



Politechnika Gdańska

Wydział Elektroniki,
Telekomunikacji i
Informatyki



Katedra: Katedra Inżynierii Wiedzy

Imię i nazwisko dyplomanta: **Karol Gasiński**

Nr albumu: 106287

Forma i poziom studiów: Stacjonarne jednolite studia magisterskie

Kierunek studiów: Informatyka,
Inżynieria dokumentu

Praca dyplomowa magisterska

Temat pracy: **Porównanie metod wizualizacji terenu
w czasie rzeczywistym**

Kierujący pracą: dr inż. Jacek Lebieź
Konsultant pracy: mgr inż. Krzysztof Mieloszyk

Zakres pracy: Celem pracy jest zbadanie wydajności różnych technik wizualizacji terenu zadanego opisem matematycznym. Techniki te zostaną przebadane pod względem szybkości wykonania oraz jakości tworzonego obrazu na różnych poziomach szczegółowości.

Gdańsk, 2010

*Pracę tę dedykuję mojemu dziadkowi,
który zawsze mnie wspierał.*

Spis treści

1	Wstęp	1
2	Wprowadzenie do wizualizacji terenu	2
2.1	Klasyczne modele wizualizacji	2
2.1.1	Model rastrowy	2
2.1.2	Nieregularna siatka terenu	3
2.2	Podstawowe optymalizacje	4
2.2.1	Backface culling	4
2.2.2	Frustum culling	4
2.3	Algorytmy o zmiennej rozdzielczości siatki	6
2.3.1	Bottom-Up i Top-Down	6
2.3.2	Reorganizacja siatki terenu	7
2.3.3	Algorytm ROAM	11
2.4	Podsumowanie technik klasycznych	14
3	Teoria sferycznych modeli opisu terenu	15
3.1	Wprowadzenie do sferycznych metod wizualizacji	15
3.2	Współrzędne sferyczne	15
3.3	Opis matematyczny	16
3.4	Wielościany foremne	17
3.5	Podstawowe kryteria	18
3.6	Quad-Sfera	18
3.6.1	Projekcja CAR	19
3.6.2	Siatka terenu	20
3.6.3	Zniekształcenia	21
3.7	Cube-Sfera	21
3.7.1	Projekcje TAN i TSC	22
3.7.2	Projekcja CSC	26
3.7.3	Projekcja QSC	30
3.7.4	Siatka terenu	33
3.8	Ico-Sfera	34
3.8.1	Projekcje	34
3.8.2	Siatka terenu	36
4	Środowisko implementacyjne	38
4.1	Konfiguracja sprzętowa	38
4.1.1	Wymagana generacja kart graficznych	38
4.2	Użyte biblioteki i oprogramowanie	39
4.2.1	Ngine 3.0	39
4.2.2	Środowisko IDE	41
4.3	CubeResampler	41

5	Implementacja	42
5.1	Struktura programu	42
5.1.1	Diagram klas	42
5.2	Autorski algorytm HRTMR	54
5.2.1	Przygotowanie sektorów	54
5.2.2	Triangulacja i rendering	57
5.3	Odświeżenie stanu wizualizacji	59
5.3.1	Korekcja odległości	60
5.3.2	Reorganizacja siatki	63
5.3.3	Test widoczności sektorów	66
5.3.4	Macierz sektora i korekcja precyzji	66
5.4	Rendering	67
5.4.1	Vertex Shader	67
5.4.2	Fragment Shader	71
6	Eksperymenty	73
6.1	Środowisko testowe	73
6.2	Złożoność sceny	75
6.3	Czas renderingu	75
6.4	Poprawność wizualizacji	75
6.5	Obciążenie	79
7	Wnioski	83

1 Wstęp

Trójwymiarowy obraz przedstawiający ukształtowanie terenu nazywamy jego wizualizacją. Wizualizacje takie mają szerokie zastosowanie w dzisiejszym świecie, począwszy od turystyki, a na planowaniu działań taktycznych kończąc. Dla przykładu wykorzystuje się je w geodezji do badania przekrojów gruntu. Dzięki temu można oznaczyć obszary posiadające złoża surowców naturalnych oraz przykładowe punkty wydobywania. Tymczasem w urbanistyce pomagają przy projektowaniu zagospodarowania miejskiego. Jednak ich głównym zastosowaniem są obecnie symulatory lotów oraz rynek gier komputerowych.

Wzrastające możliwości sprzętu komputerowego jak i oczekiwania rynku sprawiają, że wizualizacje te muszą posiadać coraz lepszą jakość generowanego obrazu. Wymaga się od nich również prezentowania coraz większych i dokładniejszych zestawów danych. Czynniki te zmuszają programistów do opracowywania algorytmów potrafiących przetwarzać wspomniane dane w efektywny sposób.

Celem pracy jest zbadanie wydajności różnych technik wizualizacji terenu zadanego opisem matematycznym. Do wykonania tego zadania została stworzona aplikacja zawierająca implementację dwóch metod sferycznych: Quad-Sfery i Cube-Sfery. Techniki bazujące na tych metodach zostaną przebadane pod względem szybkości wykonania oraz jakości tworzonego obrazu na różnych poziomach szczegółowości.

2 Wprowadzenie do wizualizacji terenu

Techniki wizualizacji terenu są tematem intensywnych badań od przeszło trzydziestu lat. Przez ten czas powstało wiele rozwiązań i algorytmów. Mają one na celu osiągnięcie jak najlepszych efektów wizualnych przy jednoczesnym zachowaniu zadowalającego poziomu wydajności. Poniższy rozdział zajmuje się wprowadzeniem do tego szerokiego zagadnienia.

2.1 Klasyczne modele wizualizacji

Najprostszą i zarazem najpopularniejszą metodą wizualizacji terenu jest przedstawienie go w postaci połałdowanej powierzchni. Powierzchnia taka jest tworzona za pomocą zbioru trójkątów. Wierzchołki, na których je rozpięto są punktami, w których pomierzono wysokość terenu. Wszystkie modele bazujące na opisanej metodzie nazywane są w tej pracy klasycznymi modelami wizualizacji. Ma to na celu odróżnienie ich od modeli sferycznych.

Informacje o ukształtowaniu terenu, niezbędne do wygenerowania obrazu, są przechowywane w specjalnie do tego celu opracowanych strukturach danych. Struktury te mają znaczący wpływ na to, jaką funkcjonalność będzie posiadała wspomniana wizualizacja oraz jakie algorytmy będzie można w niej zastosować. Obecnie najczęściej stosowane są dwie metody przechowywania danych o ukształtowaniu terenu, rastrowa i nieregularnej sieci.

2.1.1 Model rastrowy

Pierwszym i najbardziej rozpowszechnionym sposobem opisu terenu jest model rastrowy zwany potocznie mapą wysokości. Należy on do grupy numerycznych modeli terenu NMT (ang. *DEM - digital elevation model*). Opisywany obszar dzieli się równomiernie na komórki o zadanym rozmiarze, który jednoznacznie określa dokładność modelu. Następnie dla każdej komórki oblicza się średnią wysokość i zapisuje ją w pamięci.

Za pomocą modelu rastrowego można więc przybliżyć funkcję dwóch zmiennych, które odpowiadają pozycji na płaszczyźnie, i zwracającej wysokość terenu

w tym punkcie. Wartości x i y dla których zapisuje się uśrednioną wysokość są dyskretne ze względu na ograniczoną dokładność pomiaru. Aby uzyskać wysokość w dowolnym punkcie wskazanym przez współrzędne ciągłe, należy posłużyć się algorytmami interpolującymi najbliższe wartości w zbiorze danych.

Niewątpliwą zaletą modelu rastrowego jest prosty sposób przechowywania danych. Umożliwia on implementację stronicowania tych danych z dysku do pamięci głównej, dzięki czemu aplikacja jest w stanie przetwarzać większe zbiory informacji. Możliwe jest również manipulowanie danymi, co wzbogaca mechanizm wizualizacji o dodatkową funkcjonalność jak sprawdzanie kolizji czy dokonywanie dynamicznej edycji terenu.

2.1.2 Nieregularna siatka terenu

Stała gęstość opisu wysokości terenu w modelu rastrowym jest często wadą. Ukształtowanie terenu jest zróżnicowane. W niektórych obszarach jest on bardzo pofałdowany, pełen wzniesień, skarp i nierówności. W innych obszarach, dla przykładu na pustyniach czy w dolinach jest on mniej szczegółowy i prostszy w opisie. Najlepszym rozwiązaniem byłoby więc opisać go z różną liczbą próbek wysokości w zależności od jego ukształtowania. Taką metodę opisu terenu nazywa się nieregularną siatką trójkątów (ang. *TIN - triangulated irregular network*).

Model TIN nie posiada nadmiarowych danych tak jak ma to miejsce w przypadku modelu rastrowego. Dokładność, z jaką zostanie zapisana informacja o danym obszarze, jest definiowana w postaci liczby wierzchołków przechowujących informacje o jego ukształtowaniu.

Powstała nieregularna siatka trójkątów jest w stanie przechowywać ukształtowanie terenu ze znacznie mniejszym nakładem wielokątów niż metoda rastrowa o takiej samej dokładności. Niestety wadą metody TIN jest skomplikowany zapis danych sprawiający problemy przy ich przetwarzaniu. Każdy wierzchołek musi zawierać nie tylko informację o jego wzniesieniu, tak jak ma to miejsce w modelach NMT, ale również jego pozycję i wskaźy na wierzchołki sąsiednie. Problemy te sprawiają, że programiści aplikacji wizualizujących teren częściej

skłaniają się ku modelowi rastrowemu.

2.2 Podstawowe optymalizacje

W wyniku analizy danych opisujących ukształtowanie terenu, powstają siatki składające się z milionów trójkątów. Jest to stanowczo za duża ilość informacji do przetworzenia w czasie rzeczywistym. Aby rozwiązać ten problem stworzono wiele algorytmów, które mają na celu zoptymalizowanie procesu renderingu. Pierwszymi z nich były algorytmy określające widoczność badanego trójkąta.

2.2.1 Backface culling

Podstawowym algorytmem stosowanym do określenia widoczności danej płaszczyzny jest "backface culling". W dosłownym tłumaczeniu oznacza to odrzucanie obszarów skierowanych do obserwatora "plecami". Polega on na wyznaczeniu wektora L skierowanego od obserwatora do centrum badanego trójkąta oraz jego wektora normalnego N . Na podstawie tych dwóch wektorów, korzystając z ich skalarnego mnożenia, oblicza się cosinus kąta zawartego między nimi. Jeżeli wyliczona wartość jest ujemna, trójkąt jest skierowany w stronę obserwatora. Jeżeli wartość jest dodatnia, trójkąt jest niewidoczny i zostaje pominięty.

$$V = \vec{L} \cdot \vec{N} \quad (1)$$

Opisany algorytm jest obsługiwany sprzętowo przez współczesne karty graficzne. W przypadku wizualizacji terenu można go jednak wykorzystać do stworzenia bardziej złożonych testów widoczności. Testy takie zaimplementowane po stronie jednostki centralnej CPU mogą określać widoczność całych sektorów terenu.

2.2.2 Frustum culling

Algorytm backface culling pozwala jedynie odrzucić trójkąty niewidoczne z powodu ich skierowania tyłem do obserwatora. Nie bierze ona jednak pod uwagę informacji o ich położeniu na scenie względem widza. Dla przykładu trójkąty,

które są skierowane w stronę obserwatora, ale znajdują się za jego plecami zostaną przez ten test uznane za widoczne, mimo że w rzeczywistości tak nie jest. Aby poprawnie rozwiązać takie przypadki, powstał algorytm zwany "Frustum cullingiem".

Wszystko, co jest widoczne na scenie podlega projekcji perspektywicznej na płaszczyznę rzutowania, która odzwierciedla powierzchnię ekranu. Ponieważ wraz ze wzrostem odległości obiekty stają się coraz mniejsze, zdefiniowano odległość graniczną określaną jako pole widzenia. W odległości tej znajduje się płaszczyzna obcinania. Obiekty znajdujące się za tą płaszczyzną nie są wyświetlane na ekranie, ponieważ znajdują się za daleko od obserwatora. Jeżeli na obu tych płaszczyznach zaznaczy się po cztery punkty odpowiadające narożnikom widocznego obszaru na ekranie, można za ich pomocą wyznaczyć cztery płaszczyzny łączące płaszczyznę obcinania z płaszczyzną rzutowania. Płaszczyzny te będą się znajdować na granicy pola widzenia kolejno z prawej, lewej, górnej i dolnej strony ekranu. Powstała w ten sposób bryła stanowi ostrosłup ścięty i nazywana jest Frustumem.

Dla każdej ściany frustumu wyznacza się równanie płaszczyzny w ten sposób, aby jej wektor normalny był skierowany do środka bryły. Następnie do uzyskanych równań podstawia się pozycje wierzchołków badanego trójkąta. Jeżeli wszystkie sześć równań zwróci wartość dodatnią, wierzchołek znajduje się wewnątrz bryły obcinania. Analizując wyniki dla wszystkich trzech wierzchołków można określić czy badany trójkąt jest poza frustumem, przecina go lub znajduje się wewnątrz. Dla ostatnich dwóch przypadków określa się trójkąt jako widoczny i przesyła dalej do wyświetlenia.

Również algorytm Frustum Culling jest wykonywany sprzętowo przez karty graficzne dla aktualnej macierzy projekcji. W przypadku wizualizacji terenu jest on jednak rozszerzony o testowanie sfer lub brył opisujących sektory mapy. Z tego powodu podobnie jak Backface Culling jest on implementowany indywidualnie po stronie CPU.

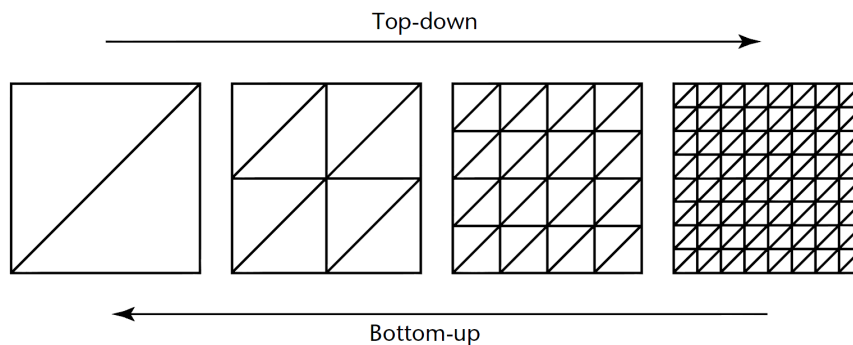
2.3 Algorytmy o zmiennej rozdzielczości siatki

Najważniejszą właściwością wykorzystywaną przez algorytmy wizualizujące jest fakt, że wraz ze wzrostem odległości widza od obiektu jego obraz maleje. Dzieje się tak z powodu zastosowanej projekcji perspektywicznej. W związku z tym, ilość szczegółów obserwowanego obiektu, jaką jest się w stanie rozpoznać jest coraz mniejsza. Wykorzystując tę cechę powstały algorytmy upraszczające siatkę terenu wraz ze wzrostem jej odległości od widza. Algorytmy takie nazywane są algorytmami o zmiennej rozdzielczości terenu. Są one stosowane w najbardziej wymagających aplikacjach czasu rzeczywistego. W większości przypadków można je podzielić na dwie kategorie.

2.3.1 Bottom-Up i Top-Down

Podejście "Bottom-Up" polega na założeniu, że w momencie rozpoczęcia pracy algorytm posiada maksymalnie szczegółową siatkę terenu do wizualizacji. Następnie na podstawie pozycji obserwatora i kierunku, w którym skierowany jest jego wzrok, algorytm upraszcza siatkę terenu eliminując niewidoczne detale. W ten sposób można otrzymać siatkę terenu o optymalnie uproszczonej geometrii dla żadanego poziomu dokładności. Niestety technika ta wymaga przetworzenia maksymalnej ilości danych w pierwszych krokach algorytmu, co wiąże się z koniecznością użycia dużej mocy obliczeniowej jak również ze znacznym zużyciem pamięci.

Technika "Top-Down" zakłada odwrotne podejście w stosunku do "Bottom-Up" (rys. 1). Cały wizualizowany teren jest opisany za pomocą minimalnej liczby wielokątów, zazwyczaj równej dwóm lub czterem trójkątom tworzącym bazową płaszczyznę. Następnie siatka terenu jest uszczegóławiana poprzez kolejne podziały i wybrzuszenia do momentu osiągnięcia oczekiwanego poziomu dokładności. Taki sposób tworzenia geometrii terenu umożliwia również stosowanie testów widoczności ograniczających przetwarzany zbiór danych do obszaru widocznego przez obserwatora. Podejście "Top-Down" wymaga również mniej zasobów i mocy obliczeniowej od techniki "Bottom-Up". Z tego powodu jest ono



Rysunek 1: Optymalizacja siatki terenu za pomocą technik Top-Down i Bottom-Up.

częściej stosowane w wielu wizualizacjach.

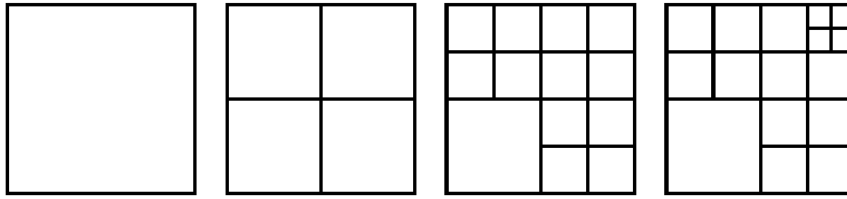
Z wymienionych powyżej dwóch technik wyewoluowała metoda hybrydowa. Korzysta ona z zależności czasowych występujących pomiędzy kolejnymi klatkami generowanego obrazu. Mianowicie, dla każdej kolejnej klatki bada ona przesunięcie obserwatora względem poprzedniej pozycji w terenie. Na podstawie otrzymanych danych upraszcza ona obszary, które po Δt znalazły się dalej od widza a uszczegóławia te do których się on zbliżył. Algorytmem bazującym na tej technice jest np. ROAM [Du97].

2.3.2 Reorganizacja siatki terenu

Algorytmy o zmiennej rozdzielczości terenu potrzebują odpowiedniej struktury danych do zarządzania podziałem przestrzeni. Struktura taka musi być w stanie opisać zmieniającą się wraz z odległością jakość siatki i pozwalać na jej dynamiczne przebudowywanie. Idealnym do tego celu rozwiązaniem są struktury drzewiaste, spośród których wyróżnia się drzewa czwórkowe i drzewa binarne.

Drzewo czwórkowe

W drzewie czwórkowym cały teren jest na początku opisany za pomocą jednego kwadratowego sektora. Następnie, gdy poziom dokładności jest za mały, algorytm dokonuje podziału tego sektora na cztery identyczne podsektory. Sektor



Rysunek 2: Przykład podziału przestrzeni przy użyciu drzewa czwórkowego.

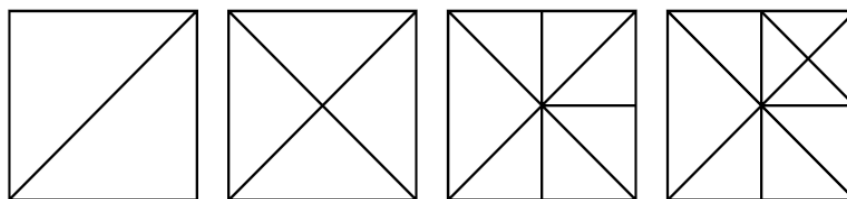
dzielony staje się sektorem rodzicielskim (ang. *parent node*) natomiast powstające sektory jego sektorami potomnymi (ang. *child node*). Każdy z sektorów potomnych w wyniku podziału zajmuje 25% powierzchni swojego rodzica. Powstałe w ten sposób sektory są nowymi kandydatami do podziału. Algorytm ten jest powtarzany rekurencyjnie, do momentu uzyskania odpowiedniego poziomu dokładności (rys. 2).

Gdy sektor jest zbyt szczegółowy wykonuje się operację odwrotną polegającą na usunięciu go wraz z jego trzema sąsiadami i zastąpieniu przez sektor rodzicielski. Należy tu jednak pamiętać o tym, aby najpierw sprawdzić, czy któryś z czwórki sektorów nie posiada sektorów potomnych. Jeżeli tak jest, należy najpierw wywołać tę metodę rekurencyjnie dla wspomnianych podsektorów.

W wyniku omówionych operacji powstaje ostateczny zestaw sektorów opisujących teren. Aby wygenerować siatkę terenu, każdy z sektorów można potraktować jako parę trójkątów rozpiętych na jego wierzchołkach. Wysokość wierzchołków jest następnie pobierana z mapy wysokości opisanej metodą NMT tworząc finalną siatkę.

Drzewo binarne

Alternatywą dla drzewa czwórkowego jest drzewo binarne. Strukturą startową dla takiego drzewa jest kwadratowa podstawa podzielona po przekątnej na dwie równe części. Części te są sektorami początkowymi i mają kształt trójkątów prostokątnych o równej długości ramion. W przeciwieństwie do drzewa czwórkowego każdy sektor dzieli się na dwa sektory potomne. Podział taki następuje



Rysunek 3: Podział przestrzeni oparty na drzewie dwójkowym.

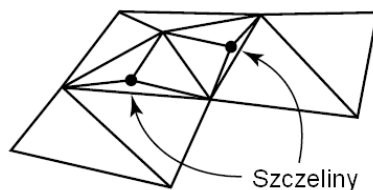
poprzez wstawienie wierzchołka pośrodku przeciwprostokątnej i utworzeniu nowej krawędzi od niego do wierzchołka w kącie prostym (rys. 3).

Zaletą drzew binarnych jest fakt, że każdy sektor sąsiaduje z sektorami, które mogą się od niego różnić tylko o jeden poziom podziału. Dzięki temu płynność zmiany rozdzielczości terenu jest zachowana. Drzewa czwórkowe nie posiadają domyślnie takiej zdolności, przez co trzeba stosować dodatkowe operacje mające na celu zachowanie płynności przejść. Drzewa binarne potrafią sobie także łatwo radzić z powstającymi nieregularnościami w siatce (takimi jak szczeliny czy T-połączenia).

