Volume Rendering

Realizacja algorytmów Marching Cubes oraz Direct Volume Rendering

Katarzyna Stepień, Alicja Wójcik

ABSTRACT

Renderowanie wolumetryczne to potężna technika w grafice komputerowej, umożliwiająca wizualizację zbiorów danych wolumetrycznych, takich jak obrazy medyczne, symulacje naukowe czy skany przemysłowe. Niniejsza dokumentacja projektu opisuje implementację i użytkowanie systemu renderingu wolumetrycznego, który wykorzystuje najnowsze algorytmy do generowania przekonujących wizualizacji z trójwymiarowych pól skalarów.

Keywords: volume rendering, marching cubes, ray casting, wizualizacja 3d

1 VOLUME RENDERING

Rendering wolumetryczny to zaawansowana technika graficzna, która umożliwia generowanie trójwymiarowych wizualizacji przestrzennych danych, znanych jako dane wolumetryczne. W odróżnieniu od tradycyjnych technik renderingu, które operują na powierzchniach obiektów, volume rendering skupia się na reprezentacji pełnego wolumenu danych, co czyni ją szczególnie przydatną do wizualizacji obrazów medycznych, symulacji naukowych i innych trójwymiarowych danych przestrzennych [4].

W procesie volume renderingu, każdy element wolumetrycznego zbioru danych, nazywany wokselem (skrót od "volume pixel"), jest poddawany analizie i przekształcaniu w kolor oraz intensywność światła. To podejście pozwala na uzyskanie realistycznych i pełnych informacji wizualizacji struktur trójwymiarowych, nawet jeśli te struktury są złożone i znajdują się w różnych głębokościach w przestrzeni.

Metody renderingu wolumetrycznego są wszechstronne i znajdują zastosowanie w wielu dziedzinach, od medycyny po grafikę komputerową, oferując unikalne możliwości wizualizacji trójwymiarowych danych. W dziedzinie medycyny, technika ta odgrywa kluczową rolę w diagnostyce obrazowej, umożliwiając precyzyjną reprezentację struktur anatomicznych z obrazów tomograficznych i rezonansu magnetycznego.

W naukach przyrodniczych i inżynierii, metody volume rendering pozwalają na wizualizację wyników złożonych symulacji numerycznych. Dzięki tej technice, badacze mogą analizować złożone zjawiska przestrzenne, takie jak przepływy płynów czy dynamika strukturalna, co jest niezwykle istotne w projektowaniu nowoczesnych konstrukcji, analizie materiałów czy badaniach geologicznych.

W dziedzinie przemysłu, metody volume rendering stosuje się w analizie struktury i właściwości trójwymiarowych obiektów. Przykładowo, w badaniach materiałów, technika ta umożliwia precyzyjną ocenę mikrostruktury substancji, co ma kluczowe znaczenie dla doskonalenia nowoczesnych materiałów inżynieryjnych.

Nie tylko w obszarze naukowym, ale także w grafice komputerowej i przemyśle rozrywkowym, volume rendering odgrywa znaczącą rolę. W produkcji gier komputerowych i filmów animowanych, metody te są wykorzystywane do generowania realistycznych efektów świetlnych, dymu, czy ognia, co przyczynia się do tworzenia immersyjnych i atrakcyjnych wizualnie doświadczeń dla użytkowników. Zastosowanie tych różnorodnych metod wolumetrycznych obejmuje więc szerokie spektrum dziedzin, otwierając nowe perspektywy w wizualizacji trójwymiarowych danych i ich interpretacji.

OMÓWIENIE WYBRANYCH ALGORYTMÓW VOLUME RENDERING

1.1 Algorytm Marching Cubes

Algorytm Marching Cubes jest powszechnie używaną i efektywną techniką generowania trójwymiarowej grafiki na podstawie danych przestrzennych. Ten zaawansowany algorytm, zaproponowany pierwotnie przez Lorensa i Cline'a w 1987 roku, jest szczególnie użyteczny w dziedzinie wizualizacji medycznej, geologii, oraz innych dziedzin, gdzie trójwymiarowa reprezentacja danych jest kluczowa. Algorytm Marching Cubes pozwala na wizualizację danych przestrzennych w postaci trójwymiarowych obiektów, a jego popularność wynika z jego zdolności do generowania powierzchni w oparciu o skalarne dane, takie jak gęstość, temperatura czy inne właściwości fizyczne.

