Politechnika Łódzka
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Implementacja algorytmu śledzenia promieni w technologii Vulcan

Implementation of the Ray Tracing algorithm in Vulcan technology

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr hab. inż. Piotr Napieralski
Dyplomant: Jarosław Suchiński
Nr albumu: 216894
Kierunek: Informatyka
Specjalność: Technologie gier i symulacji komputerowych

Łódź, luty 2022

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Spis treści

1 Wstęp

1.1 Wstęp

Metoda śledzenia promieni (ang. ray tracing) używana jest do generowania obrazów komputerowych już od roku 1968. Po raz pierwszy zaproponowana została przez Arthura Appela w 1968 [2]. W 1979 roku Tunera Whitted'a zaproponował rekursywną metodę śledzenia promieni [1]. Polegała ona na analizie przebiegu promieni pomiędzy źródłem światła a obserwatorem. Technika ta pozwala na generowanie fotorealistycznych obrazów jak i wizualizacji o dużym stopniu realizmu. Popularność tej metody wynika z realistycznej symulacji transportu światła, w porównaniu z innymi metodami renderingu, takimi jak np. rasteryzacja. Dzięki sposobowi generowania obrazów tą metodą, efekty takie jak odbicia i cienie, stają się naturalnym wynikiem działania algorytmu. Złożoność obliczeniowa nie pozwalała do tej pory używać tej metody w aplikacjach czasu rzeczywistego. Od 2018 roku, dzięki rozwojowi technologii i stworzeniu akceleratorów graficznych dedykowanych tej metodzie, stało się możliwe generować obrazy metodą śledzenia promieni w czasie rzeczywistym. Na rynku istnieje już kilka gier używających tę metodę, a kolejne są w fazie produkcji. Metoda śledzenia promieni w obecnej formie wykorzystywana przy produkcji gier jest stosunkowo nowa i wszystko wskazuje na to, że będzie stawała się coraz bardziej popularna w przemyśle gier komputerowych.

1.2 Cel i zakres pracy

W pracy dyplomowej zaprezentowano sposób implementacji śledzenie promieni dla aplikacji czasu rzeczywistego. Aplikacja ma na celu wyświetlenie fotorealistycznej sceny oraz umożliwienie użytkownikowi poruszania się po scenie. Ponadto moim celem jest stworzenie aplikacji, która działałaby płynnie wyświetlając co najmniej 60 klatek na sekundę.

Implementacja opiera się na technologii Vulkan. Została ona wybrana, ponieważ pozwala uzyskać bardzo wysoką wydajność aplikacji, która jest kluczowym aspektem przy implementacji tak wymagającej metody jak śledzenie promieni. Ponadto udostępnia funkcjonalności do wykorzystania sprzętowego wsparcia śledzenia promieni. Zostało to osiągnięte poprzez udostępnienie programiście znacznie większej kontroli nad akceleratorem graficznym niż w technologiach poprzednich generacji, takich jak OpenGL czy DirectX 11. Wadą tego rozwiązania jest to, iż programista musi szczegółowo i precyzyjnie zaprogramować sterowanie działaniem karty graficznej przez co program taki jest bardziej skomplikowany i obszerniejszy. Wysiłek jest zdecydowanie opłacalny, ponieważ pozwala uzyskać znaczny przyrost wydajności.

2 Historia metody śledzenia promieni

Ray tracing pierwszy raz został wykorzystany do wygenerowania obrazu komputerowego przez Artura Appel'a w 1968 roku [2]. Użył tej metody do znalezienia punktów powierzchni znajdujących się najbliżej kamery oraz użył promieni wtórnych by określić czy dany punkt znajduje się w cieniu.

Pionierem w tej dziedzinie został jednak Turner Whitted, który w 1979 roku opisał rekursywny algorytm, pozwalający na generowanie obrazów z materiałami lustrzanymi, refrakcyjnymi (np. szkło) oraz rzucających cienie. Wyrenderował on wówczas również kilkunastosekundową animację pt. „The Compleat Angler" prezentującą efekty jego pracy [3]. W jego pracy przełomowe było przeformułowanie algorytmu z bycia skupionym na widoczności powierzchni na algorytm skupiony na dystrybucji światła. Znaczenie tej pracy nadaje również fakt, iż był on inspiracją dla późniejszych prac o rozproszonym śledzeniu promieni oraz „unbiased path tracing" (metoda ray tracingu nie wprowadzająca systematycznych błędów lub odchyleń w przybliżeniu radiancji), co z kolei doprowadziło do opracowania równania renderingu i w efekcie możliwości generowania fotorealistycznych obrazów bazując na rzeczywistych właściwościach materiałów.

Ray tracing rozwijany był głównie na uczelniach i innych ośrodkach badawczych, gdzie najpotężniejsze dostępne maszyny były wykorzystywane do rozwoju grafiki komputerowej w tym zakresie. Początkowa metoda ta była implementowana do renderowania obrazu przy pomocy procesora CPU, co było jedyną dostępną opcją i niosło ze sobą bardzo niską wydajność. Znaczny przyrost wydajności nastąpił w momencie, gdy technikę tę można było zaimplementować do operowania na akceleratorze graficznym GPU, ponieważ posiadały one znacznie wyższą liczbę rdzeni mogących jednocześnie renderować więcej pikseli obrazu.

Wraz z postępem technologicznym ray tracing zaczął być wykorzystywany w branży filmowej. Wciąż był jednak technologią drogą, wymagającą kosztujących dziesiątki tysięcy dolarów stacji roboczych oraz czasu na wyrenderowanie realistycznie oświetlonych scen, na które nie każdy mógł sobie pozwolić. Jednym ze studiów wykorzystujących tą technikę jest Pixar, który użył jej w animacja „Cars" powstającej w 2004 roku do wygenerowania odbić lustrzanych np. na maskach samochodów [4].

Współcześnie ray tracing jest podstawą rendererów takich jak vray, Arnold (w programie 3dsMax) czy eevee, cycles (w programie Blender) i pozwala generować fotorealistyczne obrazy na większości komputerów o ile długi czas renderingu nie stanowi problemu.

Pomimo znacznego rozwoju w kwestii akceleratorów graficznych GPU oraz ogromnego skoku wydajności w aplikacjach czasu rzeczywistego ray tracing był implementowany w demach technologicznych co najwyżej w formie niezbyt skomplikowanej sceny z pojedynczym źródłem światła. Sprzyjał temu szczególnie rozwój mechanizmu podprogramu obliczeniowego GPU (ang. compute shader). Pomimo takiego kroku milowego technika ta była zbyt kosztowna by mogła być wykorzystana w jakiejkolwiek grze.

W grach komputerowych śledzenie promieni w czasie rzeczywistym stało się faktem dopiero w roku 2018 dzięki sprzętowemu wsparciu dla ray tracingu w kartach graficznych z serii RTX 2000 autorstwa Nvidii. Jedną z pierwszych gier, która wykorzystuje tą cechę nowych kart jest Battlefild V, gdzie odbicia lustrzane są generowane poprzez śledzenie promieni.

3 Metoda śledzenia promieni

Ray tracing jest w zamyśle bardzo prostym algorytmem, bada w jaki prymityw (najczęściej trójkąt) na scenie trafi promień wychodzący z kamery, przechodzący przez dany piksel. Można to zobrazować w pseudokodzie za pomocą podwójnej pętli:

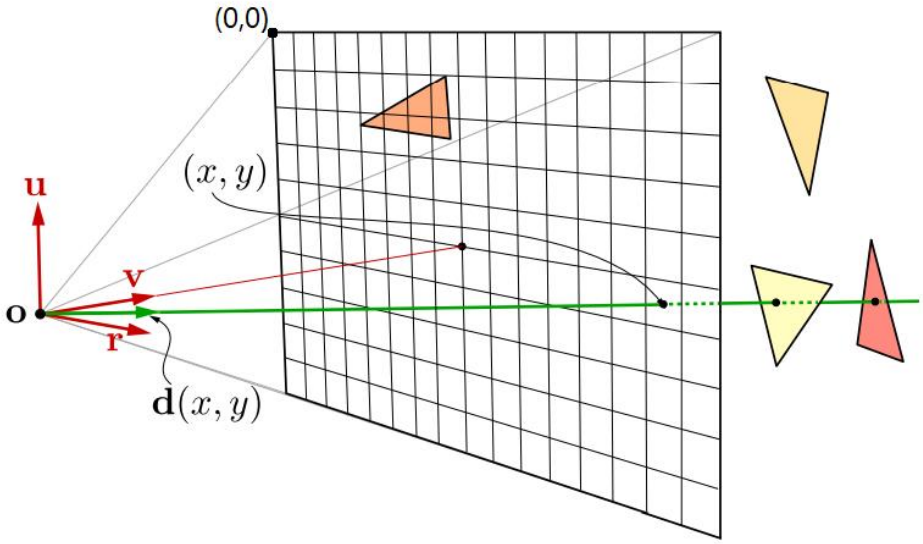
```
for (P in pixels)
    for (T in triangles)
        determine if ray through P hits T
```

Oczywiście to najprostsze wykorzystanie ray tracingu nie dające lepszych rezultatów niż rasteryzacja. Jednak jeśli zostanie dodane rekurencyjne śledzenie dodatkowych promieni z punktów trafień poprzednich, można w szybki i prosty sposób osiągnąć globalną iluminacje (GI, realistyczne rozchodzenie się światła i oświetlenia sceny), cienie, okluzję otoczenia (AO, ang. ambient occlusion) czy odbicia lustrzane.

Niestety śledzenie dodatkowych promieni potrzebuje wykładniczo więcej mocy obliczeniowej wraz z każdą kolejną głębokością rekurencji, dlatego ważne jest by ograniczyć to poprzez określenie maksymalnej głębokości rekurencji. Co więcej bolączką ray tracingu jest znalezienie trójkąta, który koliduje z promieniem, nie przeprowadzając testu przecięcia z każdym trójkątem w scenie. W tym właśnie celu wykorzystywane są przestrzenne struktury danych takie jak drzewo brył ograniczających (ang. Bound Volume Hierarchy, BVH), które znacznie przyspiesza ten proces, dzięki czemu można osiągnąć wydajność nawet $\boldsymbol{O}(\boldsymbol{log\ n})$, gdzie $\boldsymbol{n}$ to liczba trójkątów w scenie. Niemniej jednak zbudowanie takiej struktury danych jest dość kosztowne, jak również bieżące uaktualnianie jej, gdy zmienia się scena (np. gdy kamera porusza się i obraca). Przez wiele lat obsługa tego mechanizmu leżała po stronie programisty, obecnie wykorzystując najnowsze karty graficzne jest to zaimplementowane sprzętowo, co usprawnia ten proces. Jest to jednak stosunkowo nowa technologia i w następujących latach można spodziewać się postępujących optymalizacji w tym zakresie.

3.1 Podstawy ray tracingu

Śledznie promieni można opisać poprzez dwie metody **śledzenie** (ang. **trace**) oraz **cieniowanie** (ang. **shade**). Pierwsza z nich odpowiada za geometryczną część algorytmu, która jest odpowiedzialna za znalezienie najbliższego punktu przecięcia promienia z prymitywami (trójkątami) na scenie, natomiast druga generuje kolor trafionego punktu w zależności od właściwości trafionego obiektu. Faza cieniowania może również wywoływać kolejne śledzenie i cieniowanie, by uzyskać więcej danych.



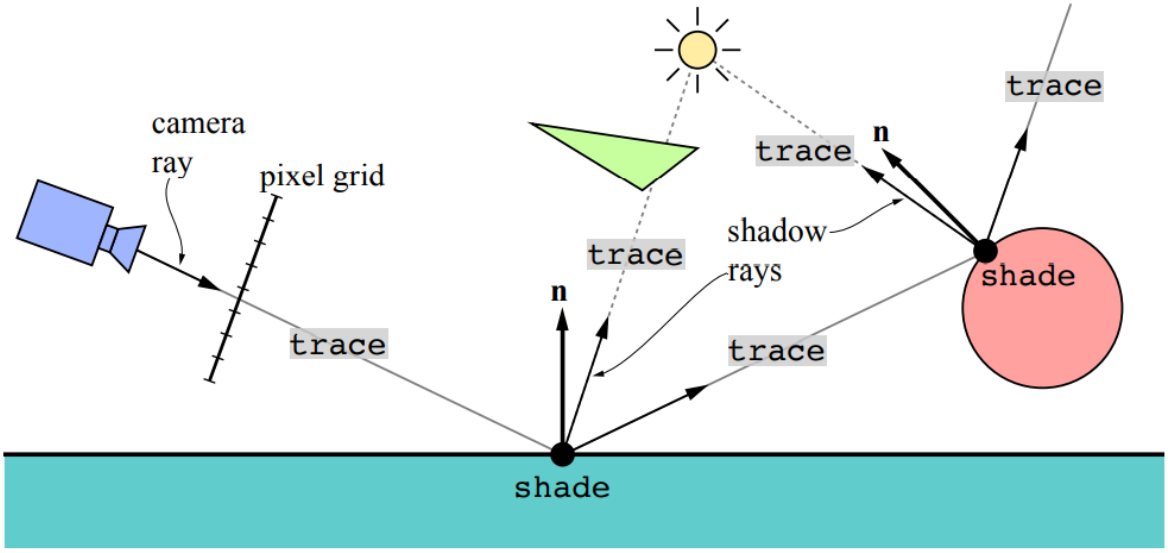
Rysunek 3.1 Ustawienia kamery i promienia [5]

By określić kolor piksela należy wystrzelić promień z kamery przez piksel obrazu i obliczyć jego kolor na podstawie zebranych danych. Takie promienie nazywane sa promieniami kamery. Ustawienie kamery jest zobrazowane na Rysunek 3.1. Dla danych całkowitych współrzędnych piksela (x, y), gdzie x rośnie w prawo, a y w lewo od lewego górnego rogu obrazu, pozycji kamery (ang. camera) $\boldsymbol{c}$, wektorów kamery $\{\boldsymbol{r}, \boldsymbol{u}, \boldsymbol{v}\}$ (prawy (ang. **r**ight), do góry (ang. **u**p), widoku (ang. **v**iew)) oraz rozdzielczości obrazu $w \times h$ (ang. **w**idth, **h**eight) obliczane są zmienne do wzoru promienia $\boldsymbol{ray}(t) = \boldsymbol{o} + t\boldsymbol{d}$

$$\begin{aligned} \boldsymbol{o} &= \boldsymbol{c} \\ \boldsymbol{temp}(x, y) &= af\left(\tfrac{2(x+0.5)}{w} - 1\right)\boldsymbol{r} - f\left(\tfrac{2(y+0.5)}{h} - 1\right)\boldsymbol{u} + \boldsymbol{v}, \\ \boldsymbol{d}(x, y) &= \frac{\boldsymbol{temp}(x,y)}{\|\boldsymbol{temp}(x,y)\|} \end{aligned} \tag{3.1}$$

gdzie $\boldsymbol{d}$ to znormalizowany wektor kierunku promienia i jest zależny od $f = \tan(\phi/2)$, przy czym ϕ to pionowe pole widzenia kamery, natomiast $a = w/h$ określa proporcje obrazu. Ponadto $\boldsymbol{temp}$ jest wektorem tymczasowym (ang. temporary) użytym w celu znormalizowania wektora kierunku kamery. Do współrzędnych piksela (x, y) dodawana jest stała wartość 0.5 w celu wyśrodkowania promienia wewnątrz piksela.

W naiwnej implementacji fazy śledzenia dla każdego piksela wykonywano by test przecięcia promienia z każdym z n trójkątów w scenie. Taki algorytm prezentowałby złożoność obliczeniową $O(n)$, co jest nieakceptowanie wolne i absolutnie nie nadawałoby się do użytku. Dlatego by uzyskać znacznie lepszą wydajność rzędu $O(\log n)$, należy wykorzystać przestrzenną strukturę danych taką jak drzewo brył ograniczających (BHV) czy drzewo k-wymiarowe (*k*-d tree). Struktura ta jest nazywana przyspieszającą (ang. acceleration structure), ponieważ pozwala znacznie przyspieszyć działanie algorytmu.



