

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

Spis treści

Wstęp	3
1. Podstawy Ray Tracingu	4
1.1. Definicje i oznaczenia	4
1.2. Algorytm ray tracingu	4
1.3. Badanie przecięcia promienia ze sferą	7
1.4. Badanie przecięcia promienia z trójkątem	9
2. Optymalizacja - Bounding Volume Hierarchy	12
2.1. Optymalizacja przecinania promienia z trójkątem	12
2.2. Zasada działania BVH	12
2.3. Konstrukcja drzewa BVH	14
2.4. Przechodzenie przez drzewo BVH	14
2.5. Surface Area Heuristic	17
3. Materiały i modele oświetlenia	20
3.1. Model oświetlenia lokalnego	20
3.2. Model oświetlenia globalnego	24
4. Biblioteki i narzędzia	29
4.1. DirectX 11	29
4.2. Win32	29
4.3. ImGui	29
4.4. Assimp	30
4.5. Stb_Image	30
4.6. Visual Studio	30
4.7. Premake	30
5. Prezentacja silnika	31
5.1. Interfejs	31

Spis treści

5.2. Elementy silnika	32
5.3. Opis potoku graficznego	32
5.4. Implementacja algorytmów	34
5.4.1. Implementacja testu przecięcia promień-sfera	35
5.4.2. Implementacja testu przecięcia promień-trójkąt	36
5.4.3. Implementacja testu przecięcia promienia z AABB	37
5.4.4. Konstrukcja BVH	38
5.4.5. Przechodzenie przez drzewo BVH	43
5.4.6. Główna logika ray tracingu	45
Podsumowanie	48
5.1. Testy	48
5.2. Zakończenie	50
Bibliografia	51

Wstęp

W 2018 roku NVIDIA przedstawiła światu nową generację kart graficznych nazywanych RTX. Od tego roku każda kolejna seria poczawszy 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 obiektami 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 będzie 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 różne wydajności.

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 $v = (x, y, z)$.
- 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).

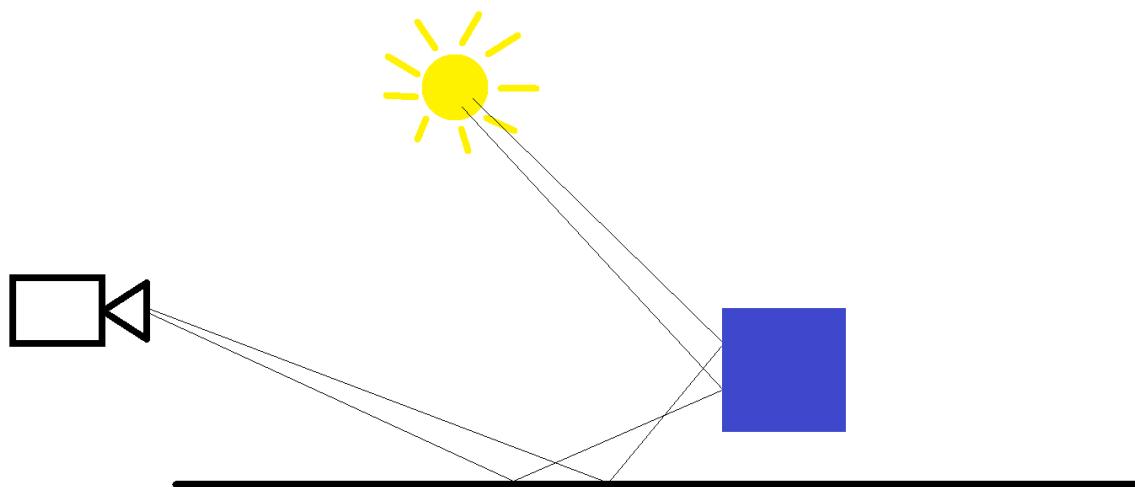
1.2. Algorytm ray tracingu

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. Klasyczny

1. Podstawy Ray Tracingu

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 przedstawia opisaną ideę algorytmu. Promienie wychodzą z kamery i uderzają w obiekt, następnie generowane są kolejne promienie, które w pewnym momencie trafiają do źródła światła.



Rysunek 1.1.: Rysunek przedstawia ideę ray tracingu.

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 w skład światła widzialnego wchodzą fale o różnej długości. Zakres długości fal dla światła widzialnego przez człowieka wynosi około 380 nm–780 nm [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 nm–780 nm.

Budowę oka ludzkiego można podzielić na 3 błony [4]:

- zewnętrzna błona włóknista składająca się z twardówki i rogówki,
- środkowa naczyniowa złożona z tęczówki, ciała rzęskowego i naczyniówka,

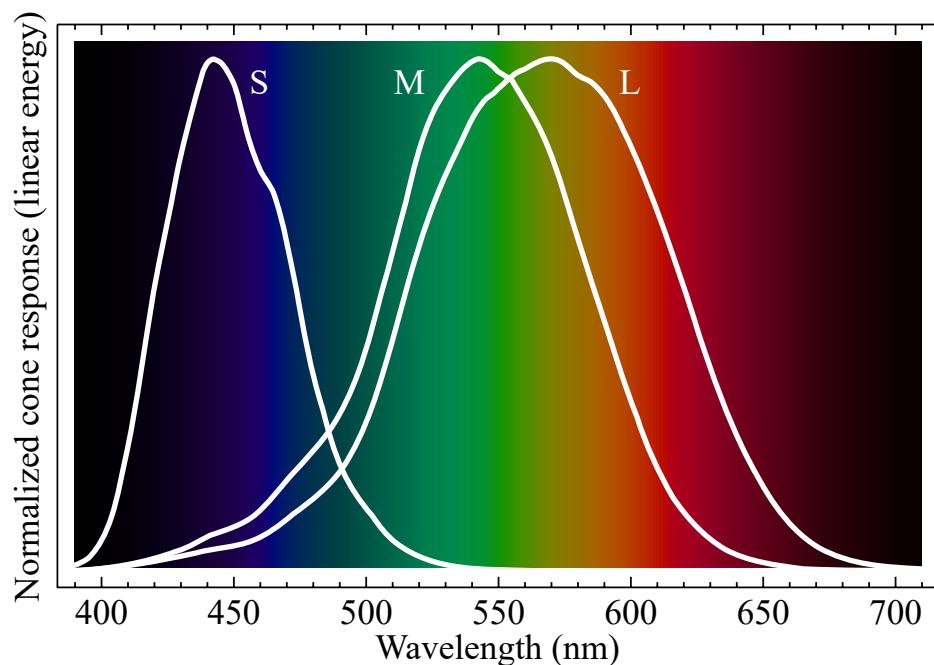
1. Podstawy Ray Tracingu

- wewnętrznej czuciowej (siatkówka).

Promień światła przechodząc przez żrenicę i soczewkę trafiają na siatkówkę. W siatkówce odbywają się procesy fizyczne i biochemiczne, które przetwarzają bodziec świetlny na bodziec nerwowy. Odpowiadają za to tzw. czopki i prećiki. W oku ludzkim znajduje się około 7 milionów czopków i 130 milionów prećików [4].

Czopki można podzielić pod względem czułości na inne długości fali:

- Czopki typu L – maksymalna czułość około 564 nm. Odpowiedzialne za detekcję światła czerwonego.
- Czopki typu M – maksymalna czułość około 534 nm. Odpowiedzialne za detekcję światła zielonego.
- Czopki typu S – maksymalna czułość około 420 nm. Odpowiedzialne za detekcję światła niebieskiego.



Rysunek 1.2.: Obrazek przedstawia znormalizowaną czułość spektralną poszczególnych czopków. [5]

Czopki odpowiadają za widzenie kształtu i barw przedmiotów w jasnym oświetleniu. Prećiki zajmują się przygotowaniem oka do słabego oświetlenia i do rozróżniania zarysów przedmiotów [4].

