

Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki

Daniel Nadolny
nr albumu: 312887

Praca inżynierska
na kierunku informatyka

Ray Tracing w czasie rzeczywistym

Opiekun pracy dyplomowej
doktor Jakub Narębski
Wydział Matematyki i Informatyki

Toruń 2026

Pracę przyjmuję i akceptuję

Potwierdzam złożenie pracy
dyplomowej

.....

data i podpis opiekuna pracy

.....

data i podpis pracownika dziekanatu

Spis treści

Wstęp	3
1 Podstawy Ray Tracingu	5
1.1 Definicje i oznaczenia	5
1.2 Algorytm ray tracingu	6
1.3 Badanie przecięcia promienia ze sferą	7
1.4 Badanie przecięcia promienia z trójkątem	9
2 Optymalizacja	11
2.1 Bounding Volume Hierarchy	11
2.2 Zasada działania BVH	11
2.3 Przechodzenie przez drzewo BVH	13
2.4 Surface Area Heuristic	14
3 Materiały i modele oświetlenia	16
3.1 Materiały w grafice komputerowej	16
3.2 Model oświetlenia lokalnego	17
3.3 Model oświetlenia globalnego	18
4 Prezentacja silnika	22
4.1 Biblioteki	22
4.2 Elementy silnika	24
4.3 Opis potoku graficznego	24
4.4 Implementacja algorytmów	27
4.4.1 Implementacja testu przecięcia promień-sfera	27

4.4.2	Implementacja testu przecięcia promień-trójkąt	28
4.4.3	Przechodzenie przez drzewo BVH	29
4.4.4	Główna logika ray tracingu	31
Podsumowanie		34
4.5	Testy	34
4.6	Zakończenie	36
Bibliografia		37

Wstęp

W 2018 roku NVIDIA przedstawiła światu nową generację kart graficznych nazwanych RTX. Od tego roku każda kolejna seria począwszy od serii 20 do teraz (seria 50) jest wyposażona w tzw. RT Cores. Rdzenie RT są specjalnie stworzone do przyspieszania obliczeń związanych ze śledzeniem promieni (dalej będę używał nazwy ray tracing), w szczególności przy testowaniu przecięcia promienia z trójkątem i przechodzenia przez strukturę danych zwaną BVH (ang. bounding volume hierarchy). Odpowiednikiem RT Cores w kartach graficznych od AMD są "Ray accelerators". Ray tracing jest bardzo wymagającym algorytmem pod względem obliczeniowym. Dodanie powyższych rozwiązań sprzętowych do GPU pozwoliły programistom implementowanie ray tracingu w czasie rzeczywistym np. w grach, gdzie obecnie w jednej scenie może pojawić się kilka milionów trójkątów (dotychczas ray tracing wykorzystywany był głównie w filmach).

W ramach pracy inżynierskiej stworzony został silnik graficzny przedstawiający ray tracing w czasie rzeczywistym, napisany jest w języku C++, wykorzystując bibliotekę DirectX 11 i win32. Interfejs użytkownika stworzony został za pomocą biblioteki ImGui.

Pierwszy rozdział tej pracy będzie poświęcony przedstawieniu podstaw ray tracingu, od opisania idei algorytmu do wyprowadzenia dwóch podstawowych procedur badania przecięć promienia z obiekta w scenie (promień-sfera i promień-trójkąt). W rozdziale drugim poruszone będzie zagadnienie optymalizacji silnika używając struktury danych BVH. W następnym rozdziale zostanie opisane zagadnienie materiałów i modeli oświetlenia wykorzystywanych w grafice komputerowej. Oba zagadnienia mają największy wpływ na aspekty wizualne. W czwartym rozdziale opisany będą

dzie stworzony silnik i implementacje przedstawionych wcześniej algorytmów. W końcowej części pracy przedstawione zostaną wyniki testów wydajnościowych programu. Testy były przeprowadzone przed optymalizacją silnika i po optymalizacji, aby wykazać różnice wydajności.

Rozdział 1

Podstawy Ray Tracingu

1.1 Definicje i oznaczenia

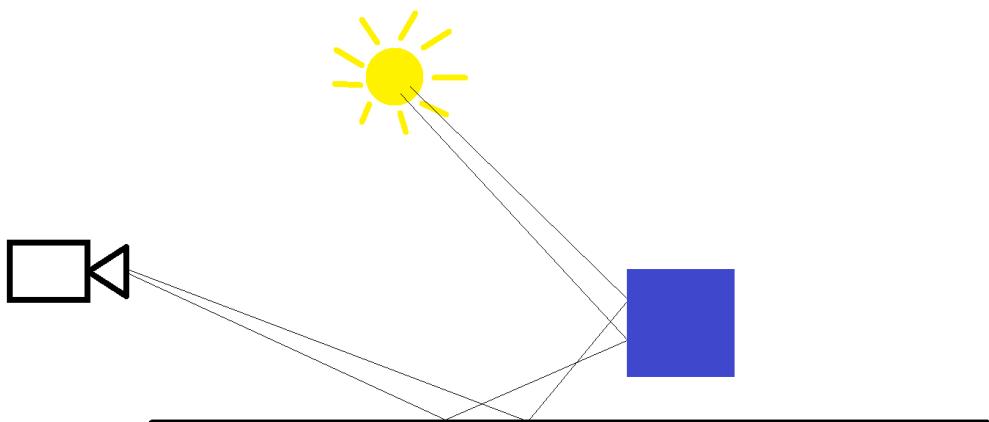
W dalszej części będę posługiwać się takimi oznaczeniami:

- a - wartość skalarna.
- C - punkt w przestrzeni trójwymiarowej.
- v - wektor, w większości przypadków $v = (x, y, z)$.
- n - wektor normalny. Wektor prostopadły do danej powierzchni.
- $a \cdot b$ - iloczyn skalarny.
- $a \times b$ - iloczyn wektorowy.
- Promień - promień (ang. ray) jest podstawową strukturą w ray tracingu. Składa się on z punktu początkowego (ang. origin) i kierunku (ang. direction). Oba elementy zdefiniowane są jako wektory trójwymiarowe, z czego kierunek jest wektorem znormalizowanym (długość równa 1).

1.2 Algorytm ray tracingu

Algorytm ray tracingu znany jest już od 1979 roku, kiedy John Turner Whitted opublikował artykuł "An improved illumination model for shaded display" opisujący rekurencyjny ray tracing [1].

Ideą algorytmu jest naśladowanie światła. W opisie zachowania się światła i jego oddziaływania z materią korzysta się z teorii falowej i optyki geometrycznej. W rzeczywistości światło porusza się po liniach prostych, od źródła np. Słońca. Ray tracing działa odwrotnie, tzn. źródłem promieni jest "oko" kamery i od niego wychodzi światło w generowaną scenę, ponieważ jak w rzeczywistości mózg człowieka "renderuje" obraz korzystając tylko z tych promieni, które padają na siatkówkę w oku. Gdy wystrzelony z kamery promień uderza w jakiś obiekt punkt przecięcia staje się punktem początkowym kolejnego promienia (rekurencyjna natura algorytmu)[2].



Rysunek 1.1: Rysunek przedstawia ideę ray tracingu. Promienie wychodzą z kamery i uderzają w obiekty, następnie generowane są kolejne promienie, które w pewnym momencie trafiają do źródła światła.