Szczeliny i T-połączenia

W trakcie podziału przestrzeni powstają krawędzie, na których stykają się sektory o różnej rozdzielczości. W takiej sytuacji wierzchołek mniejszego i zarazem dokładniejszego sektora będzie się znajdował na krawędzi sektora większego, mniej dokładnego. Dla większego sektora wysokość w tym punkcie krawędzi jest interpolowana na podstawie wysokości wierzchołków, na których jest ona rozpięta. Tymczasem dla mniejszego sektora wysokość w tym punkcie zostanie odczytana z mapy wysokości. Taki przypadek nazywa się szczeliną, ponieważ prawie zawsze wysokości te różnią się tworząc w siatce otwór (rys. 4).

Zaproponowano wiele rozwiązań mających na celu zapobiec temu problemowi. Pierwszym rozwiązaniem było wstawienie dodatkowego trójkąta uzupełniającego siatkę w miejscu szczeliny. Byłby on rozpięty na wierzchołkach tworzących krawędź większego sektora i wierzchołku sektora sąsiedniego. Rozwiązanie



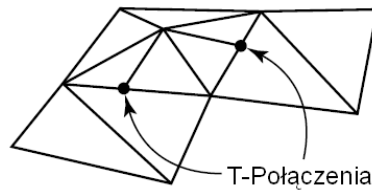
Rysunek 4: Przykład szczeliny występującej na granicy sektorów.

takie nie sprawdza się, gdy różnica między poziomami dokładności jest na tyle duża, że sektor tworzący krawędź styka się z więcej niż dwoma sektorami sąsiednimi. Powoduje ona również powstawanie dziwnie wyglądających nierówności terenu podobnych do klifów.

W związku z tym zdecydowano, że można by obniżyć pozycje wierzchołków znajdujących się na współdzielonej krawędzi. Na podstawie położenia wierzchołka oblicza się wysokość krawędzi większego sektora w tym miejscu a następnie przypisuje ją wierzchołkowi. Takie podejście sprawia że dokładniejsze sektory "wygładzają się" na połączeniu z sektorami prostszymi tworząc tak zwane T-połączenie (rys. 5).

Niestety brak dokładności w obliczeniach zmiennoprzecinkowych sprawia, że podczas renderingu połączenie takie jest niedokładne i z klatki na klatkę powstają losowe prześwity w mapie. Są one widoczne w postaci migających pojedynczych pikseli obrazu. Dzieje się tak, ponieważ linia tworząca krawędź mniejszego sektora nie musi pokrywać się z odcinkiem linii tworzącej krawędź większego sektora. Oznacza to, że przy określaniu przynależności pikseli do krawędzi, niektóre z nich mogą zostać pominięte tworząc prześwit.

Kolejnym sposobem na drodze ku rozwiązaniu tego problemu była propozycja, aby wstawiać dodatkowy wierzchołek również na krawędzi większego sektora. Wierzchołek taki znajdowałby się w tym samym miejscu, co wierzchołek dokładniejszego sektora i posiadał tę samą wysokość. Dodatkowy wierzchołek oznacza jednak konieczność podzielenia trójkąta, na którego krawędzi się on znajduje. Jeżeli na tej samej krawędzi sektora stykają się więcej niż dwa inne sektory, trzeba wstawiać odpowiednio więcej wierzchołków. Wspomniany trójkąt



Rysunek 5: T-połączenia powstające przy braku płynnych przejść między sektorami.

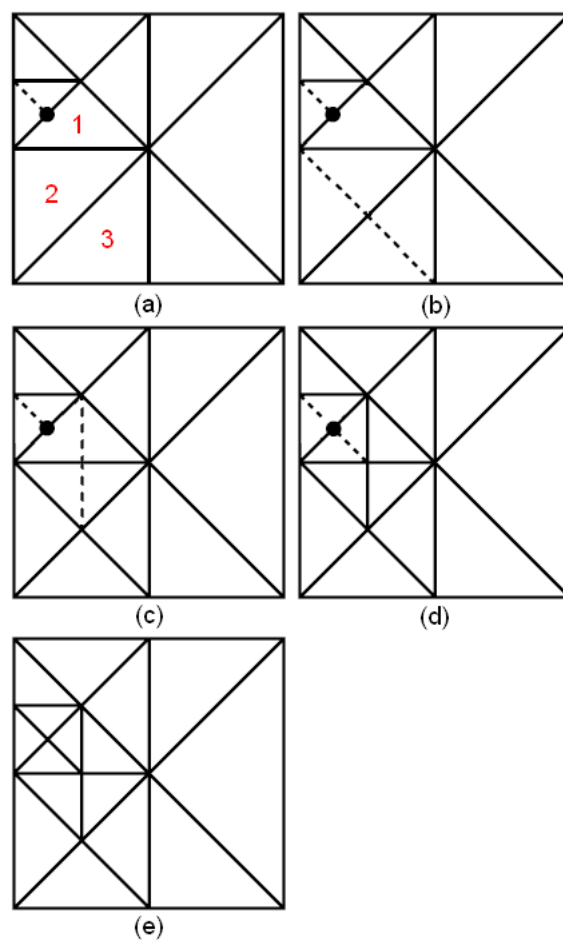
zostaje wtedy zamieniony przez wachlarz trójkątów.

Informacji o ukształtowaniu terenu jest zazwyczaj tak dużo że musi ona być na bieżąco doładowywana z dysku do pamięci. Podstawową jednostką dla takiej operacji są zazwyczaj sektory, na które teren jest podzielony. Bazując na tym założeniu powstała idea takiego tworzenia siatki, aby siatka każdego z sektorów była niezależna od siatek sektorów sąsiednich. Dzięki temu można by przetwarzać je równolegle. Aby tego dokonać zaproponowano, aby wszystkie sektory posiadały na krawędziach maksymalną liczbę wierzchołków. Oznacza to, że sektory nie są upraszczane na krawędziach a jedynie w środku. Takie podejście zapewnia ciągłość siatki, ale jej poziomowi nieregularności brak płynności i można w niej dojrzeć krawędzie podziału.

Ostatnie z rozwiązań okazało się idealne dla drzew binarnych. Łączy ono w całość cechy wspomnianych wcześniej rozwiązań wykorzystując przy tym specyfikę struktury danych. Gdy sektor ma zostać podzielony na dwa dokładniejsze, wiąże się to z identycznym podziałem sektora współdzielącego z nim przeciwprostokątną. Jeżeli sektor sąsiedni jest większy niż sektor, który ma zostać podzielony, najpierw dokonuje się rekurencyjnie podziału na sąsiedzie. Co prawda powstała w ten sposób siatka posiada dodatkowe trójkąty, jednak jest ona płynna i nie zawiera ani szczelin ani T-połączeń (rys. 6).

2.3.3 Algorytm ROAM

Algorytm ROAM (*Real-time Optimally Adapting Meshes*) został opracowany w 1997 roku w oparciu o mechanizm drzewa binarnego. Zyskał on sobie uz-



Rysunek 6: Podział siatki zachowujący jej spójność.

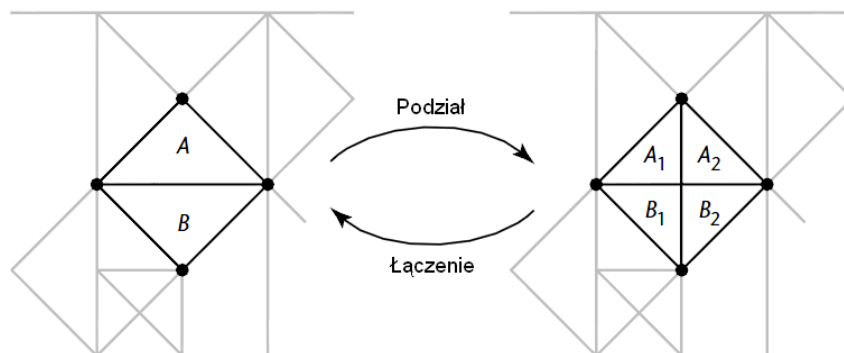
nanie w środowisku programistów gier komputerowych ze względu na możliwość tworzenia spójnej siatki terenu w czasie rzeczywistym. Co ważne siatka taka, mimo że posiada zmienną rozdzielczość, nie jest obciążona problemem szczelin czy T-połączeń. Dzieje się tak dzięki wykorzystaniu wspomnianego wcześniej wymuszonego podziału rekurencyjnego.

Istotą algorytmu jest reorganizacja siatki w oparciu o wagi jej sektorów. Każdy sektor jest oceniany na podstawie jego odległości od widza, rozmiaru w przestrzeni ekranu i poprawności tworzonego obrazu. Jeżeli waga sektora wynosi 0 oznacza to, że jest on niewidoczny lub znajduje się tak daleko od widza, że nie ma żadnego wpływu na finalny obraz. Jeżeli waga jest większa, oznacza to, że sektor jest bliżej obserwatora i ma większy wpływ na generowaną klatkę. Sektory są oceniane w znormalizowanej skali, co oznacza, że dla maksymalnego zbliżenia się do obserwatora waga jest najwyższa i równa 1.

Kolejnym istotnym elementem algorytmu są dwie kolejki opisujące siatkę. Pierwsza z nich zawiera w sobie opis wszystkich sektorów, ich wagi i informacje o sektorach sąsiednich. Druga kolejka zawiera informacje o tak zwanych "diamentach". Diamentem nazywa się czwórkę sektorów o tym samym poziomie podziału tworzących rąb. Krawędziami tego rombu są krawędzie przeciwprostokątne poszczególnych sektorów. Sektory te posiadają wspólny wierzchołek w centrum rombu, który jest wierzchołkiem przyprostokątnym dla każdego z nich.

Pojedynczy przebieg pętli reorganizującej siatkę terenu zaczyna się od ponownego wyznaczenia wag wszystkich sektorów. Następnie w pierwszej pętli dokonuje się uproszczenia sektorów o za niskiej wadze. Dokonuje się tego przechodząc po liście diamentów i sprawdzając czy maksymalna waga sektora w diamencie znajduje się poniżej progu łączenia. Jeżeli tak jest, sektory zostają połączone parami zamieniając diament w dwa sektory o mniejszym poziomie podziału (rys. 7).

Ostatnim krokiem jest przejście w pętli po wszystkich sektorach i dokonanie podziału tych z nich, których wagi są wyższe od wartości granicznej podziału. W obu przypadkach przejście po elementach kolejki dokonuje się w pętli do momentu usunięcia wszystkich sektorów z określonego przedziału wag. Gdy wagi



Rysunek 7: Przykład reorganizacji siatki w algorytmie ROAM.

wszystkich sektorów znajdują się między wartościami granicznymi podziału i łączenia siatka jest optymalna i gotowa do wyświetlenia.

Algorytm ROAM jest więc przykładem algorytmu hybrydowego łączącego w sobie cechy metod Top-Down i Bottom-Up. Obecnie jest to jeden z najczęściej stosowanych algorytmów do generacji siatki terenu.

2.4 Podsumowanie technik klasycznych

Istnieje wiele technik wizualizacji terenu. Wszystkie one bazują na założeniu, że teren można przedstawić w postaci pofałdowanej płaszczyzny. Techniki takie nie są jednak w stanie poprawnie wizualizować rzeczywistego terenu posiadającego horyzont. Nie są one również w stanie wizualizować ciał niebieskich, które bardziej zbliżone są do sfery aniżeli płaszczyzny. W pracy tej zostaną porównane metody wizualizacji terenu oparte na sferze jako alternatywa dla klasycznych metod wizualizujących.

3 Teoria sferycznych modeli opisu terenu

3.1 Wprowadzenie do sferycznych metod wizualizacji

Każdy obszar terenu przeznaczony do wizualizacji można przedstawić jako wycinek sfery o określonej szerokości i wysokości kątowej. Dla małych wartości obszar taki zachowuje się jak w przypadku klasycznych metod wizualizacji, gdzie teren opiera się na podstawie prostokątnej płaszczyzny. W przypadku większych wartości kątowych wzrasta stopień krzywizny terenu, przez co obszar taki zyskuje własność horyzontu bez potrzeby ingerencji w wizualizowane dane. Dla maksymalnych wartości kątowych otrzymuje się wizualizację w pełni sferycznego terenu właściwą dla ciał niebieskich takich jak planety czy księżyce. Ta własność sprawia, że sferyczne metody wizualizacji są uniwersalne, dzięki czemu posiadają one potencjalnie szerokie pole zastosowań.

3.2 Współrzędne sferyczne

Pozycje punktu w sferycznym układzie odniesienia opisuje się za pomocą trzech składowych tworzących koordynaty sferyczne (ρ, ϕ, θ) . Pierwsza składowa ρ oznacza odległość punktu od centrum sfery. Dla operacji na jednostkowej sferze jest ona domyślnie równa 1 i z tego powodu nie jest ona uwzględniana we wzorach. W dalszych etapach tworzenia wizualizacji, gdy określana jest finalna pozycja punktu, ρ można opisać wzorem 2, gdzie jest ono wyrażone jako suma promienia $R > 0$ i wzniesienia E ponad poziom 0.

$$\rho = R + E \tag{2}$$

Składowa ϕ określa kątowe odchylenie w płaszczyźnie XZ od dodatniej osi Z i może przyjmować wartości od 0° do 360° . Kierunek jej wzrostu jest przeciwny do ruchu wskazówek zegara (przy obserwacji z dodatniej strony osi Y). Dla przykładu, punkt o składowej ϕ równej 90° leży na dodatniej osi X w prawoskrętnym układzie odniesienia. Składowa ϕ w przypadku globu ziemskiego odpowiada długości geograficznej.

Ostatnia składowa θ określa kąt odchylenia od płaszczyzny XZ w stronę osi Y . Gdy punkt znajduje się bezpośrednio na północnym biegunie, jego współrzędna θ jest równa 90° . Podążając w dół, θ osiąga 0° na równiku by ostatecznie dojść do -90° na biegunie południowym. Składowa θ w przypadku globu ziemskiego odpowiada szerokości geograficznej.

Współrzędne sferyczne nie zostały ustandaryzowane, dlatego możliwe jest, że w innych dokumentach będą one opisane przy pomocy innych oznaczeń lub będą miały inne zakresy. Należy o tym pamiętać podczas implementacji wzorów.

3.3 Opis matematyczny

Architektura dzisiejszych kart graficznych nie pozwala na wyświetlanie powierzchni zadanych opisem matematycznym a jedynie zbiorów wielokątów. Najprostszym wielokątem jest trójkąt, dlatego też jest to podstawowy prymityw geometryczny przetwarzany w potoku renderującym. Aby więc wyświetlić model opisany wzorem matematycznym należy zamienić go na zbiór trójkątów przybliżających jego kształt. Czym więcej trójkątów zostanie przeznaczonych na ten cel i czym gęściej będą one ułożone, tym lepiej będą one oddawać powierzchnię przedstawionego modelu.

Ponieważ klasyczne metody wizualizacji posiadają prostokątną podstawę, jej przybliżenie za pomocą zbioru trójkątów jest zawsze idealne. Problem powstaje w przypadku technik posiadających za podstawę idealną sferę. Opis matematyczny sprowadzający się do pozycji na scenie i promienia musi zostać zamieniony na zbiór trójkątów.

Nie istnieje metoda potrafiącą idealnie oddać sferyczność kuli za pomocą zbioru płaszczyzn. Powstały model zawsze będzie jedynie przybliżeniem do pewnego stopnia dokładności. Przy zwiększaniu stopnia powiększenia uwidoczni się nierówność powierzchni. Czym większe takie powiększenie będzie tym wyraźniej widoczne będzie odstępstwo bryły od zamierzonej powierzchni sfery.

Aby rozwiązać ten problem można by stworzyć na tyle szczegółowy model, aby artefakty wynikające z jego niedokładności były niezauważalne dla obserwa-

tora. Jeżeli jednak wizualizowany teren będzie znacznych rozmiarów, złożoność takiego modelu byłaby na tyle duża, że nie możliwe byłoby jego przetwarzanie w czasie rzeczywistym.

3.4 Wielościany foremne

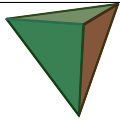
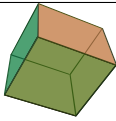
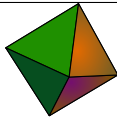
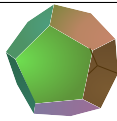
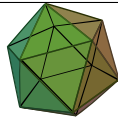
Skoro nie możliwe jest posiadanie sfery jako podstawy, a model wystarczająco dobrze ją przedstawiający byłby zbyt skomplikowany dla aplikacji czasu rzeczywistego, należy wybrać za podstawę model, który można uszczegóławiać rekurencyjnie. W ten sposób bez względu na stopień powiększenia model będzie zachowywał zadany stopień dokładności a co za tym idzie prawidłowy poziom wizualny.

Pozostaje więc wybrać bryłę która nadaje się do rekurencyjnego podpodziału. Ze względu na specyfikę zagadnienia bryła taka musi posiadać możliwość identycznego uszczegóławiania się w dowolnym jej punkcie. Tak jak powierzchnia sfery jest wszędzie jednakowa tak samo powierzchnia bryły bazowej powinna w każdym jej punkcie dzielić się w ten sam sposób, dążąc do oczekiwanego kształtu.

Jako rozwiązanie, samoistnie nasuwa się propozycja użycia jednego z wielościanów foremnych, jako bryły pełniącej podstawę wizualizacji sferycznej. Wszystkie one składają się z jednakowych ścian równomiernie rozłożonych w przestrzeni.

Istnieje pięć różnych wielościanów foremnych. Czworoscian, ośmiościan i dwudziestościan foremny składają się z trójkątnych ścian o równych kątach. Także sześcian i dwunastościan foremny, po podzieleniu ich ścian w równomierny sposób, mogą zostać przedstawione w postaci zbioru trójkątów. Jak już wcześniej wspomniano, jest to bardzo ważne ze względu na fakt że karty graficzne operują wyłącznie na danych takiego typu.

To jaki wielościan foremny będzie podstawą sferycznej wizualizacji, ma duży wpływ na to, jakie algorytmy zostaną użyte do jej dalszej implementacji. Różnice w ilościach krawędzi, orientacji względem siebie ścian jak i sposobie ich podziału wpływają na dalszy sposób przetwarzania danych.

					
Nazwa	Tetrahedron (czworościan)	Hexahedron (sześcián)	Octahedron (ośmiościan)	Dodecahedron (dwunastościan)	Icosahedron (dwudziestościan)
Podstawa ściany	trójkąt równoboczny	kwadrat	trójkąt równoboczny	pięciokąt foremny	trójkąt równoboczny
Ściany	4	6	8	12	20
Wierzchołki	4	8	6	20	12
Krawędzie	6	12	12	30	30

Tablica 1: Porównanie cech wielościanów foremnych

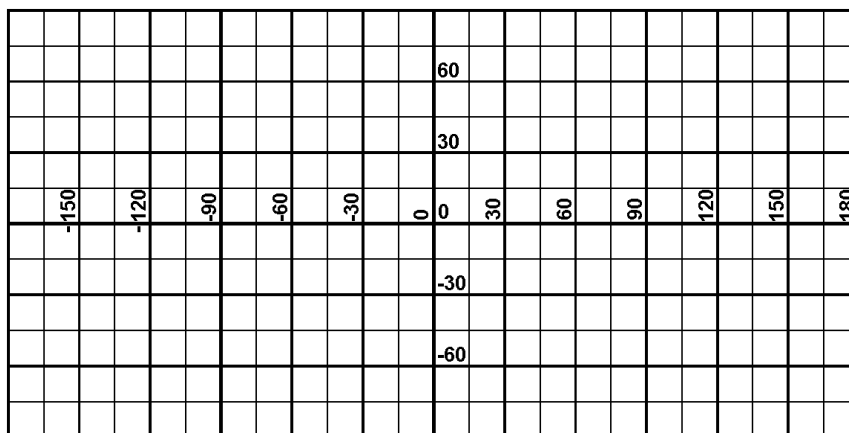
3.5 Podstawowe kryteria

Z założenia sferyczna wizualizacja terenu ma za zadanie wybrzuszyć lub uczynić wklęsłą powierzchnię idealnej sfery, przybliżanej rekurencyjnie za pomocą odpowiednich algorytmów, w taki sposób, aby uzyskać pofałdowanie siatki odpowiadające ukształtowi terenu. Informacje o wybrzuszeniu siatki w danym punkcie są zapisane w mapie wysokości. Mając ten fakt na uwadze należy tak projektować aplikację wizualizującą, aby mapa wysokości nakładała się w prosty sposób na powierzchnię sfery przy jednoczesnym minimalnym zużyciu zasobów pamięci.

Nie jest to proste, ponieważ nie istnieje technika rzutująca powierzchnię sfery na płaszczyznę bez zniekształceń. Powstało wiele metod odwzorowania, które w różnym stopniu radzą sobie z powstałymi zniekształceniami. Należy więc wybrać rozwiązanie, które ma zachowany właściwy stosunek pomiędzy stopniem zniekształcenia nakładanej mapy a nadmiarowością danych potrzebną do ich częściowego lub całkowitego zniwelowania.

3.6 Quad-Sfera

Biorąc pod uwagę powyższe kryteria stworzono podstawową sferyczną wizualizację terenu zwaną Quad-Sferą. Jej nazwa pochodzi od bazowego elementu budującego siatkę terenu, czyli czworoboku. Istotą tej techniki jest wykorzysta-



Rysunek 8: Projekcja Plate Carrée CAR.

tanie cech najpopularniejszej projekcji kuli do generacji jej siatki bazowej.

3.6.1 Projekcja CAR

Równoodległościowe odwzorowanie walcowate (CAR - *Plate Carrée Cylindrical Projection*) jest jedną z pierwszych metod kartograficznych. Zostało ono wynalezione przez Marinosa z Tyru około setnego roku naszej ery w celu sporządzenia mapy morza śródziemnego i Europy. Odwzorowanie to polega na zrzutowaniu kuli na powierzchnię boczną walca a następnie rozpostarcie jej na płaszczyźnie.