1. Podstawy Ray Tracingu

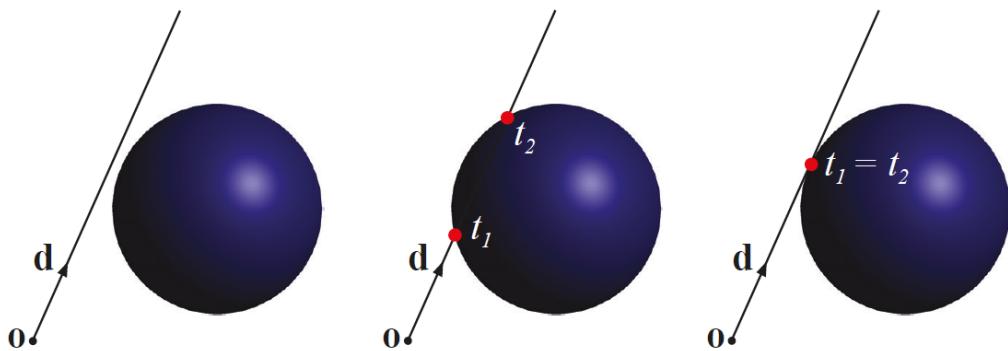
Bazując na budowie ludzkiego oka i informacji w jaki sposób człowiek interpretuje kolory stworzono model RGB. RGB jest modelem addytywnym, nazywane są tak modele w których wynikiem dodania do siebie podstawowych barw (czerwonego, zielonego i niebieskiego) jest barwa biała, w modelach subtraktywnych wynikiem jest kolor czarny.

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

Minusem algorytmu jest jego złożoność obliczeniowa. Dla przykładu, bez optymalizacji mając model złożony z 100 000 trójkątów, w rozdzielcości 2560x1440 (3 686 400 pikseli), mając ustawione 2 próbki na piksel i 5 odbiciach, program musiałby dla każdego piksela wykonać 1 000 000 testów.

1.3. Badanie przecięcia promienia ze sferą

Test przecięcia promienia ze sferą jest jednym z najprostszych algorytmów tego typu i idealnie nadaje się do przedstawienia procesu wyprowadzania takich algorytmów.



Rysunek 1.3.: Obrazek przedstawia możliwe warianty przecięcia sfery [2]

Wyprowadzenie testu należy zacząć od zapisania równania sfery o środku

1. Podstawy Ray Tracingu

w punkcie $\mathbf{C} = (c_x, c_y, c_z)$ i promieniu r .

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

Punkt $\mathbf{P} = (p_x, p_y, p_z)$ jest punktem leżącym na sferze. Można zapisać wektor o długości r ze środka sfery do tego punktu zapisując:

$$(\mathbf{P} - \mathbf{C}) \quad (1.2)$$

Używając powyższych oznaczeń równanie sfery można zapisać tak:

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = r^2 \quad (1.3)$$

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

Szukaną wartością w tym badaniu jest zmiana t , na podstawie której można obliczyć współrzędne punktu \mathbf{P} . Równanie opisujące promień o punkcie początkowym \mathbf{O} i kierunku zdefiniowanym przez wektor d :

$$\mathbf{P}(t) = \mathbf{O} + t\mathbf{d} \quad (1.4)$$

Wstawiając (1.4) do (1.3):

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

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

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2t\mathbf{d} \cdot (\mathbf{O} - \mathbf{C}) + (\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - r^2 = 0 \quad (1.6)$$

Jest to równanie kwadratowe o współczynnikach:

$$a = \mathbf{d} \cdot \mathbf{d}, \quad (1.7a)$$

$$b = 2\mathbf{d} \cdot (\mathbf{O} - \mathbf{C}), \quad (1.7b)$$

$$c = (\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - r^2. \quad (1.7c)$$

Otrzymujemy następujący wzór na parametr t :

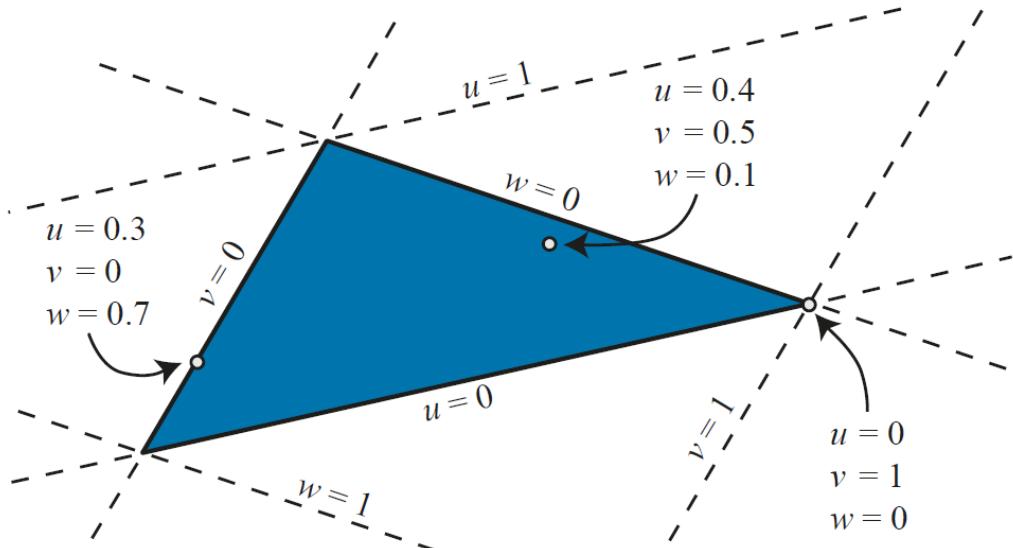
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (1.8)$$

1. Podstawy Ray Tracingu

Równanie (1.6) może mieć rozwiązań 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 punkt przecięcia promienia ze sferą jest za kamerą). W ramach oznaczenia przypadku braku rozwiązania równania (1.6) z funkcji zwracana jest wartość -1.0 .

1.4. Badanie przecięcia promienia z trójkątem

Autorami przedstawionego poniżej algorytmu są Tomas Möller i Ben Trumbore [6]. 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 [7], ale wymagają one wcześniejszego obliczenia transformacji współrzędnych z systemu globalnego do barycentrycznego. Koszt dostępu do tych danych może być zbyt duży w porównaniu z zyskami szybkości względem algorytmu Möller'a Trumbore'a.



Rysunek 1.4.: Obrazek przedstawia współrzędne barycentryczne trójkąta [2]

Definicja 1.4.1 *Współrzędne barycentryczne na trójkącie o wierzchołkach w punk-*

1. Podstawy Ray Tracingu

takie P_0, P_1, P_2 to trójkąt liczb $(w, u, v) \in \mathbb{R}$ spełniających następujące warunki:

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

Każdy punkt wewnątrz trójkąta można przedstawić jako:

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

Znalezienie punktu przecięcia sprowadza się do rozwiązyania układu równań:

$$\mathbf{O} + t\mathbf{D} = (1 - u - v)\mathbf{P}_0 + u\mathbf{P}_1 + v\mathbf{P}_2 \quad (1.9)$$

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

Po przepisaniu na postać macierzową:

$$\begin{bmatrix} -\mathbf{D} & \mathbf{P}_1 - \mathbf{P}_0 & \mathbf{P}_2 - \mathbf{P}_0 \end{bmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{O} - \mathbf{P}_0 \quad (1.10)$$

Wprowadzono następujące oznaczenia: $\mathbf{e}_1 = \mathbf{P}_1 - \mathbf{P}_0, \mathbf{e}_2 = \mathbf{P}_2 - \mathbf{P}_0, \mathbf{s} = \mathbf{O} - \mathbf{P}_0$
Używając oznaczeń układ równań ma postać:

$$\begin{aligned} -d_x t + e_{1x} u + e_{2x} v &= s_x \\ -d_y t + e_{1y} u + e_{2y} v &= s_y \\ -d_z t + e_{1z} u + e_{2z} v &= s_z \end{aligned} \quad (1.11)$$

Wtedy macierz główna układu ma postać:

$$M = \begin{bmatrix} -d_x & e_{1x} & e_{2x} \\ -d_y & e_{1y} & e_{2y} \\ -d_z & e_{1z} & e_{2z} \end{bmatrix} \quad (1.12)$$

Do rozwiązyania układu równań można wykorzystać metodę Cramera:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{D}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{D}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{D}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix} \quad (1.13)$$

1. Podstawy Ray Tracingu

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

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\mathbf{D} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{D} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{D} \end{pmatrix} \quad (1.14)$$

Wprowadzając kolejne oznaczenia, dla uproszczenia: $\mathbf{q} = \mathbf{D} \times \mathbf{e}_2$
 $\mathbf{r} = \mathbf{s} \times \mathbf{e}_1$ Otrzymane równanie:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\mathbf{q} \cdot \mathbf{e}_1} \begin{pmatrix} \mathbf{r} \cdot \mathbf{e}_2 \\ \mathbf{q} \cdot \mathbf{s} \\ \mathbf{r} \cdot \mathbf{D} \end{pmatrix} \quad (1.15)$$

Implementując algorytm tworzy się funkcję, która zwraca parametr t , dzięki niemu znany jest punkt przecięcia.