Należy się teraz zastanowić, w jaki sposób obiekty w naturze "dostają" swój kolor. Innymi słowy - dlaczego pomidor jest czerwony? Obserwując eksperyment przedstawiający rozszczepienie światła pryzmatem, lub

tęczę, możemy zauważyc, że białe światło rozszczepia się na kilka barw (w uproszczeniu). Dzieje się tak, ponieważ barwy światła mają różne długości fal. Zakres długości fal dla światła widzialnego przez człowieka wynosi: $\approx 380\text{-}780$ nanometrów[3]. Gdy promienie ze źródła uderzą w jakiś obiekt, (np. w pomidor) to materiał obiektu wchłonie w siebie pewną część światła o danej długości, a odbije resztę. Przykładowy pomidor odbije głównie barwę czerwoną, czyli fale o długości w zakresie 620-780 nm, następnie mózg interpretuje daną długość fali na odpowiedni kolor. W ray tracingu procedura jest podobna, jedynym dodatkiem do światła w scenie jest jego źródło, reszta oświetlenia pochodzi od promieni odbitych od obiektów.

Symulowanie światła pozwala programistom na uzyskanie szczegółowych i poprawnych fizycznie efektów takich jak: odbicie, refrakcja itd. Wcześniej, przed erą ray tracingu w czasie rzeczywistym, programiści musieli korzystać z różnych sztuczek aby zaimplementować te efekty, dla przykładu odbicia tworzone były poprzez zrenderowanie danego obiektu drugi raz ale odwrotnie w np. kałurzy, lustrze. Korzystając z ray tracingu efekt odbicia jest dużo prostszy w implementacji.

Minusem algorytmu jest jego złożoność obliczeniowa. Dla przykładu, bez optymalizacji BVH mając model złożony z 100 000 trójkątów, w rozdzielcości 2560x1440, mając ustawione 2 próbki na piksel i 5 odbiciach, program musiałby dla każdego piksela wykonać (w tej rozdzielcości mamy 3686400 pikseli) 1 000 000 testów.

1.3 Badanie przecięcia promienia ze sferą

Test przecięcia promienia ze sferą jest jednym z najprostszych algorytmów w tej tematyce i idealnie nadaje się do przedstawienie procesu wyprowadzania takich algorytmów.

Należy zacząć od zapisania równania sfery o środku w punkcie $C = (c_x, c_y, c_z)$.

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2 \quad (1.1)$$

Punkt $P = (p_x, p_y, p_z)$ jest punktem leżącym na sferze. Można zapi-

sać wektor o długości r ze środka sfery do tego punktu zapisując:

$$(\mathbf{P} - \mathbf{C})$$

Teraz równanie sfery można zapisać tak:

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = r^2 \quad (1.2)$$

Każdy punkt \mathbf{P} spełniający to równanie leży na sferze.

Szukaną wartością w tym badaniu jest zmienna t , na podstawie której można obliczyć współrzędne punktu P . Równanie opisujące promień o punkcie początkowym O i kierunku D :

$$\mathbf{P}(t) = \mathbf{O} + t\mathbf{D} \quad (1.3)$$

Wstawiając (1.3) do (1.2):

$$(\mathbf{O} + t\mathbf{D} - \mathbf{C}) \cdot (\mathbf{O} + t\mathbf{D} - \mathbf{C}) = r^2 \quad (1.4)$$

Nie ma potrzeby rozpisywania całego iloczynu skalarnego, zamiast tego można zapisać:

$$(\mathbf{D} \cdot \mathbf{D})t^2 + 2t\mathbf{D} \cdot (\mathbf{O} - \mathbf{C}) + (\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - r^2 = 0 \quad (1.5)$$

Jest to równanie kwadratowe o współczynnikach:

$$a = \mathbf{D} \cdot \mathbf{D}, b = 2\mathbf{D} \cdot (\mathbf{O} - \mathbf{C}), c = (\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - r^2$$

Wzór na paramter t :

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Równanie (1.5) może mieć rozwiązania dodatnie jak i ujemne. Silnik, aby poprawnie generować obrazy dla wielu sfer w scenie, potrzebuje wartości najbliższej i dodatniej (wartość ujemna świadczy o tym, że sfera jest za kamerą). W ramach oznaczenia przypadku braku rozwiązania równania (1.5)

z funkcji zwracana jest wartość -1.0 .

1.4 Badanie przecięcia promienia z trójkątem

Autorami przedstawionego poniżej algorytmu są Tomas Möller i Ben Trumbore [4]. W odróżnieniu od innych popularnych algorytmów szukających punkt przecięcia promienia z trójkątem, ten nie oblicza równania płaszczyzny wyznaczanej przez trójkąt, tylko opiera się na samych wierzchołkach trójkąta. Istnieją szybsze algorytmy np. algorytm Douga Baldwina i Michaela Webera [5], ale wymagają one dodatkowych informacji razem z trójkątami.

Definicja 1.4.1. *Współrzędne barycentryczne na trójkącie o wierzchołkach w punktach P_0, P_1, P_2 to trójka liczb $(w, u, v) \in \mathbb{R}$, dzięki którym możemy przedstawić każdy punkt na trójkącie w formie:*

$$f(u, v) = (1 - u - v)\mathbf{P}_0 + u\mathbf{P}_1 + v\mathbf{P}_2.$$

Liczby (w, u, v) spełniają następujące warunki:

$$w + u + v = 1, \quad w \geq 0, \quad u \geq 0, \quad v \geq 0.$$

[DODAC RYSUNEK Z REAL TIME RENDERING]

Znalezienie punktu przecięcia sprowadza się do rozwiązania układu równań:

$$\mathbf{O} + t\mathbf{D} = (1 - u - v)\mathbf{P}_0 + u\mathbf{P}_1 + v\mathbf{P}_2 \quad (1.6)$$

Gdzie $\mathbf{O} = (o_x, o_y, o_z)$ to punkt początkowy promienia, a $\mathbf{D} = (d_x, d_y, d_z)$ to kierunek.

Po przepisanu na postać macierzową:

$$\begin{bmatrix} -\mathbf{D} & \mathbf{P}_1 - \mathbf{P}_0 & \mathbf{P}_2 - \mathbf{P}_0 \end{bmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{O} - \mathbf{P}_0 \quad (1.7)$$

Pomocnicze oznaczenia: $e_1 = P_1 - P_0$, $e_2 = P_2 - P_0$, $s = O - P_0$
 Używając oznaczeń układ równań ma postać:

$$\begin{aligned} -d_x t + e_{1x} u + e_{2x} v &= s_x \\ -d_y t + e_{1y} u + e_{2y} v &= s_y \\ -d_z t + e_{1z} u + e_{2z} v &= s_z \end{aligned} \quad (1.8)$$

Wtedy macierz główna układu ma postać:

$$M = \begin{bmatrix} -d_x & e_{1x} & e_{2x} \\ -d_y & e_{1y} & e_{2y} \\ -d_z & e_{1z} & e_{2z} \end{bmatrix} \quad (1.9)$$

Do rozwiązyania układu równań można wykorzystać metodę Cramera:
 ra:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-D, e_1, e_2)} \begin{pmatrix} \det(s, e_1, e_2) \\ \det(-D, s, e_2) \\ \det(-D, e_1, s) \end{pmatrix} \quad (1.10)$$

Z własnością iloczynu mieszanego wiadomo, że: $(a \times b) \cdot c = \det(a, b, c)$
 Wtedy równanie można zapisać tak (zmieniając kolumny macierzy wyznacznik zmienia znak na przeciwny):

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(D \times e_2) \cdot e_1} \begin{pmatrix} (s \times e_1) \cdot e_2 \\ (D \times e_2) \cdot s \\ (s \times e_1) \cdot D \end{pmatrix} \quad (1.11)$$

Wprowadzając kolejne oznaczenia, dla uproszczenia: $q = D \times e_2$
 $r = s \times e_1$ Otrzymane równanie:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{q \cdot e_1} \begin{pmatrix} r \cdot e_2 \\ q \cdot s \\ r \cdot D \end{pmatrix} \quad (1.12)$$

W implementacji funkcja zwraca parametr t , dzięki niemu znany jest punkt przecięcia.