Podczas omawiania implementacji Algorytmu Marching Cubes, warto skupić się na jego krokach, od budowy siatki punktów po generowanie trójkątów reprezentujących powierzchnię obiektu. Dodatkowo, istotnym aspektem będzie zrozumienie sposobu radzenia sobie z kwestiami efektywności i optymalizacji, aby umożliwić płynne i szybkie generowanie grafiki w zależności od złożoności danych wejściowych. Algorytm Marching Cubes stanowi fascynującą dziedzinę grafiki komputerowej i wizualizacji danych, a jego zastosowania obejmują nie tylko dziedziny medyczne, ale także symulacje fizyczne, projektowanie gier i wiele innych obszarów.

Algorytm marching cubes przekształca siatkę zbinaryzowanych za pomocą tzw. *isoValue* punkty siatki w obiekt 3D skonstruowany z trójkątnych poligonów. Wejściem algorytmu to wokselizacja (punkty w przestrzeni symbolicznie połączone siatką) reprezentująca obszar skalarny $v = f(x,y,z)$. Wartości w punktach mogą być zbinaryzowane, bądź dostarczony zostanie próg (*isoValue*), powyżej/poniżej którego punkty o danej wartości zostaną uznane za znajdujące się wewnątrz bądź na zewnątrz tworzonego obiektu.

Algorytm polega na "trawersowaniu kostką"(analizie kolejnych 8 punktów siatki tworzących najmniejszy możliwy sześcian) po dostarczonych danych. Konfiguracja wierzchołków sześcianu jest następnie zapisywana jako 8-bitowa liczba w formacie binarnym - gdzie każdy bit sygnalizuje czy dany wierzchołek ma znajdować się wewnątrz bryły, bądź na zewnątrz. Następnie dla każdego zestawu wierzchołków korzysta się z przygotowanej tablicy ($amg.LookupTable, LUT$) o $2^8 = 256$ konfiguracjach poligonów dla każdego możliwego układu wierzchołków. Poniżej przedstawiony jest fragment tablicy (1)

Układ wierzchołków	Układ trójkątów
0x0	-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
0x109	0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
0x203	0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
...	...

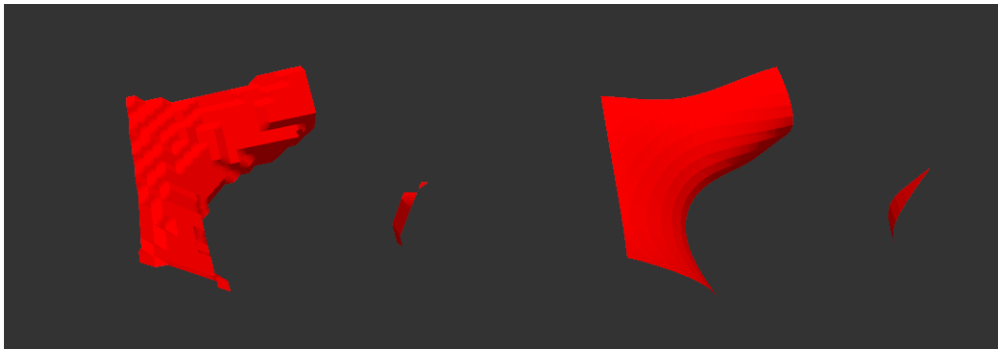
Tabela 1. Fragment LUT [2]

Istnieje również wersja algorytmu z bardziej okrojoną LUT - ogranicza się przypadki symetryczne. Wprowadza to jednak większą złożoność obliczeniową oraz zwiększa ryzyko błędu [1].

Trójkąty składające się na powstającą bryłę są złożone z trzech krawędzi przecinających się w określonych punktach. W podstawowej wersji algorytmu za punkty przecięcia uznaje się średnią odległość między wierzchołkami tworzącymi linię, którą przetną obie krawędzie. Usprawnieniem algorytmu jest zastosowanie interpolacji liniowej do wyznaczania punktów przecięcia - możliwe jest uzyskanie w ten sposób gładkich powierzchni o skomplikowanej geometrii.
W naszym projekcie użyty jest następujący wzór (1), obliczający wektor współrzędnych punktu przecięcia na podstawie współrzędnych wierzchołków v_1 i v_2 oraz ich wartości f:

$$g(v_1, v_2) = v_1 + \frac{(isoValue - f(v_1)) * (v_2 - v_1)}{f(v_2) - f(v_1)} \tag{1}$$