W wyniku projekcji równoleżniki zostają odwzorowane jako poziome równoległe do siebie linie, natomiast południki jako prostopadłe do nich linie pionowe. Walec styka się z kulą na jej równiku gdzie projekcja jest najdokładniejsza. Ponieważ odległości pomiędzy równoleżnikami jak i południkami są stałe, powstaje zniekształcenie odwzorowania, które rośnie wraz z odległością od równika i osiąga swoje maksimum na biegunach. Projekcja ta nie jest więc równo powierzchniowa. Oznacza to znaczną nadmiarowość danych przechowywanych w mapie, same zaś bieguny przedstawione są w postaci linii będących jej górną i dolną krawędzią.

Pomimo wymienionych wad projekcja ta stała się standardem w aplikacjach komputerowych z powodu prostoty przekształcenia pozycji punktu na mapie we współrzędne geograficzne na sferze. Pozycja w pionie bezpośrednio odpowiada szerokości geograficznej z przedziału $[-90,90]$ stopni, natomiast pozycja w poziomie długości geograficznej z przedziału $[-180,180]$ stopni. Standardowe równanie dla środka układu współrzędnych w środku mapy wygląda zatem następująco:

$$\phi = x \quad (3)$$

$$\theta = y \quad (4)$$

Równanie dla znormalizowanego układu współrzędnych z centrum w lewym dolnym rogu mapy przyjmuje zaś postać jak poniżej.

$$\phi = 360x - 180 \quad (5)$$

$$\theta = 180y - 90 \quad (6)$$

3.6.2 Siatka terenu

Prostota wizualizacji Quad Sfery polega na wykorzystaniu przez nią dwóch cech odwzorowania walcowego CAR. Pierwszą z nich jest łatwość, z jaką oblicza się pozycję na sferze dla zadanych koordynat na mapie. Posiadając wspomnianą pozycję punktu na mapie można w prosty sposób obliczyć jego współrzędne geograficzne przeskalowując wartości w osiach x i y do przedziałów $[-180,180]$ i $[-90,90]$. Następnie za pomocą otrzymanej szerokości i wysokości geograficznej wyznacza się pozycję punktu na powierzchni jednostkowej sfery. Drugą ważną cechą projekcji CAR jest fakt, że południki i równoleżniki dzielą powierzchnię mapy na równe kwadratowe obszary.

Najprostszym sposobem na stworzenie bazowej sfery jest więc przejście po kolejnych kwadratowych odcinkach mapy, dla każdego z nich wyznaczając pozycję

cje ich wierzchołków na sferze. Następnie za pomocą dwóch trójkątów rozpina się na tych wierzchołkach prostokątną powierzchnię w wyniku czego powinien powstać zamknięty model.

Minimalna ilość sektorów potrzebnych do stworzenia zamkniętej bryły równa jest ośmiu czworobokom. W takim przypadku dwa wierzchołki każdego z czworoboków posiadają tę samą pozycję w jednym z dwóch biegunów. Powstała w ten sposób bryła to ośmiościan foremny. Przyjmuje się więc że bazową bryłą dla techniki Quad Sfery jest zmodyfikowany ośmiościan foremny.

3.6.3 Zniekształcenia

Ceną jaką należy ponieść, w zamian za prostotę tworzenia siatki i nakładania na nią mapy wysokości, jest zniekształcenie terenu wzrastające wraz ze zbliżaniem się do biegunów, gdzie osiąga ono swoje maksimum. Ze względu na bezpośrednią zależność łączącą kształt siatki z projekcją mapy wysokości, kwadratowe sektory znajdujące się bliżej biegunów są coraz bardziej odkształcone. Odkształcenie to widoczne jest pod postacią zgniecenia równoleżnikowego aż do całkowitego spłaszczenia sektorów przy samych biegunach. Ponieważ jednak z tego samego powodu mapa terenu wraz ze zbliżaniem się do biegunów posiada coraz to większą nadmiarowość danych, oba te zniekształcenia znoszą się dając w rezultacie prawie całkowicie poprawny obraz.

Nie oznacza to jednak, że nadmiarowość danych i geometrii nie istnieje. Jest ona jedynie ukryta, przez ostateczny wynik wizualizacji, jakim jest obraz na ekranie. Obraz ten wydaje się prawie całkowicie poprawny poza samymi biegunami gdzie wciąż widoczne jest odkształcenie terenu. O ile więc nie jest to problemem z punktu widzenia użytkownika wizualizacji, o tyle staje się to ogromnym problemem z punktu widzenia samej aplikacji.

3.7 Cube-Sfera

Biorąc pod uwagę fakt występowania niedopuszczalnych zniekształceń siatki w technice Quad Sfery, pracowano nad znalezieniem techniki łączącej w sobie

prostotę projekcji mapy terenu z równomiernym rozłożeniem siatki. Propozycją takiego modelu wizualizacji terenu jest Cube Sfera.

3.7.1 Projekcje TAN i TSC

Jedną z metod odwzorowania powierzchni sfery na płaszczyźnie jest projekcja gnomiczna TAN. Jest to najstarsza znana metoda kartograficzna wynaleziona przez Talesa z Miletu około sześćsetnego roku przed naszą erą. W metodzie tej powierzchnia sfery jest rzutowana na styczną do niej płaszczyznę na zasadzie przedłużonego promienia. Oznacza to, że dla każdego punktu na sferze prowadzi się półprostą zaczynającą się w środku sfery i przechodzącą przez ten punkt. Miejsce w którym prosta przetnie się z płaszczyzną styczną jest punktem odwzorowania. W związku z tym, projekcja ta ma najmniejsze zniekształcenie w punkcie styczonym i wraz ze zwiększaniem się odległości od niego na płaszczyźnie, wzrasta stopniowo nadmiarowość przechowywanych w niej danych.

Za pomocą projekcji gnomicznej nie można więc zrzutować nawet połowy sfery. Z tego powodu do zrzutowania całej sfery potrzeba więcej niż dwóch, stycznych do niej płaszczyzn docelowych. Optymalnym rozwiązaniem jest sześciian opisany. Każda jego ściana służy jako płaszczyzna rzutowania dla projekcji gnomicznej (rys. 10). Takie połączenie projekcji gnomicznych w celu osiągnięcia pełnego obrazu powierzchni sfery zwane jest projekcją TSC (*Tangential spherical cube*).

Projekcje sześciennie

W projekcjach sześciennych takich jak TSC, powierzchnia sfery odwzorowywana jest równocześnie na sześciu płaszczyznach będących ścianami sześciianu. Aby więc wyprowadzić wzór na odwzorowanie punktu ze sfery (lub jego odwrotność, czyli nałożenie punktu na sferę z mapy) należy przyjąć jednolitą notację pozwalającą określić, której ściany w danym momencie dotyczą wspomniane obliczenia. Jeżeli rozłoży się siatkę sześciianu na płaszczyźnie w taki sposób jak jest to pokazane na rysunku 9, można dokonać jednoznacznego ponumerowa-

0				
1	2	3	4	
5				

Rysunek 9: Rozkład i numeracja ścian sześcienniej mapy na płaszczyźnie.

nia jej ścian. Dla przykładu, północny biegun będzie się znajdował w centrum ścianki o numerze 0, a punkt o współrzędnych 0°N , 90°E w centrum ścianki o numerze 2.

Gdy numeracja ścian jest znana, można przejść do wyznaczenia ściany, na której zostanie odwzorowany aktualnie przetwarzany punkt. Do tego celu potrzebne są zmienne l , m oraz n , które oblicza się podstawiając koordynaty kątowe wspomnianego punktu, do zestawu wzorów 7-9:

$$l = \cos \theta \cos \phi \quad (7)$$

$$m = \cos \theta \sin \phi \quad (8)$$

$$n = \sin \theta \quad (9)$$

W następnym kroku należy podstawić wyznaczone zmienne do tabeli 2. Aby określić, która ściana jest aktywna, sprawdza się, dla którego wiersza wartość zmiennej ζ jest maksymalna. Inaczej mówiąc, która z wartości n , l , m , $-l$, $-m$, $-n$ jest największa. Wskazany w ten sposób wiersz zawiera numer docelowej ściany, oraz koordynaty kątowe dla jej centrum. Na jego podstawie określa się również wartości pozostałych dwóch zmiennych ξ i η , które w wybranej projekcji zostaną podstawione do finalnych wzorów.

Odwzorowanie i rzutowanie danych

Aby odwzorować punkt w projekcji TSC wystarczy wyznaczyć z ogólnych równań docelową ścianę a następnie wartości χ i ψ według równań 10-11. Posiadając te informacje można obliczyć ostateczne położenie punktu używając wzorów 12-

Ściana	ξ	η	ζ	ϕ_c	θ_c
0	m	$-l$	n	0°	90°
1	m	n	l	0°	0°
2	$-l$	n	m	-270° lub 90°	0°
3	$-m$	n	$-l$	-180° lub 180°	0°
4	l	n	$-m$	-90° lub 270°	0°
5	m	l	$-n$	0°	-90°

Tablica 2: Relacje między współrzędnymi geograficznymi a numerem ściany.

13.

$$\chi = \frac{\xi}{\zeta} \quad (10)$$

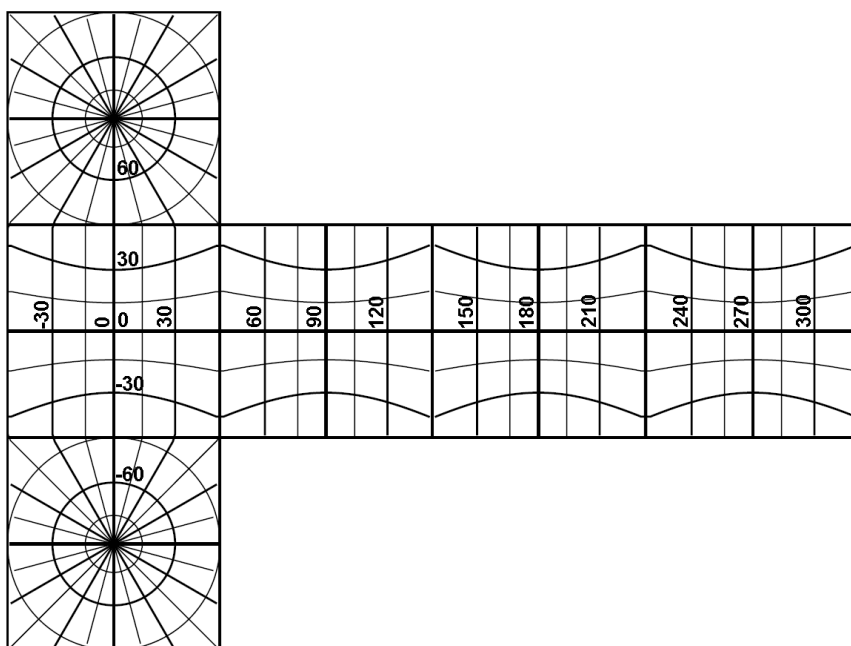
$$\psi = \frac{\eta}{\zeta} \quad (11)$$

$$x = \phi_c + 45^\circ \chi \quad (12)$$

$$y = \theta_c + 45^\circ \psi \quad (13)$$

Ze względu na prostotę przedstawionych obliczeń projekcja TSC zyskała duże uznanie w grafice komputerowej, gdzie równania te są rozwiązywane sprzętowo przez układ GPU. Przyjęto, że mapa sześcienna jest opisana na jednostkowej kuli, a wszystkie jej wierzchołki posiadają koordynaty z przedziału $[-1,1]$. Wystarczy wtedy podać jako parametr znormalizowany wektor położenia punktu na sferze, bez potrzeby jego zamiany na współrzędne kątowe. Układ karty graficznej dokona automatycznie obliczeń wyznaczając pozycję na mapie i odczytując z niej wartość. Mapy zapisane za pomocą projekcji TSC są potocznie nazywane Cube Mapami.

Aby móc korzystać ze wspomnianej mapy należy ją najpierw wygenerować. Do tego celu służą odwrotne równania do przedstawionych powyżej. W trakcie generacji mapy, dla każdego punktu na jej powierzchni trzeba wyznaczyć punkt



Rysunek 10: Projekcja TSC południków i równoleżników.

na sferze a następnie zapisać znajdującą się tam wartość.

$$\chi = \frac{x - \phi_c}{45^\circ} \quad (14)$$

$$\psi = \frac{y - \theta_c}{45^\circ} \quad (15)$$

W celu wyliczenia szerokości i wysokości geograficznej danego punktu na sferze, należy najpierw wyznaczyć wartości χ i ψ za pomocą odwróconych równań 12-13 tak jak to przedstawiają wzory 14-15. Posiadając te zmienne można wyliczyć wartości podstawowe ξ , η oraz ζ ze wzorów 16, 17 i 18.

$$\zeta = \frac{1}{\sqrt{1 + \chi^2 + \psi^2}} \quad (16)$$

$$\xi = \chi \zeta \quad (17)$$

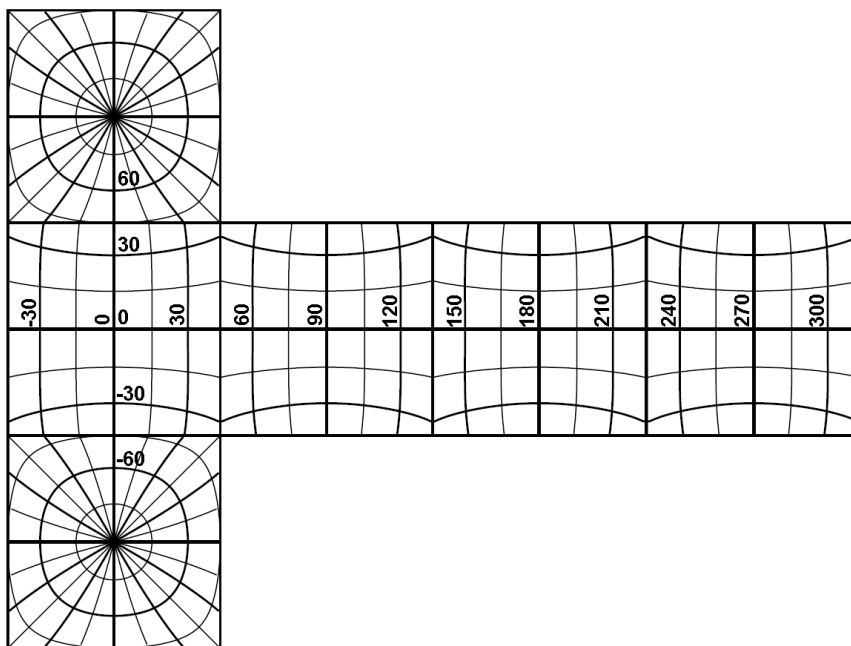
$$\eta = \psi \zeta \quad (18)$$

Ostatni etap odwrotnej projekcji jest identyczny dla wszystkich projekcji sześciennych i polega na wyznaczeniu wartości n , m i l na podstawie numeru aktualnej ściany i wartości ξ , η oraz ζ . Do tego celu wykorzystuje się wcześniej wspomnianą tabelę 2. Na zakończenie, odwracając równania 7-9 wyznacza się koordynaty kątowe punktu na sferze.

Niewątpliwą zaletą projekcji TSC jak i wszystkich projekcji opartych o sześćcian jest fakt odwzorowania sfery na kwadratowe ściany. Są one łatwe do przechowywania w pamięci komputera tak jak w przypadku projekcji CAR.

3.7.2 Projekcja CSC

Projekcja TSC nie jest jednak wolna od wad. W stosunku do projekcji CAR posiadającej dwa ekstremalne punkty odkształcenia, projekcja TSC jest lep-



Rysunek 11: Projektcja CSC.

sza, ponieważ rozkłada całkowite odkształcenie siatki pomiędzy osiem punktów wyznaczanych przez wierzchołki sześciennej mapy. Odkształcenie to jednak wciąż istnieje powodując nierównomierne obciążenie przy doładowywaniu mapy terenu.

Rozwiązaniem tego problemu byłoby znalezienie sześciennej projekcji równopowierzchniowej. Poszukiwania wzorów potrafiących opisać taką projekcję zostały rozpoczęte w 1974 roku na potrzeby programu COBE (*Cosmic Background Explorer*) zainicjowanego przez Departament Lotnictwa Marynarki Wojennej USA. Wynikiem przeprowadzonych badań była opracowana przez F.K. Chana i E.M. O'Neill'a projekcja CSC (*COBE Quadrilateralized Spherical Cube*) opisana w raporcie [Ch75] (rys. 11).

Projekcja ta działa poprzez zmodyfikowanie wzorów projekcji TSC w ten sposób, aby odkształcić rzutowaną powierzchnię sfery, rozciągając ją równomiernie na powierzchni sześcianu opisującego.

Odwzorowanie na sferę

Do momentu wyznaczeni wartości χ i ψ za pomocą równań 10-11 projekcja CSC działa tak samo jak projekcja TSC. Zmiana w przeprowadzanych obliczeniach następuje w momencie wyznaczania wartości x i y na ścianie mapy. We wzorach dla projekcji TSC wartości χ i ψ zostają zastąpione przez funkcję dwóch zmiennych, mającą za zadanie korygować równopowierzchniowo współrzędne na sześcianie:

$$x = \phi_c + 45^\circ F(\chi, \psi) \quad (19)$$

$$y = \theta_c + 45^\circ F(\psi, \chi) \quad (20)$$

Rozwinięcie funkcji $F(\alpha, \beta)$:

$$\begin{aligned} F(\chi, \psi) = & \chi\gamma^* + \chi^3(1 - \gamma^*) + \\ & \chi\psi^2(1 - \chi^2) \left[\Gamma + (M - \Gamma)\chi^2 + (1 - \psi^2) \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} C_{ij} \chi^{2i} \psi^{2j} \right] \\ & + \chi^3(1 - \chi^2) \left[\Omega_1 - (1 - \chi^2) \sum_{i=0}^{\infty} D_i \chi^{2i} \right] \end{aligned} \quad (21)$$

Do obliczenia wspomnianej funkcji niezbędne jest posiadanie zestawu stałych, które korygują przesunięcie kątowe (tabela 3). Wartości te zostały wyznaczone w trakcie badań nad projektem COBE.

Odwrotna projekcja do mapy

Dane terenu zazwyczaj są zapisane w wygodnej formie za pomocą projekcji CAR, co oznacza potrzebę ich przekonwertowania z mapy prostokątnej na sześcienną. W przypadku wybrania projekcji CSC jako docelowej, niezbędne wzory do przeprowadzenia odwrotnej projekcji są bardzo podobne do wzorów dla projekcji TSC. Mianowicie różnica występuje jedynie przy wyznaczaniu wartości χ

γ^*	=	1.37484847732
M	=	0.004869491981
Γ	=	-0.13161671474
Ω_1	=	-0.159596235474
C_{00}	=	0.141189631152
C_{10}	=	0.0809701286525
C_{01}	=	-0.281528535557
C_{20}	=	-0.178251207466
C_{11}	=	0.15384112876
C_{02}	=	0.106959469314
D_0	=	0.0759196200467
D_1	=	-0.0217762490699

Tablica 3: Stałe stosowane w projekcie COBE dla projekcji CSC.

oraz ψ .

$$\chi = f\left(\frac{x - \phi_c}{45^\circ}, \frac{y - \theta_c}{45^\circ}\right) \quad (22)$$

$$\psi = f\left(\frac{y - \theta_c}{45^\circ}, \frac{x - \phi_c}{45^\circ}\right) \quad (23)$$

Dla funkcji $f(\alpha, \beta)$ będącej istotą odwrotnej projekcji CSC powinno się przyjąć N równe 6. Zestaw stałych P niezbędnych do jej rozwiązania zawarto w tabeli 4.

$$f(X, Y) = X + X(1 - X^2) \sum_{j=0}^N \sum_{i=0}^{N-j} P_{ij} X^{2i} Y^{2j} \quad (24)$$

Posiadając wartości χ oraz ψ można już postępować identycznie jak w przypadku projekcji TSC używając wzorów 16, 17, 18 i tabeli 2 do wyznaczenia szerokości i wysokości geograficznej punktu na sferze. Istotnym jest fakt, że projekcja CSC wraz z odwrotną projekcją CSC nie pokrywają się idealnie, co udowadnia pewne niedokładności występujące przy ich użyciu.

P_{00}	=	-0.27292696	P_{04}	=	0.93412077
P_{10}	=	-0.07629969	P_{50}	=	0.25795794
P_{01}	=	-0.02819452	P_{41}	=	1.71547508
P_{20}	=	-0.22797056	P_{32}	=	0.98938102
P_{11}	=	-0.01471565	P_{23}	=	-0.93678576
P_{02}	=	0.27058160	P_{14}	=	-1.41601920
P_{30}	=	0.54852384	P_{05}	=	-0.63915306
P_{21}	=	0.48051509	P_{60}	=	0.02584375
P_{12}	=	-0.56800938	P_{51}	=	-0.53022337
P_{03}	=	-0.60441560	P_{42}	=	-0.83180469
P_{40}	=	-0.62930065	P_{33}	=	0.08693841
P_{31}	=	-1.74114454	P_{24}	=	0.33887446
P_{22}	=	0.30803317	P_{15}	=	0.52032238
P_{13}	=	1.50880086	P_{06}	=	0.14381585

Tablica 4: Stałe stosowane przy odwrotnej projekcji do mapy CSC.