Rozdział 2

Optymalizacja

2.1 Bounding Volume Hierarchy

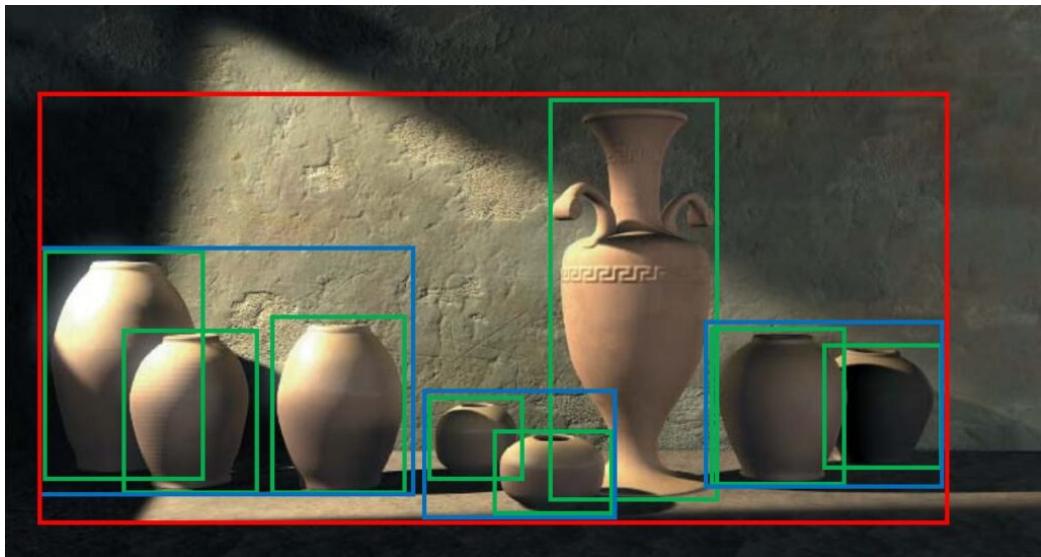
W stworzonym silniku graficznym w danej chwili jest kilka stałych wartości: rozdzielcość (np. 1920x1080 pikseli), liczba odbić promienia, liczba promieni na piksel i liczba trójkątów w scenie. Jedyną zmienną, którą można optymalizować jest liczba testów przecięcia promień-trójkąt (dla uproszczenia algorytmu sfery są pominięte). Do takiej optymalizacji można posłużyć się tzw. drzewami BVH - Bounding Volume Hierarchy.

Idea algorytmu jest prosta - pomijać te trójkąty których promień na pewno nie przetnie. Bez optymalizacji, aby sprawdzić czy promień przecina się z jakimś trójkątem, trzeba przejść przez całą tablicę trójkątów modelu i każdy przetestować, w przypadku użycia BVH większość pomijamy badając tylko te najbliższe punktowi przecięcia.

2.2 Zasada działania BVH

W BVH dzielimy dany model na prostopadłościany zwane "bounding box". Proces podziału rozpoczyna się od korzenia (ang. root) jako prostopadłościan okalający cały obiekt (lub całą scenę), następnie rekurencyjnie dzielimy model na coraz to mniejsze prostopadłościany, aż do zadanej granicy np. 2 trójkątów w jednym boxie. BVH ma strukturę drzewa binarnego

(tak jest w stworzonym silniku), gdzie w wierzchołkach mieszą się kolejne prostopadłościany z podziału, a w liściach znajdują się trójkąty [2].



Rysunek 2.1: Obrazek przedstawiający BVH
[6]

W strukturze BVH używa się różnych typów "bounding box", w ray tracingu popularnym wyborem są "axis-aligned bounding box" (AABB). AABB tworzone są poprzez wyznaczenie dwóch punktów: prawego górnego punktu i lewego dolnego punktu. Maksymalny punkt AABB wyznaczany jest poprzez wyszukanie największych wartości na każdej osi ze zbioru trójkątów, każdy trójkąt ma 3 punkty. Punkt minimalny wyznaczany jest analogicznie poprzez szukanie najmniejszej wartości.

Można zapisać to tak:

Aby stworzyć strukturę drzewa należy teraz podjąć decyzję w jaki sposób dzielić model (AABB boxy). Dla prostoty algorytmu dzielić można wzdłuż najdłuższej osi [2]. Następnie dzielimy trójkąty na dwie grupy poprzez sprawdzanie czy jego centroid¹ leży w jednym boxie, czy w drugim.

¹Centroid to punkt przecięcia median trójkąta (mediana to odcinek łączący wierzchołek ze środkiem przeciwnego boku) [7]

Algorithm 1 Wyznaczanie AABB

```
Triangles ← [ (v1, v2, v3), ... ]  
for each triangle t in Triangles do  
    pmin ← ( min(Pmin, t.v1) )  
    pmin ← ( min(Pmin, t.v2) )  
    pmin ← ( min(Pmin, t.v3) )  
    pmax ← ( max(Pmax, t.v1) )  
    pmax ← ( max(Pmax, t.v2) )  
    pmax ← ( max(Pmax, t.v3) )  
end for
```

2.3 Przechodzenie przez drzewo BVH

Najważniejszym punktem w procedurze przechodzenia przez drzewo BVH jest testowanie przecięcia promienia z AABB. W tym przypadku test nie musi zwracać dokładnego punktu przecięcia, ale samą informację, czy do przecięcia doszło lub (jeśli potrzebna) odległość od punktu początkowego promienia do przecięcia. Jedną z metod jest tzw. slab test.

Slab test polega na sprawdzaniu czy promień przecina wszystkie płaszczyzny wyznaczane przez osie x, y, z. Dla każdej osi należy obliczyć, AABB box zdefinowany jest jako dwa punkty $P_{min} = (x_{min}, y_{min}, z_{min})$, $P_{max} = (x_{max}, y_{max}, z_{max})$, parametr t , który można użyć do obliczenia punktu przecięcia używając równanie promienia (1.3):

$$t_1 = \frac{P_{min} - O}{D} \quad (2.1)$$

$$t_2 = \frac{P_{max} - O}{D} \quad (2.2)$$

Następnym krokiem jest wybranie wartości największej i najmniejszej:

$$t_{min} = \min(t_1, t_2) \quad (2.3)$$

$$t_{max} = \max(t_1, t_2) \quad (2.4)$$

Następnie należy obliczyć wartości:

$$t_{near} = \max(t_{min_x}, t_{min_y}, t_{min_z}) \quad (2.5)$$

$$t_{far} = \min(t_{max_x}, t_{max_y}, t_{max_z}) \quad (2.6)$$

Jeśli $t_{near} \leq t_{far}$ to promień jest na części wspólnej wyznaczonej przez płaszczyzny, czyli przecina AABB box, jeśli $t_{near} \geq t_{far}$ lub $t_{far} < 0$ promień nie trafił w box.

Jeśli promień przetnie box AABB, wtedy trzeba sprawdzić czy wierzchołek ze struktury BVH jest liściem. Jeśli tak (liść ma w sobie trójkąty), to uruchamia się test przecięcia promień-trójkąt. W przypadku wierzchołków, które nie są liśćmi, funkcja wchodzi w rekurencje dla lewego potomka i prawnego potomka.