Rysunek 3.2 Ilustracja śledzenia i cieniowania przez promień [5]

Rozumiejąc fazy śledzenia i cieniowania, opisanie programu śledzącego promienie jest już stosunkowo proste. Równanie 3.1 jest wykorzystane do stworzenia promienia, który rozpoczyna się w pozycji kamery i przechodzi przez środek piksela. Promień ten jest wykorzystany w fazie śledzenia do znalezienia punktu trafienia, dla którego zostanie obliczony kolor w fazie cieniowania. Proces ten zilustrowany jest na Rysunek 3.2. Bardzo ważnym aspektem jest tutaj fakt, iż faza cieniowania może wywołać dodatkowe fazy śledzenia innych promieni, bo otrzymać dodatkowe informacje o trafionym punkcie. Te nowe promienie mogą posłużyć do określenia np. naświetlenia lub zacieniowania danego punktu, do rekursywnego śledzenia odbicia lustrzanego lub refrakcji. Głębokość promienia to termin określający, ile promieni zostało rekursywnie śledzonych. Jest on bardzo przydatny, ponieważ im wyższa jest maksymalna głębokość promienia tym lepszą jakość obrazu można uzyskać, natomiast dzieje się to kosztem wydajności, dlatego jest ważnym czynnikiem optymalizacyjnym. Na Rysunek 3.2 promień wychodzący z kamery ma głębokość równą 1, natomiast dalszy odbity od płaskiej podłogi i trafiający w czerwoną kulę ma głębokość 2.

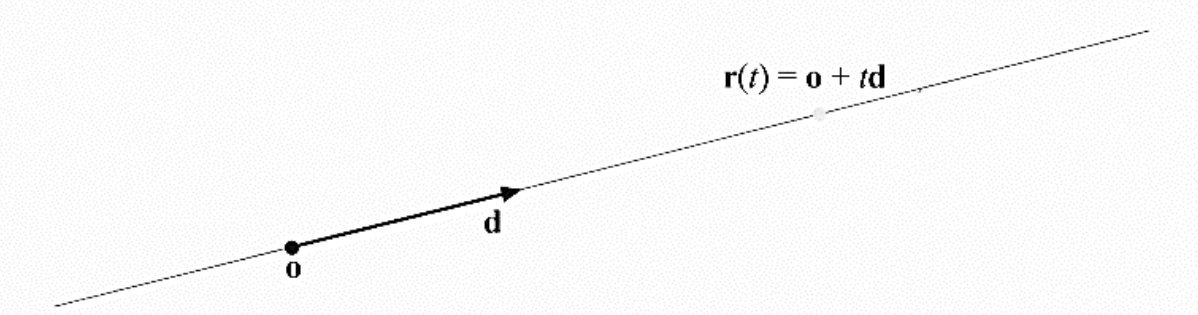
3.2 Test przecięcia promienia z trójkątem

Fundamentem, który sprawia, że wszystkie algorytmy śledzenia promieni mogą działać jest możliwość wyznaczenia punktu geometrii na scenie, w który promień trafia. Służy do tego test przecięcia promienia z trójkątem geometrii, jednak przed rozpoczęciem rozważań na ten temat należy zdefiniować równania promienia oraz trójkąta.

Promień, można opisać wzorem:

$$\boldsymbol{ray}(t) = \boldsymbol{o} + t\boldsymbol{d} \tag{3.2}$$

gdzie $\boldsymbol{o}$ to punkt początkowy (ang. origin), a $\boldsymbol{d}$ to znormalizowany wektor kierunkowy (ang. direction), natomiast t jest skalarem określającym konkretny punkt na półprostej opisanej równaniem promienia i jest to zilustrowane na Rysunek 3.3.



Rysunek 3.3 Ilustracja promienia [5]

Punkt opisany na trójkącie przedstawia się wzorem:

$$\boldsymbol{f}(u,v) = (1-u-v)\boldsymbol{p_0} + u\boldsymbol{p_1} + v\boldsymbol{p_2} \tag{3.3}$$

gdzie (u, v) to współrzędne barycentryczne, a więc takie które spełniają warunki $u \geq 0$, $v \geq 0$ oraz $u + v \leq 1$. Warto zauważyć, że są to te same współrzędne używane do mapowania tekstur czy interpolacji koloru.

W tym momencie obliczanie punktu przecięcia danego promienia $\boldsymbol{ray}(t)$ z trójkątem $\boldsymbol{f}(u,v)$ sprowadza się do rozwiązania równania $\boldsymbol{ray}(t) = \boldsymbol{f}(u,v)$ co zgodnie ze wzorami 3.1 i 3.2 rozwija się do postaci równania 3.4.

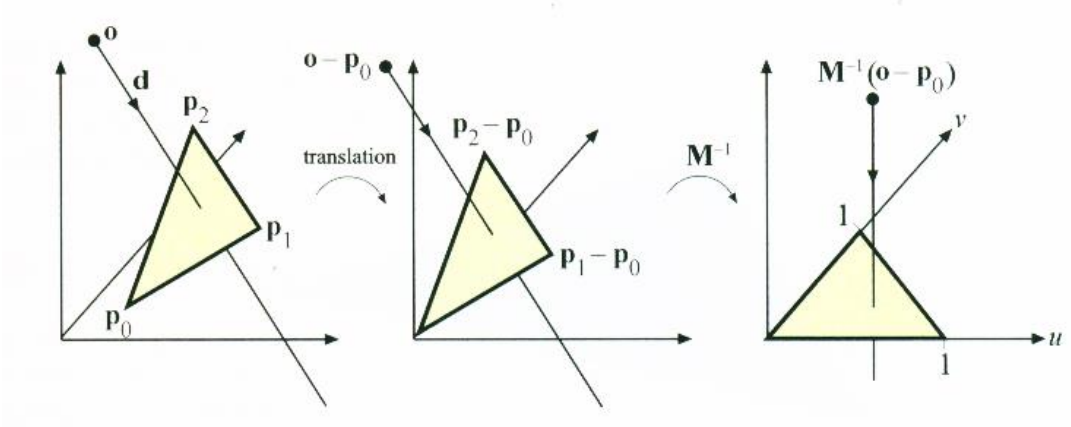
$$\boldsymbol{o} + t\boldsymbol{d} = (1-u-v)\boldsymbol{p_0} + u\boldsymbol{p_1} + v\boldsymbol{p_2} \tag{3.4}$$

Równanie 3.4 można z kolei przekształcić do postaci 3.5.

$$(-\boldsymbol{d} \quad \boldsymbol{p_1} - \boldsymbol{p_0} \quad \boldsymbol{p_2} - \boldsymbol{p_0}) \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \boldsymbol{o} - \boldsymbol{p_0} \tag{3.5}$$

W rezultacie współrzędne barycentryczne (u, v) oraz odległość t od punktu początkowego promienia można obliczyć rozwiązując otrzymany układu równań liniowych 3.5.

Przekształcenia te mogą zostać zobrazowane geometrycznie jako przesunięcie trójkąta do początku układu współrzędnych oraz jego transformacja do trójkąta jednostkowego na płaszczyźnie yz, gdzie kierunek d promienia jest równoległy do osi x. Ilustruje to Rysunek 3.4. Jeżeli $M = (-d \quad p_1 - p_0 \quad p_2 - p_0)$ jest macierzą równania 3.5, wtedy rozwiązanie można uzyskać mnożąc równanie 3.5 z macierzą M^{-1}.



Rysunek 3.4 Przekształcenie trójkąta i promienia [5]

Oznaczając przez $e_1 = p_1 - p_0$, $e_2 = p_2 - p_0$ oraz $s = o - p_0$, rozwiązanie równania 3.4 można uzyskać wykorzystując metodę Cramera:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{det(-\boldsymbol{d}, \boldsymbol{e_1}, \boldsymbol{e_2})} \begin{pmatrix} \det(\boldsymbol{s}, \boldsymbol{e_1}, \boldsymbol{e_2}) \\ \det(-\boldsymbol{d}, \boldsymbol{s}, \boldsymbol{e_2}) \\ \det(-\boldsymbol{d}, \boldsymbol{e_1}, \boldsymbol{s}) \end{pmatrix} \tag{3.5}$$

Z algebry liniowej wiadomo, że $\det(a, b, c) = |a\ b\ c| = -(a \times c) \cdot b = -(c \times b) \cdot a$. Wykorzystanie wspomnianej równości w równaniu 3.5 pozwala uzyskać:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\boldsymbol{d} \times \boldsymbol{e_2}) \cdot \boldsymbol{e_1}} \begin{pmatrix} (\boldsymbol{s} \times \boldsymbol{e_1}) \cdot \boldsymbol{e_2} \\ (\boldsymbol{d} \times \boldsymbol{e_2}) \cdot \boldsymbol{s} \\ (\boldsymbol{s} \times \boldsymbol{e_1}) \cdot \boldsymbol{d} \end{pmatrix} = \frac{1}{\boldsymbol{q} \cdot \boldsymbol{e_1}} \begin{pmatrix} \boldsymbol{r} \cdot \boldsymbol{e_2} \\ \boldsymbol{q} \cdot \boldsymbol{s} \\ \boldsymbol{r} \cdot \boldsymbol{d} \end{pmatrix}, \tag{3.6}$$

gdzie $q = d \times e_2$ oraz $r = s \times e_1$. Ponieważ te iloczyny wektorowe są wykorzystywane więcej niż raz, zapisanie ich w zmiennych może przyspieszyć obliczenia.

Jeśli można do obliczeń wykorzystać więcej pamięci, by zredukować liczbę kalkulacji, to równanie 3.6 można również zapisać w postaci równania 3.7.

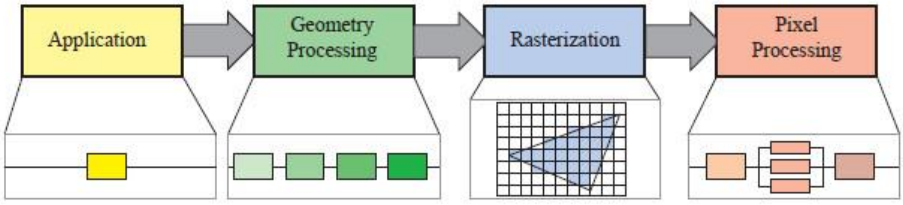
$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\boldsymbol{d} \times \boldsymbol{e_2}) \cdot \boldsymbol{e_1}} \begin{pmatrix} (\boldsymbol{s} \times \boldsymbol{e_1}) \cdot \boldsymbol{e_2} \\ (\boldsymbol{d} \times \boldsymbol{e_2}) \cdot \boldsymbol{s} \\ (\boldsymbol{s} \times \boldsymbol{e_1}) \cdot \boldsymbol{d} \end{pmatrix} =$$

$$= \frac{1}{-(\boldsymbol{e_1} \times \boldsymbol{e_2}) \cdot \boldsymbol{d}} \begin{pmatrix} (\boldsymbol{e_1} \times \boldsymbol{e_2}) \cdot \boldsymbol{s} \\ (\boldsymbol{s} \times \boldsymbol{d}) \cdot \boldsymbol{e_2} \\ -(\boldsymbol{s} \times \boldsymbol{d}) \cdot \boldsymbol{e_1} \end{pmatrix} = \frac{1}{-\boldsymbol{n} \cdot \boldsymbol{d}} \begin{pmatrix} \boldsymbol{n} \cdot \boldsymbol{s} \\ \boldsymbol{m} \cdot \boldsymbol{e_2} \\ -\boldsymbol{m} \cdot \boldsymbol{e_1} \end{pmatrix}, \tag{3.7}$$

Gdzie $n = e_1 \times e_2$ to nieznormalizowany wektor normalny trójkąta, a więc wartość stała (dla geometrii statycznej), oraz $m = s \times d$. Jeśli zapiszemy p_0, e_1, e_2 oraz n dla każdego trójkąta można uniknąć wielu obliczeń przecięcia promienia i trójkąta. Największy przyrost wydajności osiągany jest poprzez unikanie obliczania iloczynu wektorowego. Należy zauważyć tutaj jednak, że przeciwstawia się to pierwotnemu założeniu algorytmu, żeby przechowywać minimalną ilość informacji o trójkącie. Niemniej jednak, jeśli szybkość obliczeń jest najważniejsza, to jest to rozsądna alternatywa. Należy jednak również wziąć pod uwagę czy dodatkowy dostęp do pamięci nie będzie bardziej kosztowny wydajnościowo aniżeli dodatkowe obliczenia. Rozstrzygnięcie tej kwestii wymaga bardzo uważnych testów, gdzie niewykluczone, iż różny sprzęt oraz oprogramowanie będą miały również wpływ na ich wynik.

4 Klasyczny potok renderingu

Klasyczny potok renderingu składa się z kilku faz: aplikacji, przetwarzania geometrii, rasteryzacji oraz przetwarzania pikseli. Przedstawia to Rysunek 4.1.



Rysunek 4.1 Fazy klasycznego potoku renderingu [6]

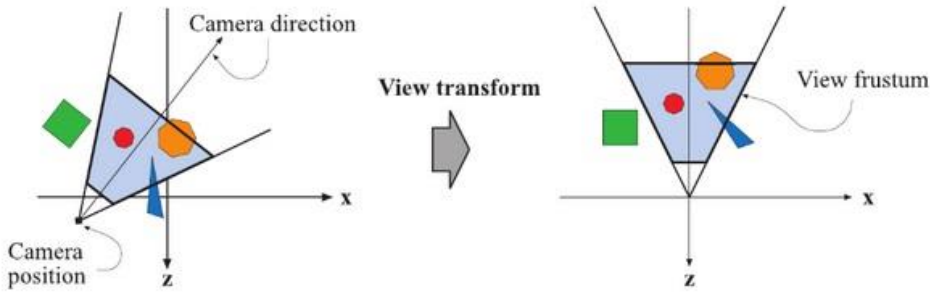
4.1 Faza aplikacji

Podczas tej fazy dostarczane są dane o scenie, obiektach, materiałach, które uznane zostaną za ważne. Prostą optymalizacją tutaj jest zastosowanie algorytmu usuwania obiektów, które nie znajdują się w bryle widzenia kamery (ang. frustum culling), a więc takich które na pewno nie znajdą się na ekranie, ponieważ są np. położone za kamerą. Dzięki temu do przetworzenia jest wysyłane znacznie mniej obiektów, a obraz zostanie wygenerowany szybciej.

4.2 Faza przetwarzania geometrii

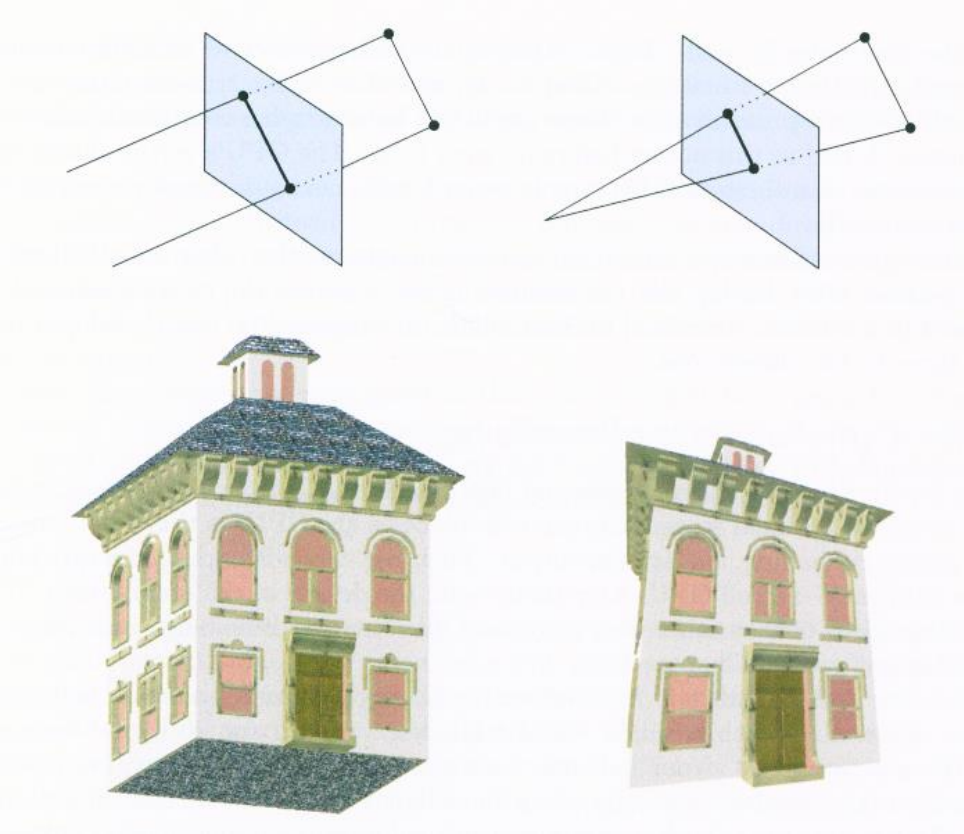
W fazie tej pojedyncze wierzchołki obiektów, trójkątów ulegają różnego typu transformacjom, projekcji czy odrzuceniu, jeśli nie spełnią odpowiednich warunków. Ogólnie faza ta decyduje co, jak i gdzie zostanie później narysowane.