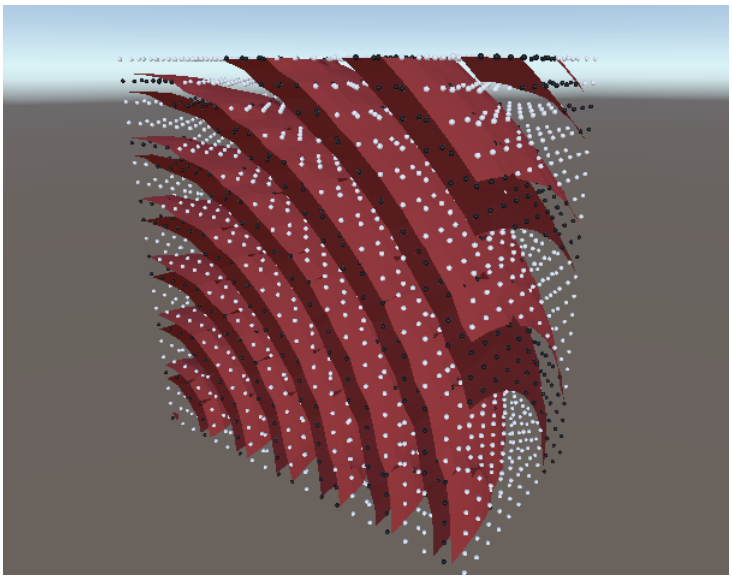
Wynik zastosowania interpolacji w algorytmie widoczny jest na poniższej grafice (1)



Rysunek 1. Wynik działania algorytmu przed i po zastosowaniu interpolacji

Algorytm posiada oczywiście znaczącą ilość ograniczeń - największym problemem jest oczywiście złożoność obliczeniowa, ale również pamięciowa - modele rosną sześciennie, co sprawia, że już średniej wielkości modele posiadają milionową liczbę wielokątów, co zajmuje znaczącą ilość miejsca w pamięci maszyny. Problemem jest również dokładność - przez odtwarzanie obrazu sześcian po sześcianie mogą tworzyć się błędy - dziury, wypukłości. Oczywiście, zwiększanie ilości punktów tworzących siatkę pozwala minimalizować niedoskonałości, ten sposób pogarsza jednak znacząco problem złożoności.

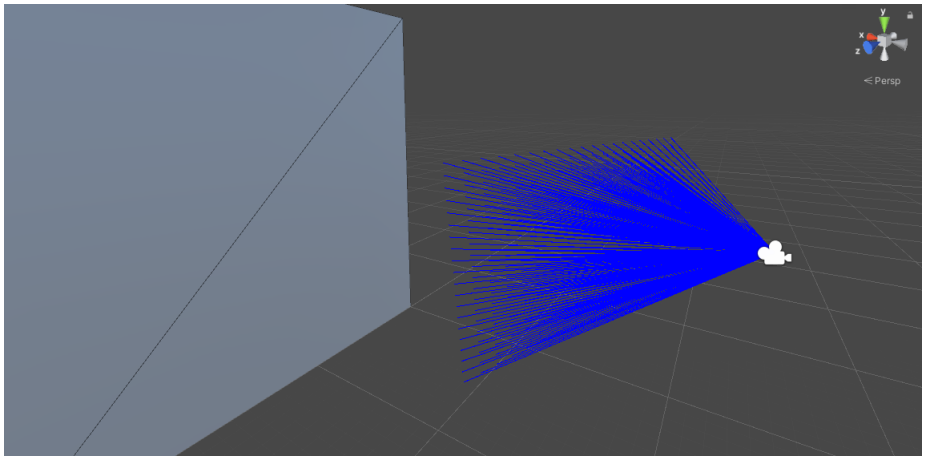
Wynik działania algorytmu prezentuje rysunek 2



Rysunek 2. Wygenerowana powierzchnia na podstawie zdyskretyzowanych punktów w przestrzeni (białe, czarne punkty - punkty znajdujące się wewnątrz i na zewnątrz struktury)