2.4 Surface Area Heuristic

Istnieją też inne metody na budowę BVH, jedną z tych metod jest SAH "Surface Area Heuristic". W tej metodzie przy każdym podziale oblicza się koszt przeprowadzania testów przecięcia promienia z danym obiektem. Wzór na koszt [8]:

$$c(A, B) = t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{i=1}^{N_B} t_{isect}(b_i) \quad (2.7)$$

- t_{trav} - czas potrzebny na ustalenie przez które dzieci przechodzi promień.
- p_A - to prawdopodobieństwo, że przez wierzchołek A przejdzie promień (odpowiednio p_B).
- $\sum_{i=1}^{N_A} t_{isect}(a_i)$ - Czas potrzebny na przeprowadzenie testu przecięcia promienia z obiektem dla każdego obiektu w danym AABB.

$$p_A = \frac{s_a}{s_b} \quad (2.8)$$

- s_a - powierzchnia danego AABB ($s_a > s_b$)

Rozdział 3

Materiały i modele oświetlenia

3.1 Materiały w grafice komputerowej

W rzeczywistości zachowanie światła padającego na dany obiekt będzie zależało od materiału z którego ten obiekt jest stworzony np. światło padające na metal będzie odbijało się inaczej od światła padającego na plastik. W grafice komputerowej chcemy symulować właściwości różnych materiałów za pomocą tzw. modeli oświetlenia.

Model oświetlenia, to matematyczny opis zachowania światła w scenie. Określa w jaki sposób obiekty powinny być renderowane tj. w jaki sposób światło odbija się od powierzchni obiektów. Modele oświetlenia możemy podzielić na:

- Modele oświetlenia lokalnego (local illumination) - kolor danej powierzchni zależy tylko od materiału, z którego powierzchnia jest zrobiona i źródła światła.
- Modele oświetlenia globalnego (global illumination) - kolor danej powierzchni zależy od materiału, z którego powierzchnia jest zrobiona, źródła światła i światła odbitego od innych obiektów. [2]

3.2 Model oświetlenia lokalnego

Jednym z najpopularniejszych modeli oświetlenia lokalnego jest model Phonga (ang. Phong reflection model). Model Phonga składa się z trzech komponentów:

- Ambient - światło otoczenia. W tym algorytmie wpływ innych obiektów na jasność i kolor danego modelu (global illumination) jest symulowane poprzez dodanie pewnej stałej wartości do jasności i koloru obiektu.
- Diffuse - światło rozproszone. Najważniejszy komponent oświetlenia, światło padające na powierzchnię odbija się we wszystkich kierunkach równomiernie [8]
- Specular - światło zwierciadlane, odblask.

Wzór przedstawiający ostateczne oświetlenie danego punktu [9]:

$$I = I_a k_a + k_d I_i (\mathbf{n} \cdot \mathbf{l}) + k_s I_i (\mathbf{r} \cdot \mathbf{v})^s \quad (3.1)$$

Gdzie:

- I_a - natężenie światła otoczenia.
- I_i - natężenie światła padającego.
- k_a - współczynnik odbicia światła otoczenia. Własność materiału.
- k_d - współczynnik odbicia rozproszonego. Określa jak bardzo obiekt jest matowy. Własność materiału.
- k_s - współczynnik odbicia zwierciadlanego. Określa jak bardzo obiekt jest błyszczący. Własność materiału.
- \mathbf{n} - wektor normalny do powierzchni.
- \mathbf{l} - znormalizowany wektor kierunku do źródła światła od badanego punktu.

- r - wektor kierunku odbicia światła obliczany według wzoru:

$$r = 2.0(\mathbf{l} \cdot \mathbf{n}) \cdot \mathbf{n} - \mathbf{l}$$

- v - znormalizowany wektor kierunku do kamery.
- s - współczynnik przedstawiający rozmiar odblasku.

W przypadku wielu źródeł światła w scenie wzór przyjmuje postać:

$$I = I_a k_a + \sum_i^{lights} k_d I_i (\mathbf{n} \cdot \mathbf{l}_i) + \sum_i^{lights} k_s I_i (\mathbf{r}_i \cdot \mathbf{v})^s \quad (3.2)$$

Ulepszeniem tego modelu jest Model Blinna-Phonga w którym oblicza się tzw. halfway vector $\mathbf{h} = \frac{\mathbf{l}+\mathbf{v}}{\|\mathbf{l}+\mathbf{v}\|}$ i zamienia się $\mathbf{r} \cdot \mathbf{v}$ na $\mathbf{n} \cdot \mathbf{h}$

3.3 Model oświetlenia globalnego

W modelach oświetlenia globalnego (ang. *global illumination*) do obliczenia natężenia światła (koloru piksela) w danym punkcie bierze się pod uwagę źródło światła jak i otoczenie - światło odbite od innych powierzchni. Efekty takie jak: realistyczne cienie, odbicia i obiekty transparentne są implementowane za pomocą algorytmów globalnego oświetlenia.

Algorytmy global illumination oparte są na tzw. *rendering equation* [2]:

$$L_o(\mathbf{p}, \mathbf{v}) = L_e(\mathbf{p}, \mathbf{v}) + \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \mathbf{L}_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \quad (3.3)$$

gdzie:

- $L_o(\mathbf{p}, \mathbf{v})$ - światło wychodzące od punktu p w kierunku \mathbf{v} (kierunek kamery).
- $L_e(\mathbf{p}, \mathbf{v})$ - światło emitowane przez powierzchnię w punkcie p w kierunku \mathbf{v} .
- Ω - to powierzchnia półsfery, która znajduje się nad punktem dla którego oblicza się wychodzące światło. Całkowanie po tej półsferyze

odpowiada sumowaniu wkładu światła docierającego ze wszystkich możliwych kierunków.

- $f(\mathbf{l}, \mathbf{v})$ - to tzw. *bidirectional reflectance distribution function* (dalej BRDF). BRDF to funkcja opisująca w jaki sposób światło nadchodzące z kierunku \mathbf{l} jest odbijane w kierunku \mathbf{v} przez dany materiał.
- $L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})$ - przychodzące światło. \mathbf{p} to punkt na powierzchni, a \mathbf{l} to kierunek. Funkcja $r(\mathbf{p}, \mathbf{l})$ zwraca punkt przecięcia się promienia, którego \mathbf{p} to punkt początkowy, a \mathbf{l} kierunek. Komponent ten oznacza, że światło przychodzące do punktu \mathbf{p} jest światłem wychodzącym od innego punktu.
- $(\mathbf{n} \cdot \mathbf{l})^+$ - iloczyn skalarny wektora normalnego powierzchni z wektorem kierunku światła, ponieważ oba wektory są znormalizowane wynikiem jest cosinus kąta pomiędzy tymi wektorami. $+$ oznacza branie tylko dodatnich wyników.

Analitycznie równanie 3.3 jest niemożliwe do rozwiązania (poza bardzo prostymi scenami), ponieważ aby rozwiązać równanie dla punktu \mathbf{p} trzeba znać wynik dla punktu wcześniejszego \mathbf{p}' , powstaje nieskończona rekurencja na nieskończonej liczbie kierunków z których światło dochodzi do punktu \mathbf{p} .

Równanie renderingu rozwiązuje się metodami numerycznymi np. metodą monte-carlo.

Przykładem modelu oświetlenia, który jest wykorzystywany w implementacji global illumination (jak i local illumination) jest model Cook-Torrance [10]. Model bazuje na tzw. *microfacet theory*, polega ona na spostrzeżeniu, że w rzeczywistości nie ma idealnie gładkich powierzchni (najbliżej jest lustro), złożone są z "mikropowierzchni", które skierowane są pod różnym kątem. Poziom nierówności powierzchni opisuje się parametrem *roughness*.

Funkcja BRDF modelu Cook-Torrance dla odbić lustrzanych[11]:

$$f_{cookTorrance} = \frac{DFG}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})} \quad (3.4)$$

gdzie:

- D - Funkcja rozkładu mikropowierzchni. Określa ona jak dużo ścianek skierowana jest w taki sposób aby odbić nadchodzący promień w stronę kamery.
- F - Współczynnik Fresnala, określa stosunek światła odbitego do światła załamanego.
- G - Funkcja określająca atenuację (tłumienie) światła wynikające z nawiązania się mikropowierzchni na siebie (mikropowierzchnie mogą się zasłaniać).
- \mathbf{n} - wektor normalny.
- \mathbf{l} - wektor do źródła światła. W tym przypadku jest to kierunek z którego przychodzi światło, tj. wektor odbicia.
- \mathbf{v} - wektor do kamery.