W fazie tej następuje transformacja wierzchołków z przestrzeni świata do przestrzeni kamery, tak by kamera znajdowała się w środku układu współrzędnych, a jej wektory $\{\boldsymbol{r}, \boldsymbol{u}, \boldsymbol{v}\}$ (3.1) odpowiadały kierunkom osi współrzędnych $\{\boldsymbol{x}, \boldsymbol{y}, -\boldsymbol{z}\}$. Obrazuje to Rysunek 4.2. Wektor widoku $\boldsymbol{v}$, może rówież wskazywać w odwrotnym kierunku, czyli $+\boldsymbol{z}$, różnica jest głównie tylko semantyczna.



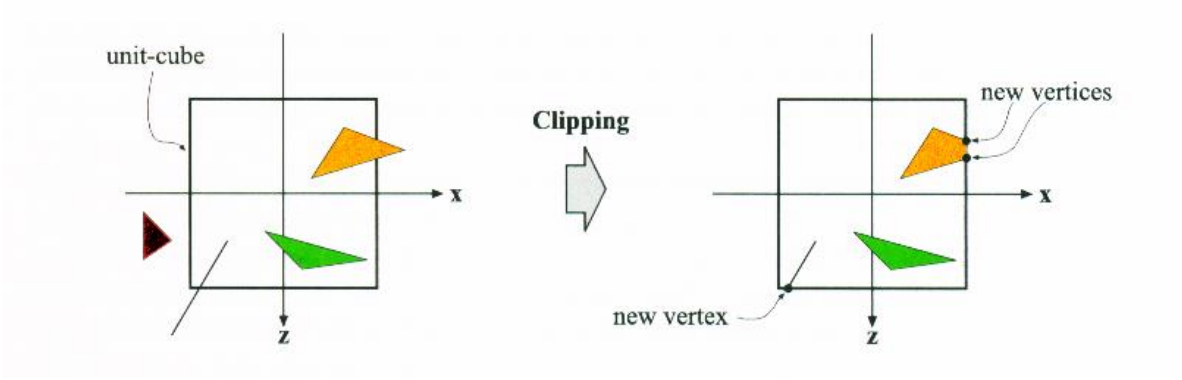
Rysunek 4.2 Transformacja widoku [6]

W tej fazie następuje też projekcja, czyli rzutowanie obiektów w pewien określony sposób. Najczęściej używana jest projekcja perspektywiczna (widoczna na Rysunek 4.3 po prawej stronie), ponieważ jest ona z godna z tym jak widzi człowiek. Niemniej jednak swoje zastosowanie ma również projekcja ortograficzna (widoczna na Rysunek 4.3 po lewej stronie), ponieważ jest przydatna np. do podglądu mapy w widoku z góry na dół (ang. top-down view), czy też przy generowaniu mapowania cieni.



Rysunek 4.3 Projekcja ortograficzna (po lewej) oraz perspektywiczna (po prawej) [6]

Następuje tutaj również proces obcinania (ang. clipping), kiedy to część geometrii, która nie znajduje się we frustrum widzenia kamery, zostaje odrzucona (czerwony trójkąt na Rysunek 4.4) lub obcięta, poprzez odrzucenie wierzchołków znajdujących się poza obrazem i wygenerowanie nowych na krawędzi obrazu (pomarańczowy trójkąt na Rysunek 4.4).



Rysunek 4.4 Odcinanie geometrii poza obszarem renderowania [6]

Później następuje mapowanie ekranu (ang. screen mapping), gdzie obraz skalowany jest z początkowej proporcji (często 1:1) do proporcji obrazu na ekranie. Dla przykładu, jeśli aplikacja będzie renderować obraz o rozdzielczości 1920x1080 pikseli to proporcje tego obrazu będą wynosić 16:9.

4.3 Faza rasteryzacji

Podczas procesu rasteryzacji z każdych kolejnych 3 wierzchołków formowany jest trójkąt, a następnie znajdowane są piksele, które ten prymityw pokrywa. Można to zobrazować w pseudokodzie za pomocą podwójnej pętli:

```
for (T in triangles)
    for (P in pixels)
        determine if P is inside T
```

Rasteryzacja jest obecna w grafice komputerowej od dekad, a karty graficzne są cały czas rozwijane, by jak najwydajniej ją realizować, dlatego proces znajdowania pikseli jest zoptymalizowany tak, by nie sprawdzać każdego piksela obrazu dla kolejnych prymitywów. Dodatkowo w tej fazie wykonywany jest test bufora głębokości (ang. depth buffer test), który ustala czy obecnie przetwarzany piksel powinien nadpisać poprzedni. Dzięki temu obiekty nieprzezroczyste mogą być wysłane do renderowania w dowolnej kolejności.

4.4 Faza przetwarzania pikseli

Tutaj ostatecznie jest obliczany kolor każdego generowanego piksela obrazu. Oczywiście kolor ten jest nadpisywany, jeśli aktualnie rozpatrywany prymityw znajduje się bliżej kamery niż poprzedni. Jest to określane mianem testu głębokości (ang. z-buffer test). W tej fazie też może następować mieszanie nowo obliczonego koloru z poprzednim (półprzezroczystość).

5 Różnice pomiędzy śledzeniem promieni, a rasteryzacją

Techniczna różnica obu algorytmów opiera się na sposobie znajdywania pikseli pokrytych przez trójkąty geometrii. W ray tracingu znajduje je bezpośrednio promień, natomiast w klasycznym potoku geometria jest rzutowana z przestrzeni trójwymiarowej na dwuwymiarową przestrzeń obrazu.

Ponadto rasteryzacja jest znacznie bardziej wydajna od ray tracingu. Od dekad jest ona rozwijana, a sprzęt (karty graficzne) optymalizowane pod jej kątem. Ray tracing w kategorii algorytmu działającego w czasie rzeczywistym jest stosunkowo nowa, tak jak i urządzenia sprzętowo to wspierające, dlatego jej wydajność jest wciąż dość ograniczona.

Ray tracing ma znaczącą przewagę nad rasteryzacją w kwestii jakości generowanego obrazu. Za jego pomocą można w szybki i prosty sposób dodać do sceny cienie, odbicia lustrzane, półprzezroczystość, wystarczy wykorzystać dodatkowe promienie. Przy rasteryzacji każdy z tych przypadków wymaga dużo pracy i komplikuje kod. By osiągnąć odbicia lustrzane (i to tylko na względnie płaskiej powierzchni) należy wyrenderować scenę jeszcze raz (co znacznie wpływa na długość generowania obrazu) ustawiając kamerę w odpowiednim miejscu i pod odpowiednim kątem względem obserwatora. Można to ominąć sztuczką nazwaną SSR (ang. Screen Space Reflection), odbicie w przestrzeni ekranu, jednak jej wadą jest to, że odbija tylko to co widać na ekranie, a więc jeśli obserwator będzie widział kałuże i pół neonowego napisu na budynku, to odbicie w kałuży będzie urwane i przełamie iluzję realności sceny. Zaimplementowanie cieni również wymaga wyrenderowania sceny ponownie jednak jest zdecydowanie bardziej skomplikowane. Render musi być wykonany z punktu, w którym znajduje się źródło światła i w odpowiednim rzucie (np. ortograficznym dla światła kierunkowego pochodzącego ze słońca). Następnie trzeba przeprowadzić skompilowane operacje, aby nałożyć te cienie na wygenerowany obraz. W przypadku półprzezroczystości wszystkie obiekty w scenie (a przynajmniej wszystkie półprzezroczyste) musza zostać posortowane i wysłane do wyrenderowania w kolejności od najdalszego względem kamery do najbliższego, ponieważ w innym przypadku kolory te nie zostaną połączone i będą się zastępować, wyświetlając błędny obraz.

Ważną różnicą jest również to, iż ray tracing do renderowania obrazu potrzebuje mieć w pamięci wszystkie dane dotyczące sceny. Rasteryzator potrzebuje do działania tylko jednego prymitywu na raz, co pozwala na dosyłanie (ang. stream) danych i optymalizację systemów zarządzania pamięcią.

6 Rendering oparty na fizyce (PBR)

PBR (ang. physically based rendering), tłumaczony na język polski jako rendering oparty na fizyce, jest zbiorem technik renderowania, które mają za zadanie naśladować fizyczne zachowanie światła, by w rezultacie generować obrazy bardziej realistyczne niż przy użyciu modeli oświetlenia Phonga czy Blinna-Phonga.

Aby model oświetlenia był uznany za bazujący na fizyce musi spełnić 3 warunki:

1. Opierać się na modelu powierzchniowym mikrościanek,
2. Podlegać zasadzie zachowania energii,
3. Używać fizycznego modelu funkcji BRDF.

Tekstury i materiały, tworzone dla renderera PBR, bazują na fizycznych właściwościach powierzchni, dzięki czemu będą one zawsze wyglądały poprawnie niezależnie od warunków oświetleniowych, co nie jest gwarantowane w żadnym innym modelu oświetlenia.

6.1 Model mikrościanek

Teoria mikrościanek (ang. microfacets) zakłada, że w mikroskopowym przybliżeniu, każda powierzchnia składa się z małych lusterek, które doskonale odbijają światło. Ułożenie tych mikrościanek może się znacznie różnić i opisywane jest przez parametr szorstkości.

Im bardziej szorstka (ang. rough) jest powierzchnia, tym bardziej chaotycznie ułożona jest każda mikrościanka na powierzchni. Światło padające na taką powierzchnię ulegnie rozproszeniu. W przypadku gładkiej (ang. smooth) powierzchni, promienie świetlne ulegną bardziej regularnemu odbiciu, skutkując mniejszym i ostrzejszym odbiciem światła. Obrazuje to Rysunek 6.1.

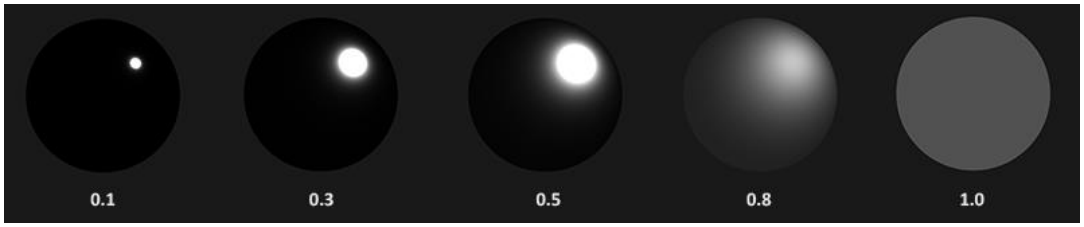
Rysunek 6.1 Powierzchnie szorstka (po lewej) i gładka (po prawej) [7]

Żadna powierzchnia nie jest całkowicie gładka na poziomie mikroskopowym, niemniej jednak wiedząc, że mikrościanki są na tyle małe, iż nie jest możliwe rozróżnienie ich na poziomie pikselowym, można statystycznie przybliżyć tą nierówność za pomocą parametru chropowatości (ang. roughness). W zależności od tego parametru można obliczyć stosunek mikrościanek podobnych do wektora połowicznego $\boldsymbol{h}$, który znajduje się w połowie odległości między wektorem światła $\boldsymbol{l}$, a wektorem patrzenia $\boldsymbol{v}$ i obliczany jest jako suma wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$, podzielona przez jego długość:

$$\boldsymbol{h} = \frac{\boldsymbol{l}+\boldsymbol{v}}{\|\boldsymbol{l}+\boldsymbol{v}\|} \tag{6.1}$$

Gdzie: $\boldsymbol{h}$ – wektor połowiczny, $\boldsymbol{l}$ – wektor światła, $\boldsymbol{v}$ – wektor patrzenia.

Im bardziej mikrościanki są przybliżone do wektora połowicznego, tym ostrzejszy i silniejszy rozbłysk. Wraz z parametrem chropowatości, który przyjmuje zakres między 0 a 1, można statystycznie określić wyrównanie mikrościanek.



Rysunek 6.2 Porównanie rozproszenia światła dla zmiennej chropowatości [7]

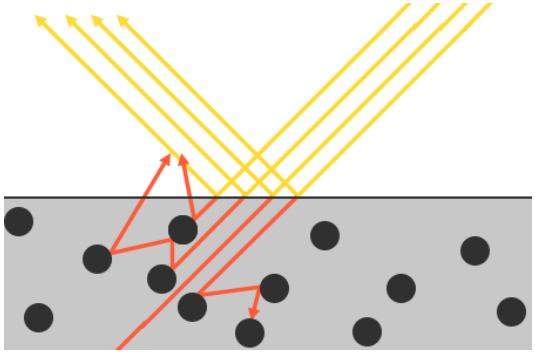
Rysunek 6.2 jasno przedstawia, iż wyższe wartości chropowatości wykazują znacznie większe rozproszenie światła na obiekcie, w przeciwieństwie do mniejszego i ostrzejszego odbicia lustrzanego występującego na gładkich powierzchniach.

6.2 Zachowanie energii

Teoria mikrościanek wykorzystuje zasadę zachowania energii, co oznacza, że energia światła wychodzącego nigdy nie może przekroczyć energii światła przychodzącego (nie dotyczy materiałów/powierzchni emitujących światło). Jak można zauważyć na Rysunek 6.2, jeśli powierzchnia, na której światło ulega odbiciu wzrasta (co jest wynikiem wzrastającej wartości parametru chropowatości), to maleje intensywność światła odbitego. Gdyby wartość światła odbitego miałaby być taka sama dla każdego piksela to powierzchnie szorstkie emitowałyby więcej energii niż do nich dociera, łamiąc zasadę zachowania energii. Dlatego na gładkich powierzchniach odbicia są znacznie bardziej intensywne, a na chropowatych przygaszone.

By zasada zachowania energii była spełniona, należy jednoznacznie rozróżnić światło rozproszone i odbite. Promień światła uderzając w powierzchnię, dzieli się na dwie części: jedna ulega refrakcji, a druga odbiciu. Pierwsza z nich załamuje się i dostaje się pod powierzchnię, a jej energia zostaje pochłonięta przez materiał obiektu, powodując oświetlenie rozproszone. Druga z nich odbija się od powierzchni nie tracąc swojej energii, dając w efekcie oświetlenie zwierciadlane.

Należy tutaj jednak wspomnieć, iż światło po załamaniu nie zostaje pochłonięte od razu. Zgodnie z zasadami fizyki, światło możemy potraktować jako wiązkę energii, która porusza się, dopóki nie wytraci całej swojej energii, a wytraca ją poprzez kolizje z cząsteczkami materii. Każdy obiekt składa się z takich małych cząsteczek, które pochłaniają część lub całość energii promienia przy każdym trafieniu w cząsteczkę, co ukazuje Rysunek 6.3.



Rysunek 6.3 Odbijanie i pochłanianie światła przez powierzchnię [7]

Promienie światła ulegające refrakcji zazwyczaj nie są w całkowicie pochłaniane przez materiał, ale odbijają się od cząsteczek pod powierzchnią, dopóki ich energia nie zostanie wyczerpana lub nie wydostaną się poza obiekt. Promienie, które wydostały się spod powierzchni wpływają na postrzegany kolor obiektu. W renderingu opartym na fizyce, ze względu na skomplikowanie obliczeń, przyjmuje się jednak, że całe światło załamane zostaje pochłonięte oraz rozproszone na bardzo niewielkim obszarze, ignorując przypadki, gdy promienie wydostają się spod powierzchni w dalszej odległości od miejsca trafienia w obiekt. Efekt ten można osiągnąć poprzez zastosowanie specjalnej metody znanej pod nazwą rozpraszanie podpowierzchniowe (ang. subsurface scattering) i pozwala znacznie poprawić jakość generowanego obrazu dla niektórych materiałów, takich jak skóra czy wosk, niestety pochłania ona dużo mocy obliczeniowej.