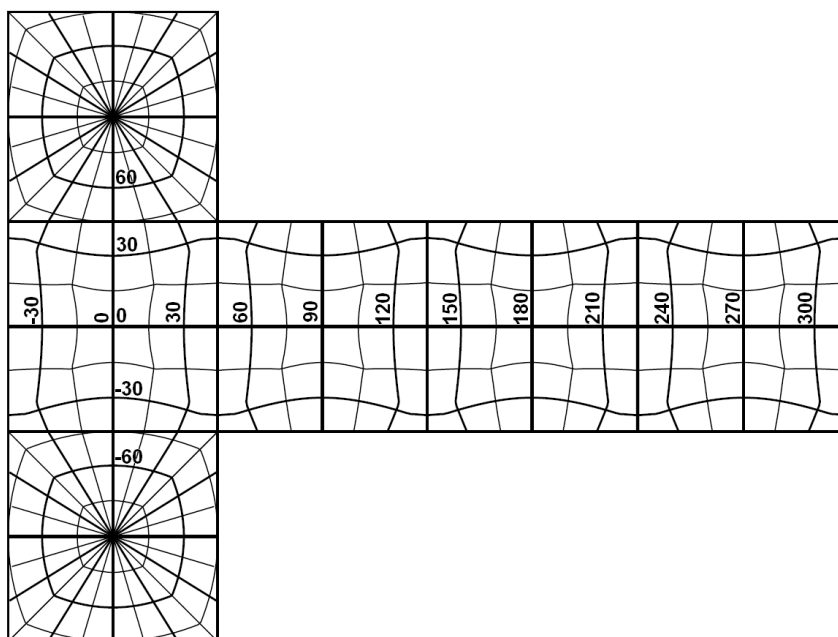
3.7.3 Projekcja QSC

Równopowierzchniowa projekcja CSC została opracowana metodą prób i błędów, co spowodowało, że wyznaczone dla niej wzory i stałe nie do końca prawidłowo wyznaczały współrzędne odwzorowania. Wykazali to E.M. O'Neill i Laubscher w 1976 roku publikując raport "Extended Feasibility Study of a Quadrilateralized Spherical Cube Earth Data Base" [Ne76] (rys. 12).

Poza dowodem na niedokładne działanie projekcji CSC w opublikowanym raporcie zostały zawarte wzory na projekcję QSC (Quadrilateralized Spherical Cube), która właściwie oddawała równopowierzchniowe odwzorowanie sfery na sześciennie.

Odwzorowanie na sferę

Dzięki wyznaczeniu ostatecznej, prawidłowej formuły dla projekcji sześcienniej, nieposiadającej nadmiarowości danych, wzory potrzebne do wykonania odwzorowania znacznie się uprościły w stosunku do wzorów dla projekcji CSC. W pierwszym kroku będącym jednakowym dla wszystkich projekcji sześciennych należy wyznaczyć numer ściany docelowej. W przypadku projekcji QSC należy jeszcze pobrać na jego podstawie wartości kątowe ϕ_c i θ_c jej centrum (z tabeli 2),



Rysunek 12: Projekcja QSC.

aby móc podstawić je do ostatecznego wzoru.

$$(x, y) = (\phi_c, \theta_c) + \begin{cases} (u, v) & \text{jeżeli } |\xi| > |\eta| \\ (v, u) & \text{w przeciwnym przypadku} \end{cases} \quad (25)$$

Wartości u i v są wyznaczane na podstawie poniższych:

$$u = 45^\circ S + \sqrt{\frac{1 - \zeta}{1 - \frac{1}{\sqrt{2 + \omega^2}}}} \quad (26)$$

$$v = \frac{u}{15^\circ} \left[\operatorname{tg}^{-1}(\omega) - \sin^{-1} \left(\frac{\omega}{\sqrt{2(1 + \omega^2)}} \right) \right] \quad (27)$$

$$\omega = \begin{cases} \eta/\xi & \text{jeżeli } |\xi| > |\eta| \\ \xi/\eta & \text{w przeciwnym przypadku} \end{cases} \quad (28)$$

$$S = \begin{cases} +1 & \text{jeżeli } \xi > |\eta| \text{ lub } \eta > |\xi| \\ -1 & \text{w przeciwnym przypadku} \end{cases} \quad (29)$$

Odwrotna projekcja do mapy

Również projekcja odwrotna jest wyznaczona w postaci dokładnych równań bez potrzeby posiadania stałych korygujących. W łatwy sposób można wyznaczyć wartości pośrednie u i v dzięki równaniu 30. Na ich podstawie, korzystając z równań od 31 do 36 wyznacza się standardowy zestaw ξ, η, ζ pozwalający z kolei obliczyć, identycznie jak w poprzednich projekcjach sześciennych, koordynaty punktu na sferze.

$$(u, v) = \begin{cases} (x - \phi_c, y - \theta_c) & \text{jeżeli } |x - \phi_c| > |y - \theta_c| \\ (y - \theta_c, x - \phi_c) & \text{w przeciwnym przypadku} \end{cases} \quad (30)$$

$$\zeta = 1 - \left(\frac{u}{45^\circ}\right)^2 \left(1 - \frac{1}{\sqrt{2 + \omega^2}}\right) \quad (31)$$

$$\omega = \frac{\sin(15^\circ \frac{v}{u})}{\cos(15^\circ \frac{v}{u}) - \frac{1}{\sqrt{2}}} \quad (32)$$

Jeżeli $|x - \phi_c| > |y - \theta_c|$ wtedy ξ i η przyjmują następujące wartości:

$$\xi = \sqrt{\frac{1 - \zeta^2}{1 + \omega^2}} \quad (33)$$

$$\eta = \xi\omega \quad (34)$$

w przeciwnym wypadku:

$$\eta = \sqrt{\frac{1 - \zeta^2}{1 + \omega^2}} \quad (35)$$

$$\xi = \eta\omega \quad (36)$$

Ze względu na złożoność obliczeniową zarówno projekcje CSC jak i QSC nigdy nie były używane w aplikacjach czasu rzeczywistego i nie posiadają zastosowania poza programami badawczymi NASA. Jednakże moc obliczeniowa sprzętu komputerowego klasy PC znajdującego się obecnie na rynku, pozwala przypuszczać, że możliwe jest już ich użycie w grach i aplikacjach wizualizacji terenu.

3.7.4 Siatka terenu

Przedstawione powyżej projekcje sześciennie stanowią mechanizm umożliwiający pobieranie informacji o ukształtowaniu terenu dla dowolnego punktu na sferze. Informacje te są niezbędne do prawidłowej generacji siatki. W przypadku wizualizacji opartych na Cube-Sferze najpopularniejszym sposobem jej generacji jest zmodyfikowany algorytm ROAM.

Klasyczny algorytm ROAM działa na płaszczyźnie. Trójkąty znajdujące się na jej krawędziach są uznawane za przypadki specjalne i podlegają nieco zmienionym zasadom postępowania niż wszystkie inne. Jak wiadomo uwzględnianie każdego takiego specjalnego przypadku komplikuje program i zwiększa zużywaną moc obliczeniową. W przypadku Cube-Sfery algorytm ROAM jest prostszy i nie posiada krawędzi. Ściany bazowego sześcianu stykają się ze sobą tworząc wzajemne sąsiedztwa. W ten sposób są one wszystkie zarządzane przez ROAM tak jakby były jedną spójną powierzchnią. Usuwa to problemy związane z przypadkami szczególnymi i ułatwia implementację.

Wyznaczenie geometrii bazowej sfery jest bardzo proste. Wystarczy dokonać normalizacji na wektorach przechowujących pozycje wierzchołków, wszystkich trójkątów opisywanych przez algorytm. Następnie można już przystąpić do wybrzuszania otrzymanej siatki na podstawie danych przechowywanych w mapie wysokości.

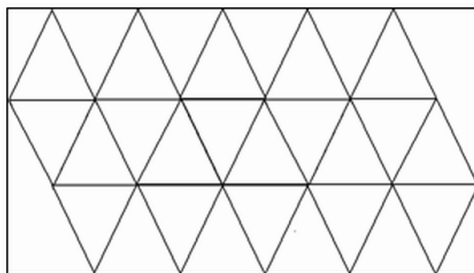
3.8 Ico-Sfera

Wielościanem foremnym najbardziej zbliżonym do sfery jest dwudziestościan foremny. Wykorzystuje się go jako podstawę dla geometrii terenu w modelach nazywanych Ico-Sferami. Stosowane przez nie projekcje i algorytmy są niestety bardzo złożone, co sprawia, że model ten prowadzi do najbardziej skomplikowanej wizualizacji sferycznego terenu spośród przedstawionych w tej pracy.

3.8.1 Projekcje

Najprostszą metodą projekcji powierzchni sfery na dwudziestościan jest zastosowanie wariantu projekcji gnomicznej. Dla każdego punktu na sferze prowadzi się półprostą zaczynającą się w jej centrum i przechodzącą przez ten punkt. Punkt w którym przetnie ona powierzchnię wpisanego w sferę dwudziestościanu jest punktem projekcji.

Należy tutaj zauważyć, że trójkąty tworzące bryłę dwudziestościanu mają swoje wierzchołki rozpięte na powierzchni sfery. Oznacza to, że każdemu takiemu



Rysunek 13: Projekcja powierzchni sfery za pośrednictwem dwudziestościanu.

trójkątowi odpowiada trójkąt sferyczny. Z powyższego wniosku od razu można zauważyć, że projekcja ta nie jest równopowierzchniowa, a gęstość próbek zmienia się od centrum trójkąta ku jego wierzchołkom.

Kolejne zniekształcenie powstaje w wyniku przechowywania otrzymanych danych na płaszczyźnie, tak jak to przedstawiono na rysunku 13. Dwudziestościan składa się z trójkątów równobocznych, co oznacza, że rozłożenie jego siatki na płaszczyznę nie sprawia problemów. Jednak dla kolejnych poziomów szczegółowości siatki nie jest to już prawdą.

Każda ścianka w wyniku podziału zostaje wybruszona, aby lepiej oddawać sferyczność kuli. Oznacza to, że trójkąty tworzące siatkę posiadają łączną większą powierzchnię od trójkąta bazowego. Z każdym kolejnym podziałem siatka zastępująca trójkąt bazowy coraz bardziej przypomina trójkąt sferyczny. Zmienia to dystrybucję błędu projekcji na bardziej równomierną, równocześnie jednak powstaje zniekształcenie związane z rozkładem siatki na mapie. Dzieje się tak, ponieważ z każdym poziomem uszczegółowienia siatki trójkąty zmieniają swój kształt, który odchodzi od trójkąta równobocznego na rzecz równoramiennego i dowolnego. Ich powierzchnia jest jednak wciąż rzutowana na równoboczny obszar tekstury.

Pomimo powstających zniekształceń projekcji, metoda ta jest używana w aplikacjach bazujących na Ico Sferze ze względu na małą złożoność obliczeniową i prosty mechanizm przechowywania danych.

W 1996 roku Max Tegmark przeprowadził badania na temat projekcji rów-

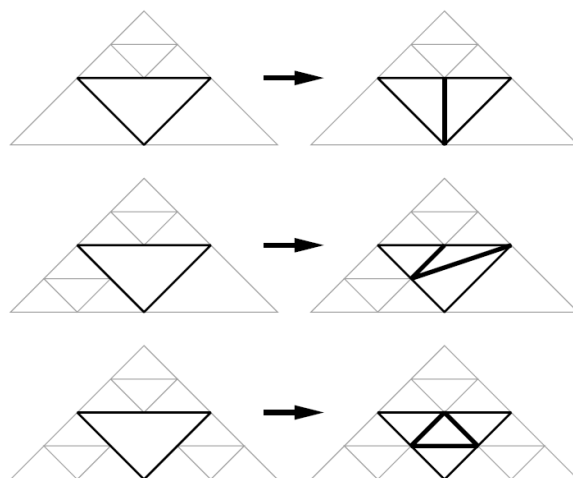
nopowierzchniowej. Miały one na celu ulepszenie mechanizmu akwizycji danych otoczenia gwiazdowego w projekcie COBE. Uzyskana w ten sposób projekcja jest bliska optymalnej. Występujący w niej błąd projekcji jest mniejszy od występującego w projekcji sześcienniej o 10% przy jednoczesnym zużyciu 20% mniej pamięci. Niestety projekcja ta nie nadaje się do aplikacji czasu rzeczywistego z powodu zaawansowanych równań, których używa oraz skomplikowanej struktury danych. Więcej szczegółów na jej temat można znaleźć w [Te96].

3.8.2 Siatka terenu

Dużą zaletą Ico Sfery jest wygenerowana za pomocą rekurencyjnego podziału siatka przybliżająca sferę bazową. Dzięki zastosowaniu dwudziestościanu jako bryły podstawowej, siatka ta osiąga bardzo szybko krzywiznę powierzchni o wysokiej jakości przy bardzo małej liczbie trójkątów.

Uszczegółowienie pojedynczego trójkąta siatki odbywa się poprzez jego podział na cztery mniejsze. Podział ten następuje według następujących kryteriów. W połowie każdej z trzech krawędzi trójkąta zostaje wstawiony dodatkowy wierzchołek. Następnie wszystkie trzy wierzchołki są łączone za pomocą krawędzi, tworząc cztery nowe trójkąty. Trójkąt otoczony przez nowopowstałe krawędzie jest trójkątem centralnym. Aby poprawnie przybliżyć powierzchnię sfery, dokonuje się wyrzuczenia podzielonego trójkąta stosując normalizację jego wierzchołków.

Generowana w ten sposób siatka terenu jest niestety niespójna. Co gorsza nie można w jej przypadku zastosować bezpośrednio algorytmu ROAM. Po skomplikowanej projekcji jest to drugi największy problem implementacji Ico Sfery. Rozwiązaniem tego problemu jest rekurencyjny podział sąsiednich trójkątów do momentu, gdy będą one tego samego rozmiaru co trójkąt, który ma zostać podzielony. Mechanizm ten działa podobnie jak w algorytmie ROAM z tą różnicą, że nie zachowuje on spójności siatki. Ma to również miejsce w przypadku drzew czwórkowych, których specjalnym przypadkiem jest siatka dwudziestościanu foremego. Gdy sąsiednie trójkąty są tego samego rozmiaru, dokonuje się po-



Rysunek 14: Przykłady "łat" zachowujących spójność siatki.

działu trójkąta. Aby zachować spójność siatki należy wtedy dokonać odświeżenia trójkątów sąsiednich, które muszą zostać zamienione specjalnymi łatami (rys. 14).

Tutaj zaczynają się wspomniane problemy. Każdy trójkąt siatki ma dualną naturę. W przypadku testu porównującego poziom podziału, dokonuje się go na trójkątach podstawowych, nie zaś na trójkątach je zastępujących i tworzących łaty. Poprawna implementacja takiego systemu musi uwzględniać tę naturę siatki. Oznacza to konieczność dynamicznej teselacji trójkątów bazowych na podstawie informacji o trójkątach sąsiednich w celu tworzenia łat usuwających szczeliny. W momencie uszczegółowienia trójkąta bazowego należy najpierw usunąć siatkę łaty a dopiero potem dokonać podziału. Wszystkie te czynniki powodują, że algorytm reorganizujący siatkę dwudziestościanu jest skomplikowany i potrzebuje więcej mocy obliczeniowej od poprzednich rozwiązań.

4 Środowisko implementacyjne

Grafika komputerowa jest bardzo szybko rozwijającą się dziedziną nauki. Nieustająca ewolucja kart graficznych wymaga ciągłego podążania za najnowszymi rozwiązaniami sprzętowymi i technikami programowania. Z tego względu aplikacja, w której zaimplementowano opisane powyżej metody wizualizacji terenu, została stworzona w oparciu o rozwiązania techniczne dostępne w czasie jej pisania.

4.1 Konfiguracja sprzętowa

Napisany przez autora program jest aplikacją typowo graficzną toteż największy nacisk podczas jej tworzenia został położony na maksymalne wykorzystanie możliwości karty graficznej. Po stronie CPU autor skupił swoją uwagę na poprawnym jej działaniu, nie zaś maksymalnej wydajności. Dzięki takim założeniom nie istnieją specyficzne wymagania dotyczące jednostki centralnej czy innych podzespołów poza samą kartą graficzną.

4.1.1 Wymagana generacja kart graficznych

Wraz z wprowadzeniem ósmej generacji kart graficznych przedstawiona została zunifikowana architektura jednostek cieniujących (USA - *Unified Shading Architecture*) umożliwiająca przetwarzanie programu cieniującego dowolnego typu na tym samym rdzeniu karty graficznej. Architektura ta umożliwiła również powstanie i sprzętową obsługę specyfikacji Shader Model 4.0. Dziewiąta i dziesiąta generacja kart graficznych nie przyniosły większych zmian w udostępnianych możliwościach jak i w wewnętrznej architekturze. Z tego też powodu implementacja programu została oparta na możliwościach specyfikacji Shader Model 4.0, a do jej uruchomienia wymagana jest minimum karta z rodziny NVidia GeForce 8000.

Aktualnie na rynek wkracza jedenasta generacja kart graficznych posiadająca sprzętową obsługę standardu Shader Model 5.0. Karty te posiadają również

nowe stopnie potoku renderującego jak programowalny teselator czy Compute Shader. Nowe możliwości pozwalają na jeszcze lepszą implementację opisanych wcześniej technik niż ma to miejsce we wspomnianej aplikacji. Należy jednak pamiętać, że w momencie rozpoczęcia nad nią prac, sprzęt ten jeszcze nie istniał.

4.2 Użyte biblioteki i oprogramowanie

4.2.1 Ngin 3.0

Opisywana aplikacja jest oparta na autorskim silniku Ngin w wersji 3.0. Silnik ten jest swego rodzaju zbiorem klas i funkcji tworzących zwarte i łatwe w obsłudze środowisko programistyczne. Jego główną zaletą jest niezależność programisty od systemu operacyjnego, architektury sprzętowej zarówno CPU jak i GPU czy bibliotek pośredniczących. Aby tego dokonać silnik oddziela właściwy kod aplikacji od środowiska wykonania za pomocą programowej warstwy abstrakcji. Warstwa ta składa się ze zbioru modułów zarówno dostępnych dla programisty jak i przed nim ukrytych i pracujących w tle. Poniżej zostały opisane najważniejsze z nich.

Kontekst renderujący

Zadaniem kontekstu renderującego jest utworzenie abstrakcyjnego modelu potoku renderującego. Programiście zostaje udostępniona warstwa logiczna przetwarzająca jego polecenia i przekazująca je dalej do konkretnego API graficznego. API to z kolei steruje kartą graficzną znajdującą się w komputerze.

Jako podstawowe API graficzne wybrana została biblioteka OpenGL w najnowszej wersji 3.2. Posiada ona pełne wsparcie dla Shader Model 4.0 i możliwości dorównujące DirectX 10.1 a nawet go przewyższające, jeżeli użyje się dodatkowych rozszerzeń, udostępnianych przez poszczególnych producentów kart graficznych. O wyborze API zdecydował także otwarty standard umożliwiający implementację OpenGL na innych platformach sprzętowych oraz uznanie w świecie naukowym.

Dzięki warstwie abstrakcji z jednej strony i mechanizmowi zależnych sprzętowo rozszerzeń specyfikacji z drugiej, kontekst renderujący jest w stanie wykorzystać posiadaną kartę graficzną w najlepszy sposób. Odciąża on również programistę od potrzeby panowania nad mnogością wersji biblioteki i historii ewolucji kolejnych rozszerzeń.

W celu poprawnego ładowania tekstur z różnych formatów graficznych silnik korzysta dodatkowo z biblioteki DevIL (wcześniej znanej jako OpenIL). Pozwala ona na łatwą dekompresję obrazów z takich formatów jak PNG, JPEG czy wiele innych. Na potrzeby tej aplikacji zostało jednak użyte jedynie dekodowanie plików PNG w formacie RGB oraz RGBA.

Konteksty urządzeń wejściowych

Obecnie istnieją dwa podstawowe konteksty urządzeń wejściowych. Kontekst obsługi klawiatury i myszy. Tak jak w przypadku kontekstu renderującego czy innych elementów opisywanego silnika, oddzielają one mechanizmy specyficzne dla konkretnego systemu operacyjnego od programisty. Pozostawiają mu one w ten sposób przejrzysty i prosty w obsłudze interfejs.

Funkcje pomocnicze

Funkcje pomocnicze tworzą zbiór klas i metod wspierających rozwój aplikacji. Główną ich funkcjonalnością jest wbudowana biblioteka GPCPU (*General Purpose Computations on CPU*). Udostępnia ona deklaracje typów wektorowych i macierzowych, tak jak ma to miejsce po stronie karty graficznej. Dołączony jest również zbiór metod i przeciążonych operatorów pozwalających operować na nich w dowolny sposób zgodnie z ich arytmetyką. W ten sposób bez większego problemu można przenosić całe funkcje i bloki kodu między programami w GLSL czy CG/HLSL wykonywanymi w karcie graficznej a programem głównym napisanym w C/C++.

Metody pomiaru czasu umożliwiają z kolei pomiar z dokładnością do mikrosekundy i są bardzo przydatne przy tworzeniu wszelkiego rodzaju benchmarków

czy dokonywaniu późniejszych optymalizacji. W funkcjach pomocniczych zawarto też takie metody jak logowanie do pliku dowolnych zdarzeń występujących w aplikacji, w celu późniejszego śledzenia przebiegu jej wykonania. Przydatne są też szablony kontenerów do przechowywania danych.

4.2.2 Środowisko IDE

Do stworzenia aplikacji użyte zostało zintegrowane środowisko programistyczne Microsoft Visual Studio 2008 Professional. Wbudowane kolorowanie składni zostało rozszerzone o kolorowanie typów macierzowych i wektorowych stosowanych w silniku graficznym oraz o kolorowanie składni GLSL w wersji 1.50. Możliwe to było dzięki zastosowaniu specjalnie do tego celu stworzonych pluginów. W wyniku modyfikacji możliwości IDE procesy tworzenia aplikacji oraz debugowania przebiegały sprawniej.

4.3 CubeResampler

CubeResampler to aplikacja pomocnicza napisana na potrzeby głównej aplikacji wizualizującej. Za zadanie ma ona przetworzyć mapę z projekcji CAR do wybranej projekcji TSC, CSC lub QSC. Dodatkowo aplikacja potrafi przetwarzać tekstury w jednym z trzech typów danych (R8G8B8A8, R8G8B8 lub R16). Wprowadzane dane powinny być zapisane w formacie RAW, to znaczy w postaci bloku bez żadnego nagłówka. Podany format danych obsługują wszystkie poważniejsze programy graficzne jak chociażby Adobe Photoshop czy darmowy IrfanView.

Program dokonuje przepróbkowania danych przy użyciu filtrowania NEAREST. Oznacza to, że dla każdego teksela wynikowej tekstury wyznaczana jest jego pozycja na sferze, a następnie na jej podstawie pozycja w teksturze wejściowej w projekcji CAR. Wyznaczona pozycja jest konwertowana do pozycji najbliższego teksela wejściowego. Realny brak filtrowania danych spowodowany jest programową implementacją jednostki próbkującej teksturę.