Wybór funkcji D i G może być różny, najpopularniejszą funkcją D jest funkcja Trowbridge-Reitz (GGX), a G to funkcja Smitha.

$$D_{GGX}(\mathbf{n}, \mathbf{h}, \alpha) = \frac{\alpha^2}{\pi((\mathbf{n} \cdot \mathbf{h})^2(\alpha^2 - 1) + 1)^2} \quad (3.5)$$

$$G(\mathbf{n}, \mathbf{v}, \mathbf{l}) = G_1(\mathbf{n}, \mathbf{v}) \cdot G_1(\mathbf{n}, \mathbf{l}) \quad (3.6)$$

$$G_1(\mathbf{n}, \mathbf{v}) = \frac{\mathbf{n} \cdot \mathbf{v}}{(\mathbf{n} \cdot \mathbf{v})(1 - k) + k} \quad (3.7)$$

gdzie:

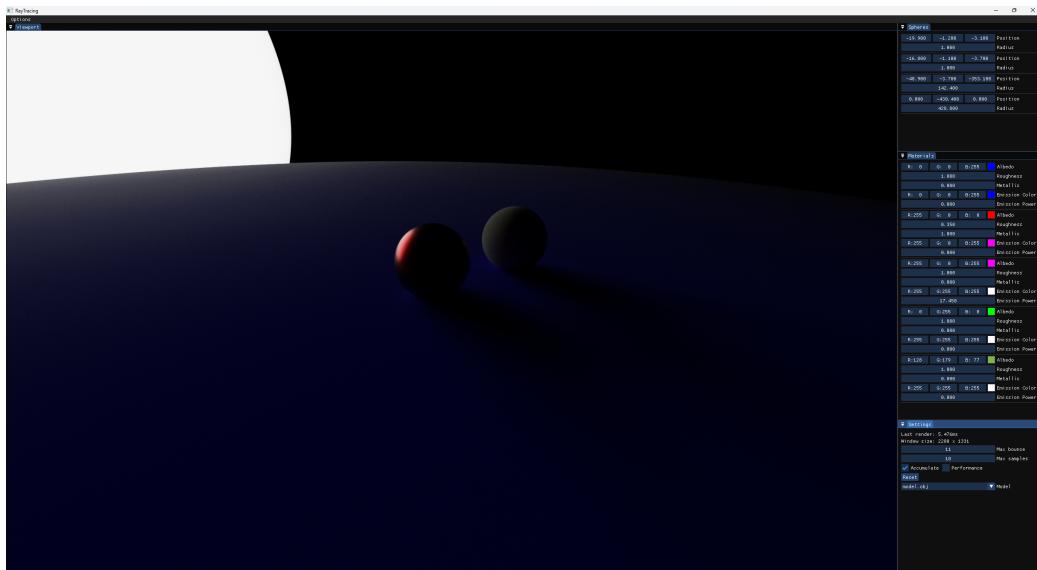
- α to parametr materiału *roughness*
- $k = \frac{(\alpha+1)^2}{8}$

Współczynnik Fresnela w większości przypadków przybliża się tzw. przybliżeniem Schlick'a[12].

$$F_{Schlick} = F_0 + (1 - F_0)(1 - \cos(\theta))^5 \quad (3.8)$$

F_0 to współczynnik odbicia światła padającego prostopadle do powierzchni. W silniku przyjmuje się bazową wartość $f_0 = 0.04$.

Przedstawiony powyżej wzór służy do opisania odbić lustrzanych (ang. specular reflection). Aby opisać światło rozproszone (ang. diffuse) można posłużyć się dowolnym modelem np. standardowym modelem Lambert'a.



Rysunek 3.1: Przykład globalnego oświetlenia - miękkie cienie. W silniku nie zaimplementowano świateł, są tylko materiały emisyjne.

Rozdział 4

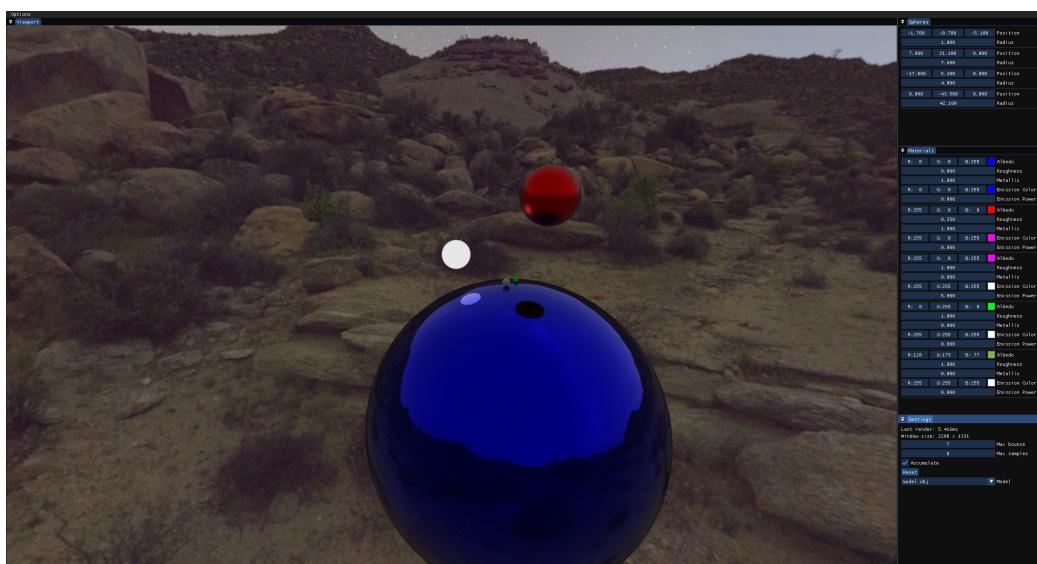
Prezentacja silnika

4.1 Biblioteki

- **DirectX 11** jest to biblioteka graficzna wykorzystywana głównie do tworzenia gier komputerowych (np. Baldur's Gate 3, Wiedźmin 3) lub innych aplikacji graficznych. Biblioteka została stworzona w 2009 roku przez firmę Microsoft. Dostępna jest tylko na komputerach z systemem windows i konsolach Xbox. W silniku wykorzystywana jest do grafiki poprzez shadery¹ takie jak: compute shader, pixel shader i vertex shader.
- **Win32** jest to interfejs programistyczny systemu Windows. Zawiera w sobie funkcje umożliwiające działanie programów w systemie. Okno prezentowanego silnika zostało stworzone za pomocą tej biblioteki.
- **ImGui** biblioteka wykorzystywana do tworzenia interfejsu użytkownika. Napisana jest w myśl paradygmatu "Immediate Mode GUI" (IM-GUI), który głównie polega na tym, że interfejs jest cały czas odświeżany, nie przechowuje on stanu między klatkami przez co jest zawsze zsynchronizowany z danymi. Biblioteka ta jest również łatwiejsza w obsłudze niż np. biblioteka QT.