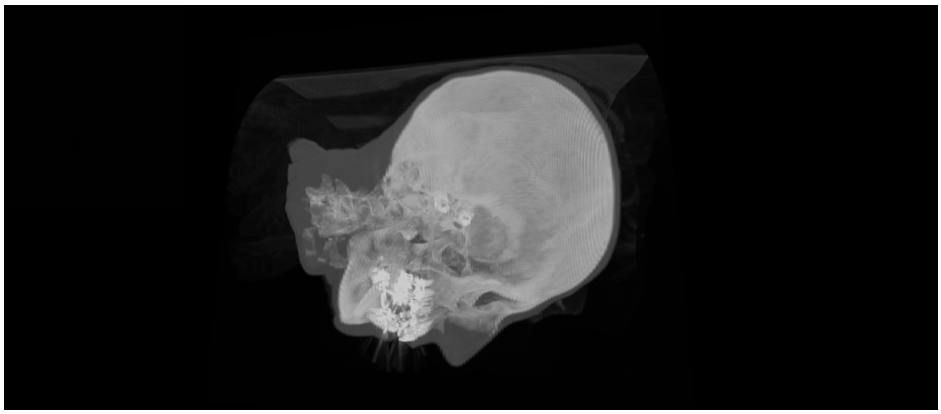
1.2 Ray casting

Ray casting to technika wykorzystująca możliwości szybkiego przetwarzania równoległego przez karty graficzne (GPU). Jej zasadniczym elementem jest wyliczanie koloru każdego piksela tekstury poprzez wysyłanie wirtualnych wiązek w przestrzeni świata [5]. Po wyliczeniu kierunku i trajektorii wiązki, można stwierdzić, przez jakie punkty przechodzi, a w związku z tym stwierdzić, czy napotyka renderowany obiekt. Zaawansowane techniki mające na celi wyliczenie stopnia oświetlenia piksela na podstawie zjawiska załamania / odbicia wiązki noszą nazwę "Ray tracing", a są one wykorzystywane przede wszystkim we współczesnych grach komputerowych i narzędziach typu Blender.



Rysunek 3. Wizualizacja wiązek wysyłanych z kamery w kierunku odpowiednich pikseli (tu dla tekstury 20x20)

W wypadku Volumetric Rendering trajektoria związki jest wykorzystywana, aby wyliczyć wartość piksela na podstawie wszystkich wokseli, przez które przechodzi. Każdy woksel należący do tekstury zawiera wartość z przedziału $[0,1]$. Można przyjąć różne metryki wyliczające ostateczną wartość dla piksela. Jedną z nich jest np. Maximum Intensity Projection, który za ostateczną wartość dla piksela przyjmuje kolor $(m,m,m,0)$, gdzie m jest największą wartością napotkaną na trajektorii wiązki, a wartości kanałów przyjmują wartości z zakresu $[0,1]$.



Rysunek 4. Wizualizacja modelu ludzkiej głowy na podstawie skanów pochodzących z tomografii komputerowej metodą Maximum Intensity Projection

Można wykorzystać też inne funkcje, np. przypisujące napotkanej maksymalnej wartości określony kolor, co może pomóc zróżnicować użytkownikowi np. różne rodzaje tkanek przy użyciu kontrastujących ze sobą kolorów.

2 IMPLEMENTACJA ALGORYTMÓW

2.1 Właściwości środowiska Unity

Unity jest silnikiem gier, który oferuje zaawansowane narzędzia do tworzenia interaktywnych środowisk wizualnych, ale może być również skutecznym narzędziem do implementacji algorytmów volume rendering. Unity oferuje kilka kluczowych korzyści dla implementacji algorytmów volume rendering. Po pierwsze, jest to środowisko programistyczne przyjazne dla użytkownika, co ułatwia rozwój algorytmów i eksperymentowanie z różnymi technikami wizualizacji. Ponadto, silnik ten oferuje wydajne mechanizmy renderowania, które umożliwiają płynne i responsywne wyświetlanie trójwymiarowych danych na ekranie.

Wreszcie, Unity zapewnia również możliwość łatwego dostosowania interfejsu użytkownika, co pozwala na tworzenie interaktywnych narzędzi do manipulacji i analizy trójwymiarowych danych. To z kolei może znacznie ułatwić pracę z algorytmami volume rendering w różnych dziedzinach, poprawiając wizualizację danych oraz ułatwiając zrozumienie ich struktury i relacji.

2.2 Omówienie implementacji algorytmu marching cubes

Działanie algorytmu zostało omówione w poprzedniej sekcji. Poniżej znajduje się szczeółowe omówienie komponentów użytych w implementacji.

MarchingCubes Klasa stanowiąca rdzeń algorytmu, dziedzicząca po MonoBehaviour. W Unity, klasa dziedzicząca po MonoBehaviour to podstawowa jednostka, która reprezentuje skrypt lub komponent, który można dołączyć do obiektów w scenie. Klasy te dziedziczą z klasy MonoBehaviour, co nadaje im specjalne cechy i umożliwia interakcję z silnikiem Unity.