Przypadki szczególne (brak trafienia w trójkąt, trójkąt znajduje się za punktem początkowym promienia) wykrywa się sprawdzając wartości pośrednie. Pierwszym sprawdzeniem jest upewnienie się, że wyznacznik główny jest różny od 0 ($\mathbf{q} \cdot \mathbf{e}_1 \neq 0$). Następnie sprawdza się czy wartości u i v spełniają warunki z 1.4.1. Ostatnim krokiem jest sprawdzenie wartości t , musi być większa od 0 (punkt przecięcia nie może być za kamerą).

2. Optymalizacja - Bounding Volume Hierarchy

2.1. Optymalizacja przecinania promienia z trójkątem

Wydajność renderowania scen za pomocą ray tracingu zależy od kilku parametrów: rozdzielczości obrazu, liczby odbić promienia, liczby promieni na piksel i liczby trójkątów w scenie. Wzrost tych wartości przekłada się na wyższą szczegółowość sceny, jednak zdecydowanie zwiększa koszt jej wyrenderowania. Głównym polem optymalizacji jest minimalizacja liczby testów przecięcia promień-trójkąt. Do takiej optymalizacji można posłużyć się tzw. drzewami BVH – Bounding Volume Hierarchy.

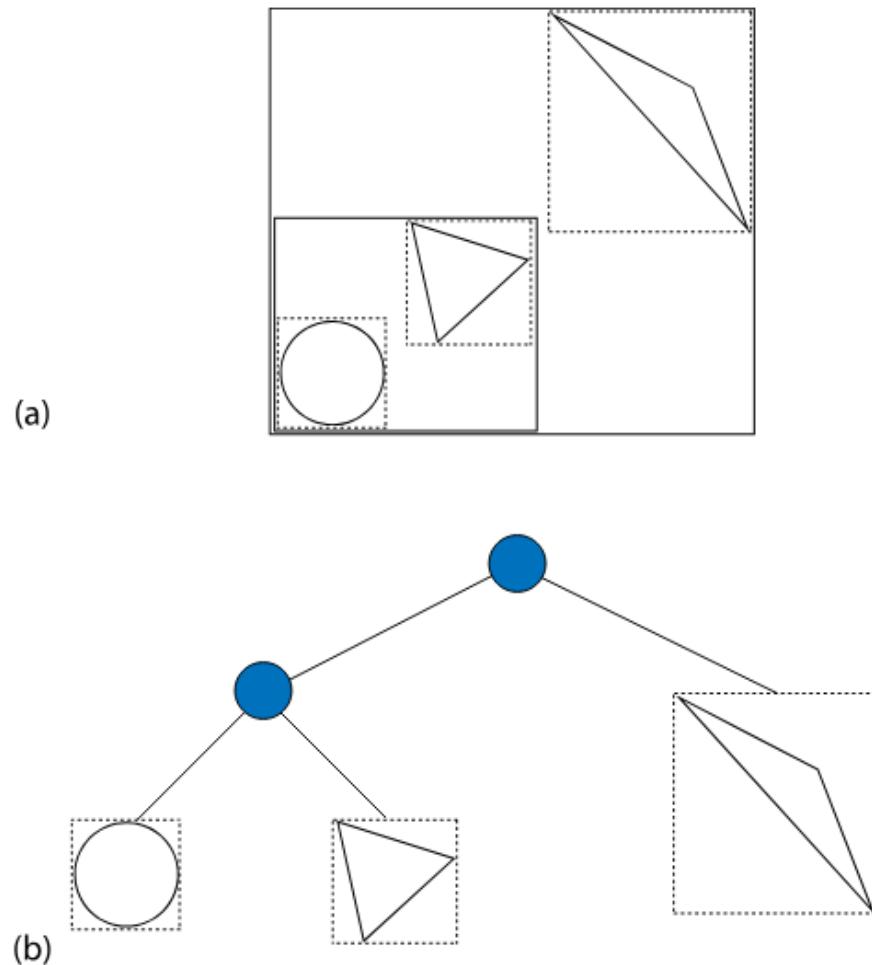
Idea algorytmu jest prosta - pomijać te trójkąty których promień na pewno nie przetnie. Bez optymalizacji, aby sprawdzić czy promień przecina się z jakimś trójkątem, trzeba przejść przez całą tablicę trójkątów modelu i każdy przetestować, w przypadku użycia BVH większość pomijamy badając tylko te najbliższe 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 mieścią się kolejne prostopadłościany z podziału, a w liściach znajdują się trójkąty [2].

W strukturze BVH używa się różnych typów „bounding box”, w ray tracingu popularnym wyborem są „axis-aligned bounding box” (AABB). AABB tworzone są po-

2. Optymalizacja - Bounding Volume Hierarchy



Rysunek 2.1.: Obrazek przedstawiający BVH. Podpunkt *a* przedstawia objętości (bboxy), a w nich obiekty. Podpunkt *b* pokazuje drzewo BVH stworzone na podstawie sceny z podpunktu *a* [8].

2. Optymalizacja - Bounding Volume Hierarchy

przez 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.

Pseudokod algorytmu wyznaczania prostopadłościanu ograniczającego (axis aligned bounding box) dla zbioru trójkątów pokazany jest jako algorytm. 1.

Algorithm 1 Wyznaczanie AABB

```
Triangles  $\leftarrow [ (v_1, v_2, v_3), \dots ]$ 
 $p_{\min} \leftarrow (\infty, \infty, \infty)$ 
 $p_{\max} \leftarrow (-\infty, -\infty, -\infty)$ 
for each triangle  $t$  in Triangles do
     $p_{\min} \leftarrow (\min(P_{\min}, t.v_1))$ 
     $p_{\min} \leftarrow (\min(P_{\min}, t.v_2))$ 
     $p_{\min} \leftarrow (\min(P_{\min}, t.v_3))$ 
     $p_{\max} \leftarrow (\max(P_{\max}, t.v_1))$ 
     $p_{\max} \leftarrow (\max(P_{\max}, t.v_2))$ 
     $p_{\max} \leftarrow (\max(P_{\max}, t.v_3))$ 
end for
```