Należy nadmienić, iż takie zjawisko występuje w przypadku powierzchni niemetalicznych (dielektryki) i nie ma zastosowania do powierzchni metalicznych. Na nich całe światło, które uległo refrakcji, zostaje pochłonięte i nie wykazuje żadnego rozproszenia. Ze względu na tą różnicę metale i dielektryki są inaczej traktowane w modelu PBR.

Równanie odbicia światła

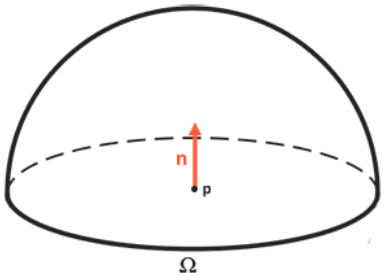
Fundamentem oświetlenia modelu PBR jest równanie odbicia światła (ang. reflectance equation):

$$\boldsymbol{L_o(p, \omega_o)} = \int_{\boldsymbol{\Omega}} \boldsymbol{f_r(p, \omega_i, \omega_o)} \cdot \boldsymbol{L_i(p, \omega_i)} \cdot (\boldsymbol{n} \cdot \boldsymbol{\omega_i})\, \boldsymbol{d\omega_i} \tag{6.2}$$

Gdzie:
$\boldsymbol{L_o}$ – suma radiancji świateł z punktu $\boldsymbol{p}$ w kierunku obserwatora $\boldsymbol{\omega_o}$,
$\boldsymbol{\Omega}$ – przestrzeń hemisfery na punkcie $\boldsymbol{p}$ na którym badamy irradiancję,
$\boldsymbol{f_r}$ – dwukierunkowa funkcja rozkładu odbicia,
$\boldsymbol{L_i}$ – radiancja światła,
$(\boldsymbol{n} \cdot \boldsymbol{\omega_i})$ – cos kąta pomiędzy wektorem normalnym a kierunkiem radiancji światła.

Wynikiem tego równania jest kolor $\boldsymbol{L_o}$ obserwowanego w przestrzeni punktu $\boldsymbol{p}$, a składa się na niego kilka zmiennych.

Przede wszystkim na postrzegany kolor ma wpływ jego oświetlenie. Ponieważ światło nadchodzi ze wszystkich kierunków to, aby je poznać należy obliczyć natężanie promieniowania z każdego kierunku na hemisferze $\boldsymbol{\Omega}$ wokół obserwowanego punktu $\boldsymbol{p}$, co obrazuje Rysunek 6.4.



Rysunek 6.4 Hemisfera $\boldsymbol{\Omega}$ wokół punktu $\boldsymbol{p}$ [7]

Natężenie promieniowania opisuje energię źródła światła zgromadzoną nad rzutowanym obszarem na jednostkowej sferze i dane jest wzorem:

$$\boldsymbol{I} = \frac{\boldsymbol{d\phi}}{\boldsymbol{d\omega}} \tag{6.3}$$

Gdzie:

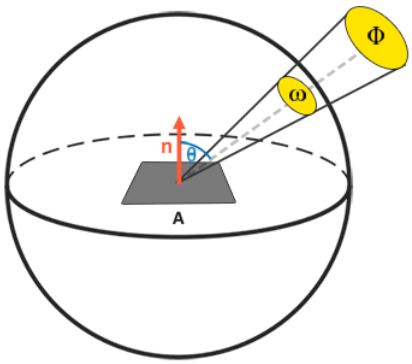
$\boldsymbol{\phi}$ – strumień promieniowania, który jest wartością fizyczną określającą energię przenoszoną przez fale elektromagnetyczne. W grafice komputerowej takie podejście byłoby niepraktyczne i dlatego upraszczając, energię tą koduje się jako trójkolorowy wektor światła RGB.

$\boldsymbol{\omega}$ – kąt bryłowy opisujący wielkość kształtu rzutowanego na jednostkową sferę, a który można rozumieć jako kierunek z objętością.

Wykorzystawszy zmienne z równania 6.3 można teraz zdefiniować radiancję $\boldsymbol{L}$ jako całkowitą energię zgromadzoną na obszarze $\boldsymbol{A}$ na kąt bryłowy $\boldsymbol{\omega}$ światła o strumieniu promieniowania $\boldsymbol{\phi}$ i przedstawić wzorem oraz zobrazować na Rysunek 6.5.

$$\boldsymbol{L} = \frac{\boldsymbol{d^2\phi}}{\boldsymbol{dA\, d\omega\, cos\theta}} \tag{6.4}$$

Gdzie:
$\boldsymbol{L}$ – radiancja światła,
$\boldsymbol{A}$ – obszar powierzchni,
$\boldsymbol{\phi}$ – strumień promieniowania,
$\boldsymbol{\omega}$ – kąt bryłowy,
$\boldsymbol{\theta}$ – kąt padania światła względem wektora normalnego.



Rysunek 6.5 Radiancja na obszarze $\boldsymbol{A}$ [7]

Jak można zauważyć we wzorze 6.4, radiancja jest skalowane przez $cos\theta$ – kosinus kąta między wektorem normalnym powierzchni, a wektorem kierunku światła. Dzieję się tak ponieważ światło jest najsilniejsze, gdy pada prostopadle na powierzchnię i maleje, gdy kąt między tymi wektorami rośnie.

Aby radiancja była użyteczna we wzorze 6.2, należy przyjąć jeszcze, że kąt bryłowy $\boldsymbol{\omega}$ oraz obszarze $\boldsymbol{A}$ są nieskończenie małe i zamieniają się w wektor kierunku światła $\boldsymbol{\omega}$ oraz punkt $\boldsymbol{p}$. Dzięki temu można obliczyć radiancję dla pojedynczego promienia padającego na konkretny punkt w przestrzeni.

W równaniu odbicia światła 3.2 pojawia się całka $\int_{\boldsymbol{\Omega}}$, ponieważ pozwala ona wykonać obliczenia dla wszystkich takich pojedynczych promieni docierających na ten punkt $\boldsymbol{p}$ poprzez hemisferę $\boldsymbol{\Omega}$.

6.3 BRDF

Dwukierunkowa funkcja rozkładu odbicia (ang. bidirectional reflective distribution function, BRDF) jest funkcją, która skaluje nadchodzącą radiancję w oparciu o właściwości materiału powierzchni. Na przykład, jeśli powierzchnia ma idealnie gładką powierzchnię (jak lustro), funkcja BRDF zwróci 0.0 dla wszystkich przychodzących promieni światła $\boldsymbol{\omega_i}$ z wyjątkiem jednego promienia, który ma ten sam kąt (odbicia) jak wychodzący promień $\boldsymbol{\omega_o}$, dla którego funkcja zwraca 1.0.

Celem funkcji jest przybliżenie właściwości odbijających i refrakcyjnych materiału w oparciu o wcześniej omawianą teorię mikrościanek. Aby BRDF był oparty na fizyce, musi przestrzegać prawa zachowania energii, tj. suma odbitego światła nigdy nie może przekraczać ilości światła przychodzącego. Choć w grafice komputerowej jest kilka takich funkcji to prawie wszystkie potoki renderowania w czasie rzeczywistym są oparte na Cook-Torrance BRDF, którą można opisać równaniem 6.5.

$$\boldsymbol{f_r = k_d f_{lambert} + k_s f_{cook-torrance}} \tag{6.5}$$

Gdzie $\boldsymbol{k_d}$ jest współczynnikiem określającym ilość światła rozproszonego, natomiast $\boldsymbol{k_s}$ jest współczynnkiem światła odbitego. $\boldsymbol{k_d}$ skaluje rozproszenie Lamberta określone równaniem 6.6.

$$\boldsymbol{f_{lambert} = \frac{c}{\pi}} \tag{6.6}$$

Gdzie $\boldsymbol{c}$ to albedo, kolor powierzchni. Jest ono podzielone przez $\boldsymbol{\pi}$ w celu znormalizowania światła rozproszonego, ponieważ całka $\boldsymbol{L_o}$ jest skalowana przez $\boldsymbol{\pi}$.

Istnieją również inne równania lepiej opisujące światło rozproszone, jednak są one bardziej wymagające obliczeniowo, a na potrzeby aplikacji czasu rzeczywistego rozproszenie Lambertowskie jest zupełnie wystarczające.

Część odbita światła jest bardziej zaawansowana i opisana równaniem 6.7.

$$\boldsymbol{f_{cook-torrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}} \tag{6.7}$$

Równanie 6.7 składa się z trzech funkcji w liczniku oraz współczynnika normalizującego w mianowniku. Celem każdej z funkcji opisanych symbolami $\boldsymbol{D}$, $\boldsymbol{F}$, $\boldsymbol{G}$, jest przybliżenie właściwości odbicia lustrzanego powierzchni.

$\boldsymbol{D}$ – funkcja **D**ystrybucji wektorów normalnych (ang. normal **D**istribution function), podstawowa funkcja przybliżania mikrościanek, opisująca w jaki sposób mikrościanki na powierzchni są wyrównane do wektora połowicznego (opisany w rozdziale 6.1), na który wpływ ma chropowatość powierzchni.

$\boldsymbol{F}$ – równanie **F**resnel'a, opisuje współczynnik odbicia powierzchni przy różnych kątach.

$\boldsymbol{G}$ – funkcja **G**eometrii, opisuje właściwość samo-zacieniania mikrościanek, występującą na stosunkowo szorstkich powierzchniach, gdy jedne mikrościanki rzucają cień na inne, redukując w ten sposób ilość odbitego go światła.

Powyższe funkcje mają wiele wersji, jedne są bardziej realistyczne inne wydajniejsze. Badania nad różnymi typami aproksymacji przeprowadził Brian Karis z Epic Games [8]. W badaniach można odnaleźć funkcje używane w Unreal Engine 4 (jednym z najpopularniejszych silników do tworzenia aplikacji 3D), dlatego też ich wersje zostaną opisane dalej, a są to: Trowbridge-Reitz GGX dla $\boldsymbol{D}$, przybliżenie Fresnela-Schlicka dla $\boldsymbol{F}$ oraz Smitha-Schlicka-GGX dla $\boldsymbol{G}$.

6.4 Funkcja dystrybucji wektorów normalnych

Rozważana w tej pracy wersja funkcji dystrybucji wektorów normalnych nazywana jest Trowbridge-Reitz GGX i przyjmuje postać równania 6.8.

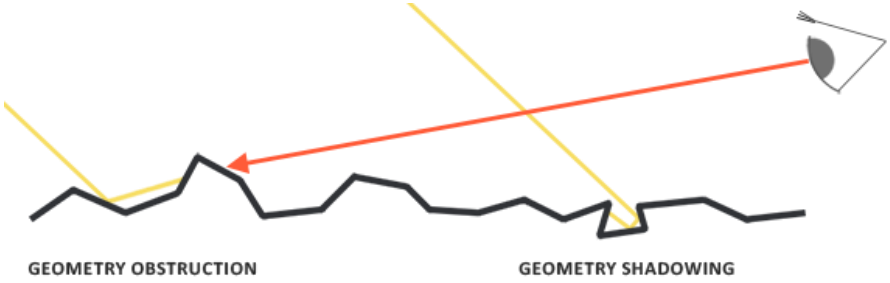
$$\boldsymbol{NDF_{GGX_{TR}}}(\boldsymbol{n},\boldsymbol{h},\boldsymbol{r}) = \frac{\boldsymbol{r^2}}{\boldsymbol{\pi((n\cdot h)^2(r^2-1)+1)^2}} \tag{6.8}$$

Gdzie:
$\boldsymbol{n}$ – wektor normalny,
$\boldsymbol{h}$ – wektor połowiczny,
$\boldsymbol{r}$ – wartość chropowatości**.**

Kiedy chropowatość jest niska (a więc powierzchnia jest gładka), znaczna ilość mikrościanek jest dopasowywana do wektorów połowicznych w małym promieniu. Ze względu na to wysokie zagęszczenie NDF wyświetla bardzo jasny punkt. Natomiast na chropowatej powierzchni, gdzie mikrościanki są ustawione w znacznie bardziej losowych kierunkach, znajduje się znacznie większa liczba wektorów połowicznych wyrównanych do mikrościanek, lecz mniej zagęszczonych co w efekcie daje słabszy połysk na większej powierzchni.

6.5 Funkcja geometrii

Funkcja geometrii statystycznie przybliża względne pole powierzchni, gdzie mikrościanki przesłaniają się nawzajem, powodując zasłonięcie promieni świetlnych przedstawione na Rysunek 6.6.



Rysunek 6.6 Samozaciemnianie się powierzchni [7]

Podobnie jak NDF funkcja geometrii przyjmuje parametr chropowatości materiału, gdzie bardziej szorstkie powierzchnie mają większe prawdopodobieństwo zacieniania mikrościanek. Rozważana funkcja to Schlick-GGX, która powstała z połącznia aproksymacji GGX i Schlick-Beckmann'a i przyjmuje postać równania 6.9.

$$\boldsymbol{G_{SchlickGGX}(n, v, k)} = \frac{\boldsymbol{n \cdot v}}{\boldsymbol{(n \cdot v)(1-k)+k}} \tag{6.9}$$

Gdzie $\boldsymbol{k}$ jest mapowaniem parametru $\boldsymbol{r}$ na odpowiednią wartość w zależności od zastosowanego oświetlenia. Równanie 6.10 jest stosowany dla bezpośredniego źródła światła, natomiast równanie 6.11 dla oświetlenia IBL (ang. Image Based Lightning), czyli opartego na obrazie (np. cubemapie).

$$\boldsymbol{k_{direct}} = \frac{\boldsymbol{(r+1)^2}}{\boldsymbol{8}} \tag{6.10}$$

$$\boldsymbol{k_{IBL}} = \frac{\boldsymbol{r^2}}{\boldsymbol{2}} \tag{6.11}$$

Aby skutecznie przybliżyć geometrię, należy wziąć pod uwagę zarówno kierunek patrzenia (zasłonięcie geometrii), jak i wektor kierunku światła (cienie geometrii). Metoda Smith'a uwzględnia obie te zmienne i wyraża się równaniem 6.12.

$$\boldsymbol{G(n, v, l, k)} = \boldsymbol{G_{SchlickGGX}(n, v, k) G_{SchlickGGX}(n, l, k)} \tag{6.12}$$

Implementacja tej metody, w połączeniu ze zmienną wartością chropowatości $\boldsymbol{r}$, daje rezultaty widoczne na Rysunek 6.7.

Rysunek 6.7 Wizualizacja funkcji geometrii dla zmiennej wartości chropowatości [7]

6.6 Równanie Fresnela

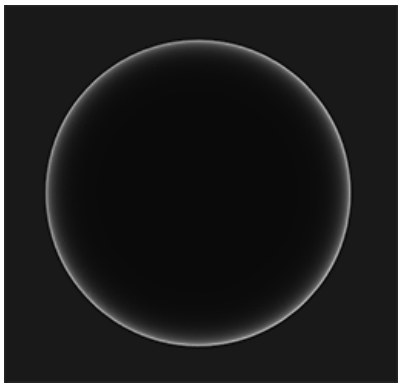
Równanie Fresnela opisuje stosunek światła odbitego do załamanego, w zależności od kąta pomiędzy wektorem patrzenia a wektorem normalnym powierzchni.

Każdy materiał ma podstawowy współczynnik odbicia $\boldsymbol{F_0}$, określany podczas obserwacji jego powierzchni idealnie prostopadle (pod kątem 0° do wektora normalnego jego powierzchni). Gdy kąt ten zaczyna rosnąć, powierzchnia coraz bardziej zaczyna odbijać promienie, aż do kąta 90°, kiedy to teoretycznie odbija całe docierające do niej światło. Zjawisko to jest znane pod nazwą efektu Fresnela i obrazuje je Rysunek 6.8.