Marching cubes zleca stworzenie siatki punktów oraz jej przetworzenie, zajmując się tworzeniem nowych obiektów w grze na podstawie dostarczonych danych wynikowych, tworząc zadane struktury 3D. Obsługuje również rotację nowopowstałych obiektów jak jednością, stosując grupowanie.

GridGenerator Klasa służąca do proceduralnego generowania danych wejściowych - algorytm Marching Cubes idealnie nadaje się do wizualizacji ciekawych funkcji matematycznych. Klasa zwraca punkty w przestrzeni zgrupowane w sześcienne struktury danych (GridCell) wraz z przypisanymi do nich wartościami.

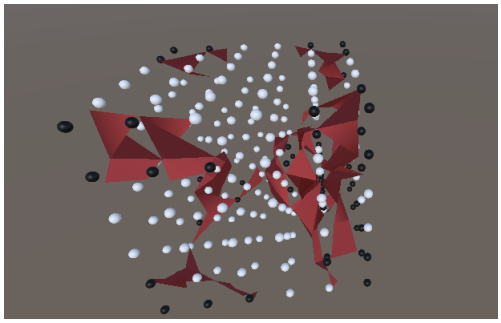
LongList Specjalna wersja listy stworzona przez limitacje pamięciowe, zdolna do przechowywania dużo większej ilości danych. Zostanie omówiona bardziej szczegółowo w sekcji napotkane problemy oraz ich rozwiązania.

MeshWrapper Klasa zajmująca się właściwym algorytmem. Przyjmuje siatkę punktów przestrzeni, przetwarza je "kostka po kostce"i zwraca gotowy obiekt typu mesh (siatka poligonów).

MarchingCubesSettings klasa umożliwiająca przekazanie przez użytkownika parametrów do funkcji. Możliwe jest wybranie 3 funkcji matematycznych do prezentacji działania algorytmu wraz z odpowiednimi parametrami dla nich (threshold dla szumu Perlina i radius dla kuli i funkcji "ser"). Udostępniona jest też opcja wizualizacji siatki w przestrzeni, jest jednak ona bardzo wymagająca sprzętowo.

2.3 Marching cubes - napotkane problemy oraz ich rozwiązania

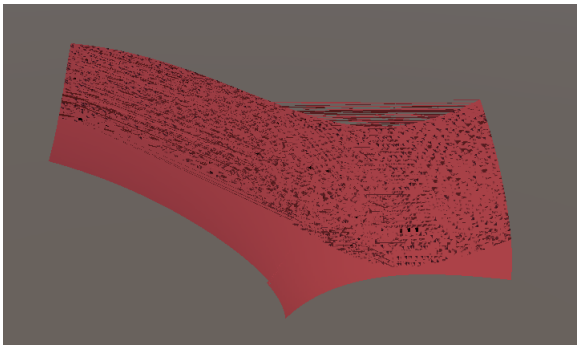
Jednym z pierwszych, znaczących problemów okazała się być znajomość środowiska unity. Alorytm wykonywany jest na najbardziej podstawowych strukturach grafiki 3D, co wymagało głębszego poznania mechanizmów silnika. Przykładowym błędem związanym z brakiem tej wiedzy było nieprawidłowe układanie punktów przestrzeni w sześciany - kolejność wierzchołków wpływała na działanie algorytmu, często go psując (rys. 5)



Rysunek 5. Dziury powstałe przez nieprawidłową kolejność wierzchołków w badanych sześcianach

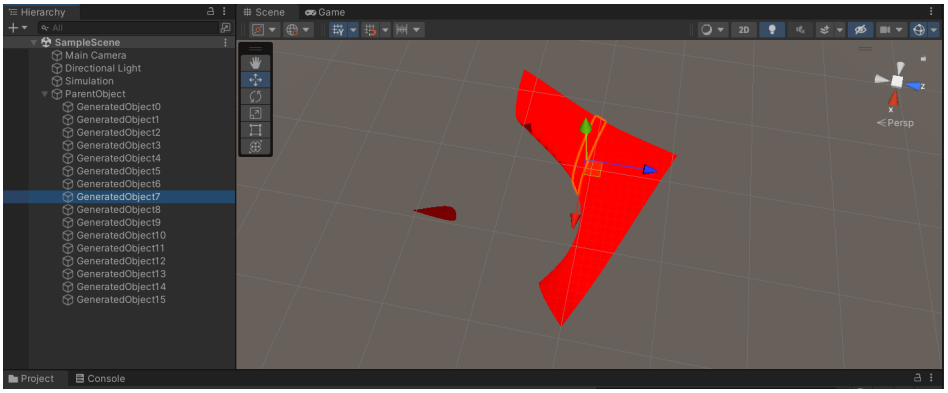
Największym problemem podczas implementacji marching cubes okazały się ograniczenia sprzętowe. Samo unity okazało się być sporą przeszkodą - ogranicza ilość wierzchołków struktury pojedynczego obiektu do ok. 65 tys. Jeszcze większym problemem okazały się być ograniczenia pamięciowe struktur

