowadzący do graficznego przedstawienia danego zbioru danych.

X - Rozmiar kawałka wolumenu w osi X.

Y - Rozmiar kawałka wolumenu w osi Y.

Z - Rozmiar kawałka wolumenu w osi Z.

C_{max} - Maksymalna możliwa do zaalokowania ilość kawałków w danej konfiguracji silnika.

VBO - Vertex Buffer Object, bufor przechowujący dane wierzchołkowe w OpenGL API.

klient - Aplikacja komunikująca się z kontekstem OpenGL (CPU).

serwer - Kontekst OpenGL aplikacji (GPU).

a - długość pół osi elipsoidy w osi X.

b - długość pół osi elipsoidy w osi Y.

c - długość pół osi elipsoidy w osi Z.