5 Implementacja

Na podstawie metod wizualizacji sferycznej, opisanych w poprzednich rozdziałach, autor stworzył aplikację, która ma na celu umożliwienie ich porównania. Porównanie to zostanie przeprowadzone na podstawie ich wydajności i jakości generowanego obrazu. Zaimplementowane zostały dwie spośród wymienionych technik, podstawowa Quad-Sfera i bardziej zaawansowana Cube-Sfera.

5.1 Struktura programu

Aplikacja ta jest traktowana jako program typowo badawczy, dlatego też, jej głównymi użytkownikami będą osoby zaznajomione z tematem sferycznej wizualizacji terenu. Do ich dyspozycji zostaje przekazany interfejs użytkownika posiadający zestaw opcji, za pomocą których można zmieniać wartości sterujące przebiegiem wykonania algorytmów wizualizacji. Umożliwia to przeprowadzenie badań wydajnościowych na podstawie ustalonego stanu zmiennych i wbudowanego testu.

Test polega na uruchomieniu animacji o stałym torze kamery i czasie wykonania. W trakcie całego czasu jego trwania powstaje plik zawierający zapis wszystkich parametrów aplikacji. Przykładem takich parametrów są między innymi czas potrzebny na wygenerowanie kolejnej klatki czy czas, jaki procesor zużywa w celu reorganizacji siatki. Powstałe w ten sposób zbiory danych mogą być w dowolny sposób analizowane w celu opracowania rezultatów.

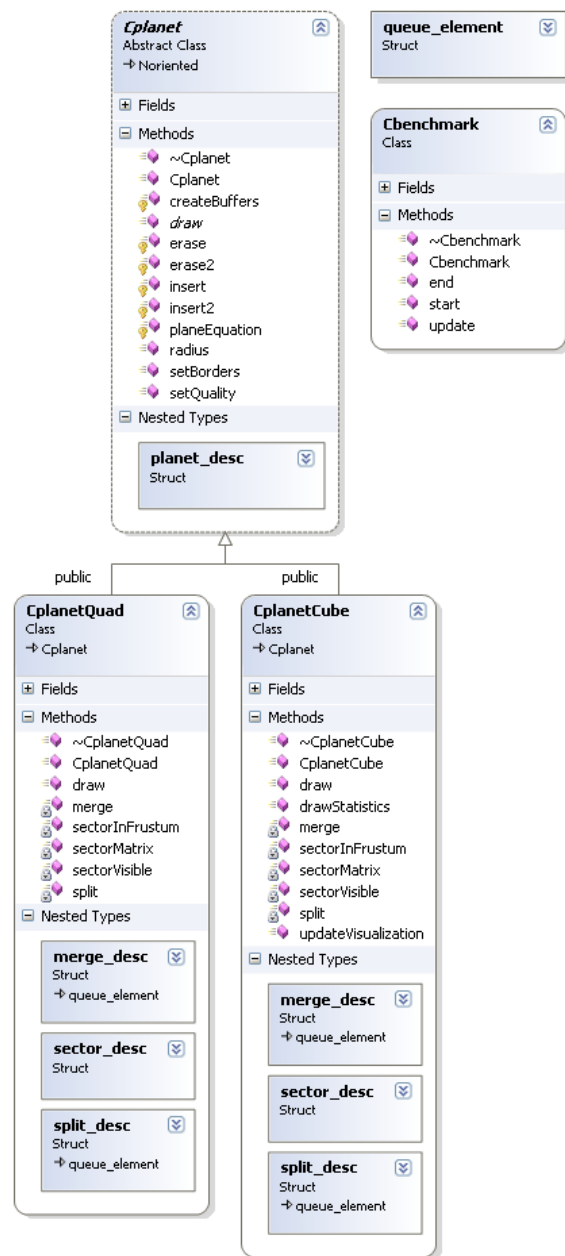
5.1.1 Diagram klas

Poniżej zostały opisane klasy tworzące wizualizację terenu wraz z ich metodami.

Klasa: Cplanet

Klasa ta jest podstawową klasą reprezentującą sferyczną wizualizację terenu. Posiada ona zestaw bazowych metod będących elementem wspólnym dla wszystkich rodzajów wizualizacji (rys. 15).

Klasa bazowa: Noriented



Rysunek 15: Diagram klas tworzących wizualizację.

Klasy potomne: CplanetQuad, CplanetCube

Metody:

```
Cplanet::~~Cplanet();
```

- Destruktor.

```
Cplanet::Cplanet();
```

- Konstruktor.

```
void Cplanet::createBuffers(uint8 lvl);
```

- Tworzy zbiór buforów geometrii reprezentujących pojedynczy sektor w różnych poziomach szczegółowości.

- Na wejściu otrzymuje ilość poziomów szczegółowości, dla których ma utworzyć bufor geometrii.

- W przypadku wartości spoza zakresu tworzy podstawowy bufor geometrii.

```
virtual void Cplanet::draw(uint32 mode, Ncam* view, Ncam* display) = 0;
```

- Definiuje funkcję odpowiedzialną za odświeżenie stanu wizualizacji i wyświetlenie geometrii na ekranie. Metoda ta musi zostać zdefiniowana przez wszystkie klasy dziedziczące.

- Na wejściu otrzymuje następujące parametry: tryb, w jakim ma być wyświetlony teren, wskaz na kamerę obserwatora, dla którego ma być odświeżony stan wizualizacji, wskaz na rzeczywistą kamerę, dla której ma być wyświetlona wizualizacja.

- W przypadku trybu poza zakresem wybierany jest tryb domyślny. W przypadku jednego pustego wskaźnika dla kamery przyjmuje się odświeżenie i rendering z pozycji drugiej dostępnej kamery. W przypadku obu pustych wskaźników metoda przerywa swoje wykonanie.

void Cplanet::erase(struct queue_element* elem);

- Usuwa wskazany element z dwukierunkowej listy sektorów.
- Jeżeli wskaz na element jest pusty metoda przerywa swoje wykonanie.

void Cplanet::erase2(struct queue_element* elem);

- Usuwa wskazany element z dwukierunkowej listy diamentów.
- Jeżeli wskaz na element jest pusty metoda przerywa swoje wykonanie.

bool Cplanet::insert(struct queue_element* pos, struct queue_element* elem);

- Wstawia element we wskazanym miejscu listy sektorów.
- Na wejściu otrzymuje wskaz na miejsce, w którym ma być wstawiony element i wskaz na wstawiany element.
- Na wyjściu zwracana jest prawda, jeżeli operacja się powiodła.
- Jeżeli wskaz na element jest pusty, metoda przerywa swoje wykonanie. Jeżeli wskaz na miejsce jest pusty, metoda dodaje element na początku listy.

bool Cplanet::insert2(struct queue_element* pos, struct queue_element* elem);

- Wstawia element we wskazanym miejscu listy diamentów.
- Na wejściu otrzymuje wskaz na miejsce, w którym ma być wstawiony element i wskaz na wstawiany element.
- Na wyjściu zwracana jest prawda, jeżeli operacja się powiodła.
- Jeżeli wskaz na element jest pusty, metoda przerywa swoje wykonanie. Jeżeli wskaz na miejsce jest pusty, metoda dodaje element na początku listy.

double4 Cplanet::planeEquation(double3* v0, double3* v1, double3* v2);

- Metoda oblicza współczynniki równania płaszczyzny, na której znajdują się wskazane punkty.
- Na wejściu metoda otrzymuje wskaźniki na trzy punkty.
- Na wyjściu metoda zwraca wektor współczynników równania płaszczyzny.
- Jeżeli punkty nie spełniają warunków równania płaszczyzny lub któryś z ich wskaźników jest pusty, zwracany wektor jest pusty.

float Cplanet::radius(void);

- Zwraca aktualny promień wizualizowanej sfery terenu.

bool Cplanet::setBorders(double merge, double split);

- Ustawia dolną granicę łączenia sektorów i górną granicę ich dzielenia dla kryterium reorganizacji siatki.
- Metoda zwraca prawdę, jeżeli została wykonana poprawnie.
- Jeżeli wartości nie zawierają się w zakresie (0..1) lub dolna wartość jest większa od górnej, metoda kończy swoje działanie.

bool Cplanet::setQuality(uint8 lvl);

- Metoda określa, z jaką aktualną jakością mają być przetwarzane sektory. Inaczej mówiąc jak szczegółowa siatka ma być nakładana na każdy sektor.
- Na wejściu metoda otrzymuje poziom szczegółowości.
- Metoda zwraca prawdę, jeżeli została wykonana poprawnie.
- Jeżeli przekazany poziom szczegółowości nie zawiera się w poprawnym zakresie, metoda przerywa swoje wykonanie.

Klasa: CplanetQuad

Klasa implementująca wizualizację opartą o ośmiościan foremny.

Klasa bazowa: Cplanet

Klasy potomne: brak

Metody:

CplanetQuad::~~CplanetQuad();

- Destruktor wizualizacji. Dokonuje zwolnienia wszystkich zasobów używanych przez wizualizację.

CplanetQuad::CplanetQuad(float radius);

- Konstruktor wizualizacji. Inicjuje wszystkie struktury danych dla wizualizacji i przypisuje jej określony promień.

- Na wejściu konstruktor otrzymuje promień podstawy wizualizacji sferycznej.

void CplanetQuad::draw(uint32 mode, Ncam* view, Ncam* display);

- Metoda dokonująca odświeżenia stanu wizualizacji względem pierwszej kamery i jej wyrenderowania względem drugiej kamery w zadanym trybie.

- Na wejściu otrzymuje następujące parametry: tryb, w jakim ma być wyświetlony teren, wskaz na kamerę obserwatora, dla którego ma być odświeżony stan wizualizacji, wskaz na rzeczywistą kamerę, dla której ma być wyświetlona wizualizacja.

- W przypadku trybu poza zakresem wybierany jest tryb domyślny. W przypadku jednego pustego wskaźnika dla kamery przyjmuje się odświeżenie i rendering z pozycji drugiej dostępnej kamery. W przypadku obu pustych wskaźników metoda przerywa swoje wykonanie.

bool CplanetQuad::merge(struct merge_desc* diamond);

- Dokonuje połączenia czterech sektorów tworzących diament w dwa większe.
- Na wejściu otrzymuje wskaz na diament, który ma zostać uproszczony.
- Zwraca prawdę, jeżeli łączenie sektorów się powiodło.
- Jeżeli podany wskaz jest pusty przerywa swoje wykonanie.

bool CplanetQuad::sectorInFrustum(struct split_desc* sec, double4* planes, double3 *vertices);

- Sprawdza, czy wskazany sektor znajduje się we frustumie.
- Na wejściu otrzymuje wskaz na badany sektor, wskaz na tablicę z parametrami równań płaszczyzn frustumu oraz wskaz na tablicę wierzchołków frustumu.
- Metoda zwraca prawdę, jeżeli sektor, choć częściowo znajduje się wewnątrz bryły frustumu.
- Jeżeli którykolwiek ze wskazów jest pusty, metoda zwraca fałsz.

float4x4 CplanetQuad::sectorMatrix(struct split_desc* sec, double3 corr);

- Metoda generuje spakowaną macierz przenoszącą geometrię sektora z przestrzeni sektora do przestrzeni jednostkowej sfery z uwzględnieniem korekcji precyzji.
- Na wejściu metoda otrzymuje wskaz na przetwarzany sektor oraz wektor korekcji precyzji.
- Na wyjściu metoda zwraca spakowaną macierz transformacji.
- Jeżeli wskaz na sektor jest pusty, metoda zwraca macierz jednostkową.

```
bool CplanetQuad::sectorVisible(struct split_desc* sec, double3 view);
```

- Metoda sprawdza czy dany sektor jest skierowany w stronę widza.
- Na wejściu metoda otrzymuje wskaz na sektor oraz wektor patrzenia obserwatora.
- Metoda zwraca prawdę, jeżeli sektor jest widoczny.
- Jeżeli wskaz na sektor jest pusty, metoda zwraca fałsz.

```
void CplanetQuad::split(struct split_desc* sec);
```

- Metoda dokonuje podziału wskazanego sektora na dwa mniejsze. Jeżeli zachodzi taka potrzeba, wcześniej dokonywany jest rekurencyjny podział sektorów sąsiednich.
- Na wejściu metoda otrzymuje wskaz na sektor do podziału.
- Jeżeli wskaz jest pusty, metoda przerywa swoje wykonanie.

Klasa: CplanetCube

Klasa implementująca wizualizację sferyczną w oparciu o sześcian.

Klasa bazowa: Cplanet

Klasy potomne: brak

Metody:

```
CplanetCube::~CplanetCube();
```

- Destruktor wizualizacji. Dokonuje zwolnienia wszystkich zasobów używanych przez wizualizację.

```
CplanetCube::CplanetCube(float radius);
```

- Konstruktor wizualizacji. Inicjuje wszystkie struktury danych dla wizualizacji i przypisuje jej określony promień.
- Na wejściu konstruktor otrzymuje promień podstawy wizualizacji sferycznej.

```
void CplanetCube::draw(uint32 mode, Ncam* view, Ncam* display);
```

- Metoda dokonująca odświeżenia stanu wizualizacji względem pierwszej kamery i jej wyrenderowania względem drugiej kamery w zadanym trybie.
- Na wejściu otrzymuje następujące parametry: tryb, w jakim ma być wyświetlony teren, wskaz na kamerę obserwatora, dla którego ma być odświeżony stan wizualizacji, wskaz na rzeczywistą kamerę, dla której ma być wyświetlona wizualizacja.
- W przypadku trybu poza zakresem wybierany jest tryb domyślny. W przypadku jednego pustego wskaźnika dla kamery przyjmuje się odświeżenie i rendering z pozycji drugiej dostępnej kamery. W przypadku obu pustych wskaźników metoda przerywa swoje wykonanie.


```
void CplanetCube::drawStatistics(double3 position, Ncam* view, Ncam* display);
```

- Metoda dokonuje wyświetlenia sferycznej statystyki stanu wizualizacji dla celów pomiarowych. Statystyki zostają wyświetlone we wskazanym miejscu biorąc pod uwagę kamerę widza, dla której reorganizowana jest siatka, i kamerę obserwatora, dla którego siatka ta jest wyświetlana.
- Na wejściu metoda otrzymuje pozycję, w której mają być wyświetlone statystyki oraz wskazy na dwie kamery bazowej i obserwatora.
- W przypadku jednego pustego wskaźnika dla kamery przyjmuje się odświeżenie i rendering z pozycji drugiej dostępnej kamery. W przypadku obu pustych wskaźników metoda przerywa swoje wykonanie.

bool CplanetCube::merge(struct merge_desc* diamond);

- Dokonuje połączenia czterech sektorów tworzących diament w dwa większe.
- Na wejściu otrzymuje wskaz na diament, który ma zostać uproszczony.
- Zwraca prawdę, jeżeli łączenie sektorów się powiodło.
- Jeżeli podany wskaz jest pusty, przerywa swoje wykonanie.

bool CplanetCube::sectorInFrustum(struct split_desc* sec, double4* planes, double3 *vertices);

- Sprawdza, czy wskazany sektor znajduje się we frustumie.
- Na wejściu otrzymuje wskaz na badany sektor, wskaz na tablicę z parametrami równań płaszczyzn frustumu oraz wskaz na tablicę wierzchołków frustumu.
- Metoda zwraca prawdę, jeżeli sektor choć częściowo znajduje się wewnątrz bryły frustumu.
- Jeżeli którykolwiek ze wskazów jest pusty, metoda zwraca fałsz.

float4x4 CplanetCube::sectorMatrix(struct split_desc* sec, double3 corr);

- Metoda generuje spakowaną macierz przenoszącą geometrię sektora z przestrzeni sektora do przestrzeni jednostkowej sfery z uwzględnieniem korekcji precyzji.
- Na wejściu metoda otrzymuje wskaz na przetwarzany sektor oraz wektor korekcji precyzji.
- Na wyjściu metoda zwraca spakowaną macierz transformacji.
- Jeżeli wskaz na sektor jest pusty, metoda zwraca macierz jednostkową.

```
bool CplanetCube::sectorVisible(struct split_desc* sec, double3 view);
```

- Metoda sprawdza czy dany sektor jest skierowany w stronę widza.
- Na wejściu metoda otrzymuje wskaz na sektor oraz wektor patrzenia obserwatora.
- Metoda zwraca prawdę, jeżeli sektor jest widoczny.
- Jeżeli wskaz na sektor jest pusty, metoda zwraca fałsz.

```
void CplanetCube::split(struct split_desc* sec);
```

- Metoda dokonuje podziału wskazanego sektora na dwa mniejsze. Jeżeli zachodzi taka potrzeba, wcześniej dokonywany jest rekurencyjny podział sektorów sąsiednich.
- Na wejściu metoda otrzymuje wskaz na sektor do podziału.
- Jeżeli wskaz jest pusty, metoda przerywa swoje wykonanie.

```
void CplanetCube::updateVisualization(uint8 f1method, uint8 f2method);
```

- Metoda dokonuje odświeżenia stanu pomocniczej wizualizacji statystyk.
- Na wejściu metoda otrzymuje numer wartości, jaka ma być wizualizowana za pomocą wybrzuszenia sfery oraz numer wartości, która ma być wizualizowana za pomocą koloru sfery statystyk.
- Jeżeli któraś z wartości jest poza zakresem, przyjmuje się wartości domyślne.

Klasa: Cbenchamrk

Klasa zawiera implementację systemu, generującego pliki ze zrzutami stanu wizualizacji.

Klasa bazowa: brak

Klasy potomne: brak

Metody:

Cbenchamrk::~Cbenchmark(void);

- Destruktor.

Cbenchamrk::Cbenchmark(void);

- Ustawia swój stan na gotowy do rozpoczęcia.

void Cbenchamrk::end(void);

- Kończy zrzucanie danych pomiarowych do pliku i zamyka go.
- Nie posiada parametrów wejściowych.
- W przypadku braku rozpoczęcia pomiarów przerywa swoje wykonanie.

void Cbenchamrk::start(bool freelook);

- Tworzy nowy plik z zapisem parametrów wizualizacji i rozpoczyna ich zapis.
- Na wejściu otrzymuje informację, czy pomiar został wykonany według zapamiętanej ścieżki kamery, czy za pomocą ręcznego sterowania. Informacja ta jest zapisywana w tworzonym pliku.
- W przypadku braku zakończenia poprzednich pomiarów przerywa swoje wykonanie.

void Cbenchamrk::update(Cplanet* src);

- Oblicza czas, który upłynął od poprzedniego zapisu parametrów i zapisuje go wraz z nowopobranymi parametrami do otwartego pliku.
- Na wejściu otrzymuje wskaz na klasę opisującą wizualizowany teren.
- W przypadku braku rozpoczęcia pomiarów lub przekazaniu pustego wskaźnika przerywa swoje wykonanie.

5.2 Autorski algorytm HRTMR

W wyniku badań nad metodami sferycznej wizualizacji terenu powstał autorski algorytm nazwany HRTMR (*Hybryd Real-Time Mesh Refinement*). Podstawowym założeniem dla tego algorytmu, była możliwość tworzenia dynamicznie modyfikowalnej siatki terenu z wykorzystaniem możliwości przetwarzania równoległego dzisiejszych kart graficznych.

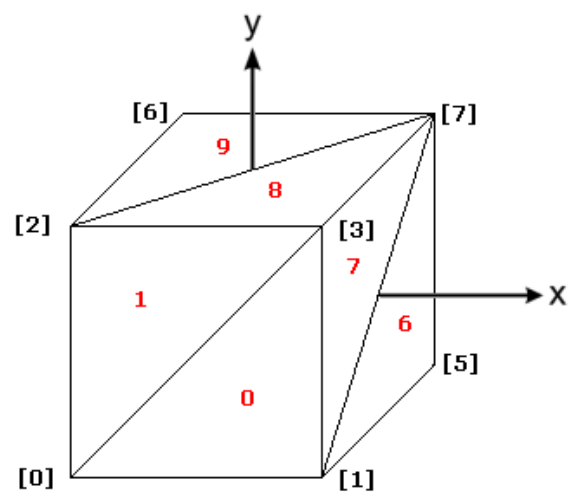
Elementem bazowym w zaproponowanym algorytmie jest sektor. Sektor można opisać jako wybrzuszony wycinek powierzchni sfery o podstawie w kształcie trójkąta sferycznego. Tak jak w klasycznych algorytmach mapa terenu jest opisywana za pomocą zbioru trójkątów, tak w algorytmie HRTMR ukształtowanie terenu opartego na sferze jest opisane za pomocą zbioru sektorów tworzących jej powierzchnię.

Ostatecznie proces wizualizacji terenu za pomocą zaproponowanego algorytmu można podzielić na dwie części. Pierwsza z nich polega na optymalnym podziale sfery na sektory. Gdy dobiegnie ona końca następuje druga polegająca na wybrzuszeniu sektorów widocznych przez obserwatora zgodnie z wartościami zapisanymi w mapie wysokości.

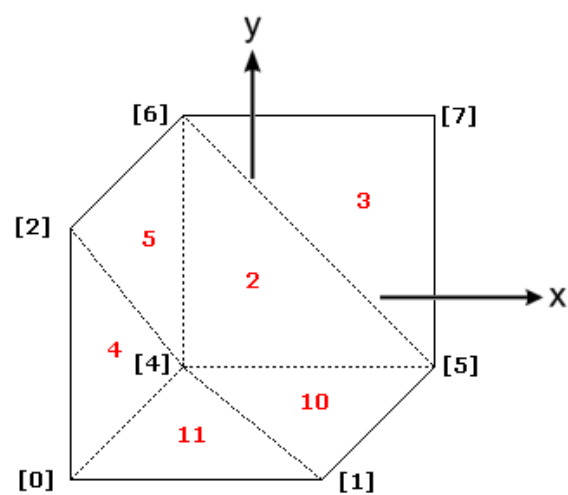
5.2.1 Przygotowanie sektorów

Podział powierzchni sfery na sektory odbywa się na CPU i jest podobny do klasycznych algorytmów typu ROAM. W przypadku Cube-Sfery podstawą podziału jest zestaw dwunastu sektorów dzielących równomiernie powierzchnię kuli. Są one rozpięte na ośmiu wierzchołkach wpisanego w nią sześcianu, tak jak jest to przedstawione na rysunkach 16 i 17 (dla ułatwienia, w dalszej części pracy sektory będą przedstawiane jako trójkąty rozpięte na ich wierzchołkach, należy jednak pamiętać, że posiadają one krzywiznę).