danych - bez ostrzerzeń listy nie były w stanie miećić ilości przetwarzanych przez algorytm obiektów, idących w miliony, co skutkowało niezauważalnymi przeciekami pamięci, różnicami w długościach list które powinny być równe i w efekcie niepoprawnymi strukturami 3D (rys. 9)



Rysunek 6. Niepoprawny obiekt powstały przez *overflow* pamięci

Problemy te wymusiły zrestrukturyzowanie całego programu - stworzona została specjalna klasa *LongList*, kontener składający się z wielu list o zadanej maksymalnej długości, zabezpieczający przez cichym przepełnianiem pamięci. Sam obiek został równierz podzielony na mniejsze obiekty, zgrupowane potem w całość w celu manipulacji w przestrzeni (rys. 7). Rozwiązanie to spowodowało kolejny błąd - ze względu na współdzielenie skryptu przez wiele obiektów generacja wizualizacji siatki w przestrzeni musiała zostać przeniesiona do nowego obiektu i skryptu.



Rysunek 7. Większa ilość obiektów składająca się na wygenerowaną powierzchnię 3D

Przeszkodą nie do przeskoczenia okazały się oczywiście ograniczenia obliczeniowe. Nawet po naprawieniu i zabezpieczeniu algorytmu jest on ograniczony przez własną, niezwykle wysoką złożoność obliczeniową. Długie obliczenia nie tylko obciążają sprzęt, ale w skrajnych wypadkach destabilizują środowiko Unity, czasem całkowicie zawieszając program. Z tego względu zrezygnowałyśmy z wizualizacji danych medycznych za pomocą algorytmu Marching Cubes - ze względu na dużą dokładność nawet mały fragment przekraczał by nasze możliwości sprzętowe.

2.4 Omówienie implementacji algorytmu ray casting

Algorytm Ray Casting w wykonywanym projekcie jest realizowany przez współpracujące ze sobą:

- Komponent Unity: **RayTracingBase**, mający za zadanie przekazanie danych, dotyczących między innymi aktualnego położenia kamery w scenie do shadera.
- Shader Unity: **RayTracingShader**, zawierający fragment shader, który wylicza dla wartości dla każdego piksela.

Szczegółowo, komponent **RayTracingBase** składa się z:

- Funkcja **Start()** - wczytuje teksturę z plików i zapisuje ją jako zasób (Unity Asset), zapisuje ID poszczególnych właściwości shadera w prywatnych atrybutach klasy, inicjalizuje materiał bazujący na shaderze **RayTracingShader** i przekazuje do materiału trójwymiarową teksturę, która ma zostać wyrenderowana.
- Funkcja **Update()** - obsługuje poruszanie się kamery, oraz wywołuje funkcję *_setShaderNumericValues*, która przekazuje do materiału parametry takie jak punkt przekroju tekstury, próg binaryzacji, parametry dotyczące kamery, macierz transformacji z przestrzeni lokalnej na przestrzeń świata, a także wartości opisujące gradient wykorzystywany w metodzie przypisującej pikselom o zadanej wartości ustalony kolor.

Zasadniczą częścią shadera jest **Fragment Shader**, w którym dokonywane są wszystkie obliczenia związane z ray castingiem.

```
fixed4 frag (v2f i) : SV_Target
{
    if (i.vertex.x < 256)
    {
        if (i.vertex.y < 256)
        {
            const float3 p = float3(i.vertex.x / 256, slice_position,
                                    i.vertex.y / 256);
            float val = tex3D(volume, p).r;
            return float4(val, val, val, 1);
        }
    }
    float3 view_point_local = float3(i.uv - 0.5, 1) * view_params;
    const float3 view_point_world = mul(cam_local_to_world_matrix,
                                        float4(view_point_local, 1));

    ray r;
    r.origin = _WorldSpaceCameraPos;
    r.direction = normalize(view_point_world - r.origin);
    return get_ray_value(r);
}
```

Pierwsza część shadera wewnątrz instrukcji warunkowych wizualizuje przekrój przez trójwymiarową teksturę o rozmiarze 256 x 256 - pobiera wartość pikseli z tekstury 3D i wyrysowuje ją na teksturę 2D.