Równanie efektu Fresnela jest matematycznie skomplikowane, dlatego na potrzeby grafiki komputerowej upraszcza się je używając aproksymacji Fresnel-Schlick'a, wyrażonego równaniem 6.13.

$$\boldsymbol{F_{Schlick}(h, v, F_0)} = \boldsymbol{F_0} + \boldsymbol{(1 - F_0)(1 - (h \cdot v))^5} \tag{6.13}$$

$\boldsymbol{F_0}$ reprezentuje podstawowy współczynnik odbicia powierzchni, który obliczany jest za pomocą współczynników refrakcji (ang. indices of refraction) lub IOR i co można zauważyć na Rysunek 6.8, im bardziej wektor patrzenia staje się prostopadły do wektora normalnego powierzchni sfery tym silniejszy jest efekt Fresnela, a tym samym odbicia.



Rysunek 6.8 Wizualizacja efektu Fresnela [7]

Równanie Fresnela ma jednak subtelności do których należy się odnieść. Przede wszystkim, przybliżenie Fresnela-Schlicka jest zdefiniowane tylko dla dielektryków (niemetali). Obliczając F_0 dla przewodników (metali) za pomocą ich współczynników refrakcji nie zachowują się one poprawnie i należy dla nich użyć innego równania Fresnela. Ze względu na tę właściwość, podstawowe współczynniki odbicia są wstępnie obliczane, a wartość funkcji interpolowana w zależności od kąta patrzenia, dzięki czemu równanie może być używane zarówno z materiałami metalicznymi jak i niemetalicznymi.

Podstawowe współczynniki odbicia powierzchni można znaleźć w dużych bazach danych [9]. Poniższa tabelka prezentuje wartości F_0 dla jednych z bardziej powszechnych materiałów:

Tabela 6.1 Wartości F_0 dla powszechnych materiałów [9]

Materiał	F_0 (Liniowy)	F_0 (sRGB)	Kolor
Woda	(0.02, 0.02, 0.02)	(0.15, 0.15, 0.15)	
Plastik / Szkło (niski)	(0.03, 0.03, 0.03)	(0.21, 0.21, 0.21)	
Plastik (wysoki)	(0.05, 0.05, 0.05)	(0.24, 0.24, 0.24)	
Szkło (wysoki) / Rubin	(0.08, 0.08, 0.08)	(0.31, 0.31, 0.31)	
Diament	(0.17, 0.17, 0.17)	(0.45, 0.45, 0.45)	
Żelazo	(0.56, 0.57, 0.58)	(0.77, 0.78, 0.78)	
Miedź	(0.95, 0.64, 0.54)	(0.98, 0.82, 0.76)	
Złoto	(1.00, 0.71, 0.29)	(1.00, 0.86, 0.57)	
Aluminium	(0.91, 0.92, 0.92)	(0.96, 0.96, 0.97)	
Srebro	(0.95, 0.93, 0.88)	(0.98, 0.97, 0.95)	

Warto zauważyć, że współczynnik odbicia podstawowego jest reprezentowany przez trójkolorowy wektor RGB oraz na to, iż dielektryki (niemetale) mają wszystkie 3 wartości równe, w przeciwieństwie do przewodników (metali). Jest to związane z tym, iż metale mogą w różnym stopniu odbijać konkretne pasma światła, przez co kolor odbitego światła może być zabarwiony.

6.7 Równanie odbicia Cook-Torrance'a

Zrozumiawszy BRDF oraz jej funkcje składowe można ją podstawić do równania odbicia światła 6.2 otrzymując równanie 6.14.

$$\boldsymbol{L_o(p,\omega_o)} = \int_{\boldsymbol{\Omega}} \left(\boldsymbol{k_d}\frac{\boldsymbol{c}}{\boldsymbol{\pi}} + \boldsymbol{k_s}\frac{\boldsymbol{DFG}}{\boldsymbol{4(\omega_o \cdot n)(\omega_i \cdot n)}}\right)\boldsymbol{L_i(p,\omega_i)n \cdot \omega_i d\omega_i} \tag{6.14}$$

Należy jednak zwrócić tutaj uwagę na to, że równanie nie jest matematycznie poprawne, ponieważ współczynnik światła odbitego $\boldsymbol{k_s}$ jest już uwzględniony w funkcji Fresnela $\boldsymbol{F}$. Należy więc usunąć ten współczynnik z równania, by otrzymać ostateczne równanie 6.15, poprawnego modelu oświetlenia opartego na fizyce (PBR).

$$\boldsymbol{L_o(p,\omega_o)} = \int_{\boldsymbol{\Omega}} \left(\boldsymbol{k_d}\frac{\boldsymbol{c}}{\boldsymbol{\pi}} + \frac{\boldsymbol{DFG}}{\boldsymbol{4(\omega_o \cdot n)(\omega_i \cdot n)}}\right)\boldsymbol{L_i(p,\omega_i)n \cdot \omega_i d\omega_i} \tag{6.15}$$

7 Implementcja

Wybór technologii Vulkan

Vulkan to niskopoziomowy, wieloplatformowy interfejs programowania aplikacji (ang. application programming interface, API) pozwalający komunikować się z kartą graficzną. Został stworzony i jest rozwijany przez Grupę Khronos – konsorcjum zrzeszające wiele firm branży IT, które zajmuje się tworzeniem otwartych standardów API. Jest on swoistym następcą starszego API, zwanego OpenGL, również zarządzanego przez to konsorcjum. Vulkan od swojego poprzednika różni się wieloma rzeczami, lepiej skaluje się z procesorami wielordzeniowymi czy zapewnia niższy narzut sterownika, dzięki czemu pozwala uzyskać lepszą wydajność. Najważniejszym jednak rozwinięciem jest możliwość wykorzystania sprzętowego wsparcia śledzenia promieni, co nie jest dostępne przy użyciu poprzednika. Jego bezpośrednim rywalem jest API Microsoftu, DirectX 12, jednak pozwala ono jedynie na pisanie aplikacji na platformy Windows oraz Xbox.

Vulkan został wybrany na potrzeby tego projektu ze względu na swoją wieloplatformowość oraz dlatego, że jestem zaznajomiony z językiem programowania shaderów GLSL, wykorzystywanym w tej technologii.

7.1 Wybór procesora graficznego (GPU)

Aby wykorzystać sprzętowe wsparcie śledzenia promieni udostępnione przez API Vulkan, należy posiadać kartę graficzną implementującą taką funkcjonalność. Ja w tym celu zaopatrzyłem się w laptopa z kartą graficzną NVIDIA RTX 2060, która spełnia te wymagania.

7.2 Podstawy technologii Vulkan

Vulkan jest niskopoziomowym API, które udostępnia bardzo rozbudowany dostęp do karty graficznej, jednak powoduje to, że kod wymagany do utworzenia i skonfigurowania aplikacji jest bardzo obszerny i zajmuje setki linii kodu. Z tego względu w mojej pracy znajdą się tylko wybrane fragmenty kodu, do których będę się odnosił szczegółowiej.

Mając na uwadze dwa różne niżej opisane potoki renderowania (rasteryzacja i ray tracing), między którymi można się przełączać w czasie działania aplikacji, program został napisany tak, by generować obraz jedną z tych metod i zapisywać go w teksturze, by w następnym kroku wyświetlić go za pomocą quad'a (prostokąta) na ekranie.

7.3 Konfiguracja początkowa renderera

Na początku należy utworzyć instancję aplikacji Vulkan, której przekazane są informacje o tym jakie rozszerzenia i warstwy urządzania są wymagane do działania aplikacji. Tak na przykład rozszerzenie "`VK_KHR_surface`" pozwoli na wyświetlanie obrazu w przestrzeni okna aplikacji. Na podstawie takich wymagań program odpytuje dostępne w systemie urządzania fizyczne (karty graficzne) i wybiera takie, które je spełnia najlepiej.

Dla wybranego urządzenia fizycznego tworzone są Kolejki (ang. Queues). Są to obiekty pozwalające na wykorzystanie możliwości urządzenia do generowania grafiki czy przeprowadzania obliczeń ogólnego przeznaczania. Zbierają one komendy (ang. commands), polecenia do wykonania przez GPU.

Do wykorzystania urządzenia fizycznego, tworzone jest urządzenie logiczne, interfejs, który decyduje jakie funkcjonalności urządzenia fizycznego zostaną aktywowane oraz pozwala na manipulowanie zasobami karty graficznej.

Następnie należy utworzyć okno systemowe, w którym będzie wyświetlana aplikacja. W tym celu została wykorzystana lekka biblioteka narzędziowa GLFW (ang. Graphics Library Framework), która podobnie jak Vulkan, jest wieloplatformowa i pozwala uzyskać dostęp do powierzchni (ang. surface) poprzez funkcję:

```
glfwCreateWindowSurface(instance, window, nullptr, &surface);
```

By wyświetlić wyrenderowaną scenę na uzyskanej powierzchni należy użyć struktury łańcucha wymiany (ang. swap chain). Jego podstawowym zadaniem jest zapewnienie, że obraz (struktura Vulkana `VkImage`) do którego renderowana jest scena, jest inny niż obecnie wyświetlany. Dzięki temu zawsze będzie wyświetlana pełna, kompletna wyrenderowana scena, a nie jej niedokończona część. Przy każdym renderze, łańcuch wymiany dostarcza nieużywany obraz, program renderuje do niego scenę, a następnie jest zwracany do łańcucha, by zostać podmieniony z obecnie wyświetlanym obrazem i pojawić się na ekranie.

Z obrazu tego nie można jednak korzystać bezpośrednio. Należy użyć strukturę widoku obrazu `VkImageView`, która definiuje, jak uzyskać dostęp do obrazu oraz jak uzyskać dostęp do konkretnych jego części. Widok obrazu należy z kolei opakować w bufor klatki `VkFramebuffer`, który może go zastosować w celu przechowania danych o kolorze, bufora głębokości (ang. depth buffer) czy bufora szablonowego (ang. stencil buffer). Bufor klatki może adresować wiele widoków.

Przejście renderowania (ang. render pass) jest kolejnym obiektem do utworzenia przed przejściem do etapu potoku graficznego (ang. graphics pipeline). Opisuje ono typy obrazów, które są używane operacji renderowania, opisuje jak obraz ma zostać użyty oraz jak traktować jego zawartość. Przejście renderowania jest więc jedynie opisem obrazu, a właściwy obraz zostaje powiązany z nim poprzez bufor klatki.

7.3.1 Potok graficzny (Graphics Pipeline)

Dwa podstawowe, programowalne etapy klasycznego potoku graficznego, które są użyte w tej aplikacji to program wierzchołków (ang. vertex shader) oraz program fragmentów (ang. fragment shader).

Dane napływające do potoku w formie wierzchołków, są dalej przekazywane do programu wierzchołków, gdzie ich pozycje są transformowane do formy znormalizowanych współrzędnych urządzenia (ang. normalized device coordinates, NDC). Głównym zadaniem tego etapu jest przekształcenie obiektu z przestrzeni świata do przestrzeni ekranu:

```
gl_Position = ubo.proj * ubo.view * vec4(worldPos, 1.0);
```

Później następuje rasteryzacja tych obiektów we fragmenty obrazu. Wywoływany jest wtedy program fragmentów, który oblicza oświetlenie i końcowy kolor piksela:

```
vec3 N = normalize(fragNormal);
vec3 diffuse = computeDiffuse(mat, L, N);
vec3 specular = computeSpecular(mat, viewDir, L, N);

// Result
outColor = vec4(lightIntensity * (diffuse + specular), 1);
```

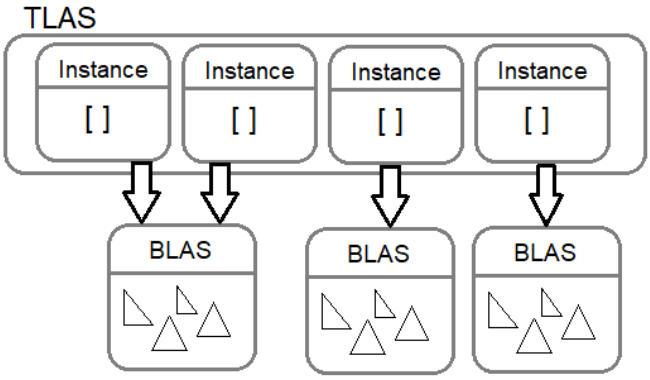
7.4 Rozszerzenie renderera o metodę śledzenia promieni

Podczas tworzenia instancji do wymaganych rozszerzeń należy dodać `"VK_KHR_ray_tracing_pipeline"`, które odblokuje funkcjonalności sprzętowego wsparcia śledzenia promieni oraz `"VK_KHR_acceleration_structure"`, które pozwoli na wykorzystanie struktur przyspieszających.

Struktury przyspieszające (ang. acceleration structure, AS) są niezwykle ważne dla wydajnego ray tracingu, ponieważ znacznie redukują liczbę testów przecięcia promienia z trójkątem. Zazwyczaj zaimplementowane są sprzętowo jako struktura hierarchiczna. W Vulkan API użytkownikowi udostępnione są dwa poziomy: pojedyncza struktura przyspieszająca wysokiego poziomu (ang. top-level acceleration structure, TLAS), która może wskazywać na dowolną liczbę struktur przyspieszających niskiego poziomu (ang. bottom-level acceleration structures, BLAS), aż do limitu określonego przez właściwości karty graficznej za pomocą `VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxInstanceCount`. Przyjęło się, iż BLAS odpowiada pojedynczym modelom 3D na scenie, natomiast TLAS odpowiada całej scenie poprzez macierze transformacji, które wskazują na BLAS'y i rozmieszczają je w przestrzeni sceny.

Struktury przyspieszające niskiego poziomu przechowują dane o wierzchołkach. Zbudowane są z jednego lub więcej buforów wierzchołków, gdzie każdy ma swoją własną macierz transformacji (niepowiązaną z macierzą w TLAS), która pozwala na przechowywanie wielu modeli rozmieszczonych wewnątrz jednego BLAS'u. Warto zauważyć, że jeśli obiekt jest tworzony kilka razy w tym samym BLAS'ie, jego geometria zostanie zduplikowana. Może to mieć pozytywny wpływ na wydajność na statycznych, nieinstancjonowanych elementach sceny. Z zasady im mniej struktur niskiego poziomu, tym wyższa wydajność.

Struktury przyspieszające wysokiego poziomu zawierają instancje obiektów, gdzie każdy ma swoją macierz transformacji oraz odnośnik do odpowiedniego BLAS'a. Dobrze całą tę zależność obrazuje to Rysunek 7.1.



Rysunek 7.1 Ilustracja struktur przyspieszających niskiego i wysokiego poziomu

Wykorzystując obiekty Vulkan API `VkAccelerationStructureGeometryKHR` oraz `VkAccelerationStructureGeometryTrianglesDataKHR`, do których zostają załadowane dane o wierzchołkach, a także ustawiając flagę `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR`, budowane są kolejne struktury niskiego poziomu dla każdego modelu występującego na scenie.

Następnie dla wszystkich instancji tych obiektów tworzone są struktury Vulkana `VkAccelerationStructureInstanceKHR`, zawierające macierz transformacji oraz odnośnik do struktury niskiego poziomu poprzez adres pamięci na karcie graficznej `VkDeviceAddress`. Wszystkie nowostworzone obiekty są pakowane do nowego obiektu typu `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR`, by zostać zbudowane do struktury przyspieszającej wysokiego poziomu.

7.4.1 Potok śledzenia promieni (Ray Tracing Pipeline)

W przeciwieństwie do potoku graficznego, podczas śledzenia promieni wszystkie programy cieniujące (shadery) muszą być dostępne jednocześnie, ponieważ decyzja o ich wykonywaniu jest podejmowana podczas działania aplikacji. W tym celu używana jest tabela programów cieniujących (ang. Shader Binding Table, SBT).

Skompilowane programy cieniujące potoku śledzenia promieni ładowane są do struktur `VkShaderModule`, a następnie pakowane są do struktury tworzącej potok `VkRayTracingPipelineCreateInfoKHR`. W czasie budowania potoku, kompilowana jest także tabela programów cieniujących, do której dostęp można uzyskać za pomocą funkcji `vkGetRayTracingShaderGroupHandlesKHR`. Następnie na karcie graficznej należy zaalokować pamięć z użyciem flagi VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR, by zaznaczyć jakie jest przeznaczenie tej pamięci, a następnie skopiować tam skompilowaną tabelę SBT.