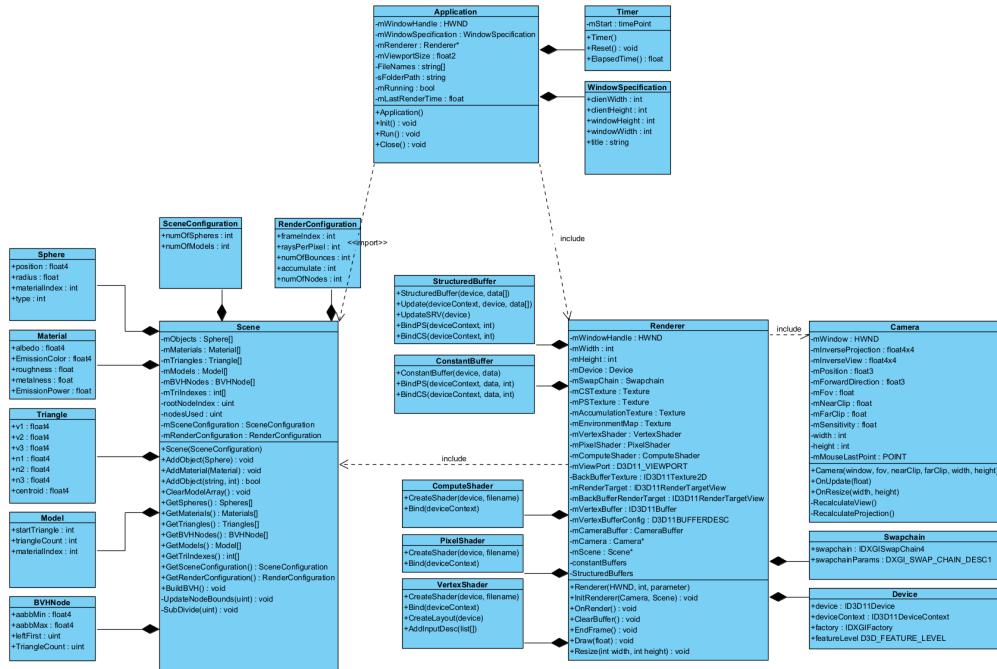
¹Shader to program działający na karcie graficznej

- **Assimp** Open Asset Import Library jest to biblioteka umożliwiająca łatwe ładowanie modeli 3D w różnych formatach np. obj, glb. Napisana jest w języku C/C++.
- **Stb_Image** biblioteka wykorzystywana do ładowania tekstur w różnych formatach.



Rysunek 4.1: Zdjęcie przedstawia interfejs silnika.

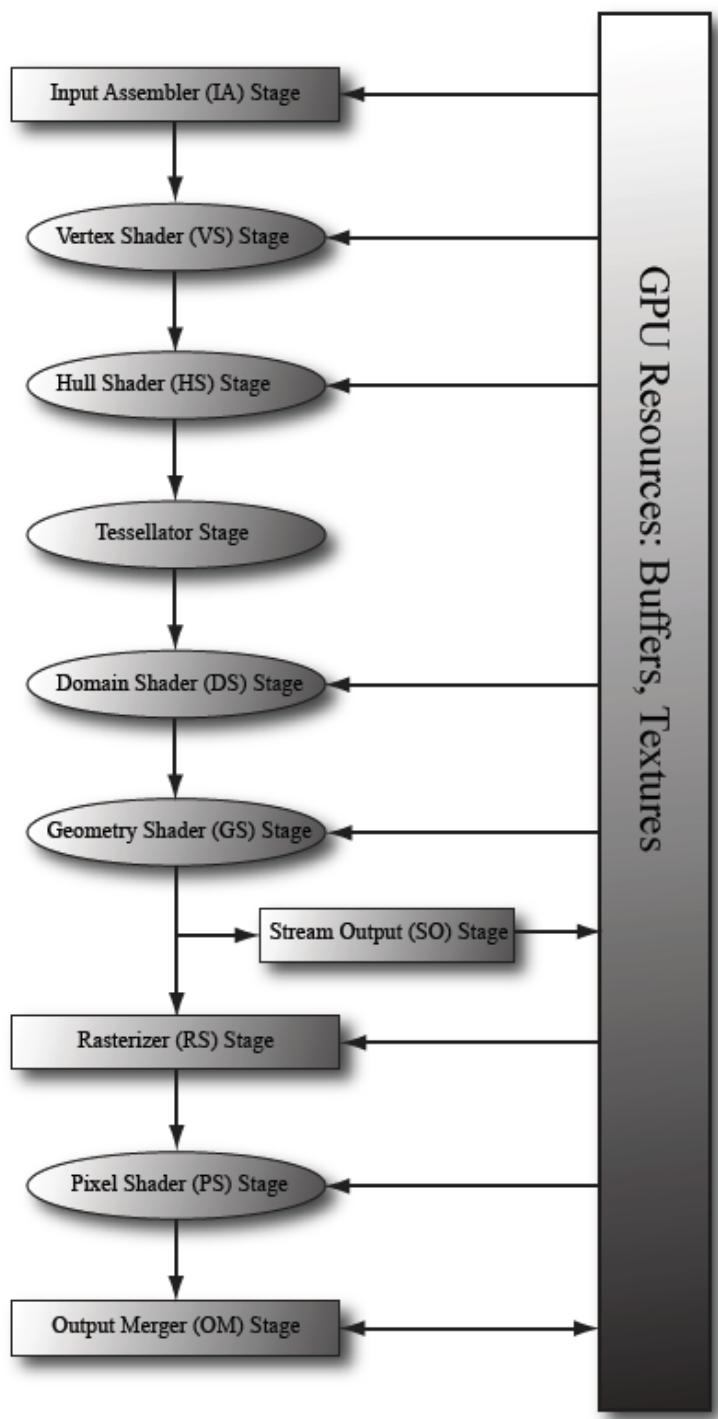
4.2 Elementy silnika



Rysunek 4.2: Schemat silnika.

4.3 Opis potoku graficznego

Potok graficzny (ang. *graphics pipeline* lub *rendering pipeline*) jest to sekwencja kroków które należy wykonać aby otrzymać obraz na ekranie. Na rysunku 4.3 widoczny jest cały potok graficzny używany w DX11. Strzałki oznaczają przepływ danych np. vertex shader otrzymuje dane z fazy Input assembler, wykonuje na nich swoje obliczenia i przekazuje dalej do hull shadera.



Rysunek 4.3: Zdjęcie przedstawia potok graficzny w DirectX 11. [13]

Najważniejszymi shaderami w potoku są vertex shader i pixel shader (pixel shader czasem nazywany jest fragment shaderem np. w OpenGL).

- **Vertex shader** zajmuje się przekształcaniem samych wierzchołków np. nakłada przekształcenia. Vertex shader wykonuje się dla każdego przekazanego wierzchołka na karcie graficznej przez co jest bardzo szybki [13].
- **Pixel shader** zajmuje się kolorem danego piksela, to tutaj wykonuje się kod implementujący np. oświetlenie w scenie.

Poza potokiem graficznym znajduje się compute shader. Compute shader jest shaderem ogólnego przeznaczenia, można w nim implementować np. ray tracing.

W prezentowanym silniku potok graficzny wygląda następująco:

Compute shader → (Vertex Shader → Pixel Shader) → ImGui →
Prezentacja w oknie

W compute shaderze zaimplementowana jest cała logika ray tracingu, shader zapisuje wyniki obliczeń jako kolor RGBA do tekstury 2D. Następnie tekstura przyjmowana jest jako wejście do pixel shadera, który zamienia wejściową teksturę mającą format R16G16B16A16 (16 bitów na piksel) na format standardowy R8G8B8A8 (8 bit na piksel).

Silnik wykonuje tzw. rendering to texture, jest to technika polegająca na zapisaniu całej sceny do tekstury, która zostaje nałożona na wybrany obiekt. Standardowym podejściem jest teksturowanie prostokąta, który ma taki sam rozmiar jak okno aplikacji, innym szybszym sposobem jest narysowanie dużego (wykraczającego poza ekran programu) trójkąta [14, 15], takie podejście sprawia, że vertex shader musi przekształcić tylko 3 wierzchołki, a nie 4 jak przy prostokącie (przy użyciu tzw index buffer nie trzeba powtarzać wierzchołków).