2.3. Konstrukcja drzewa BVH

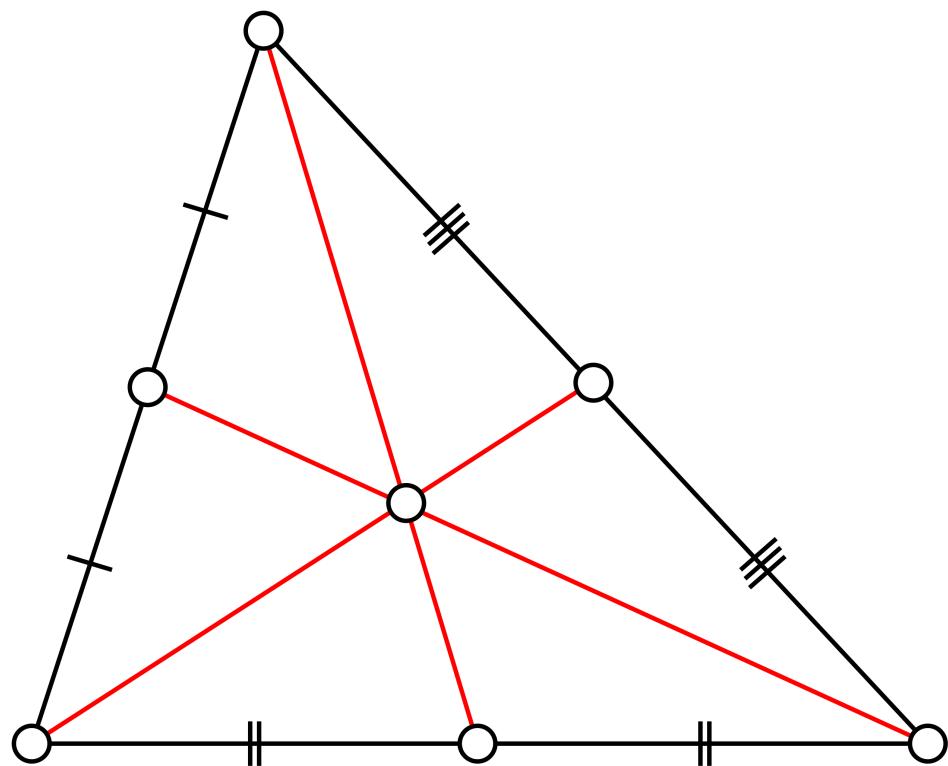
Aby stworzyć strukturę drzewa należy teraz podjąć decyzję w jaki sposób dzielić model (AABB boxy). Dla prostoty algorytmu dzielić można wzdłuż najdłuższej osi [2]. Następnie przypisujemy trójkąty do jednego lub drugiego poddrzewa BVH poprzez sprawdzanie czy jego centroid¹ leży po jednej czy drugiej stronie podziału.

2.4. 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 method.

¹Centroid to punkt przecięcia median trójkąta (mediana to odcinek łączący wierzchołek ze środkiem przeciwnego boku) [9]

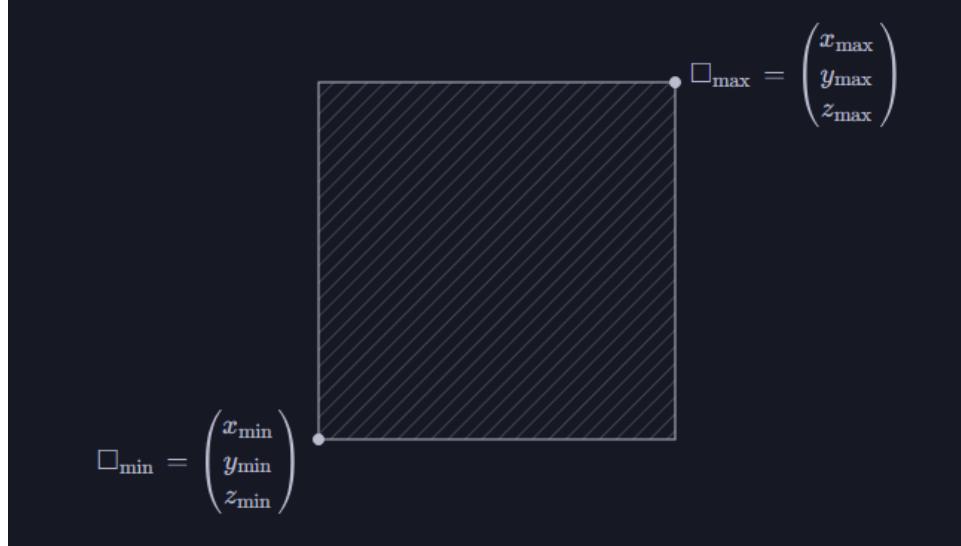
2. Optymalizacja - Bounding Volume Hierarchy



Rysunek 2.2.: Obrazek przedstawiający centroid trójkąta [9].

2. Optymalizacja - Bounding Volume Hierarchy

Slab test polega na sprawdzaniu czy promień przecina wszystkie płaszczyzny wyznaczane przez osie x, y, z. Dla każdej osi należy obliczyć parametr t , który wyznacza możliwy punkt przecięcia (1.4). AABB box zdefinowany jest jako dwa punkty $P_{min} = (x_{min}, y_{min}, z_{min})$, $P_{max} = (x_{max}, y_{max}, z_{max})$, jak na rysunku 2.3.



Rysunek 2.3.: Obrazek przedstawiający AABB [10].

Wzory na parametr t :

$$t_1 = \frac{\mathbf{P}_{min} - \mathbf{O}}{\mathbf{D}} \quad (2.1)$$

$$t_2 = \frac{\mathbf{P}_{max} - \mathbf{O}}{\mathbf{D}} \quad (2.2)$$

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.

2. Optymalizacja - Bounding Volume Hierarchy

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 prawego potomka.

Podział nie zawsze jest optymalny, jak pokazuje rysunek 2.4. W podpunkcie b widać, że AABB nachodzą się co powoduje, że sprawdzając przecięty trójkąt trzeba przejść przez dwóch potomków. Dlatego opracowano również inne metody.

2.5. Surface Area Heuristic

W tej metodzie przy konstrukcji każdego podziału oblicza się koszt przeprowadzania testów przecięcia promienia z danym obiektem. Wzór na koszt [8]:

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

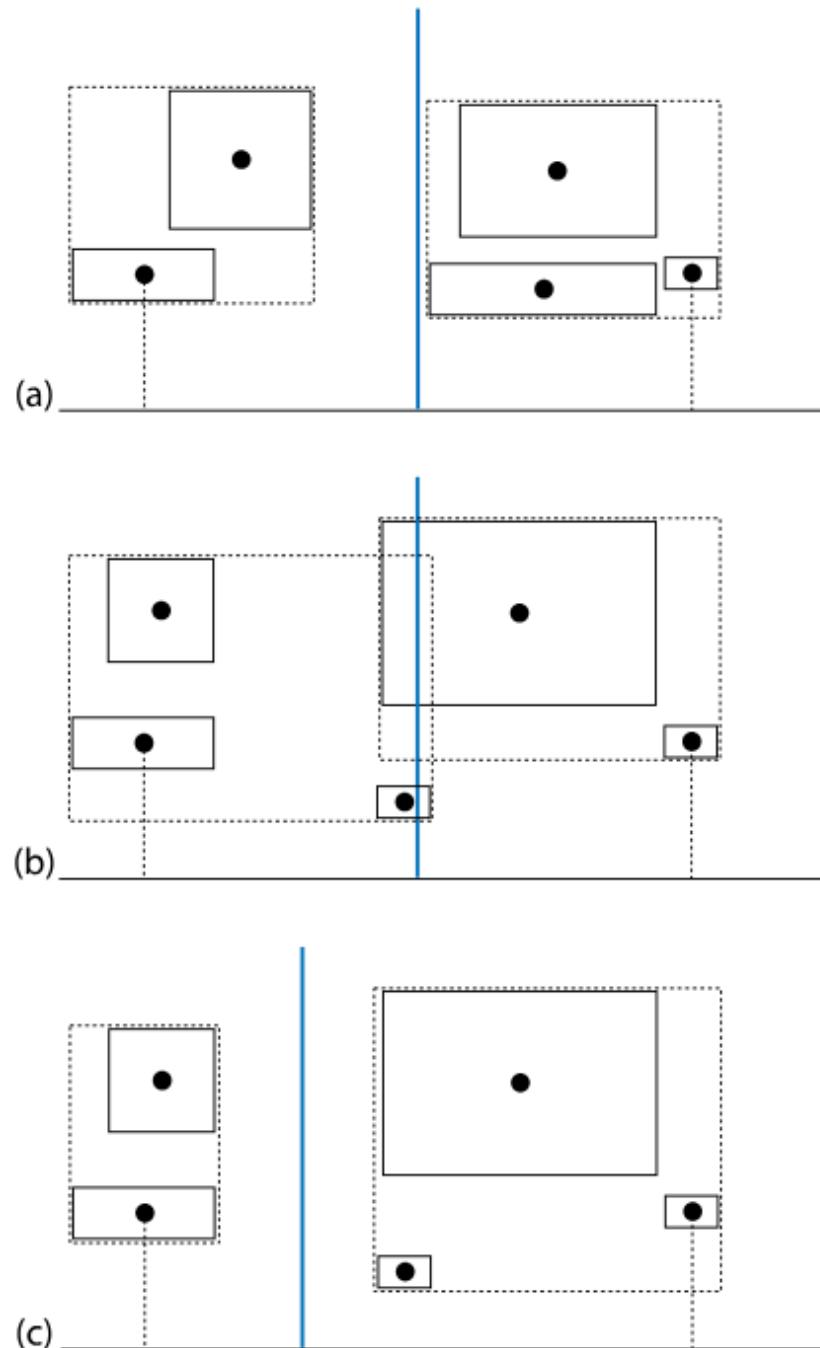
- t_{trav} - czas potrzebny na ustalenie przez które dzieci przechodzi promień.
- p_A - to prawdopodobieństwo, że przez wierzchołek A przejdzie promień (odpowiednio p_B).
- $\sum_{i=1}^{N_A} t_{isect}(a_i)$ - Czas potrzebny na przeprowadzenie testu przecięcia promienia z obiektem dla każdego obiektu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