Punktem początkowym algorytmu śledzenia promieni w Vulkan API jest program generujący promienie (ang. ray generation shader). Wykonywany jest on dla każdego piksela renderowanego obrazu i jak sama nazwa wskazuje, jego zadaniem jest obliczenie promienia wychodzącego z kamery i przechodzącego przez piksel obrazu. Wykorzystując wbudowaną funkcję uruchamiany jest algorytm śledzenia tego promienia:

```
traceRayEXT(topLevelAS,     // acceleration structure
            rayFlags,       // rayFlags
            0xFF,           // cullMask
            0,              // sbtRecordOffset
            0,              // sbtRecordStride
            0,              // missIndex
            origin.xyz,     // ray origin
            tMin,           // ray min range
            direction.xyz,  // ray direction
            tMax,           // ray max range
            0);             // payload (location = 0)
```

Wynik działania algorytmu jest przekazywany poprzez obiekt danych zwrotnych (ang. payload). Program wywołujący przekazuje swój obiekt jako rayPayloadEXT, natomiast program wywoływany odbiera go jako obiekt przychodzący rayPayloadInEXT i za jego pomocą oba programy się komunikują.

Jednym z programów wywoływanych jest program najbliższego trafienia (ang. closest hit shader). Wywoływany jest wtedy, gdy promień natrafi na najbliższą instancję geometrii. Jego zadaniem jest obliczenie oświetlenia i koloru trafionego punktu. Poza obliczonym kolorem może on zwrócić również dane do śledzenia promienia odbitego by uzyskać efekt odbicia lustrzanego:

```
vec3 origin       = worldPos;
vec3 rayDir       = reflect(gl_WorldRayDirectionEXT, normal);
payload.rayOrigin = origin;
payload.rayDir    = rayDir;
payload.done      = 0;
```

Innym programem wywoływanym może być program nietrafienia (ang. miss shader). Wykonywany jest wtedy, gdy promień nie przetnie żadnej geometrii na scenie. W tej aplikacji użyte zostały dwa tego typu programy.

Pierwszy z nich wywoływany jest, gdy promień obliczony przez program generujący promienie nie trafi na żadną geometrię, co oznacza, iż wskazuje na pustą przestrzeń i zwraca w obiekcie zwrotnym kolor wybrany dla tej przestrzeni przez użytkownika.

Drugi wykorzystywany jest we wcześniej wymienionym programie najbliższego trafienia, gdzie to z miejsca trafienia śledzony jest następny promień w kierunku położenia źródła światła. Jeśli promień nie trafi w żadną geometrię, znaczy to, że jest w pełni oświetlony i nie znajduje się w cieniu innego obiektu, a program nietrafienia przekazuje tę informację poprzez obiekt zwrotny:

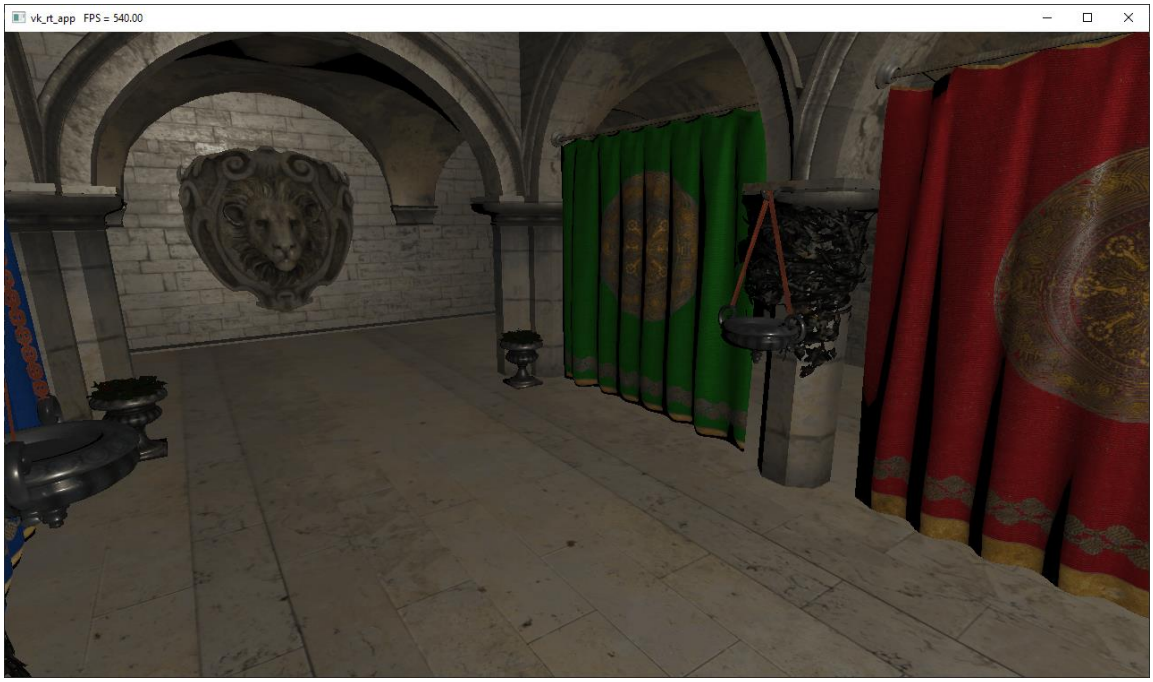
```
layout(location = 1) rayPayloadInEXT bool isShadowed;

void main()
{
  isShadowed = false;
}
```

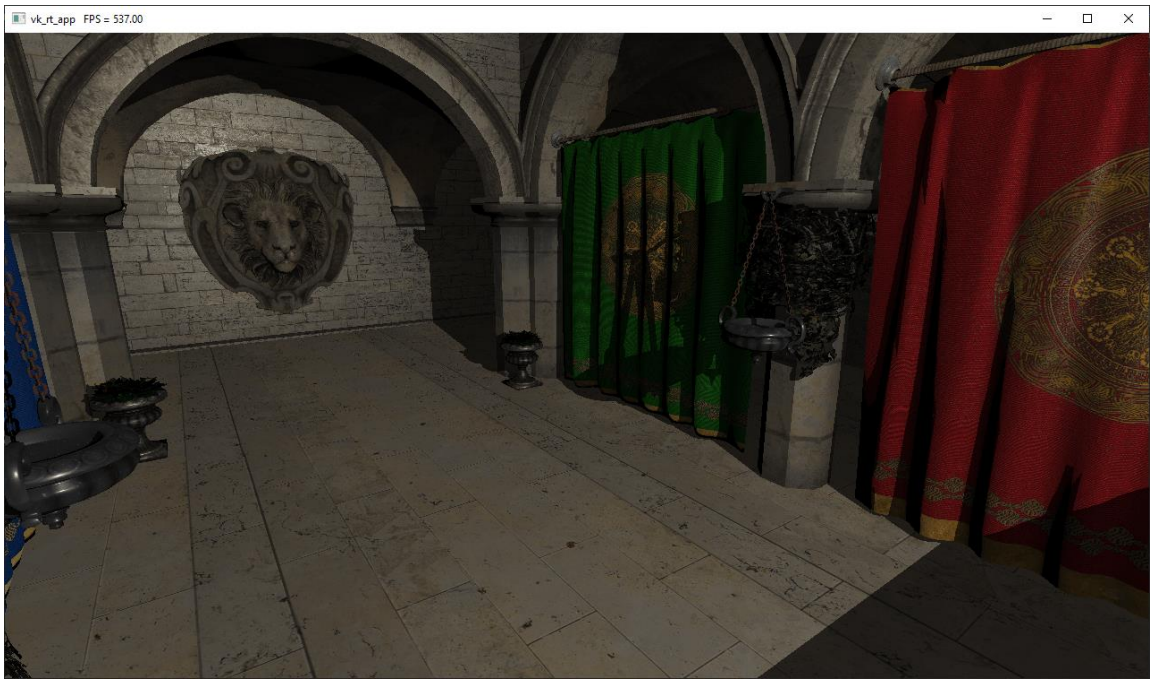
Wszystkie te elementy złożone razem tworzą potok, który przez każdy piksel obrazu śledzi promień, by określić kolor tego piksela. Jeśli nie trafi on w geometrię to zwraca kolor nieba. W przypadku przecięcia geometrii, z punktu trafiania generowany jest następny promień skierowany w kierunku światła, by uwzględnić w zwracanym kolorze czy piksel znajduje się w cieniu. Jeśli promień ten trafi w jakikolwiek model to znaczy, że znajduje się w cieniu. Ponadto, jeżeli promień przechodzący przez piksel obrazu trafi w obiekt o materiale lustrzanym to generowany jest następny promień z punktu trafienia pod kątem odbitym. Promień może być tak odbijany, aż do osiągnięcia maksymalnej dozwolonej głębokości odbicia.

7.5 Testy aplikacji

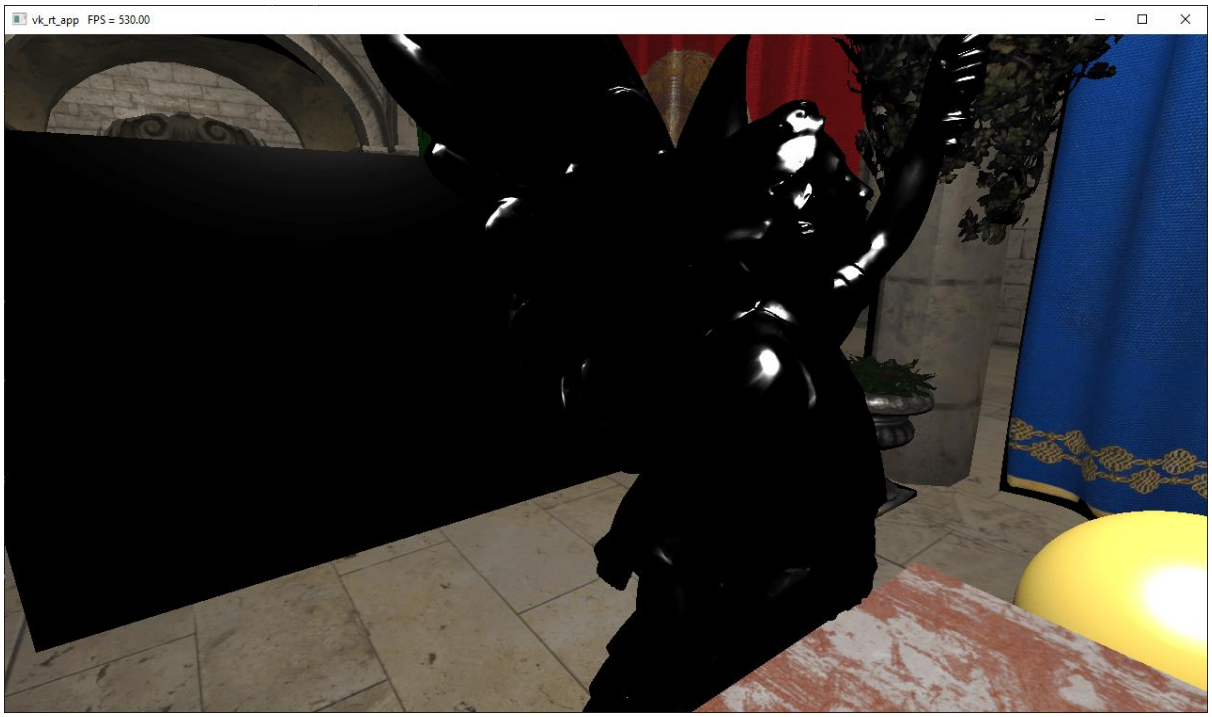
Testy aplikacji zostały przeprowadzone na laptopie z mobilnymi jednostkami CPU Intel Core i5-9300H oraz GPU NVIDIA GeForce RTX 2060. Poniższe zrzuty ekranu prezentują efekty działania programu w zależności od różnych ustawień renderera. Rozdzielczość generowanego obrazu to 1280x720 pikseli. Ich porównanie wizualne oraz wydajnościowe zostało opisane w następnym rozdziale. Należy odnotować, iż na scenie znajduje się jedno dynamiczne punktowe źródło światła, a cała geometria jest statyczna.



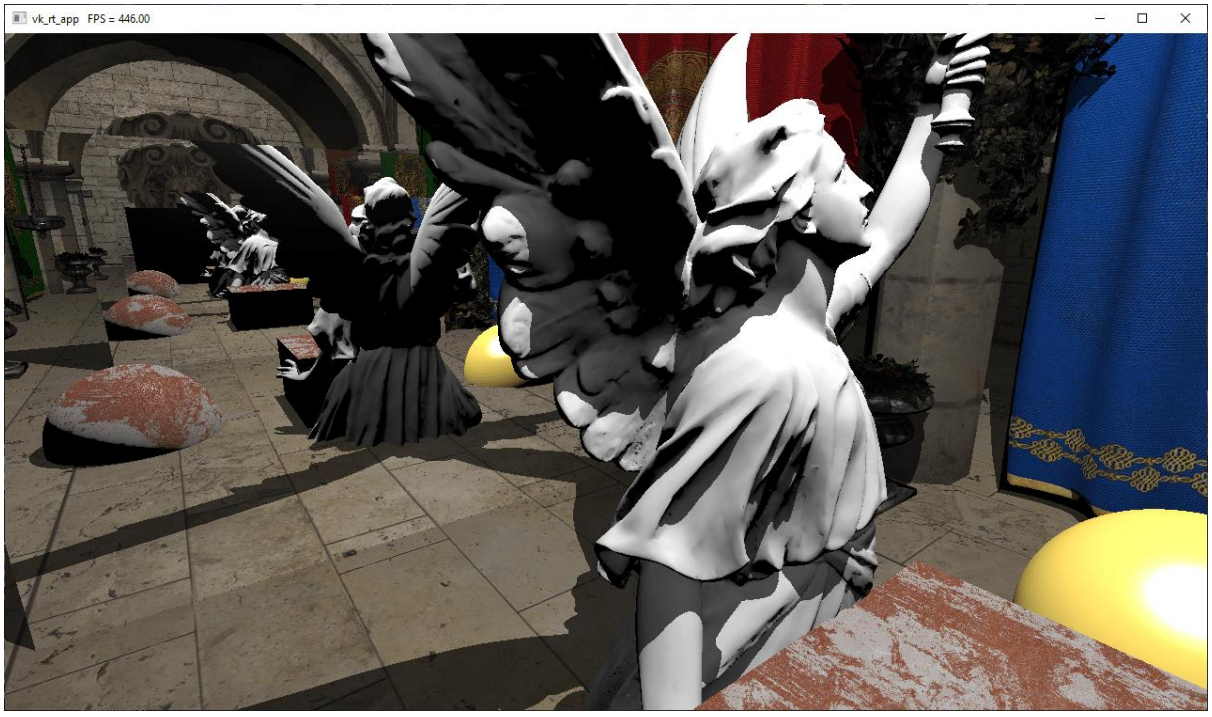
Rysunek 7.2 Ujęcie 1, rasteryzacja, brak cieni



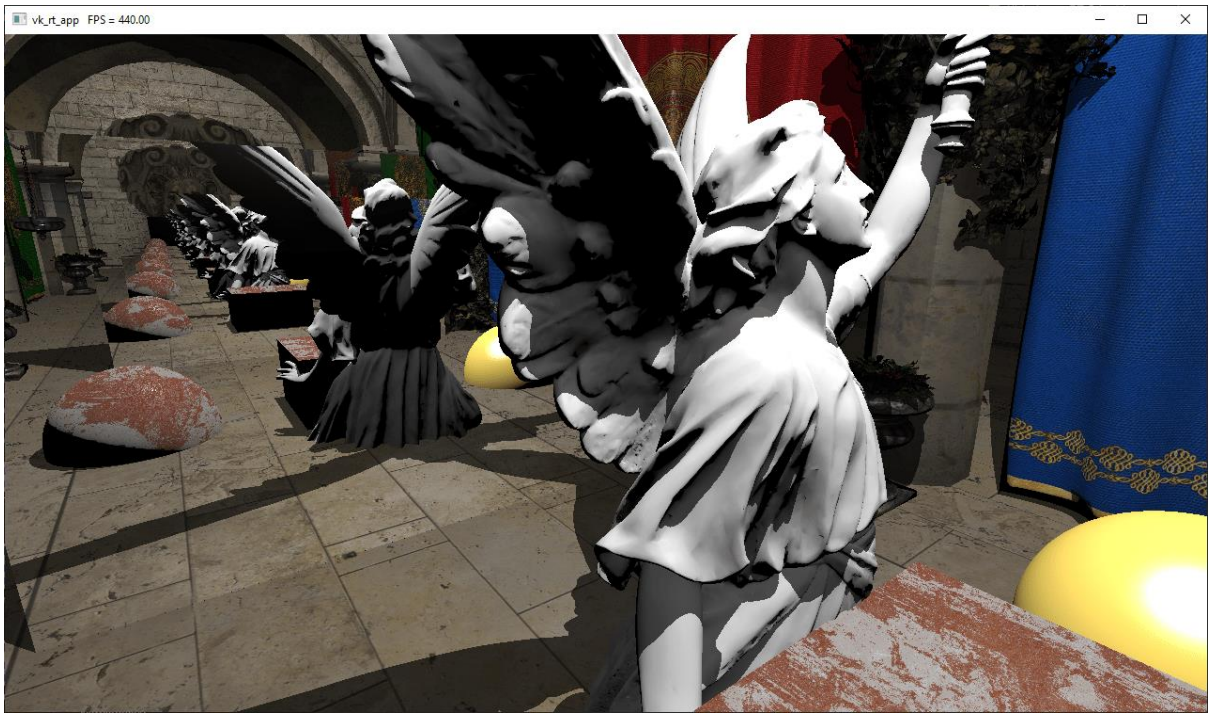
Rysunek 7.3 Ujęcie 1, ray tracing, trwarde 1-punktowe cienie