4.4 Implementacja algorytmów

W tej części przedstawione zostaną implementacje wcześniejszych omówionych algorytmów. Wszystkie napisane są w języku HLSL (ang. High-Level Shader Language).

4.4.1 Implementacja testu przecięcia promień-sfera

```
1      float SphereIntersection(Ray ray, Sphere sphere)
2      {
3          float3 spherePosition = float3(sphere.position.xyz)
4          ;
5          float3 oc = ray.origin - spherePosition;
6
7          float a = 1; //dot(ray.direction, ray.direction);
8          float b = 2.0 * dot(ray.direction, oc);
9          float c = dot(oc, oc) - sphere.radius * sphere.
10             radius;
11
12         float discriminant = b * b - 4.0f * a * c;
13
14         if (discriminant < 0.0f)
15         {
16             return -1.0f;
17         }
18
19         float t1 = (-b - sqrt(discriminant)) / (2.0f * a);
20         float t2 = (-b + sqrt(discriminant)) / (2.0f * a);
21
22         if (t1 > 0.0f && t2 > 0.0f)
23         {
24             return min(t1, t2);
25         }
26         else if (t1 > 0.0f)
```

```

27         return t1;
28     }
29     else if (t2 > 0.0f)
30     {
31         return t2;
32     }
33     else
34     {
35         return -1.0f;
36     }
37 }
```

Zmienna a ma wartość 1, ponieważ $ray.direction$ jest wektorem znormalizowanym, iloczyn skalarny dwóch tych samych znormalizowanych wektorów wynosi 1.

4.4.2 Implementacja testu przecięcia promień-trójkąt

```

1 TriangleHit TriangleIntersection(Ray ray, Triangle tri)
2 {
3     TriangleHit result;
4     result.t = -1.0f;
5     result.u = -1.0f;
6     result.v = -1.0f;
7     float3 e1 = tri.v2.xyz - tri.v1.xyz;
8     float3 e2 = tri.v3.xyz - tri.v1.xyz;
9
10    float3 q = cross(ray.direction, e2);
11    float a = dot(e1, q);
12
13    if (a > -EPSILON && a < EPSILON)
14    {
15        return result;
16    }
17
18    float f = 1 / a;
```

```

19
20     float3 s = ray.origin - tri.v1.xyz;
21     float u = f * dot(s, q);
22
23     if (u < 0.0f)
24     {
25         return result;
26     }
27
28     float3 r = cross(s, e1);
29     float v = f * dot(ray.direction, r);
30
31     if (v < 0.0f || u + v > 1.0f)
32     {
33         return result;
34     }
35
36     float t = f * dot(e2, r);
37
38     if (t > EPSILON)
39     {
40         result.t = t;
41         result.u = u;
42         result.v = v;
43         return result;
44     }
45     return result;
46 }
```

4.4.3 Przechodzenie przez drzewo BVH

```

1 int stack[32];
2 int stackPtr = 0;
3 stack[stackPtr++] = 0;
4
5 while (stackPtr > 0)
```

```

6  {
7      int nodeIdx = stack[--stackPtr];
8      BVHNode node = g_BVHNodes[nodeIdx];
9
10     float aabbDist = IntersectAABB(ray, node);
11
12     if (aabbDist < 0.0f || aabbDist >= info.hitDistance)
13     {
14         continue;
15     }
16
17     if (node.triangleCount > 0)
18     {
19         for (uint k = 0; k < node.triangleCount; k++)
20         {
21             int triIdx = g_TriIndexes[node.leftFirst + k];
22             Triangle tri = g_Triangles[triIdx];
23
24             TriangleHit hit = TriangleIntersection(ray, tri
25                                         );
26
27             if (hit.t < 0.0f)
28             {
29                 continue;
30             }
31
32             if (hit.t < info.hitDistance)
33             {
34                 info.hitDistance = hit.t;
35                 info.t = hit.t;
36                 info.hitPoint = RayAt(ray, hit.t);
37
38                 float w = 1.0f - hit.u - hit.v;
39
40                 info.normal = normalize(tri.n1.xyz * w +
41                                         tri.n2.xyz * hit.u + tri.n3.xyz * hit.v)

```

```

        ;
40         info.objectIndex = triIdx;
41         info.materialIndex = 3;
42     }
43 }
44
45 }
46 else
47 {
48     int leftChild = node.leftFirst;
49     int rightChild = node.leftFirst + 1;
50
51     if (rightChild < numOfNodes)
52     {
53         stack[stackPtr++] = rightChild;
54     }
55     if (leftChild < numOfNodes)
56     {
57         stack[stackPtr++] = leftChild;
58     }
59 }
60 }
```

Silnik używa shaderów w wersji 5 (model 5.0), ta wersja nie może używać rekurencji (rekurencja dostępna jest dla shaderów DXR używających wsparcia sprzętowego RTCores [16]), dlatego funkcja imituje rekurencje używając "stos". Z rozdziału 2 wiadomo, że jeśli $triangleCount > 0$, to wierzchołek jest liściem, po sprawdzeniu wykonuje się standardowy kod sprawdzający przecięcie z trójkątem. Jeśli wierzchołek nie jest liściem, to jego potomków dodaje się do stosu.

4.4.4 Główna logika ray tracingu

Poniżej fragment funkcji implementującej model oświetlenia opisany w rozdziale 3.

```

1   Material material = g_Materials[info.materialIndex];
2
3   light += GetEmission(material) * contribution;
4
5   ray.origin = info.hitPoint + info.normal * EPSILON;
6
7   float3 V = -ray.direction;
8
9   float3 Fdielectics = float3(0.04f, 0.04f, 0.04f);
10  float3 F0 = lerp(Fdielectrics, material.albedo.xyz,
11    material.metalness);
12
13  float cosTheta = dot(info.normal, V);
14  float3 F = FresnelSchlick(max(cosTheta, 0.0f), F0);
15
16  float reflectionChance = max(F.r, max(F.g, F.b));
17  float randomValue = RandomFloat(seed);
18  float2 xi = float2(RandomFloat(seed), RandomFloat(seed))
19    ;
20
21  if (randomValue < reflectionChance)
22  {
23
24    float3 H = SampleGGX(xi, info.normal, material.
25      roughness);
26    float3 L = reflect(-V, H);
27
28    float NdotV = saturate(dot(info.normal, V));
29    float NdotL = saturate(dot(info.normal, L));
30    float NdotH = saturate(dot(info.normal, H));
31    float VdotH = saturate(dot(V, H));
32
33    if (NdotL > 0.0f)
34    {
35      float G = G_Smith(material.roughness, NdotV,
36        NdotL);

```

```

33         float3 weight = (F * G * VdotH) / max(NdotH *
34                                         NdotV, EPSILON);
35
36         contribution *= weight / reflectionChance;
37         ray.direction = L;
38     }
39     else
40     {
41         return float3(0, 0, 0);
42     }
43     else
44     {
45         if (material.metalness >= 1.0f)
46         {
47             return light;
48         }
49         float3 L = normalize(RandomVec3OnUnitHemiSphere(
50                                         seed, info.normal));
51         float NdotL = saturate(dot(info.normal, L));
52
53         float3 kd = (1.0f - F) * (1.0f - material.metalness
54                                         );
55         contribution *= (material.albedo.xyz * kd * NdotL *
56                         2.0f) / (1.0f - reflectionChance);
57         ray.direction = L;
58     }

```

Zmienna *contribution* odpowiada za śledzenie strat energii światła podczas każdego odbicia promienia od powierzchni. Jest związana z zasadą zachowania energii, którą trzeba przestrzegać implementując PBR. Zmieniona inicjalizowana jest wartością 1.

Zmienna *light* oznacza zakumulowany kolor dla piksela, końcowy wynik obliczeń dla danej ścieżki.