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

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

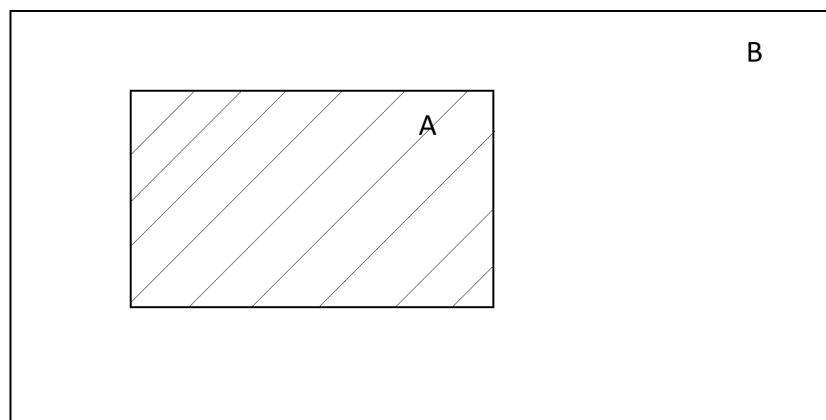
Wzór opisuje prawdopodobieństwo przecięcia powierzchni A jeśli przecięta została powierzchnia B , rysunek 2.5

2. Optymalizacja - Bounding Volume Hierarchy



Rysunek 2.4.: Obrazek przedstawiający podział na podstawie punktu środkowego centroidów na osi [8].

2. Optymalizacja - Bounding Volume Hierarchy



Rysunek 2.5.: Rysunek przedstawiający powierzchnię A i B

3. Materiały i modele oświetlenia

Zachowanie światła padającego na dany obiekt będzie zależało od materiału z którego ten obiekt jest stworzony. Na przykład światło padające na metal będzie odbiął 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 [2]:

- 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.

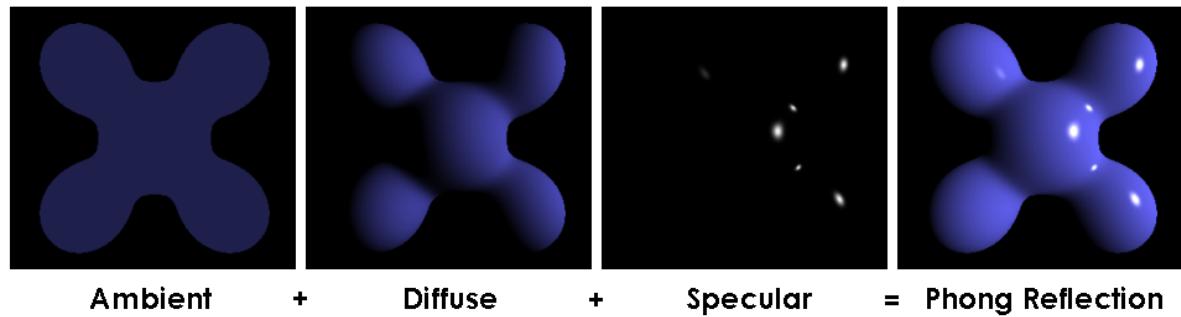
3.1. Model oświetlenia lokalnego

Jednym z najpopularniejszych modeli oświetlenia lokalnego jest model Phonga (ang. Phong reflection model). Model Phonga [11] składa się z trzech komponentów:

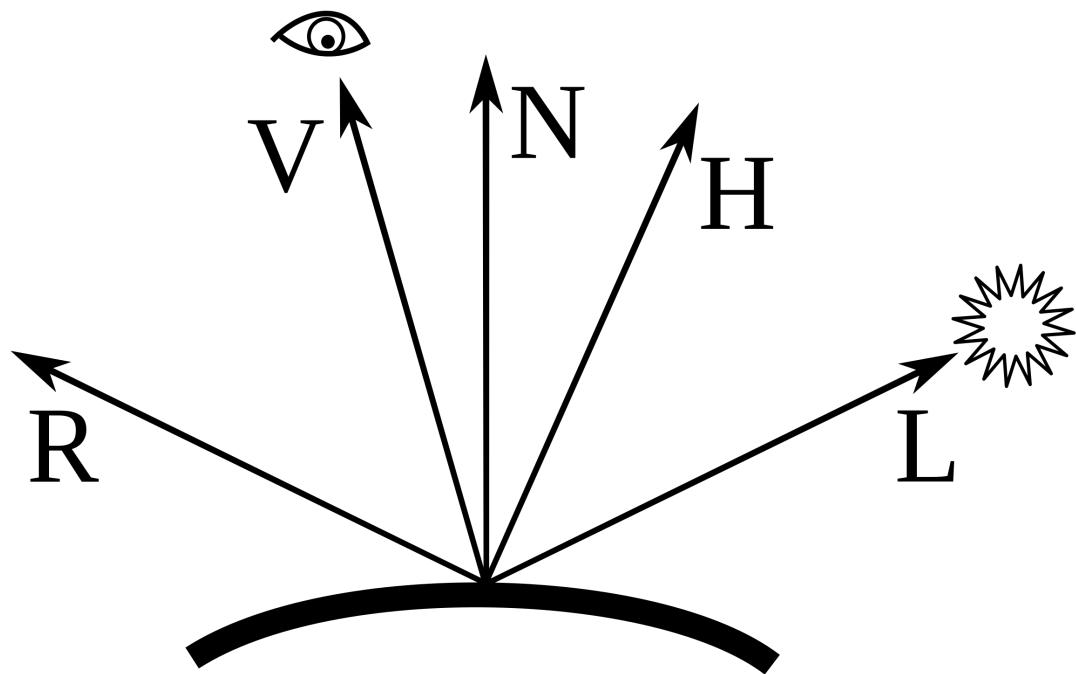
- Ambient - światło otoczenia. W tym algorytmie wpływ innych obiektów na jasność i kolor danego modelu (global illumination) jest symulowane poprzez dodanie pewnej stałej wartości do jasności i koloru obiektu.
- Diffuse - światło rozproszone. Najważniejszy komponent oświetlenia, światło padające na powierzchnię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

Wzór przedstawiający ostateczne oświetlenie danego punktu:

3. Materiały i modele oświetlenia



Rysunek 3.1.: Obrazek przedstawia komponenty tworzące model phonga [12]



Rysunek 3.2.: Wektory w modelu phonga [12]

3. Materiały i modele oświetlenia

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

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.
- \mathbf{r} - wektor kierunku odbicia światła obliczany według wzoru:

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

- \mathbf{v} - znormalizowany wektor kierunku do kamery.
- s - współczynnik przedstawiający rozmiar odblasku.