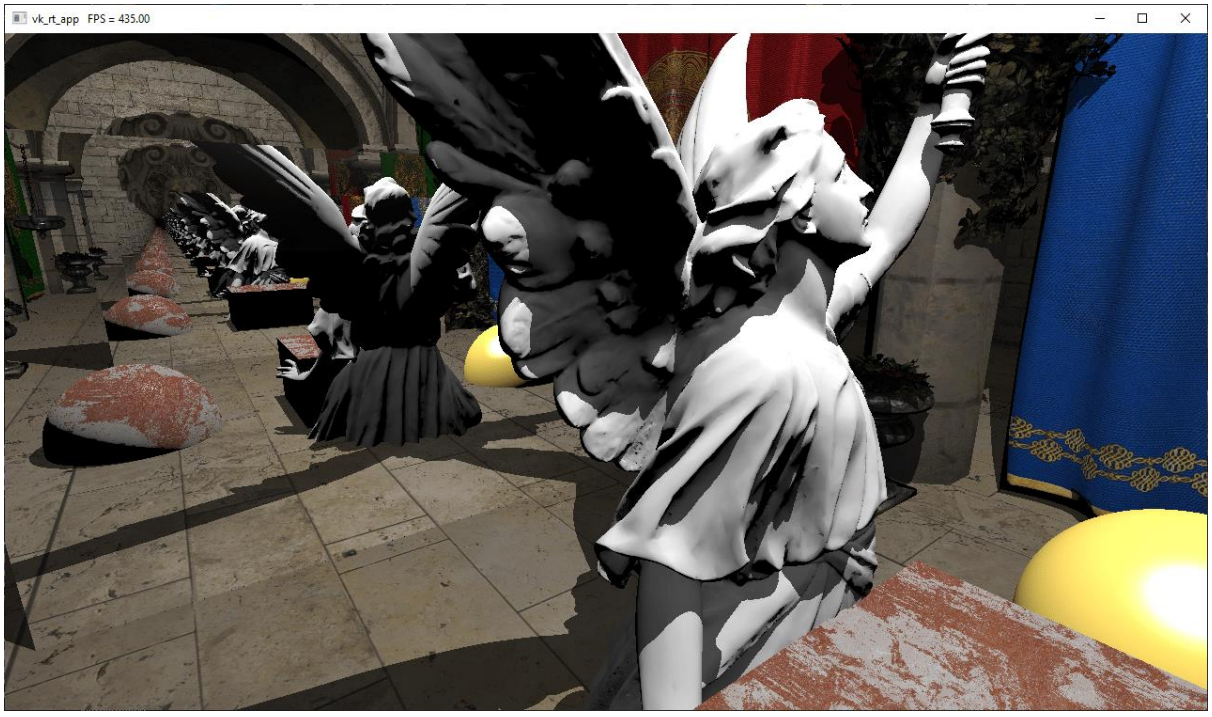
Rysunek 7.4 Ujęcie 2, rasteryzacja, brak odbić



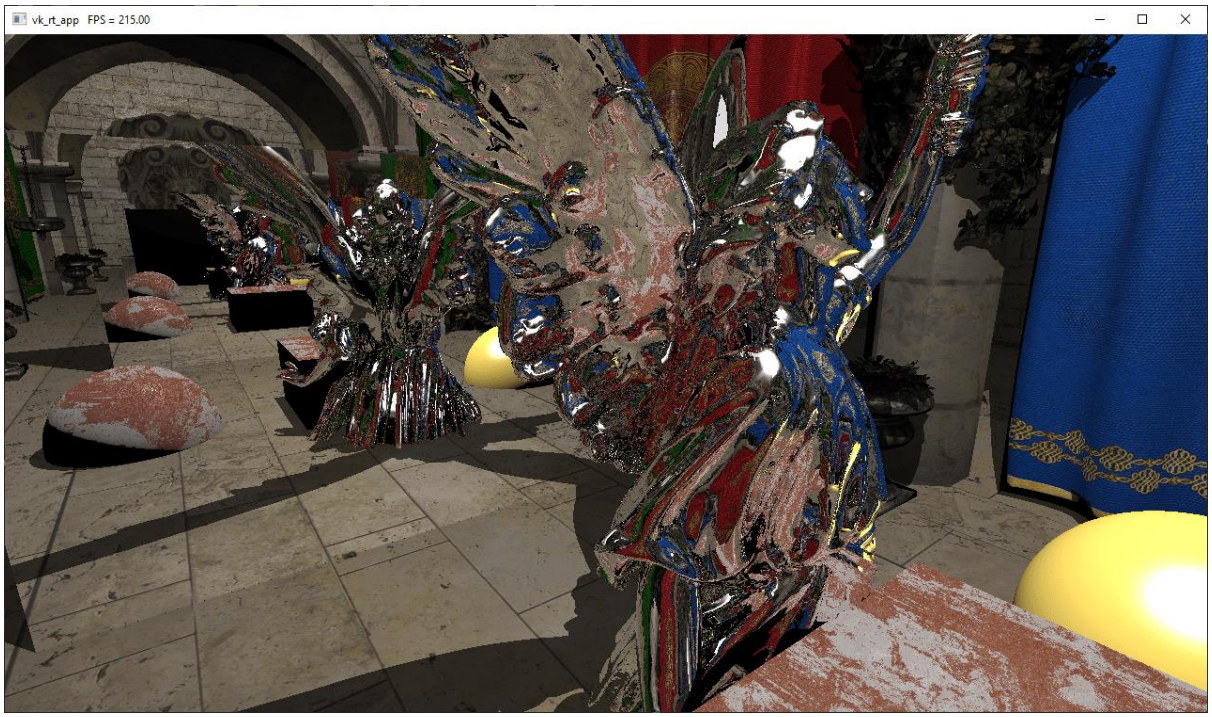
Rysunek 7.5 Ujęcie 2, ray tracing, głębokość odbić 4



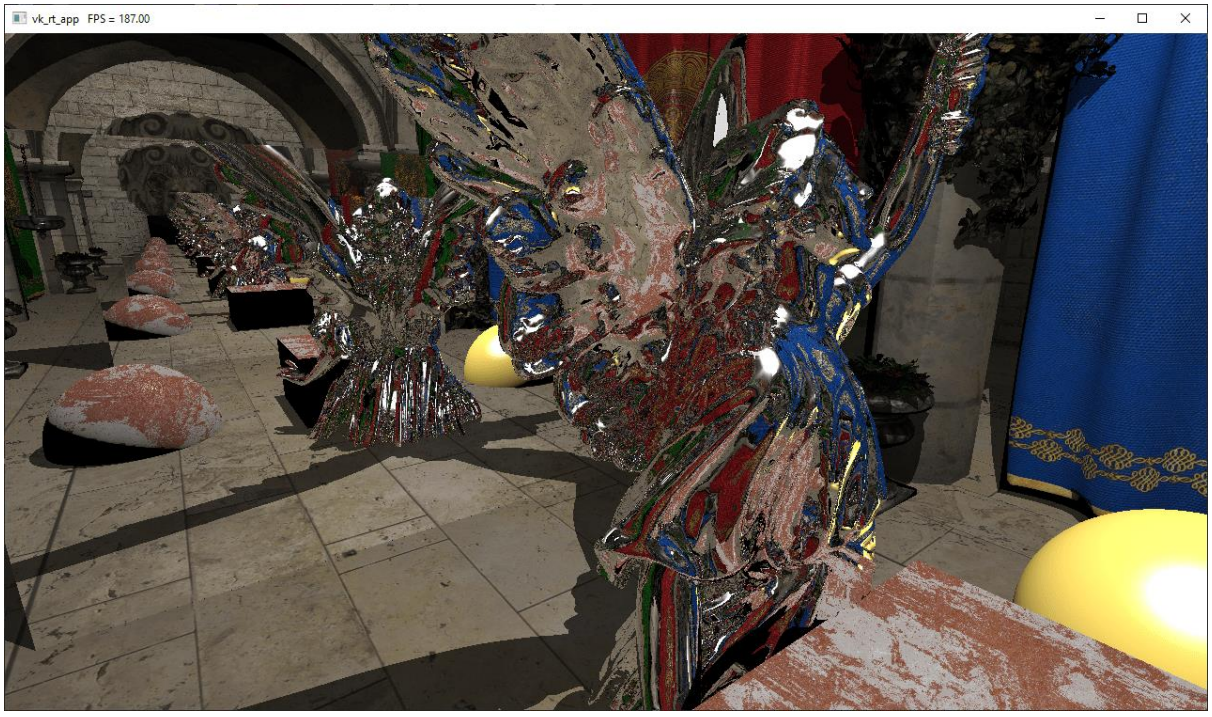
Rysunek 7.6 Ujęcie 2, ray tracing, głębokość odbić 10



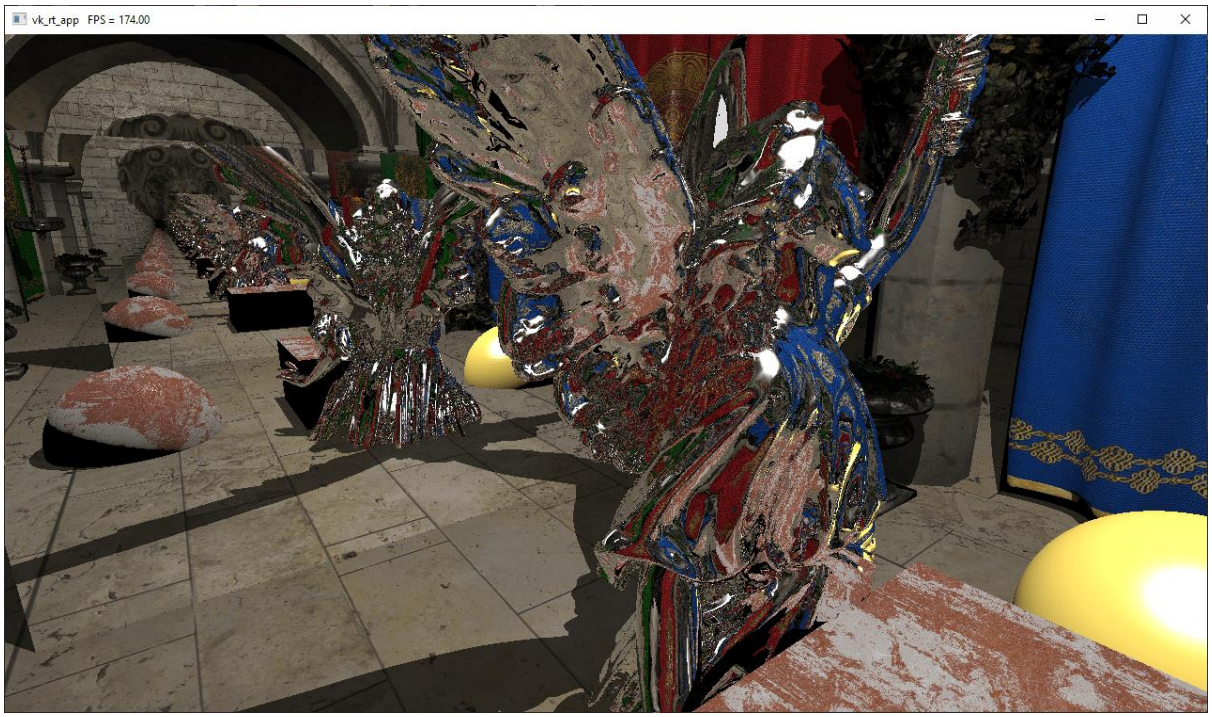
Rysunek 7.7 Ujęcie 2, ray tracing, głębokość odbić 25



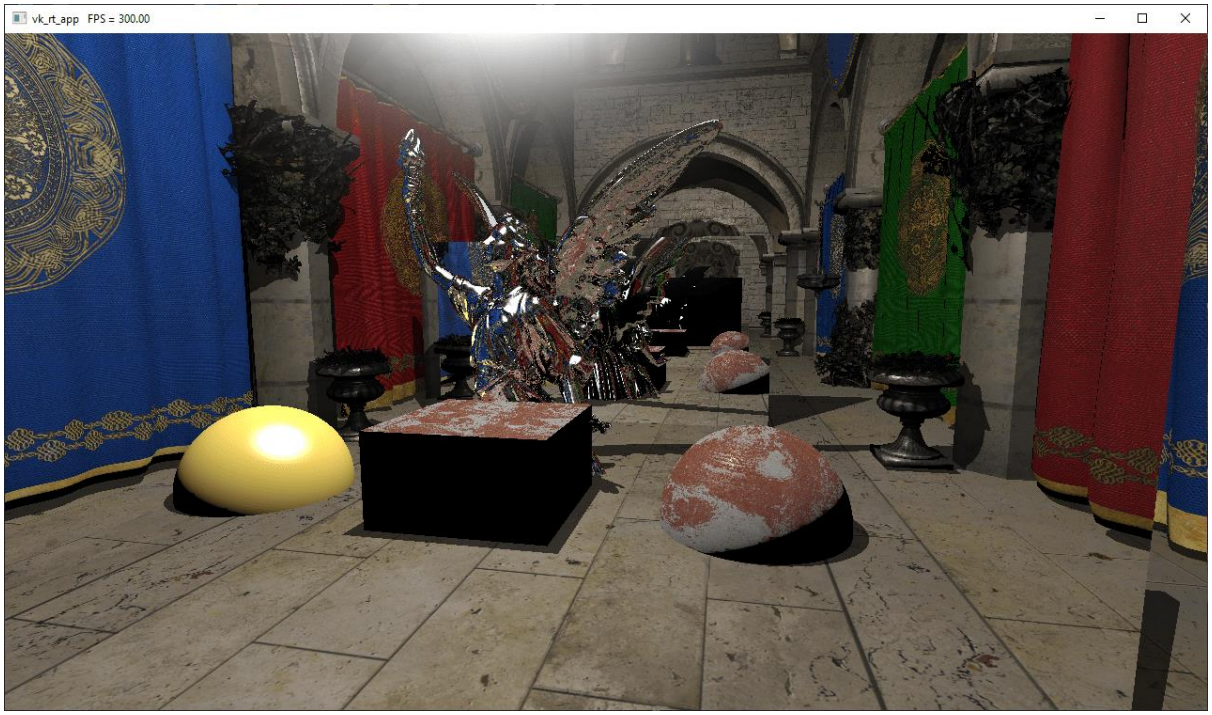
Rysunek 7.8 Ujęcie 2, ray tracing, głębokość odbić 4, materiał lustrzany na lucy