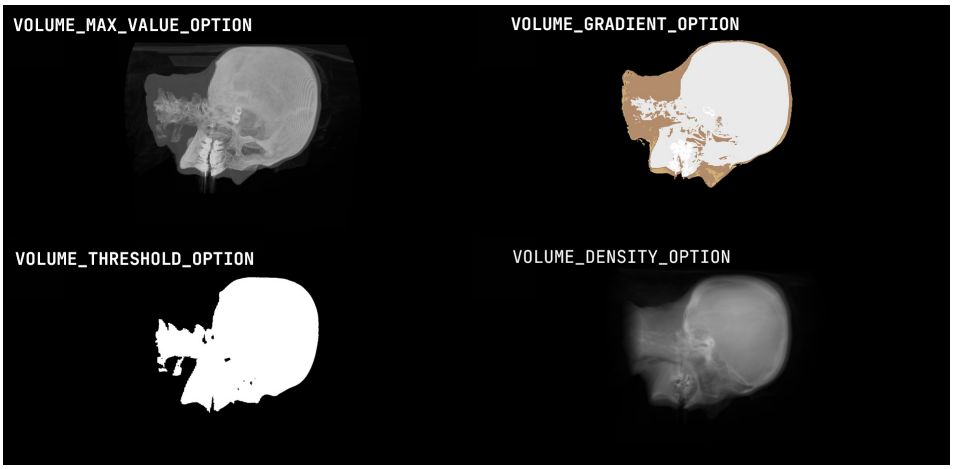
Następnie wyliczana jest pozycja piksela w przestrzeni lokalnej na rzutni, a następnie przekształcana jest w pozycję w przestrzeni świata. Tworzona jest struktura wiązki, która składa się z punktu początkowego, w tej sytuacji jest to położenie kamery, oraz kierunku, który jest wyliczany na podstawie pozycji piksela i punktu początkowego wiązki. Następnie wywoływana jest funkcja *get_ray_value(ray r)*, która ma za zadanie na podstawie wybranej metody wyliczyć wartość koloru danego piksela.
W celu przyspieszenia kompilacji shadera, wykorzystywane są dyrektywy preprocesora, które określają, jaka metoda ma zostać wykorzystana w celu obliczenia wartości piksela w wyjściowej teksturze. Dostępne opcje to:

- **VOLUME_THRESHOLD_OPTION** - jeśli wiązka napotka woksel o przypisanej wartości wyższej niż próg binaryzacji, zwraca dla piksela $(1,1,1,1)$, co odpowiada kolorowi białemu, a w przeciwnym wartość $(0,0,0,1)$, co odpowiada kolorowi czarnemu.
- **VOLUME_DENSITY_OPTION** - wysyła wiązkę o początkowej wartości równej 1, a przy przejściu przez woksel o przypisanej wartości v, zmniejsza wartość wiązki o $v/350$. Po przejściu przez całą teksturę 3D, zwraca piksel o wartości $(m,m,m,1)$, gdzie $m = 1 - \sum_{v_i \in V_r} v_i$.
- **VOLUME_MAX_VALUE_OPTION** - zwraca wartość $m,m,m,1)$ dla piksela odpowiadającą największej napotkanej wartości m spośród wszystkich wokseli, przez który przebiegła wiązka: $m = max(v_i) : v_i \in V_r$. Naywana jest w literaturze Maximum Intensity Projection

- **VOLUME_GRADIENT_OPTION** - zwraca dla piksela wartość koloru w gradiencie przekazanym przez komponent **RayTracingBase**. Po określeniu maksymalnej wartości woksela napotkanego po drodze wiązki, szuka pierwszego progu w gradiencie, który ma wartość większą od wyliczonej wartości i zwraca odpowiadający temu progowi kolor.

Nie wybranie żadnej z tych opcji przy użyciu polecenia #define powoduje zwrócenie w pełni czarnego obrazu.