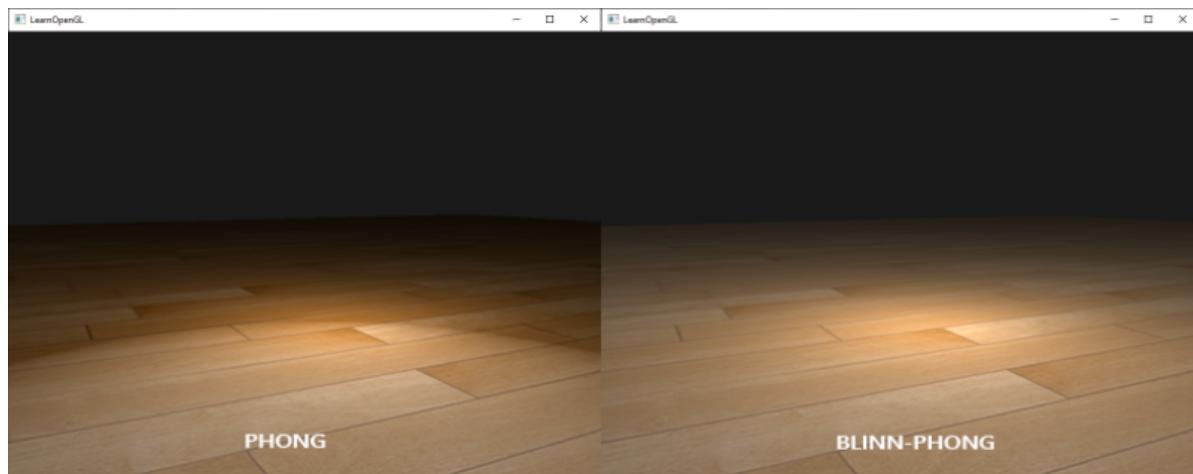
W przypadku wielu źródeł światła w scenie wzór przyjmuje postać:

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

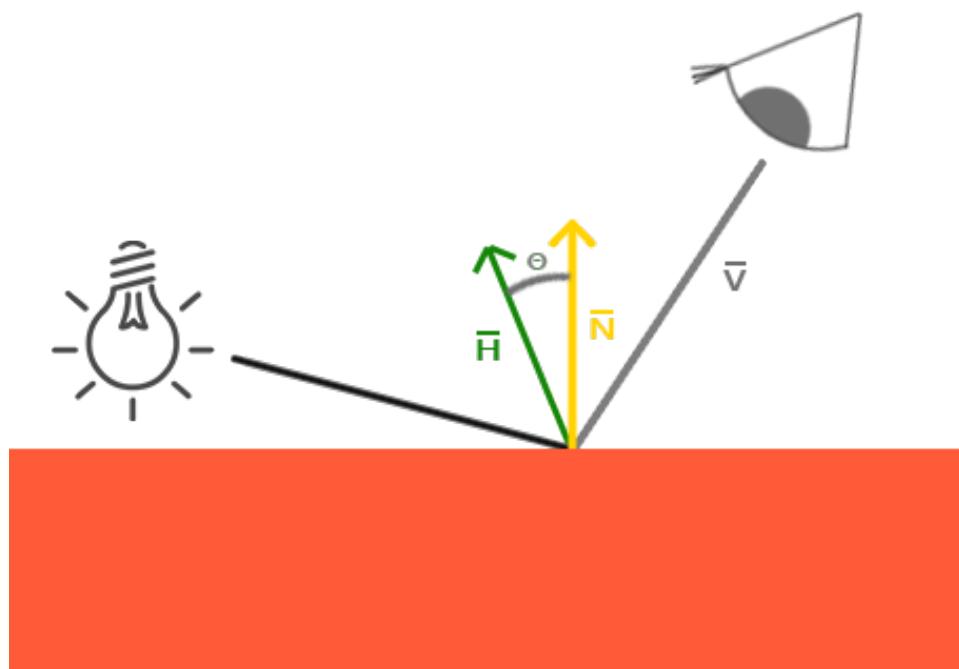
Wadą modelu Phonga jest „ucinanie się” odblasku lub jego zanik gdy kąt pomiędzy obserwatorem a wektorem odbicia jest większy niż 90 stopni, wtedy $\mathbf{r} \cdot \mathbf{v}$ jest ujemny, co doprowadza do zaniku komponentu odbicia lustrzanego (iloczyn skalarny ogranicza się od dołu 0). Problem rozwiązany jest w modelu Blinna-Phonga w którym oblicza się tzw. halfway vector (rysunek 3.4) $\mathbf{h} = \frac{\mathbf{l}+\mathbf{v}}{\|\mathbf{l}+\mathbf{v}\|}$ i zamienia się $\mathbf{r} \cdot \mathbf{v}$ na $\mathbf{n} \cdot \mathbf{h}$ [13].

$$I_{Blinn-Phong} = I_a k_a + k_d I_i (\mathbf{n} \cdot \mathbf{l}) + k_s I_i (\mathbf{n} \cdot \mathbf{h})^s \quad (3.3)$$

3. Materiały i modele oświetlenia



Rysunek 3.3.: Prówanie modeli Phonga i Blinna-Phonga [13]



Rysunek 3.4.: Rysunek przedstawia „halfway vector” [13]

3.2. 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 oświetlenia globalnego oparte są na tzw. równaniu renderingu (ang. *rendering equation*) [2]:

$$\mathbf{L}_o(\mathbf{p}, \mathbf{v}) = \mathbf{L}_e(\mathbf{p}, \mathbf{v}) + \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \mathbf{L}_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \quad (3.4)$$

gdzie:

- $\mathbf{L}_o(\mathbf{p}, \mathbf{v})$ - światło wychodzące od punktu \mathbf{p} w kierunku \mathbf{v} (kierunek kamery).
- $\mathbf{L}_e(\mathbf{p}, \mathbf{v})$ - światło emitowane przez powierzchnię w punkcie \mathbf{p} w kierunku \mathbf{v} .
- Ω - to powierzchnia półsfery, która znajduje się nad punktem dla którego oblicza się wychodzące światło. Całkowanie po tej półsferze odpowiada sumowaniu wkładu światła docierającego ze wszystkich możliwych kierunków.
- $f(\mathbf{l}, \mathbf{v})$ - to tzw. dwukierunkowa funkcja rozkładu odbicia (ang. *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.
- $\mathbf{L}_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})$ - 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.

Równanie 3.4 jest niemożliwe do rozwiązania analitycznego (poza bardzo prostymi scenami), ponieważ aby rozwiązać równanie dla punktu \mathbf{p} trzeba znać wynik

3. Materiały i modele oświetlenia

dla punktu wcześniejszego p' , powstaje nieskończona rekurencja na nieskończonej liczbie kierunków z których światło dochodzi do punktu p .

Równanie renderingu rozwiązuje się metodami numerycznymi np. metodą Monte–Carlo.

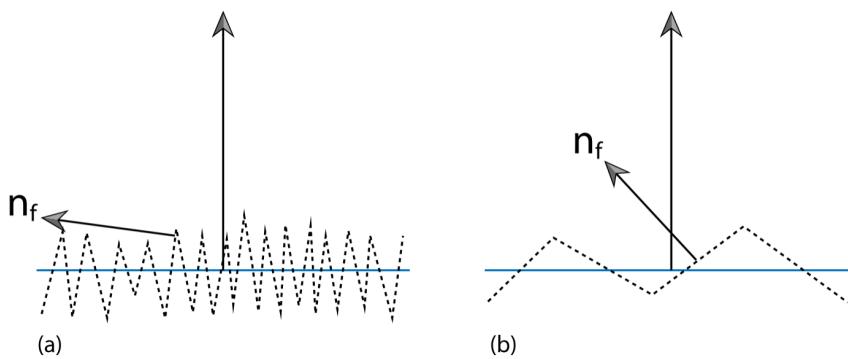
Aby przybliżyć wynik całki w równaniu 3.4 używając metodę Monte–Carlo należy wylosować N próbek i obliczyć średnią wartość funkcji pod całką.

Algorithm 2 Estymacja Monte Carlo koloru piksela (antialiasing)

```

1: totalColor  $\leftarrow (0, 0, 0)$ 
2: for  $i = 0$  to  $raysPerPixel$  do
3:    $\epsilon \leftarrow \text{RandomVec3}(seed, -0.001, 0.001)$ 
4:    $d_{jitter} \leftarrow \text{normalize}(ray.direction + \epsilon)$ 
5:    $jitteredRay \leftarrow \text{Ray}(ray.origin, d_{jitter})$ 
6:    $totalColor \leftarrow totalColor + \text{TraceRay}(jitteredRay, seed)$ 
7: end for
8: finalColor  $\leftarrow totalColor / \text{float}(raysPerPixel)$ 
```

Przykładem BRDF, który jest wykorzystywany w implementacji oświetlenia globalnego (jak i lokalnego) jest model Cook-Torrance [14]. 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 chropowatości (ang. *roughness*).