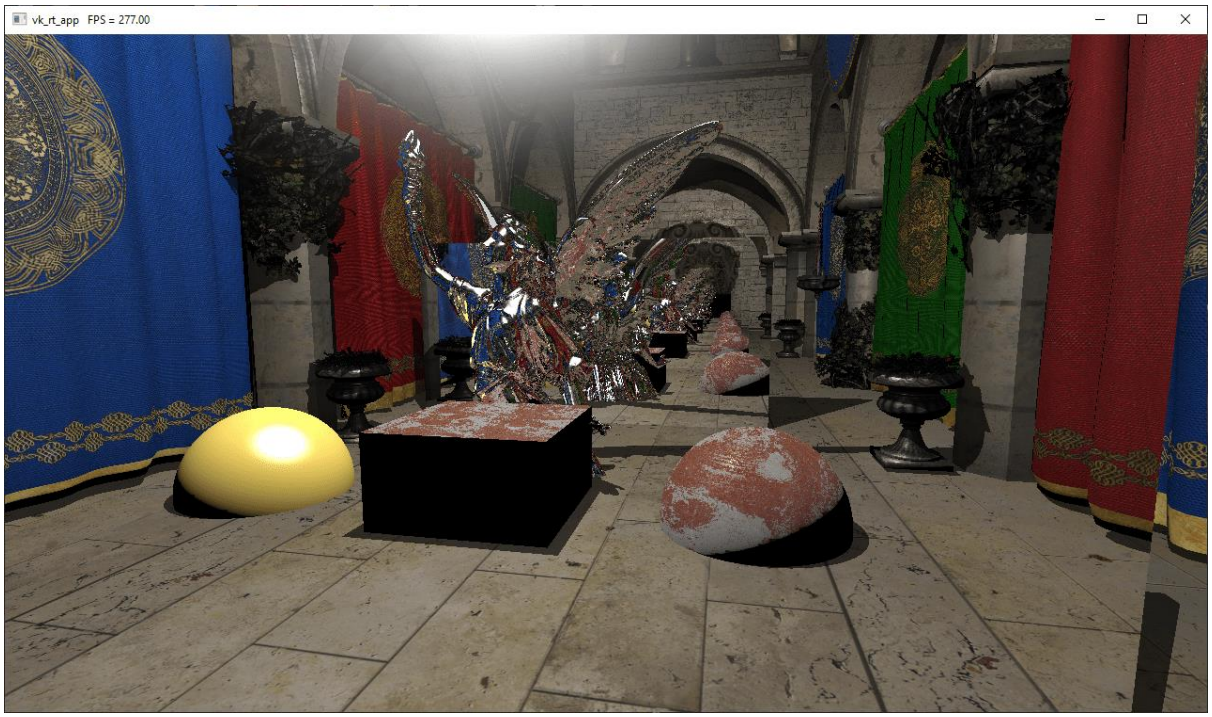
Rysunek 7.9 Ujęcie 2, ray tracing, głębokość odbić 10, materiał lustrzany na lucy



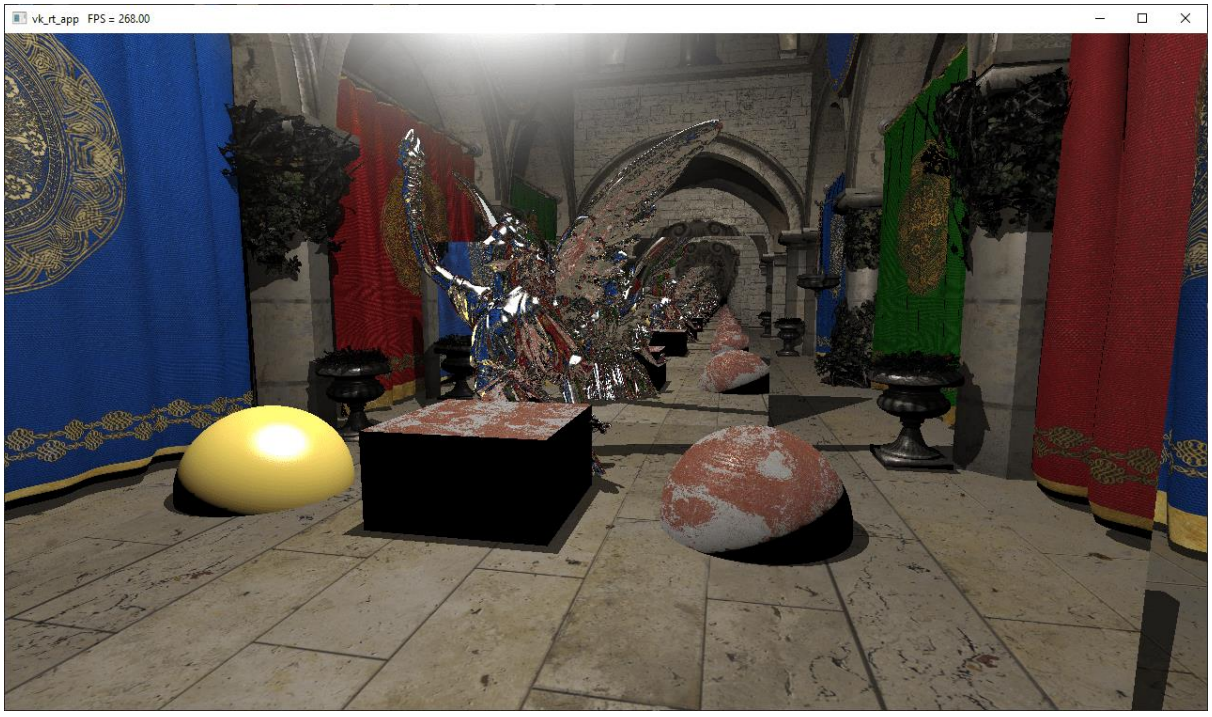
Rysunek 7.10 Ujęcie 2, ray tracing, głębokość odbić 25, materiał lustrzany na lucy



Rysunek 7.11 Ujęcie 3, ray tracing, głębokość odbić 4, materiał lustrzany na lucy



Rysunek 7.12 Ujęcie 3, ray tracing, głębokość odbić 10, materiał lustrzany na lucy



Rysunek 7.13 Ujęcie 3, ray tracing, głębokość odbić 25, materiał lustrzany na lucy

7.5.1 Testy wizualne i wydajnościowe

Porównując rysunki Rysunek 7.2 oraz Rysunek 7.3, można zauważyć, że algorytm rasteryzacji generuje obraz z częstotliwością 540 klatek na sekundę, natomiast algorytm śledzenia promieni działa minimalnie wolniej, bo z częstotliwością 537 klatek na sekundę, generując przy tym cienie na scenie, nie zaimplementowane w potoku rasteryzacji.

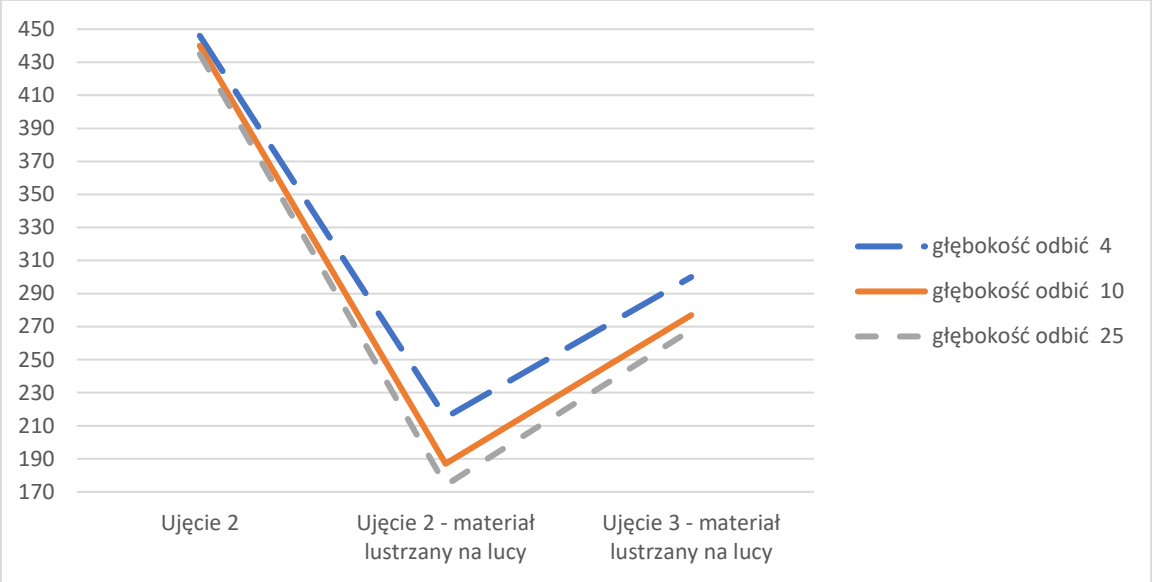
Na ujęciu 2 w kadrze znajduje się „Lucy" model o złożonej geometrii (448 880 trójkątów), co ma wpływ na wydajność i jak widać na Rysunek 7.4, podczas rasteryzacji generuje 530 klatek na sekundę.

Ujęcie to jest skierowane na dwa lustra, równolegle ustawione do siebie, co ma znaczenie dla potoku śledzenia promieni, ponieważ wywołuje wielokrotną rekursję odbijającego się promienia, tym samym wpływając na wydajność. Wpływ ten widać w liczbie generowanych klatek na sekundę względem maksymalnej głębokości rekursji promienia i tak dla głębokości 4 jest to 446 kl./s (Rysunek 7.5), dla głębokości 10 jest to 440 kl./s (Rysunek 7.6), a dla głębokości 25 jest to 435 kl./s (Rysunek 7.7). Należy jednak zauważyć, że wyższe wartości głębokości mają wpływ na coraz mniejszą liczbę pikseli (odbicia w lustrze stają się coraz mniejsze), dlatego wpływ na wydajność nie jest tak drastyczny.

W wariancie sceny, gdzie „Lucy" ma materiał lustrzany, dobrze widać jak wysoka liczba odbić może obniżyć wydajność o ponad 50% (względem sceny, gdzie obiekt nie jest lustrzany). Płynność aplikacji przy głębokości rekursji promienia 4 wynosi 215 kl./s (Rysunek 7.8), przy głębokości 10 wynosi 187 kl./s (Rysunek 7.9), a przy głębokości 25 wynosi 174 kl./s (Rysunek 7.10).

Ujęcie 3, ukazane na rysunkach Rysunek 7.11, Rysunek 7.12 oraz Rysunek 7.13, przedstawia podobny kadr do ujęcia 2, z tym, że obiekty o materiale lustrzanym generujące wysoką rekurencję promienia, zajmują mniej miejsca na obrazie. Dzięki temu skomplikowane obliczenia są przetwarzane dla mniejszej ilości pikseli, co przekłada się na wyższą płynność aplikacji i dla wymienionych wyżej rysunków wynosi kolejno 300, 277 oraz 268 kl./s.

Wydajność aplikacji w zależności od sceny oraz głębokości odbić promienia można zobrazować za pomocą Wykres 7.1. Ilustruje on dobrze, iż większa głębokość odbić ma wpływ na liczbę generowanych klatek. Wpływ ten determinuje ujęcie, a więc ustawienie kamery w scenie, ponieważ od niego zależy dla ilu pikseli promień osiągnie maksymalną głębokość. W ujęciu 2 różnica wydajności w zależności od głębokości jest niewielka, natomiast w ujęciu 2 z materiałem lustrzanym na lucy różnica ta się powiększa. Jednocześnie porównując wydajność w zależności od ujęcia, można wysnuć wnioski, iż największe znaczenie ma tutaj ilość geometrii z materiałem lustrzanym w kadrze, ponieważ na ujęciu 3 z lustrzanym materiałem, lucy zajmuje mniejszą część kadru, niż w ujęciu 2 z lustrzanym materiałem, dzięki czemu aplikacja generuje więcej klatek na sekundę.



Wykres 7.1 Ilość kl./s. w zależności od sceny oraz głębokości odbić promienia

Porównując wizualnie rysunki Rysunek 7.5, Rysunek 7.6, Rysunek 7.7, Rysunek 7.8, Rysunek 7.9, Rysunek 7.10, Rysunek 7.11, Rysunek 7.12 oraz Rysunek 7.13, można dojść do wniosku, iż głębokość promienia 4 jest zupełnie wystarczająca do stworzenia iluzji realistycznych odbić na potrzeby takich aplikacji jak gry komputerowe, gdzie wydajność jest kluczowa. Iluzja ta może zostać przełamana, gdy wystąpi sytuacja wielokrotnego rekursywnego odbicia się promienia np. w równoległych lustrach, gdzie powstanie widoczny czarny kształt. Dobrze jest to widoczne na rysunkach Rysunek 7.5, Rysunek 7.8 oraz Rysunek 7.11. Niemniej jednak taka sytuacja nie powinna mieć miejsca w świecie gry komputerowej. Może natomiast wystąpić w mniej wymagających szybkości działania scenariuszach, jak np. wizualizacjach, gdzie głębokość 10 pozwoli uzyskać zadowalające efekty (rysunki Rysunek 7.6, Rysunek 7.9, Rysunek 7.12), a głębokość 25 pozwoli uzyskać niezaprzeczalnie fotorealistyczne wyniki (rysunki Rysunek 7.7, Rysunek 7.10, Rysunek 7.13).

8 Wnioski i podsumowanie

API Vulkan jest bardzo wymagają technologią, ze względu na swoją obszerność i opisowość. W samej idei technologia ta nie jest skompilowana, ponieważ jest stworzona z myślą o czytelności i rozbita na małe części, które mają jak najlepiej opisać swoje zastosowanie. Problemem jest tutaj niestety ogromna ilość obiektów i zależności między nimi. Dla programisty, który zaczyna poznawać Vulkan API może być to mur nie do sforsowania, dlatego uważam, że najlepiej uczyć się tej technologii od weterana grafiki komputerowej. Będzie on w stanie jasno i zrozumiale przedstawić aspekty, funkcjonalności i mechanizmy technologii, jednocześnie nie dopuszczając do przygniecenia przez ogrom wiedzy jaki będzie potrzebny do przyswojenia. Stanie się samemu takim ekspertem wymaga bardzo szerokiej wiedzy nie tylko na temat API Vulkan, ale również grafiki komputerowej.

Stworzona aplikacja jasno pokazuje, że wykorzystując sprzętowe wsparcie śledzenia promieni można stworzyć program, który będzie mógł płynnie działać w czasie rzeczywistym. Choć scena jest dość prosta, to znajduje się na niej ponad 700 tyś. trójkątów, co jest podobną liczbą prymitywów, które są renderowane w każdej klatce obrazu w wielu grach komputerowych. Oczywiście w grach dzieje się znacznie więcej i w miarę dodawania różnych funkcjonalności i efektów wizualnych wydajność na pewno spadnie, to uważam, że wyniki mojej aplikacji potwierdzają, iż stworzenie niedużej gry w całości opartej o ray tracing jest zupełnie możliwe.

Z porównania wydajności wynika, iż rasteryzacja cały czas ma przewagę nad śledzeniem promieni, niemniej jednak wydaje się, że w miarę rozwoju i optymalizacji sprzętowego ray tracingu różnica ta będzie coraz mniejsza.

9 Bibliografia

[1] T. Whitted, “An improved illumination model for shaded display,” *CACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980, Accessed: Feb. 01, 2022. [Online]. Available: https://www.cs.drexel.edu/~david/Classes/Papers/p343-whitted.pdf

[2] A. Appel, “Some Techniques for Shading Machine Renderings of Solids,” in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, 1968, pp. 37–45. doi: 10.1145/1468075.1468082.

[3] Turner Whitted, *The Compleat Angler*, (1978). Accessed: Feb. 01, 2022. [Online Video]. Available: https://archive.org/details/thecompleatangler1978

[4] H. Christensen, J. Fong, and D. M. Laur D. Batali, “Ray Tracing for the Movie ‘Cars,’” 2006. Accessed: Feb. 01, 2022. [Online]. Available: https://graphics.pixar.com/library/RayTracingCars/paper.pdf

[5] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering 4th Edition Online chapter: Real-Time Ray Tracing*. 2018. Accessed: Feb. 01, 2022. [Online]. Available: https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

[6] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018.

[7] J. de Vries, “PBR chapter from Learn OpenGL tutorial.” https://learnopengl.com/PBR/Theory (accessed Feb. 01, 2022).

[8] B. Karis, “Specular BRDF Reference,” Aug. 03, 2013. https://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html (accessed Feb. 01, 2022).

[9] M. Polyanskiy, “Refractive index database.” https://refractiveindex.info (accessed Feb. 01, 2022).

10 Spis rysunków