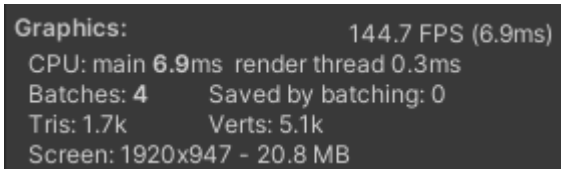
Poniżej zestawiono rezultaty wszystkich metod:



Rysunek 8. Porównanie zaimplementowanych metod ray castingu

2.5 Ray casting - napotkane problemy oraz ich rozwiązania

W przeciwieństwie do metody Marching Cubes, Ray casting nie miał żadnych problemów związanych z wydajnością ze względu na operacje wykonywane równolegle przez GPU. Dla maksymalnej testowanej rozdzielczości, 1920x947, średnia liczba klatek na sekundę wyniosła ok. 140, pomimo że dla każdej klatki obliczenia dla poszczególnych pikseli były wykonywane od nowa. W związku z tym, nie napotkano żadnych większych problemów wydajnościowych ani implementacyjnych.



Rysunek 9. Statystyki związane z wydajnością

3 WNIOSKI NA TEMAT IMPLEMENTOWANYCH ALGORYTMÓW

Obydwie metody znacząco różnią się od siebie. Pierwsza z nich, Marching Cubes, może zostać wykorzystana dla trójwymiarowych tekstur zawierających niewielką liczbę wokseli, tak by stworzone siatki nie były zbyt złożone. Można zastosować w tym wypadku gotowe materiały, przypisać je do siatek i obsłużyć oświetlenie przy użyciu istniejących narzędzi. Marching Cubes może znaleźć też zastosowanie w grach komputerowych i symulacjach, gdzie agenci wchodzą w interakcję z istniejącą siatkę, np. dokonują kolizji.

Ray casting jest niemalże w całości realizowany przez kartę graficzną - nie jest tu tworzony żaden fizyczny obiekt, z którym można wchodzić w interakcję, a jedynie tekstura wyjściowa. Jest to metoda znacznie bardziej efektywna niż algorytm Marching Cubes, który realizowany w czasie rzeczywistym (np. w wypadku w którym zmieniany byłby **isoValue**, nie miałby prawa działać bez większych spadków w

liczbie klatek na sekundę. Realizacja na GPU sprawia, że można renderować tekstury bardzo dużych rozmiarów, złożone często z więcej niż miliarda wokseli przy średniej jakości sprzęcie.

Jeśli chodzi o wykorzystanie Volume Renderingu np. w obrazowaniu medycznym, zdecydowanie lepszą metodą wydaje się Ray Casting ze względu na zwiększoną wydajność, dużą swobodę modyfikacji w czasie rzeczywistym (np. zmiana parametrów gradientu, zmiana progu binaryzacji) przy zerowym spadku klatek na sekundę, a także możliwość wizualizacji tekstur złożonych z bardzo dużej liczby wokseli, co umożliwia wykorzystywanie zdjęć o wysokiej jakości.

Marching Cubes znajduje zastosowanie przede wszystkim w grach komputerowych, gdzie wygenerowana siatka ma znaczącą przewagę ze względu na możliwość interakcji z innymi obiektami w scenie. Część obliczeń można również przenieść na GPU z wykorzystaniem tzw. Compute Shaderów, umożliwiając zwiększenie ich wydajności.

Podsumowując, obydwie metody są dobrymi metodami Volume Rendering, jednak w przypadku, który były rozważane w pracy, czyli renderowaniu obrazów na podstawie wysokiej jakości przekrojów pochodzących z tomografii komputerowej, metoda Ray Casting okazała się znacznie bardziej wydolna i jako jedyna sprostała oczekiwaniom wydajnościowym i jakościowym.

LITERATURA

[1] C. Andújar. Marching cubes algorithm, April 2014. Dostępne online na stronie Uniwersytetu Department of Computer Science (CS) and the Universitat Politècnica de Catalunya (UPC).

[2] P. Bourke. Polygonising a scalar field, May 1994. Opis i realizacja algorytmu marching cubes w OpenGL.

[3] F. Dachille, K. Kreeger, B. Chen, n. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware, 1998.

[4] C. D. Hansen and C. R. Johnson. Overview of volume rendering. In C. D. H. C. R. J. Edytor, editor, *The Visualization Handbook*, chapter 3.1. Academic Press, 2005.

[5] J. Pawasauskas. Volume visualization with ray casting, February 18, 1997. Dostępne online na stronie politechniki WORCESTER POLYTECHNIC INSTITUTE.