Rysunek 3.5.: Rysunek przedstawia mikropowierzchnie. [8]

Funkcja BRDF modelu Cook-Torrance dla odbić lustrzanych[15]:

3. Materiały i modele oświetlenia

$$f_{cookTorrance} = \frac{D(\mathbf{n}, \mathbf{h}, \alpha) F(\theta) G(\mathbf{n}, \mathbf{v}, \mathbf{l})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})} \quad (3.5)$$

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 załamaneego.
- G - Funkcja określająca atenuację (tłumienie) światła wynikające z nakładania 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 - „Ground Glass Unknown” [16]), a G to funkcja Smitha.

$$D_{GGX}(\mathbf{n}, \mathbf{h}, \alpha) = \frac{\alpha^2}{\pi((\mathbf{n} \cdot \mathbf{h})^2(\alpha^2 - 1) + 1)^2} \quad (3.6)$$

$$G(\mathbf{n}, \mathbf{v}, \mathbf{l}) = G_1(\mathbf{n}, \mathbf{v}) \cdot G_1(\mathbf{n}, \mathbf{l}) \quad (3.7)$$

$$G_1(\mathbf{n}, \mathbf{v}) = \frac{\mathbf{n} \cdot \mathbf{v}}{(\mathbf{n} \cdot \mathbf{v})(1 - k) + k} \quad (3.8)$$

gdzie:

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

Współczynnik Fresnala w większości przypadków przybliża się tzw. przybliżeniem Schlick'a[17].

$$F_{Schlick} = F_0 + (1 - F_0)(1 - \cos(\theta))^5 \quad (3.9)$$

3. Materiały i modele oświetlenia

F_0 to współczynnik odbicia światła padającego prostopadle do powierzchni. W silniku przyjmuje się bazową wartość dla dielektryków (niemetale pomijając wyjątki) $f_0 = 0.04$. Jeśli obiekt jest metalem, to wartość F_0 obliczana jest według wzoru:

$$F_{\text{dielectric}} = 0,04 \quad (3.10)$$

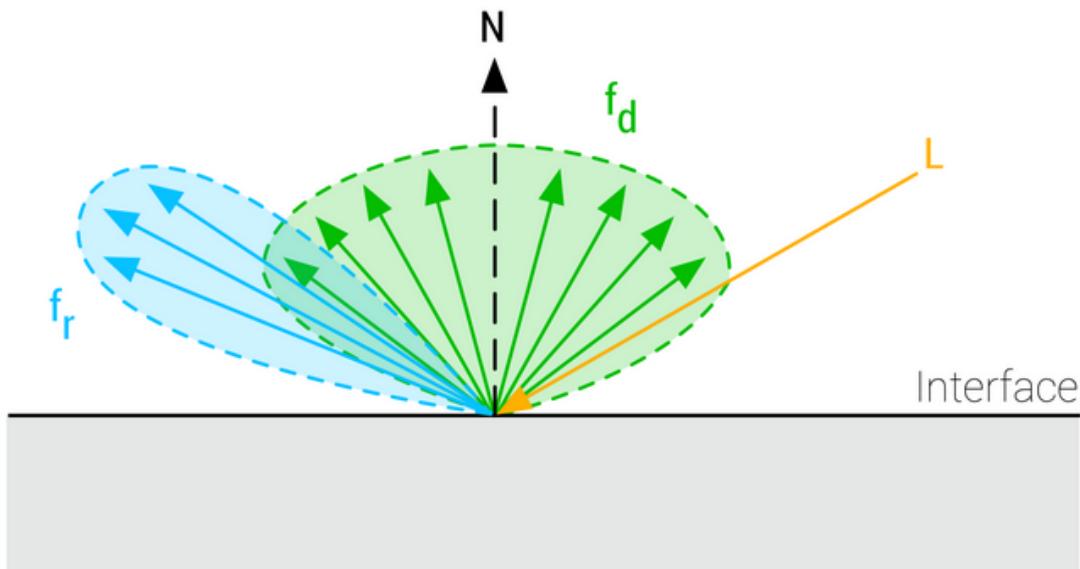
$$F_0 = \text{lerp}(F_{\text{dielectric}}, \text{material.albedo}, \text{material.metalness}) \quad (3.11)$$

Wzór 3.5 służy do opisania odbić lustrzanych (ang. specular reflection). Do pełnego modelu brakuje jeszcze części opisującej światło rozproszone, np. modelu lamberta.

$$f_{\text{diffuse}} = \frac{\sigma}{\pi} \quad (3.12)$$

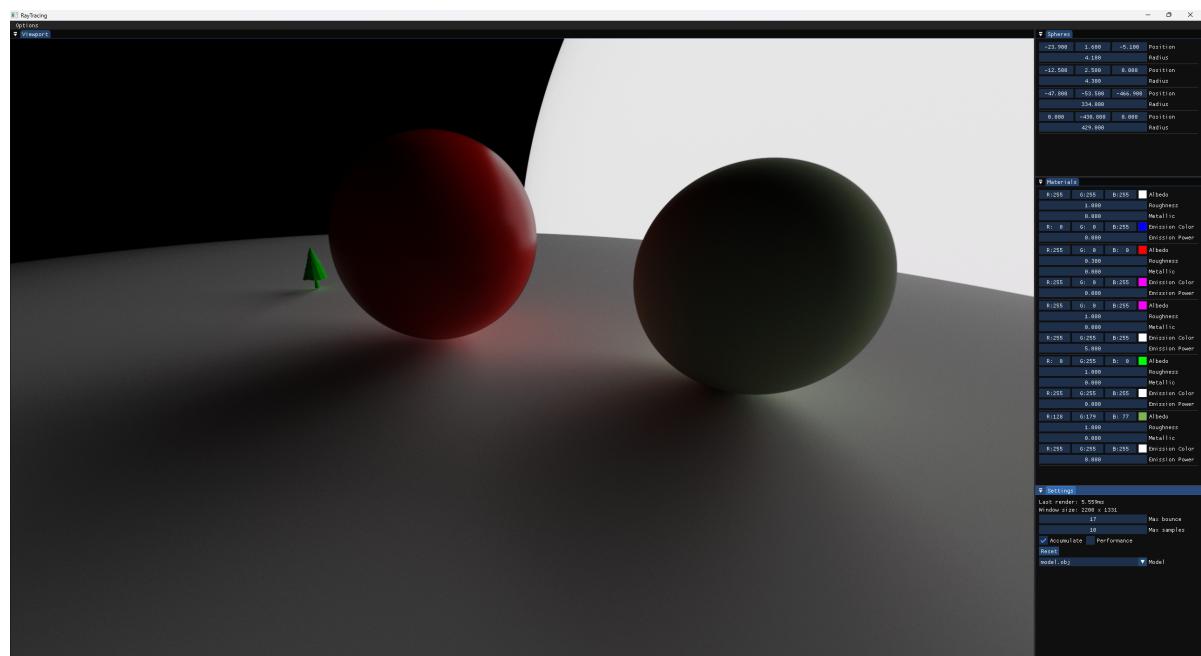
Gdzie: σ - to albedo (kolor) obiektu. Razem:

$$f(v, l) = f_{\text{diffuse}}(v, l) + f_{\text{specular}}(v, l) \quad (3.13)$$



Rysunek 3.6.: Obrazek przedstawia cały model oświetlenia [18]

3. Materiały i modele oświetlenia



Rysunek 3.7.: Przykład globalnego oświetlenia - miękkie cienie. Widać również wpływ sfery na kolor podłoża.

4. Biblioteki i narzędzia

4.1. DirectX 11

DirectX 11 [19] 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 shadery. Shader to program działający na karcie graficznej.