Podsumowanie

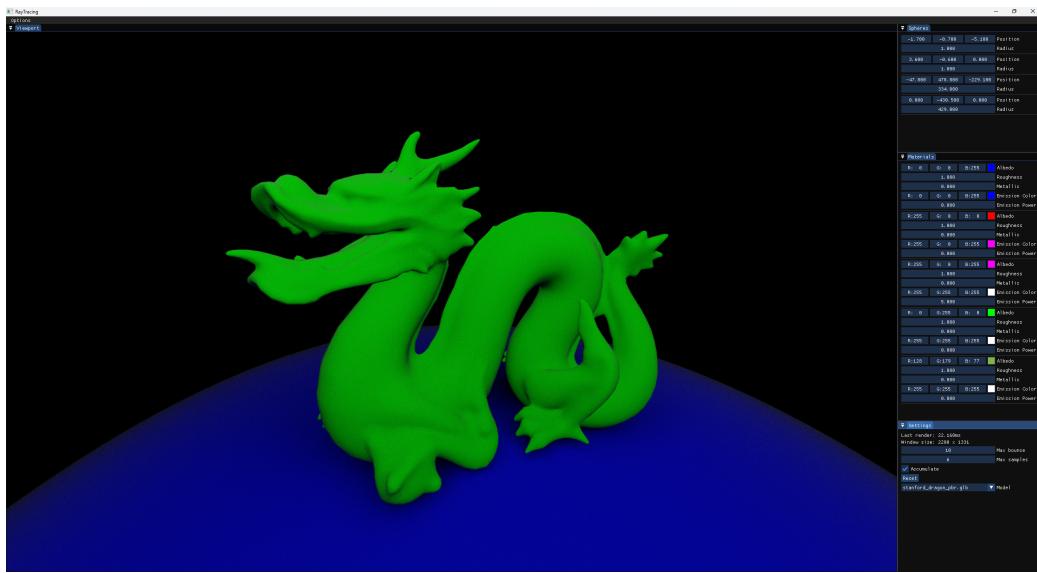
4.5 Testy

Po zaimplementowaniu drzewa BVH zostały wykonane testy wydajności silnika. Testy opierały się mierzeniu czasu jaki zajęło zrenderowanie jednej klatki w rozdzielczości 2200 x 1331, przy 10 odbiciach i 6 promieniach na piksel. Test zapisywał czasy renderowania sceny dla 3000 klatek, następnie na podstawie zgromadzonych danych wyznaczono średnią arytmetyczną czasu potrzebnego na wygenerowanie obrazu. Renderowano model Stanford dragon zbudowany z 19332 trójkątów. Testy odbywały się w konfiguracji release z włączoną optymalizacją /Ox favor speed w Visual studio 2022 na karcie graficznej AMD Radeon RX 9070 xt.

Tabela 4.1: Wyniki testu wydajności dla modelu Stanford Dragon

Wariant testowy	Średni czas renderowania klatki [ms]
Stanford Dragon (z BVH)	19.54
Stanford Dragon (bez BVH)	697.32

Nie udało się dokładnie przetestować wydajności w wariancie bez BVH. Czas renderowania modelu w rozdzielczości 2200 x 1331, 2 odbiciom i 1 promieniu na piksel wyniósł prawie 700 ms, dalsze badanie nie miało sensu. Wynik jednoznacznie potwierdza wymóg implementowania drzew BVH dla optymalizacji, bez tych struktur ray tracing w czasie rzeczywistym byłby niemożliwy.



Rysunek 4.4: Zdjęcie testowanego modelu

Przeprowadzono również testy dla mniejszego modelu zbudowanego z 264 trójkątów.

Tabela 4.2: Wyniki testu wydajności dla modelu Tree

Wariant testowy	Średni czas renderowania klatki [ms]
Tree (z BVH)	7.76
Tree (bez BVH)	60.00

Poza testami wydajności, testowano również poprawność stworzonego potoku graficznego wykorzystując wbudowane w bibliotekę DX11 funkcje.

```

1  UINT flags = D3D11_CREATE_DEVICE_SINGLETHREADED;
2
3 #ifdef _DEBUG
4         flags |= D3D11_CREATE_DEVICE_DEBUG;
5 #endif
6
7     CHECK(D3D11CreateDevice(nullptr,
8             D3D_DRIVER_TYPE_HARDWARE, nullptr,
9             flags,
```

```
9         featureLevelArray, 2, D3D11_SDK_VERSION,
10        &result.device, &result.featureLevel, &
11        result.deviceContext));
```

Flaga D3D11_CREATE_DEVICE_DEBUG aktywuje warstwę debugowania w DX11. Tryb ten rozszerza kontrolę błędów wynikających z: przesyłania nieprawidłowych danych do GPU, błędnych parametrów w funkcjach tworzących potok, wycieków pamięci itd. W przypadku wystąpienia nieprawidłowości, warstwa diagnostyczna generuje szczegółowe komunikaty w oknie wyjściowym debuggera [17].

4.6 Zakończenie

Rozwój kart graficznych sprawił, że implementacja ray tracingu/path tracingu w czasie rzeczywistym nie jest czymś niemożliwym. Przedstawiony silnik nie używa pełnych możliwości sprzętu (RTCores), jednak jego wydajność po optymalizacji pozwoliła na przedstawienie podstawowych algorytmów w temacie ray tracingu.

Bibliografia

- [1] Turner Whitted. "An Improved illumination model for shaded display". W: *Communications of the ACM* (1980).
- [2] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki i Sébastien Hillaire. *Real-Time Rendering*. 4 wyd. 2018.
- [3] Krzysztof Korna. *Fizyka w doświadczeniach*. Dostęp 24.01.2026. URL: https://www.fuw.edu.pl/~kkorona/wwwykl/skrypt_w13.pdf.
- [4] Tomas Möller i Ben Trumbore. "Fast Minimum Storage Ray-Triangle Intersection". W: *Journal of Graphics Tools* (1997).
- [5] Doug Baldwin i Michael Weber. "Fast Ray-Triangle Intersections by Coordinate Transformation". W: *Journal of Computer Graphics Techniques* (2016).
- [6] Jacco Bikker. *How to build a BVH Part 1: Basics*. Dostęp 24.01.2026. URL: <https://jacoco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/>.
- [7] Wikipedia. Dostęp 5.01.2026. URL: <https://en.wikipedia.org/wiki/Centroid>.
- [8] Matt Pharr, Jakob Wenzel i Greg Humphreys. *Physically Based Rendering. From Theory To Implementation*. 2023.
- [9] Rodolphe Vaillant's homepage. Dostęp 17.01.2026. URL: <https://rodolphe-vaillant.fr/entry/85/phong-illumination-model-cheat-sheet>.

- [10] Robert Cook i Kenneth Torrance. "A Reflectance Model for Computer Graphics". W: (1982).
- [11] *Real Shading in Unreal Engine 4*. Dostęp 24.01.2026. URL: <https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf>.
- [12] *Wikipedia*. Dostęp 24.01.2026. URL: https://en.wikipedia.org/wiki/Schlick%27s_approximation.
- [13] Frank Luna. *Introduction to 3D Game Programming with DirectX 11*. 2012.
- [14] Michał Drobot. *GCN Execution Patterns in Full Screen Passes*. Dostęp 22.01.2026. URL: <https://michaldrobot.com/2014/04/01/gcn-execution-patterns-in-full-screen-passes/>.
- [15] *Optimizing Triangles for a Full-screen Pass*. Dostęp 22.01.2026. URL: <https://wallisc.github.io/rendering/2021/04/18/Fullscreen-Pass.html>.
- [16] *DirectX Raytracing (DXR) Functional Spec*. Dostęp 23.01.2026. URL: <https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#ray-recursion-limit>.
- [17] *Software Layers*. Dostęp 24.01.2026. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-devices-layers>.