Dla implementacji Quad-Sfery przyjęto proste założenie, że sektory odpowiadają trójkątnym wycinkom walca zawierającego mapę wysokości w postaci



Rysunek 16: Numeracja ścianek i wierzchołków sześcianu bazowego.



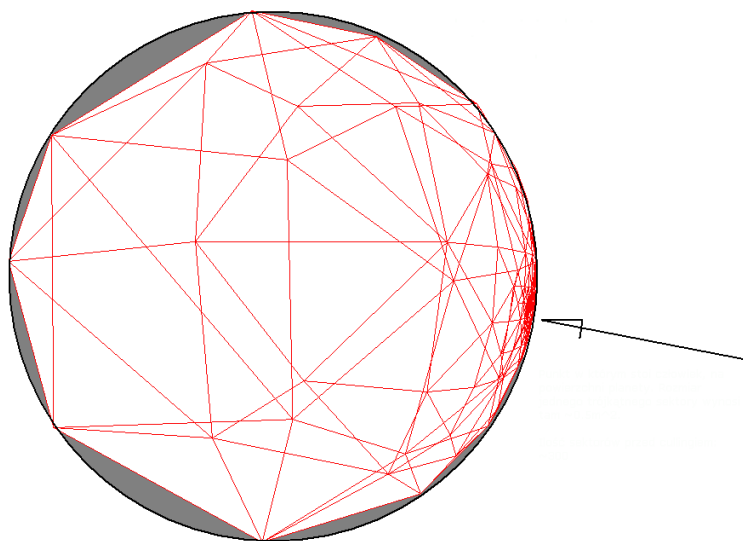
Rysunek 17: Numeracja tylnich ścianek sześcianu bazowego.

projekcji CAR. Wierzchołki tych sektorów są nanoszone na powierzchnię sfery przy użyciu tych samych mechanizmów, co dane mapy. Kolejne etapy algorytmu przebiegają identycznie dla obu metod.

Ponieważ w dalszej części wizualizacji sektory zostają poddane triangulacji, podział powierzchni sfery powinien być spójny, tak jak ma to miejsce w przypadku zwykłej siatki trójkątów. Z tego powodu sektory zorganizowane są za pomocą dwóch list, listy sektorów i listy diamentów.

Jak widać do tej pory sektory są traktowane analogicznie do zwyczajnych trójkątów. Występują tu jednak pewne różnice. W trakcie podziału lub łączenia sektorów nie jest wykorzystywana struktura drzewiasta znana z algorytmu ROAM. Nie jest ona potrzebna, ponieważ w celu odtworzenia kolejności łączy sektorów wykorzystuje się tutaj wektor bitowy. Wektor ten określa czy dany sektor przed podziałem był lewą czy prawą połową swojego rodzica, co jednoznacznie określa kierunek jego łączenia. Dodatkowo istnieją dwa znaczniki: znacznik narodzin i znacznik żywotności. Jeżeli w procesie łączenia znacznik żywotności wróci do wartości zero a aktualny poziom podziału będzie równy znacznikowi narodzin, oznacza to, że sektor ten został nowo utworzony przy tym podziale, który właśnie jest cofany i należy go usunąć. Taki mechanizm, dzięki posiadaniu jedynie liści klasycznego drzewa podziału, umożliwia wizualizowanie terenów w skali planetarnej.

Algorytm HRTMR nie posiada również wag sektorów obliczonych na etapie preprocessingu. Umożliwia to prostą modyfikację ukształtowania terenu w czasie rzeczywistym. Odświeżenie wizualizacji spowodowane jest zmianą pozycji obserwatora względem niej. Dokonywane wtedy łączenia i podziały sektorów odbywają się w oparciu o wagi obliczane dynamicznie na podstawie zadanego kryterium. Podobnie jak w algorytmie ROAM sektory o niskiej wadze błędu są upraszczane a sektory o wysokiej uszczegóławiane. Różnica polega na tym, że waga dla każdego sektora jest obliczana od nowa przy każdym odświeżeniu sceny. Kryterium dla określania wag może być różne. Najprostsze określa wagę jako stosunek odległości od widza i powierzchni sektora. Mogą jednak istnieć inne kryteria biorące pod uwagę krzywiznę horyzontu czy specyfikę terenu.



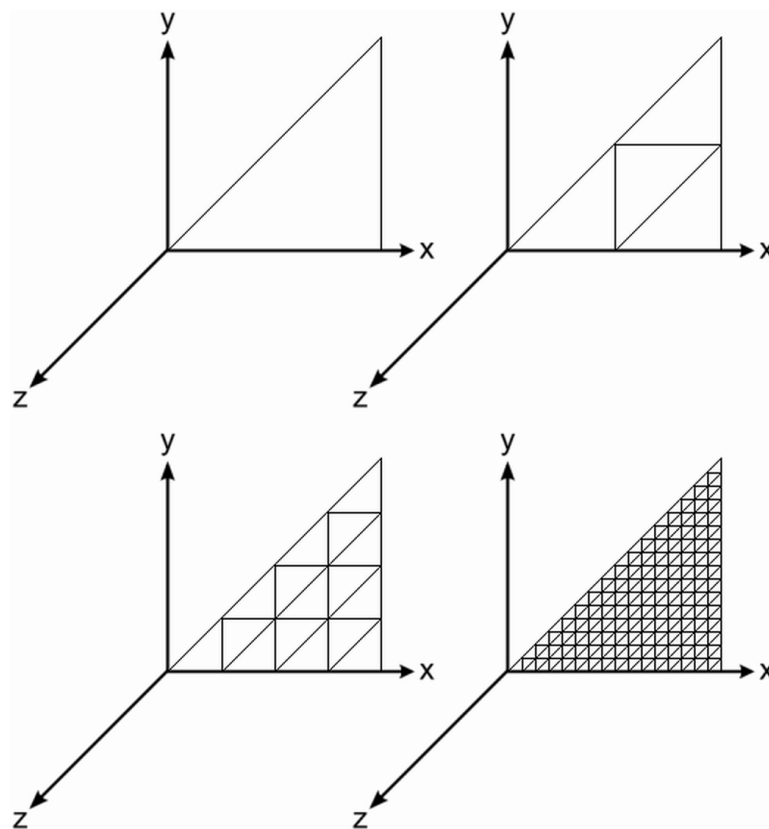
Rysunek 18: Podział sfery na sektory o różnej powierzchni (wektor przedstawia pozycję obserwatora).

Przykładowy efekt końcowy omówionego etapu został przedstawiony na rysunku 18.

5.2.2 Triangulacja i rendering

Druga część algorytmu HRTMR realizowana jest z wykorzystaniem możliwości karty graficznej po stronie GPU. Najpierw dokonuje się odrzucenia sektorów niewidocznych przez obserwatora za pomocą zmodyfikowanych testów backface cullingu i frustum cullingu (dokonują one testów widoczności z uwzględnieniem wybrzuszenia). Gdy zostaje określona lista widocznych sektorów, wywołuje się pojedynczy Draw Call odpowiedzialny za wygenerowanie i wyświetlenie siatki terenu. Draw Call jest to wywołanie graficznego API rozpoczynające rendering.

Draw Call korzysta z mechanizmu Instancingu, dzięki czemu jest w stanie przetworzyć wszystkie sektory, a co za tym idzie cały widoczny teren, za jednym razem. Oznacza to, że wszystkie wizualizowane sektory będą posiadały taką samą jakość określoną przez opisujący je bufor VBO. To właśnie dlatego w



Rysunek 19: Siatka pojedynczego sektora w różnych poziomach jakości (1,4,16,256 trójkątów).

pierwszym etapie sfera jest dzielona na sektory o różnej powierzchni.

Wspomniany wcześniej bufor VBO opisuje sektor za pomocą zbioru trójkątów stanowiących obszar, tak jak to przedstawiono na rysunku 19. Każdy wierzchołek jest w nim zapisany za pomocą pozycji w przestrzeni dwuwymiarowej XY oraz zestawu trzech wag określających wpływ trzech narożników sektora. Aby utworzyć ostateczną geometrię terenu, należy przenieść siatkę z przestrzeni sektora do przestrzeni obiektu (czyli terenu). W następnym kroku należy tę siatkę prawidłowo uszerzyć tworząc trójkąt sferyczny, aby na samym końcu możliwe było jej wyrzuczenie zgodnie z mapą wysokości.

Ponieważ każdy sektor leży w innym miejscu na powierzchni sfery, Vertex

Shader otrzymuje dodatkowo Uniform Buffer przechowujący tablicę struktur opisujących poszczególne sektory. Każda taka struktura przechowuje macierz przenoszącą narożniki sektora do przestrzeni jednostkowej sfery, na której są one rozpięte. Następnie przy użyciu trzech wag możliwe jest wyznaczenie finalnych pozycji również dla wewnętrznych wierzchołków. Jeżeli sektor opisuje bardzo mały wycinek kątowy sfery można zaniechać operacji zaokrąglania go i dokonać bezpośredniego przemnożenia wewnętrznych wierzchołków przez macierz sektora.

Gdy pozycje wierzchołków w przestrzeni obiektu są znane, próbuje się dla nich wysokość terenu z tekstury wysokości i dokonuje finalnego wybrzuszenia. Ostatnim krokiem jest ich przemnożenie przez macierz MVP w celu dokonania rasteryzacji.

Niewątpliwą zaletą algorytmu HRTMR jest możliwość płynnego przenoszenia obciążenia pomiędzy CPU a GPU nawet w trakcie działania aplikacji. Dla przykładu, aby zmniejszyć obciążenie CPU kosztem GPU wystarczy podnieść bariery podziału i łączenia sektorów jednocześnie wywołując Draw Call na buforze o gęstszej siatce. Aby zmniejszyć obciążenie GPU należy dokonać analogicznie odwrotnych operacji.

Opisany tutaj algorytm generuje siatkę wolną od szczelin i innych artefaktów zachowując przy tym płynność zmiany rozdzielczości. Zużywa on też mniej pamięci operacyjnej od klasycznego algorytmu ROAM i umożliwia przetwarzanie wielowątkowe po stronie CPU. Wszystkie te cechy sprawiają, że realne staje się przetwarzanie ogromnych obszarów terenu w skali planetarnej. Następne dwa rozdziały omawiają szczegółowo implementację przedstawionego algorytmu skupiając uwagę na zastosowanych dodatkowych korekcjach i optymalizacjach.

5.3 Odświeżenie stanu wizualizacji

Wraz ze zmianą położenia obserwatora względem terenu, należy ponownie obliczyć dane dla etapu renderingu. Zmiana taka może nastąpić, gdy obserwator zmieni swoją pozycję na scenie lub, gdy wizualizowany teren zmieni swoją orien-

tację względem obserwatora (np. rotacja planety względem własnej osi lub ruch całej planety po orbicie).

Proces odnowienia stanu wizualizacji, można podzielić na pięć etapów. Pierwszy z nich polega na przygotowaniu macierzy niezbędnej do przeprowadzenia testów widoczności, w przestrzeni jednostkowej sfery. Gdy macierz przenosząca kamerę do przestrzeni obiektu jest gotowa, następuje drugi i trzeci etap, które polegają na reorganizacji siatki terenu i dokonaniu testów widoczności. Ostatnie dwa etapy są odpowiedzialne za stworzenie finalnej macierzy MVP wizualizowanego terenu i indywidualnych macierzy poszczególnych sektorów. Rysunek 20 przedstawia kolejność podejmowanych kroków dla etapów 1, 3 i 5 tworzących macierze.

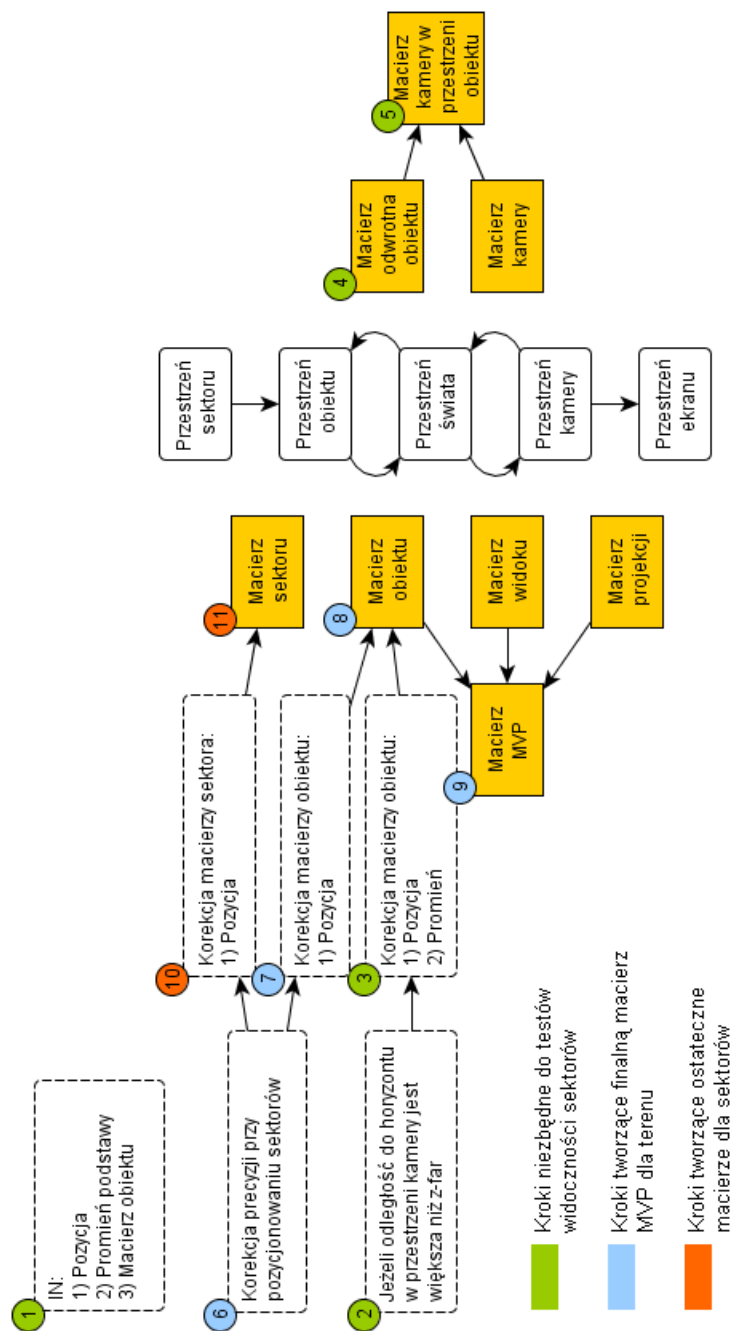
5.3.1 Korekcja odległości

W przypadku renderingu terenu istnieje potrzeba wyświetlenia w tej samej klatce dużego obszaru przy jednoczesnym zachowaniu prawidłowego przesłaniania się obiektów na scenie. Dla przykładu, teren o krzywiznie odpowiadającej krzywiznie ziemi jest widoczny na odległość 5km od obserwatora. Zwykle ustawienie wartości z-near i z-far dla takiej sytuacji skutkuje występowaniem błędów na ekranie z powodu niewystarczającej dokładności.

Aby zapobiec takiej sytuacji autor zastosował algorytm korekcji odległości. Umożliwia ona prawidłowe wyświetlanie widocznego terenu, znajdującego się w dowolnej odległości od widza. Idea algorytmu polega na założeniu, że najdalszym widocznym punktem terenu jest punkt na granicy jego horyzontu. Odległość horyzontu od obserwatora znajdującego się na wysokości h nad powierzchnią sfery o promieniu R przedstawia wzór 37.

$$H_{dis} = \sqrt{2Rh + h^2} \quad (37)$$

Przekształcając ten wzór oraz korzystając z twierdzenia Talesa, można wyznaczyć wzór na odległość horyzontu od obserwatora H_{dis}^z , wzdłuż wektora jego



Rysunek 20: Proces przygotowania macierzy (numeracja odpowiada kolejności wykonania).

wzroku \vec{L} (czyli osi Z w przestrzeni kamery). Wzór 38 korzysta z informacji o położeniu planety \vec{T}_p i obserwatora \vec{C}_p względem siebie, długości tego dystansu d oraz ogólnej odległości obserwatora od horyzontu planety H_{dis} . Po podzieleniu wartości z-far przez H_{dis}^z otrzymuje się finalny współczynnik korekcji odległości.

$$H_{dis}^z = H_{dis} * \cos\left(\arccos \frac{\vec{L} \cdot (\vec{C}_p - \vec{T}_p)}{d} - \arccos \frac{H_{dis}}{d}\right) \quad (38)$$

Aby obliczyć macierz przenoszącą kamerę do przestrzeni jednostkowej sfery (zwaną dalej macierzą frustumu), nie trzeba czekać do momentu dokonania wszystkich korekcji. Wystarczy stworzyć macierz obiektu uwzględniającą obliczony właśnie współczynnik korekcji odległości. Macierz ta posiada pozycję i skalę obiektu w przestrzeni świata, po jego pomniejszeniu i przybliżeniu (poprzez przemnożenie jego oryginalnej odległości i skali).

Macierz frustumu powstaje więc z pomnożenia macierzy odwrotnej do macierzy obiektu przez macierz kamery, przenoszącą ją do przestrzeni świata. Dzięki niej można przenieść wszystkie istotne informacje na temat kamery do przestrzeni jednostkowej sfery i w niej wykonać kolejne dwa etapy odświeżania stanu.

Listing 1: Korekcja horyzontalna odległości

```

1 // 0) Initial data for planet display
2 float4x4 planetMatrix = this->matrix();
3 float4x4 scalingMatrix;
4 double3 finalPosition = this->p;
5 float finalRadius = m_planet.r;
6
7 // 1) Calculates distance from observer position to planet
8 // horizon in look direction.
9 double3 toCamera = view->position() - this->position();
10 double distance = toCamera.length();
11 double h = distance - m_planet.r;
12 double horizon = sqrt(2.0*m_planet.r*h + h*h);
13 double zHorizon = horizon * cos( acos(
    dot(view->direction(), toCamera) / distance ) - acos(
    horizon / distance ) );
14
15 // 2) Based on distance information, factor of global distance
16 // scaling is calculated (that will be used if horizon is
17 // beyond z-buffer far clipping plane).
18 if ( abs(zHorizon) > abs(view->farClip()) &&
19     h > abs(view->farClip()) )

```

```

20     {
21         double correctionFactor = abs(view->farClip()) /
            abs(zHorizon);
22         finalPosition = this->p + (toCamera * (1.0 -
            correctionFactor));
23         finalRadius *= float(correctionFactor);
24     }
25     planetMatrix.column(3, float3(finalPosition));
26 }
27 scalingMatrix.m[0] = finalRadius;
28 scalingMatrix.m[5] = finalRadius;
29 scalingMatrix.m[10] = finalRadius;
30
31 // 3) Before second level of correction, there is calculated
32 //     matrice that will transfer frustum description (position,
33 //     corners, planes) into planet object space for Frustum
34 //     Culling.
35 float4x4 frustumMatrix =
    mul(mul(planetMatrix, scalingMatrix).invert(),
        view->modelMatrix());

```

5.3.2 Reorganizacja siatki

Reorganizacja siatki terenu odbywa się w przestrzeni jednostkowej sfery w celu zachowania maksymalnej precyzji. Jest to możliwe dzięki pracy na liczbach zmiennoprzecinkowych w zakresie [-1..1]. Ułatwia to również obliczenia pozycji sektorów na powierzchni sfery, które ograniczają się do operacji normalizacji. Proces ten, zaczyna się więc od przeniesienia pozycji obserwatora do przestrzeni jednostkowej sfery przy użyciu macierzy frustumu.

Gdy znana jest już pozycja obserwatora w przestrzeni obiektu, wywołuje się dwie pętle mające na celu skorygowanie siatki, w stosunku do nowej pozycji obserwatora. Pierwsza pętla iteruje po wszystkich "diamentach" w siatce terenu w celu odnalezienia tych, których priorytet jest za niski. Diamenty te są łączone tworząc sektory o mniejszej dokładności dzięki czemu uzyskuje się uproszczoną siatkę.

Listing 2: Uproszczenie siatki

```

1 // 6) Tessellation - Part in which geometry is reorganizd to
2 //                   get optimal visual quality with lowest
3 //                   available impact on performance.

```

```

4
5 // 6.2) Merging sectors that are redundant, e.g. sectors that
   are
6 //      too small in comparison with distance to the observer
7 //      which means they are too detailed or they are
   invisible.
8 changed = true;
9 m_cmerges = 0;
10 measure.start();
11 while(changed)
12 {
13     changed = false;
14
15     // Safe-check to break teselation if observer below
       ground level.
16     if (observerPos.length() <= 1.0f)
17         break;
18
19     // Merges to detailed sectors together to simplify the
       geometry
20     for(struct merge_desc* i =
       (CplanetCube::merge_desc*)(m_mergesH); i != NULL; )
21     {
22         // Diamond priority is max{s0,s1,s2,s3}
23         float maxp = -1.0f;
24         for(uint8 s=0; s<4; s++)
25             if (i->members[s]->priority > maxp)
26                 maxp = i->members[s]->priority;
27
28         // If sector is invisible it get's priority equal to 0.
29         if ( !sectorVisible(i->members[0],observerPos) &&
30             !sectorVisible(i->members[1],observerPos) &&
31             !sectorVisible(i->members[2],observerPos) &&
32             !sectorVisible(i->members[3],observerPos) )
33             maxp = 0.0f;
34
35         struct merge_desc* j =
           (CplanetCube::merge_desc*)(i->next);
36         if (maxp < m_merge_hi_border)
37             if (merge(i))
38             {
39                 changed = true;
40                 m_cmerges++;
41             }
42         i = j;
43     }
44 }
45 m_tmerges = measure.elapsed(EN_MICROSECONDS);

```

Pierwsza pętla kończy swoje działanie w momencie, gdy teren został wystarczająco uproszczony. Wówczas zostaje wykonana druga pętla dokonująca ponownej oceny geometrii. Każdy sektor jest oceniany biorąc pod uwagę nową pozycję obserwatora. Następnie sektory, które otrzymały wysokie wagi są au-

tomatycznie dzielone w celu uszczegółowienia siatki.

Listing 3: Ponowna ocena i uszczegółowienie sektorów

```

1  // 6.1) Splitting sectors that have too small quality (are too
2  //    big in comparison with distance to the observer.
3  changed = true;
4  m_csplits = 0;
5  measure.start();
6  while(changed)
7  {
8      changed = false;
9
10     // Safe-check to break teselation if observer below
11     //    ground level.
12     if (observerPos.length() <= 1.0f)
13         break;
14
15     // Because after split, new sectors are added at the end
16     //    of the queue
17     // it is required to only once travers it to get optimal
18     //    splits.
19     for(struct split_desc* i =
20         (CplanetCube::split_desc*)(m_splitsH); i != NULL; i =
21         (CplanetCube::split_desc*)(i->next))
22     {
23         #ifdef FLOAT_PRECISION
24         float3 center = float3((i->v0 + i->v1 + i->v2)/3.0f);
25         float side = length(i->v0 - i->v1);
26         float distance = length(observerPos - center);
27         #elif defined DOUBLE_PRECISION
28         double3 center = double3((i->v0 + i->v1 +
29             i->v2)/3.0);
30         double side = length(i->v0 - i->v1);
31         double distance = length(double3(observerPos) -
32             center);
33         #endif
34         i->priority = float(side/distance);
35         if ((side/distance) > m_split_low_border)
36         {
37             split(i);
38             changed = true;
39             m_csplits++;
40         }
41     }
42 }
43 m_tsplits = measure.elapsed(EN_MICROSECONDS);

```

Zgodnie z podstawowymi założeniami algorytmu ROAM, pętle kończą swoje działanie w momencie, gdy geometria sceny jest optymalna. Ponieważ ilość sektorów znacznie przekracza możliwości obliczeniowe karty graficznej, niezbędne jest wykonanie kolejnego etapu ograniczającego ich liczbę.

5.3.3 Test widoczności sektorów

Etap testu widoczności dokonuje hierarchicznego odrzucenia sektorów do momentu osiągnięcia niezbędnego minimalnego zbioru. W pierwszym kroku odrzucone zostają sektory skierowane do obserwatora tyłem. Należy tu zwrócić uwagę na konieczność testowania widoczności sektorów z uwzględnieniem ich późniejszego wyrzucenia.

Kolejnym krokiem jest odrzucenie sektorów, które nie znajdują się wewnątrz bryły frustumu obserwatora. Tu również uwzględnia się późniejsze wyrzucenie sektorów. W tym celu każdy test na sektorze, na prawdę wykonuje się na bryle prostopadłościanu. Jego podstawą jest sam sektor podczas, gdy wysokość równa jest maksymalnemu wzniesieniu w sektorze. Takie podejście zawyża nieco liczbę sektorów nieprawidłowo uznanych za widoczne, ale zapewnia również, że finalna klatka będzie kompletna.

Ostatni krok polega na odrzucaniu wszystkich sektorów powyżej określonego limitu, nawet, jeżeli są one widoczne. Ma to na celu zapobiegnięcie nadpisaniu bufora danych wysyłanego do karty graficznej. W przeciwnym wypadku mogłoby dojść do wycieku danych lub zamknięcia aplikacji i zresetowania sterownika graficznego.

5.3.4 Macierz sektora i korekcja precyzji

Korekcja precyzji jest niezbędna, ponieważ wierzchołki sektora po przeniesieniu do przestrzeni obiektu zostaną przeskalowane przez promień wizualizowanej planety. O ile posiadają one wystarczającą precyzję w przestrzeni jednostkowej, o tyle po przeskalowaniu przez promień, który zazwyczaj jest znacznych rozmiarów, ich położenie będzie odbiegać od oczekiwanych pozycji.

Rozwiązaniem tego problemu jest właśnie korekcja precyzji. Jeżeli wyznaczy się w przestrzeni obiektu wektor skierowany w stronę obserwatora, a następnie go znormalizuje, otrzyma się pozycję na jednostkowej sferze najbardziej zbliżoną do widza. Jeżeli przed przeskalowaniem pozycji wierzchołków, przeniesie się środek układu odniesienia do tego punktu, to wierzchołki znajdu-

jące się w bezpośrednim sąsiedztwie obserwatora prawie w ogóle nie zmieniają swojej pozycji. Wierzchołki znajdujące się dalej będą posiadały coraz większy błąd precyzji. Ostatecznie wierzchołki znajdujące się po przeciwnej stronie sfery będą posiadały dwa razy większy błąd precyzji niż w przypadku braku korekcji. Ponieważ jednak przeciwna strona sfery i tak nie będzie widoczna utrata precyzji nie ma tam znaczenia. Całkowity błąd precyzji nadal jest taki sam jednak jej jednostkowy podział jest zmieniony z równomiernego na wzrastający wraz z odległością od widza.

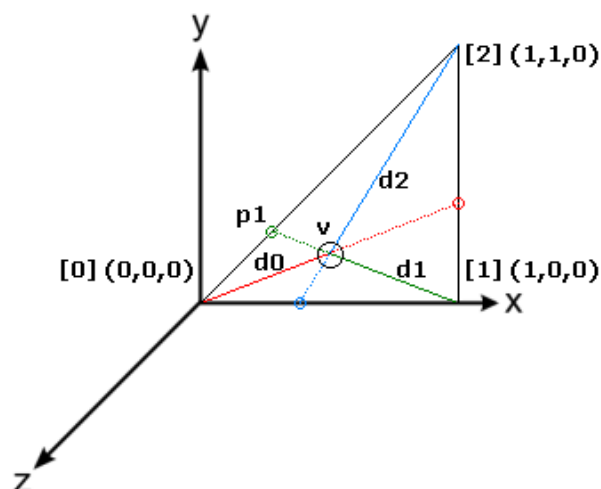
Przeniesienie środka układu odniesienia w stronę widza jest równoznaczne z odsunięciem od widza całej wizualizacji po tym wektorze o długość jej promienia. Dlatego też dopiero w tym momencie możliwe jest stworzenie finalnej macierzy MVP oraz macierzy przenoszących sektory do przestrzeni obiektu.

5.4 Rendering

Wszystkie dane niezbędne do wygenerowania obrazu na ekranie zostały przygotowane w etapie odświeżenia stanu. Teraz nadszedł czas na przetworzenie zbioru sektorów w siatkę terenu i poddanie jej rasteryzacji. Do karty graficznej jest ładowany program składający się z Vertex Shadera i Fragment Shadera. Vertex shader odpowiedzialny jest za wyznaczenie finalnych pozycji wierzchołków biorąc pod uwagę sferyczność sektorów i ukształtowanie terenu zapisane w mapie wysokości. Fragment Shader wykonuje standardowe operacje obliczające finalny kolor.

5.4.1 Vertex Shader

Vertex Shader korzysta z dwóch buforów UBO zawierających parametry. Pierwszy z nich przechowuje ogólne informacje o wizualizowanym terenie takie jak macierz MVP, promień, wektor korekcji precyzji i inne. Wszystkie sektory są wyrysowane przy użyciu jednego Draw Calla za pomocą mechanizmu Instancingu. Dlatego właśnie drugi bufor przechowuje tablicę struktur opisujących przetwarzane sektory. Wewnątrz każdej struktury znajduje się indywidu-



Rysunek 21: Interpolacja wektorów normalnych w celu usferzenia sektora.

alna macierz przenosząca wierzchołki sektora z jego przestrzeni do przestrzeni obiektu. Poza tym przechowywane są tam również wektory normalne oraz koordynaty tekstury dla każdego z narożników.

Bufor wejściowy VBO zawiera pozycję w przestrzeni XY oraz wagi określające wpływ trzech narożników sektora na przetwarzany wierzchołek siatki. Takich buforów może być więcej. W prezentowanej aplikacji istnieje ich osiem, każdy o innej gęstości siatki. Począwszy od bufora składającego się tylko z jednego trójkąta aż po bufor posiadający 16384 trójkątów. W momencie tworzenia tych buforów wyliczane są wcześniej wspomniane wagi według liniowej interpolacji. Rysunek 21 przedstawia ten proces.

Korzystając z wag, wyznacza się wektor normalny i koordynaty tekstury dla aktualnego wierzchołka. Jeżeli sektor, którego częścią jest wierzchołek, obejmuje dużą powierzchnię kątową najpierw dokonuje się jego usferzenia a dopiero potem wybrzuszenia.

Listing 4: Vertex Shader

```
1 #version 150
2 uniform samplerCube heightmap; // Height value
3 in vec2 mPosition; // Vertex position
```

```

4 in      vec3      mWeights; // Vertex weights
5
6 out     vec3      fNormal;  // Vertex normal and texcoord
7
8 // Whole planet specyfic data
9 // (shared across all sections of program)
10 layout(std140)
11 uniform Planet
12 {
13     mat4 matMVP;          // Model-View-Projection matrix for
                            planet
14     vec3 correction;      // Unit sphere center correction
15     uint level;           // Teselation swith off sector level
16     float r;              // Planet radius
17     float multiplier;     // Planet height multiplier
18 } planet;
19
20 // Sector specyfic data array
21 layout(std140)
22 uniform Sectors
23 {
24     vec4 v0;              // [v0.nx][v0.ny][v0.nz][ lvl ]
25     vec4 v1;              // [v1.nx][v1.ny][v1.nz][-----]
26     vec4 v2;              // [v2.nx][v2.ny][v2.nz][-----]
27     uint lvl;
28 } sectors[256];
29
30 void main(void)
31 {
32     // Corners positions, normals, texcoords
33     vec3 n0 = sectors[gl_InstanceID].v0.xyz;
34     vec3 n1 = sectors[gl_InstanceID].v1.xyz;
35     vec3 n2 = sectors[gl_InstanceID].v2.xyz;
36
37     // Sector level of subdivision
38     uint lvl = sectors[gl_InstanceID].lvl; //v0.z;
39
40     // Read weights for interpolation
41     float w0 = mWeights.x;
42     float w1 = mWeights.y;
43     float w2 = mWeights.z;
44
45     // Linear interpolate texcoords and normals
46     fNormal = normalize(w0*n0 + w1*n1 + w2*n2);
47
48     // Sampling terrain height in vertex position
49     float height = texture(heightmap, fNormal).x *
                    planet.multiplier;
50     vec3 elevation = fNormal.xyz * height;
51
52     vec4 base_position;
53
54     if (lvl < planet.level)
55         base_position = vec4(fNormal.xyz, 1.0);
56     else
57     {
58         mat4 sectorMatrix;

```

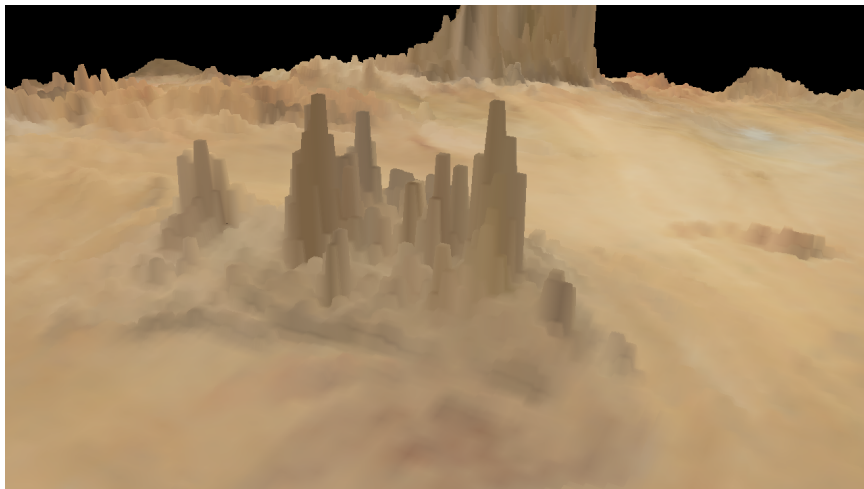
```

59
60     vec3 rb = vec3(n2 - n1);
61     vec3 rc = vec3(n1 - n0);
62
63     sectorMatrix[0] = vec4(rc, 0.0);
64     sectorMatrix[1] = vec4(rb, 0.0);
65     sectorMatrix[2] = vec4(cross(rb, rc), 0.0);
66     sectorMatrix[3] = vec4(n0, 1.0);
67
68     base_position = sectorMatrix * vec4(mPosition.x,
69         mPosition.y, 0.0, 1.0);
70 }
71 gl_Position = planet.matMVP * (base_position + vec4(elevation
72     - planet.correction, 0.0));
73 };

```

Spróbkowana mapa wysokości przechowuje wartości w postaci znormalizowanej do przedziału $[-1..1]$. Wartość ta jest przemnażana przez długość promienia w celu osiągnięcia finalnej wysokości. Przy czym wysokość ta jest określana od poziomu powierzchni sfery bazowej, co oznacza, że wartość zero jest równorzędna brakowi wybrzuszenia lub wgłębienia terenu. W przypadku wizualizacji małych terenów jak np. asteroid czy małych księżyców tekstura przechowująca mapę wysokości może być zapisana w formacie R16F. Oznacza to że każdy heksel (ang. *height element*) przechowuje swoją wartość w zmiennej typu half float. Wizualizacje większych obszarów, o rozmiarach planetarnych powinny korzystać z tekstur w formacie R32F o pełnej precyzji.

Możliwe jest dokonanie optymalizacji polegającej na przechowywaniu mapy wysokości z taką samą dokładnością, co w formacie R32F przy użyciu formatu R16UI lub nawet R16SI. Formaty te są stałoprzecinkowe i potrafią przechowywać taki sam zakres danych przy dwukrotnie mniejszym zużyciu pamięci. Karty graficzne nie potrafią jednak dokonywać interpolacji na typach stałoprzecinkowych podczas próbkowania. Należy, więc w takim przypadku być pewnym, że gęstość siatki nie będzie większa od gęstości nakładanej na nią mapy. W przeciwnym wypadku teren będzie posiadał "schodki" tak jak przedstawia to rysunek 22.



Rysunek 22: Brak interpolacji w formacie R16UI powodem "schodków".

5.4.2 Fragment Shader

Fragment Shader nie posiada żadnych elementów specyficznych dla wizualizacji terenu. Jego kod skupia się na obliczeniu równań oświetlenia i wyznaczeniu ostatecznego koloru pixela. W stworzonej aplikacji korzysta on z mapy diffuse jako źródła informacji o kolorze terenu, dzięki czemu ostateczny obraz jest prostszy w odbiorze.

Listing 5: Fragment Shader

```

1 #version 140
2 uniform samplerCube diffuse; // Diffuse color
3 uniform samplerCube heightmap; // Height value
4 in vec3 fNormal; // Normal vector
5
6 out vec4 outColor; // Final color
7
8 // Whole planet specific data
9 // (shared across all sections of program)
10 layout(std140)
11 uniform Planet
12 {
13     mat4 matMVP; // Model-View-Projection matrix for
14                 planet
15     vec3 correction; // Unit sphere center correction
16     uint level; // Teselation swith off sector level
17     float r; // Planet radius

```

```
17         float multiplier; // Planet height multiplier
18     } planet;
19
20 void main(void)
21 {
22     // Texture
23     outColor = texture(diffuse ,fNormal).xyzw;
24 };
```

6 Eksperymenty

W rozdziale tym zostanie przedstawiony przebieg eksperymentów mających na celu zbadanie wydajności zaimplementowanych metod. Środowisko, w jakim zostały przeprowadzone testy jest powtarzalne, co zapewnia wiarygodność pomiarów dla obu technik (Quad-Sfery i Cube-Sfery). Testowanie wydajności wizualizacji terenu odbyło się na bazie 80 sekundowej animacji w trakcie, której punkt położenia i orientacji obserwatora płynnie się przemieszczał względem wizualizowanego terenu. Środowisko testowe można regulować zmieniając poszczególne współczynniki sterujące wizualizacji. Są to: liczba trójkątów na sektor, dolna i górna granica reorganizacji siatki, granica usferzania sektora oraz mnożnik wybruszenia terenu. Możliwe jest również włączanie i wyłączanie kolejnych optymalizacji: backface cullingu, frustum cullingu, korekcji precyzji oraz korekcji horyzontalnej.

Najistotniejszymi badanymi czynnikami były: czas całkowity generacji klatki, czas renderingu, liczba sektorów tworzących wizualizację przed i po kolejnych optymalizacjach oraz liczba klatek na sekundę. Na ich podstawie będzie można jednoznacznie zdefiniować różnice w wydajności.

6.1 Środowisko testowe

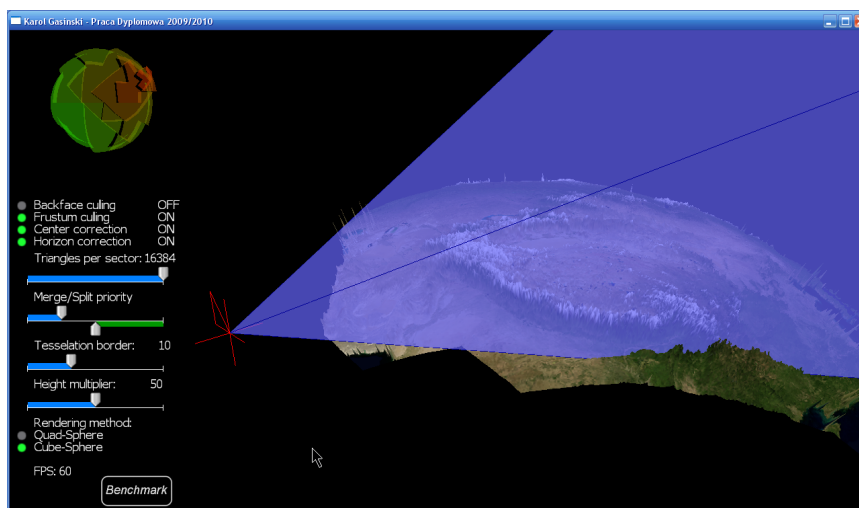
W celu zachowania maksymalnej wiarygodności pomiarów wszystkie eksperymenty zostały przeprowadzone z następującymi ustawieniami. Optymalizacje backface i frustum cullingu zostały ustawione na włączone razem z korekcją precyzji i korekcją horyzontalną. Liczbę trójkątów na sektor przyjęto jako maksymalną, równą 16384. Granice reorganizacji siatki przyjmują wartości kolejno 0,25 i 0,50, granica tesselacji jest ustawiona na 20 a mnożnik wybruszenia na 50. Wszystkie eksperymenty zostały domyślnie przeprowadzone w rozdzielczości 1280x720 na konfiguracji testowej nr. 1 (tab. 5).

Procesor	Intel Core 2 Duo T9600 2x2.80 GHz FSB 1066MHz, 6MB L2 cache, 64bit
Pamięć	4GB DDR2
Dysk	SSD 80GB
Grafika	Nvidia Quaddro FX 770M, 512MB
OS	Windows XP 32bit

Tablica 5: Konfiguracja testowa nr.1 - Laptop HP Elite Book 8530w.

Procesor	Intel Core 2 Quad Q9450 4x2.66 GHz FSB 1333MHz, 12MB L2 cache, 64bit
Pamięć	8GB DDR2 800MHz Kingston
Dysk	HDD 1TB
Grafika	Nvidia GeForce GTX 260, Zotac AMP!, 896MB
OS	Windows XP 32bit

Tablica 6: Konfiguracja testowa nr.2 - Komputer PC.



Rysunek 23: Zrzut ekranu z aplikacji testowej.

6.2 Złożoność sceny

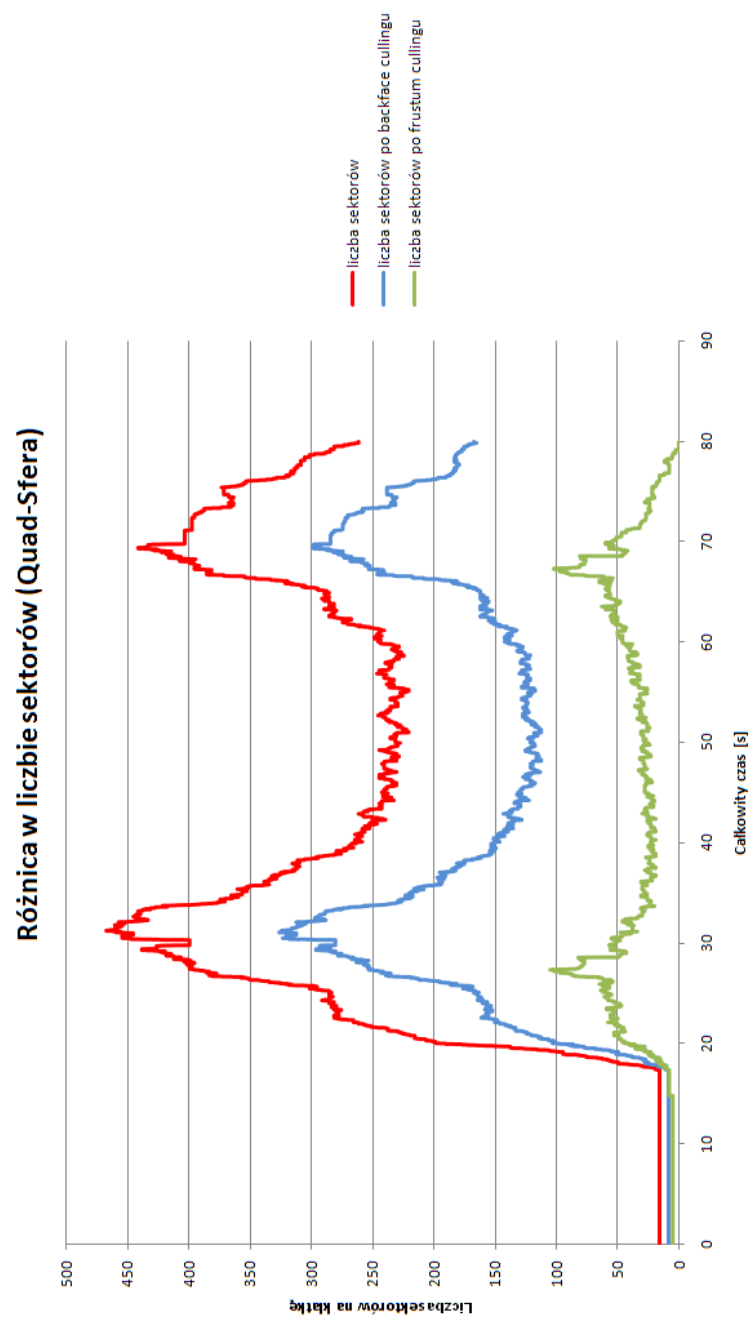
Pierwszy z przeprowadzonych eksperymentów polegał na zbadaniu ilości sektorów, z których składa się mapa terenu dla poszczególnych klatek animacji. Rysunki 24 i 25 przedstawiają wykresy wahanía ilości sektorów w czasie animacji dla technik Quad-Sfery i Cube-Sfery. Na czerwono została oznaczona pierwotna liczba sektorów przed optymalizacją, na niebiesko liczba pozostała po backface cullingu a na zielono te sektory, które przeszły pozytywnie test frustum cullingu. Przez pierwsze kilkanaście sekund oba wykresy są poziome i prawie zerowe. Odpowiada to stanowi, gdy obserwator znajduje się daleko od wizualizowanej planety. Nie następuje wówczas reorganizacja siatki, która jest w stanie początkowym. W przypadku Quad-Sfery zauważalne są dwa wzrosty szczegółowości siatki. Odpowiadają one momentom, w których obserwator przemieszczał się kolejno nad północnym i południowym biegunem wizualizacji.

6.3 Czas renderingu

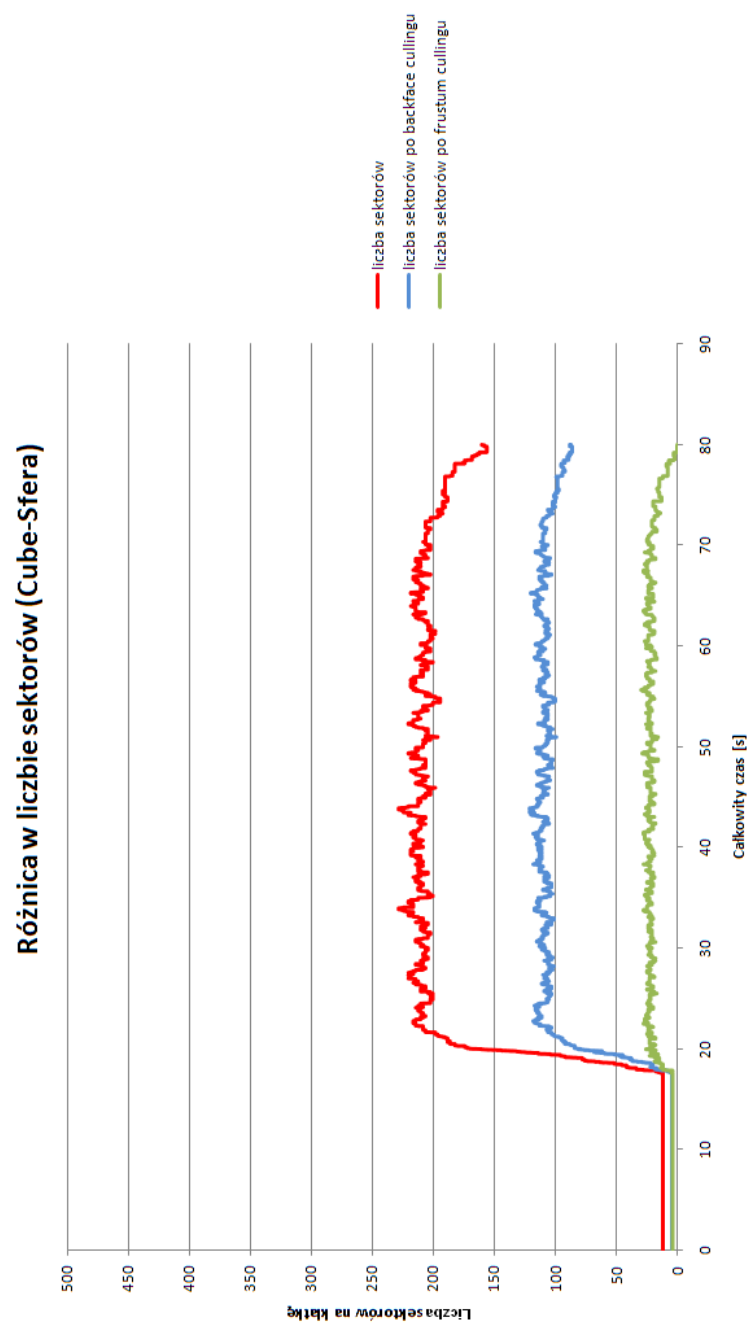
Kolejny eksperyment miał na celu zbadanie wydajności wizualizacji pod względem czasu generacji klatki oraz całkowitej liczby klatek na sekundę. Rysunek 26 przedstawia wykres czasu renderingu klatki w mikrosekundach. Podobnie jak w poprzednim eksperymencie, dla Quad-Sfery widać wyraźny wzrost czasu niezbędnego do wygenerowania klatki w momencie przejścia obserwatora nad biegunami.

6.4 Poprawność wizualizacji

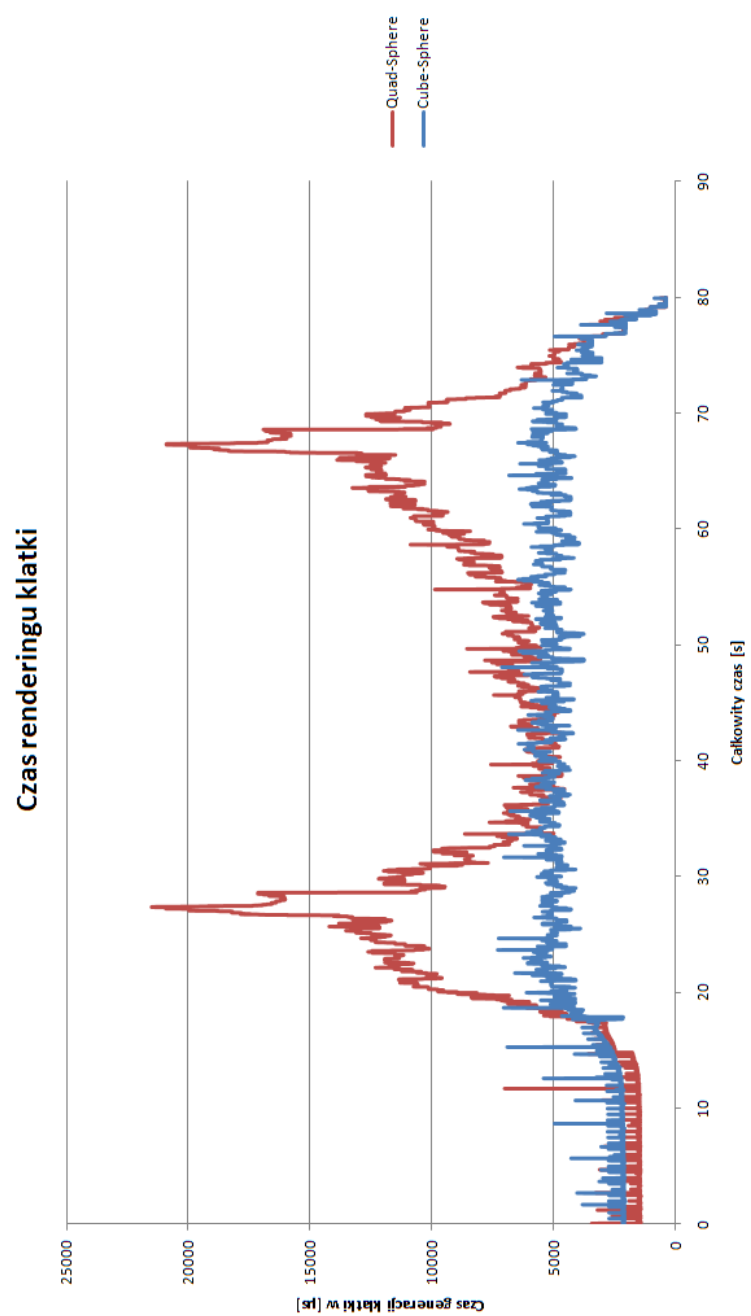
Eksperyment dotyczący poprawności wizualizacji miał na celu określenie przydatności korekcji precyzji i korekcji horyzontalnej. W pierwszej kolejności uruchomiono dwa benchmarki z włączoną i wyłączoną korekcją precyzji. Jak widać na rysunku 27 optymalizacja ta nie ma istotnego wpływu na wydajność wizualizacji. Jej efekt nie był również widoczny w przeprowadzonym benchamrku. Można by więc stawiać w wątpliwość jej zasadność. Korekcja ta ujawnia jed-



Rysunek 24: Różnica wydajności wizualizacji Quad-Sfery



Rysunek 25: Różnica wydajności wizualizacji Cube-Sfery



Rysunek 26: Czas renderingu klatki.

Tablica 7: Średnia liczba klatek na sekundę dla poszczególnych dokładności sektorów, konfiguracja testowa nr. 1.

Trójkątów	1	4	16	64	256	1024	4096	16384
FPS avg. (Quad)	700	612	697	686	612	505	368	206
FPS avg. (Cube)	924	794	918	910	793	674	489	243

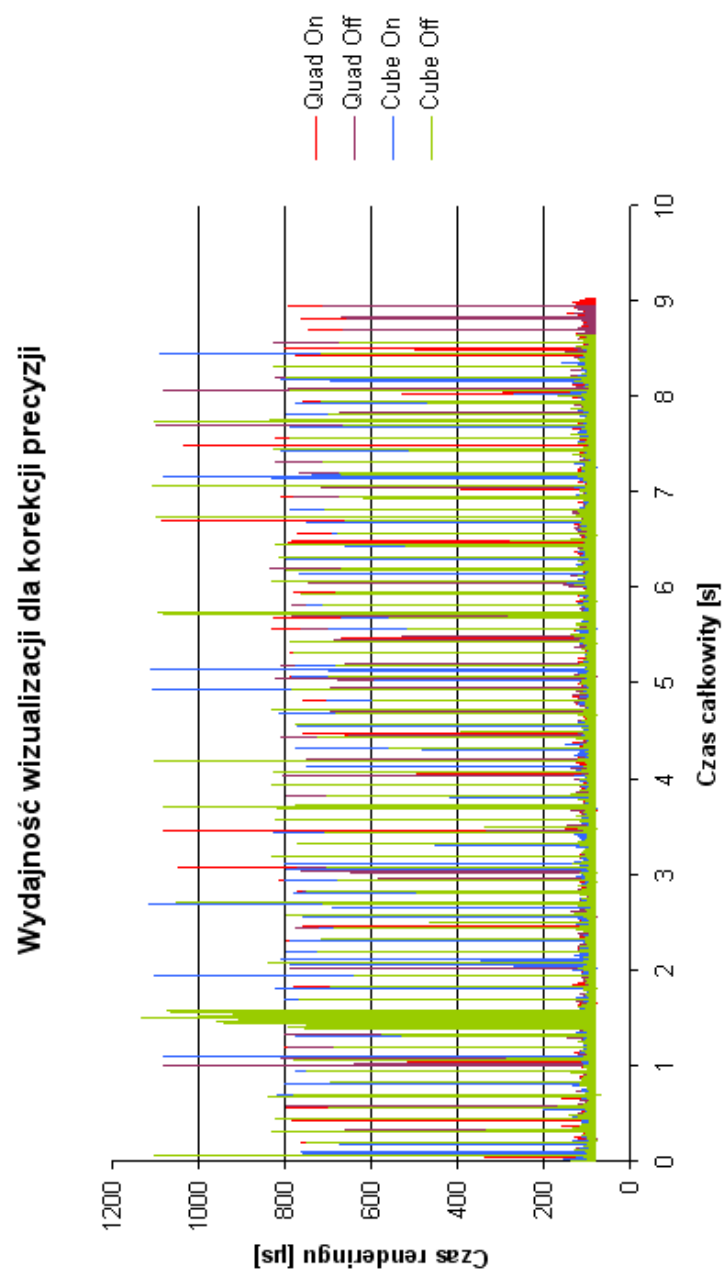
nak swoje cechy przy wizualizacjach o skali planetarnej, czego nie można było przedstawić przy użyciu standardowego benchmarku i zestawu danych. Po jej wyłączeniu teren znajduje się w złym miejscu w stosunku do obserwatora, podczas gdy po włączeniu błąd ten narasta od zera wraz ze wzrostem odległości od widza.

Kolejna część eksperymentu polegała na przeprowadzeniu tego samego benchmarku z włączoną oraz wyłączoną korekcją horyzontalną. Ona również nie wpływa na wydajność wizualizacji jednak znacząco wpływa na jej poprawność. Gdy korekcja jest wyłączona, wizualizowany teren jest całkowicie niewidoczny na początku benchmarku, gdy obserwator dopiero zbliża się do planety. Wraz z korekcją horyzontalną wizualizowany teren jest poprawnie przeskalowywany do frustumu obserwatora, dzięki czemu jest on widoczny zawsze niezależnie od odległości.

6.5 Obciążenie

Ostatni eksperyment miał na celu określenie optymalnej liczby trójkątów na sektor wizualizacji tak, aby w pełni wykorzystać możliwości współczesnych kart graficznych. W tym celu pomiary zostały przeprowadzone na dwóch komputerach (zgodnie z konfiguracjami testowymi przedstawionymi w tabelach 5 i 6).

Tabele 7 i 8 przedstawiają zależność średniej liczby klatek generowanych w ciągu sekundy w stosunku do złożoności siatki pojedynczego sektora. Na podstawie analizy przedstawionych danych, zarówno dla zintegrowanej karty



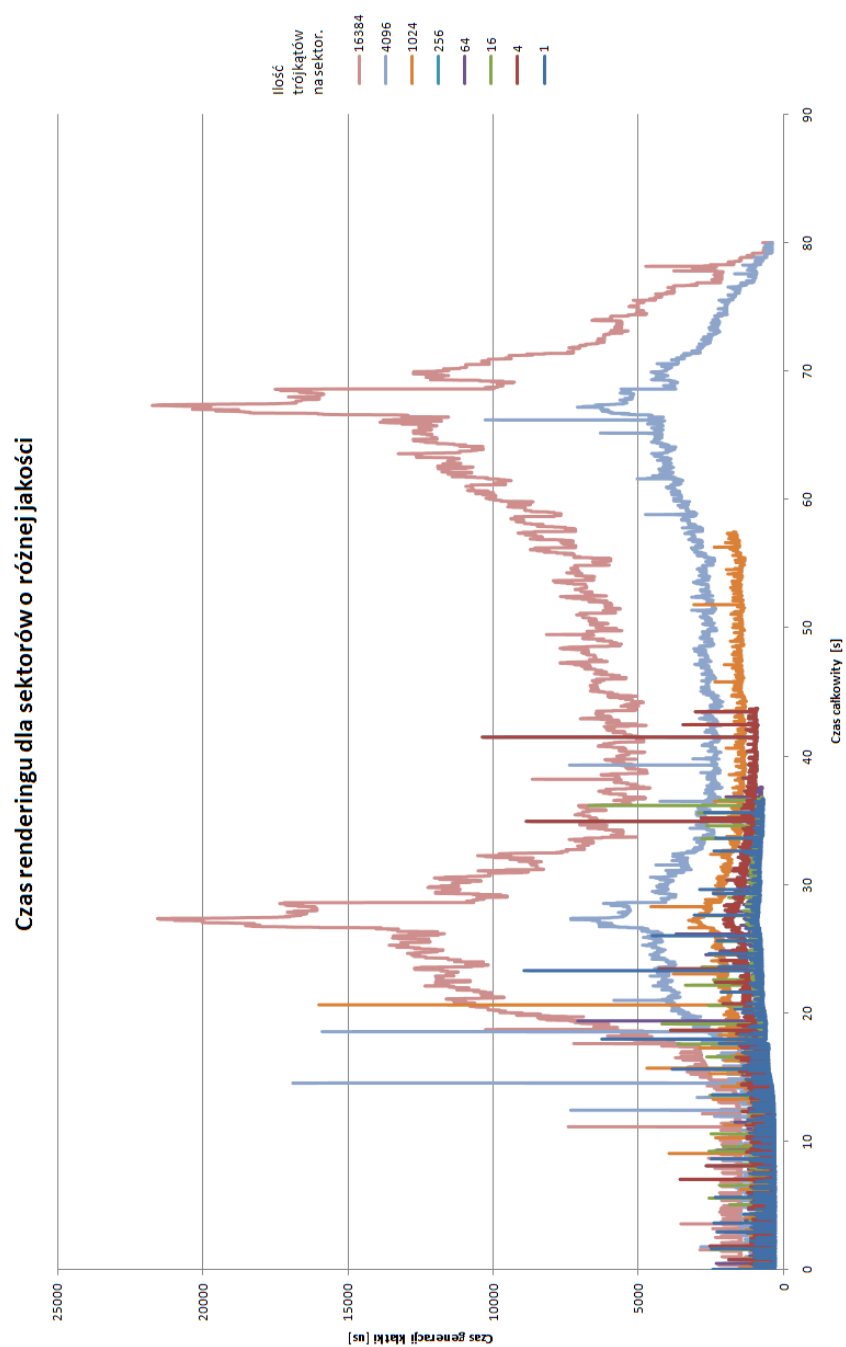
Rysunek 27

Tablica 8: Średnia liczba klatek na sekundę dla poszczególnych dokładności sektorów, konfiguracja testowa nr. 2.

Trójkątów	1	4	16	64	256	1024	4096	16384
FPS avg. (Quad)	2246	2232	2216	2182	2036	1640	949	376
FPS avg. (Cube)	3201	3159	3136	3123	2862	2284	1305	527

graficznej jak i karty dyskretnej, optymalny stosunek liczby FPS do jakości siatki terenu otrzymuje się przy 256-1024 trójkątów na sektor.

Rysunek 28 przedstawia porównanie czasów generacji klatki dla wizualizacji o różnej dokładności sektorów na pierwszej konfiguracji testowej. Przedstawione dane potwierdzają, że optymalnym rozwiązaniem jest korzystanie z sektorów składających się z 256-1024 trójkątów.



Rysunek 28: Wykresy czasu renderingu w zależności od dokładności sektorów (32000 próbek w każdej serii)

7 Wnioski

W niniejszej pracy została przedstawiona teoria podstawowych techniki sferycznej wizualizacji terenu. Jest to wciąż rozwijająca się dziedzina grafiki komputerowej, co wiąże się z mnogością istniejących rozwiązań i ich implementacji. Dwie spośród omówionych technik: Quad-Sfera i Cube-Sfera, zostały zaimplementowane przy użyciu autorskiego algorytmu HRTMR w celu dokonania porównania ich wydajności. Na podstawie otrzymanych pomiarów udowodnione zostały wady i zalety tych technik opisane w części teoretycznej.

Zgodnie z oczekiwaniami technika Cube-Sfery jest stabilna bez względu na pozycję obserwatora nad powierzchnią wizualizowanego terenu. Ilość geometrii niezbędnej do poprawnego wyświetlenia terenu w przypadkach krytycznych dla Quad-Sfery była blisko dwukrotnie większa niż dla Cube-Sfery. Wiązało się to z ponad czterokrotnym spadkiem jej wydajności w stosunku do porównywanej techniki. Wyniki te jednoznacznie udowadniają, że Cube-Sfera jest lepszym modelem do tworzenia na nim wizualizacji terenu niż Quad-Sferą.

Szanse autorskiego algorytmu na rozpowszechnienie się w aktualnej formie wśród twórców wizualizacji terenu są małe. Dzieje się tak z powodu wkraczającej na rynek nowej generacji kart graficznych posiadających układ tesselatora. Dzięki niemu twórcy aplikacji otrzymują nowe możliwości dynamicznej reorganizacji siatki terenu.

Pomimo wkraczania nowych technologii i związanych z nimi zapowiedzi przyszłych rozwiązań, algorytm HRTMR spełnia swoje założenia i może być wykorzystywany w średnio zaawansowanych projektach jako moduł odpowiedzialny za wizualizację terenu. Dołączona zaś do pracy aplikacja może być dodatkowo przydatna przy kalibracji jego parametrów dla konkretnego zastosowania.

Literatura

- [Ca02] Calabretta, M. R. & Greisen, E. W.: *Representations of celestial coordinates in FITS.*, Astronomy & Astrophysics vol. 395, 2002.
- [Ch75] Chan, F. K. & O'Neill, E. M.: *Feasibility Study of a Quadrilateralized Spherical Cube Earth Data Base*, Silver Spring, Md. : Computer Sciences Corporation, System Sciences Division, 1975.
- [Cl97] Clarke, A. C.: *3001: The Final Oddysey*, Del Rey, 1997.
- [Du97] Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich Ch. & Mineev-Weinstein M.B.: *ROAMing Terrain: Real-time Optimally Adapting Meshes*, <http://www.llnl.gov/graphics/ROAM>
- [Lu03] Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., & Huebner, R.: *Level of Detail for 3D Graphics.*, Morgan Kaufmann, 2003.
- [Ne76] O'Neill, E. M. & Laubscher, R. E.: *Extended Studies of a Quadrilateralized Spherical Cube Earth Data Base.*, Silver Spring, Md. : Computer Sciences Corporation, System Sciences Division, 1976.
- [Ro01] Rose, D., Kada, M., Ertl, T.: *On-the-Fly Adaptive Subdivision Terrain*, <http://www.vis.uni-stuttgart.de/vmv01/dl/papers/6.pdf>
- [Te96] Tegmark, M.: *An Icosahedron-Based method for pixelizing the celestial sphere*, ApJ Letters, 470, L81, 1996.
- [Vi06] Vistnes, H.: *Rendering terenu na poziomie procesora GPU w: Perelki Programowania Gier*, Tom 6, str. 517-528, Helion, 2006.