4.2. Win32

Win32 [20], jest to interfejs programistyczny systemu Windows. Zawiera w sobie funkcje umożliwiające działanie programów w systemie. Okno prezentowanego silnika zostało stworzone za pomocą tej biblioteki.

4.3. ImGui

ImGui [21] to biblioteka wykorzystywana do tworzenia interfejsu użytkownika. Napisana jest w myśl paradygmatu "Immediate Mode GUI" (IMGUI), który głównie polega na tym, że interfejs jest cały czas odświeżany, nie przechowuje on stanu między klatkami przez co jest zawsze zsynchronizowany z danymi (inaczej niż w interfejsach tworzonych za pomocą win32, retained mode). Biblioteka ta jest również łatwiejsza w obsłudze niż np. biblioteka Qt.

ImGui wykorzystuje wybrany backend (w tym przypadku backend napisany w DX11) aby renderować interfejs.

4. Biblioteki i narzędzia

4.4. Assimp

Assimp [22] – Open Asset Import Library jest to biblioteka umożliwiająca łatwe ładowanie modeli 3D w różnych formatach. W silniku wykorzystywane są formaty obj, glb. Napisana jest w języku C++. Modele 3D pobrano z internetu, linki dostępne są w pliku ModelCredits.txt, w folderze projektu.

4.5. Stb_Image

Stb_Image [23] jest to biblioteka wykorzystywana do ładowania tekstur w różnych formatach. Silnik wykorzystuje format .hdr używany w teksturach otoczenia. Biblioteka napisana jest w języku C. Tekstury pobrano ze strony „Poly Haven”, jest to darmowa biblioteka tekstur hdr, tekstur pbr (materiały) i modeli 3D.

4.6. Visual Studio

Program napisano wykorzystując IDE (ang. *Integrated development environment*) Visual Studio w wersji Community 2022 [24]. Korzystano z dodatkowych pakietów: „Programowanie aplikacji klasycznych w języku C++” i „Projektowanie gier przy użyciu języka C++”. Z rozszerzeń wykorzystano „HLSL Tools for Visual Studio”.

4.7. Premake

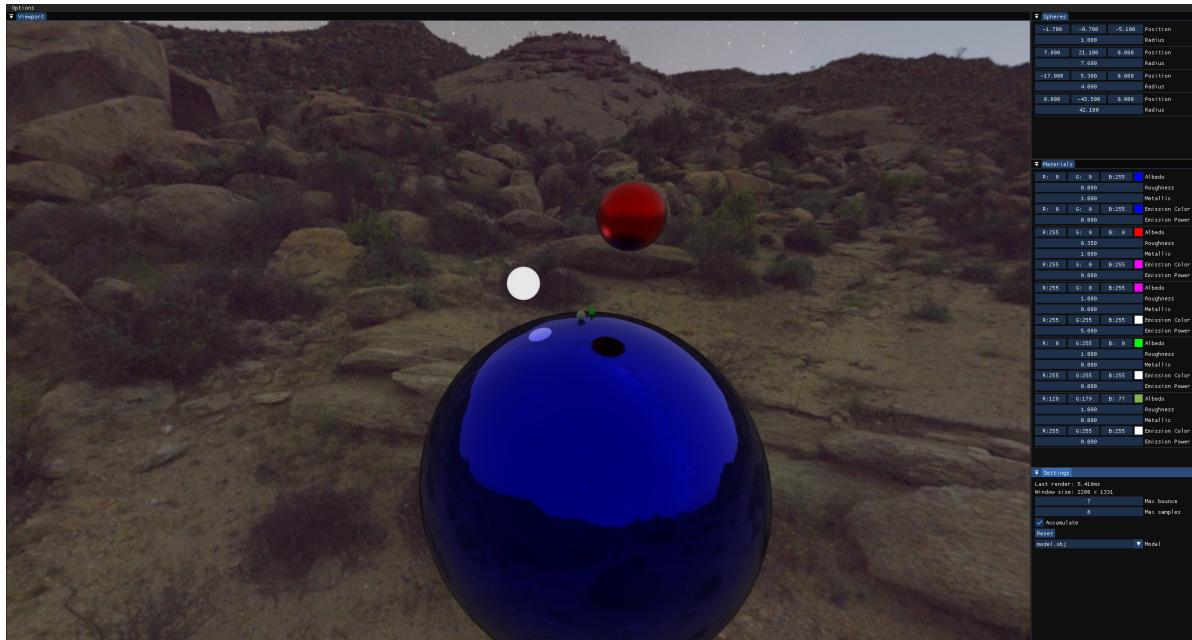
Jako narzędzie do konfiguracji projektu korzystano z Premake [25]. Premake jest to narzędzie które czytając skrypt, napisany w języku Lua, tworzy plik projektu np. solucje Visual Studio. Do pełnej automatyzacji budowania plików projektów i bibliotek dodatkowo użyto skryptów .bat.

5. Prezentacja silnika

5.1. Interfejs

Interfejs podzielony jest na 4 sekcje:

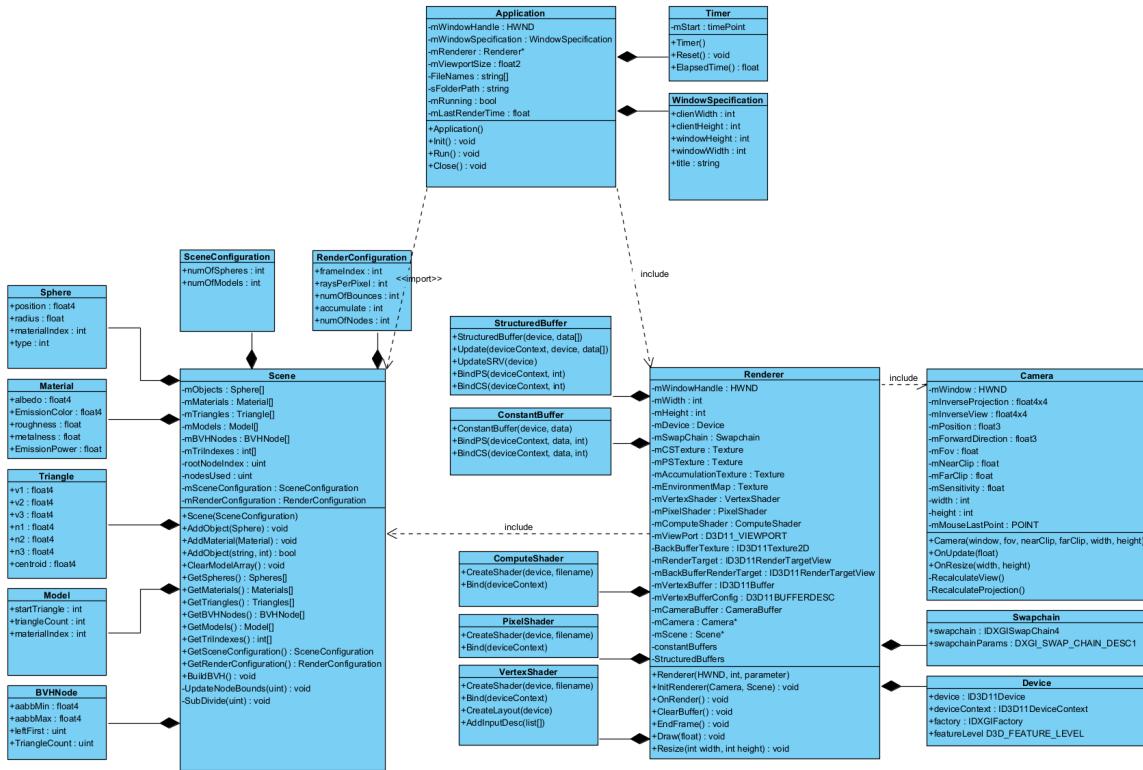
- **Viewport** – tutaj renderowany jest obraz.
- **Spheres** – ustawienia położenia i promienia każdej sfery.
- **Materials** – ustawienia parametrów materiałów.
- **Settings** – ustawienia renderowania



Rysunek 5.1.: Zdjęcie przedstawia interfejs silnika.

5. Prezentacja silnika

5.2. Elementy silnika

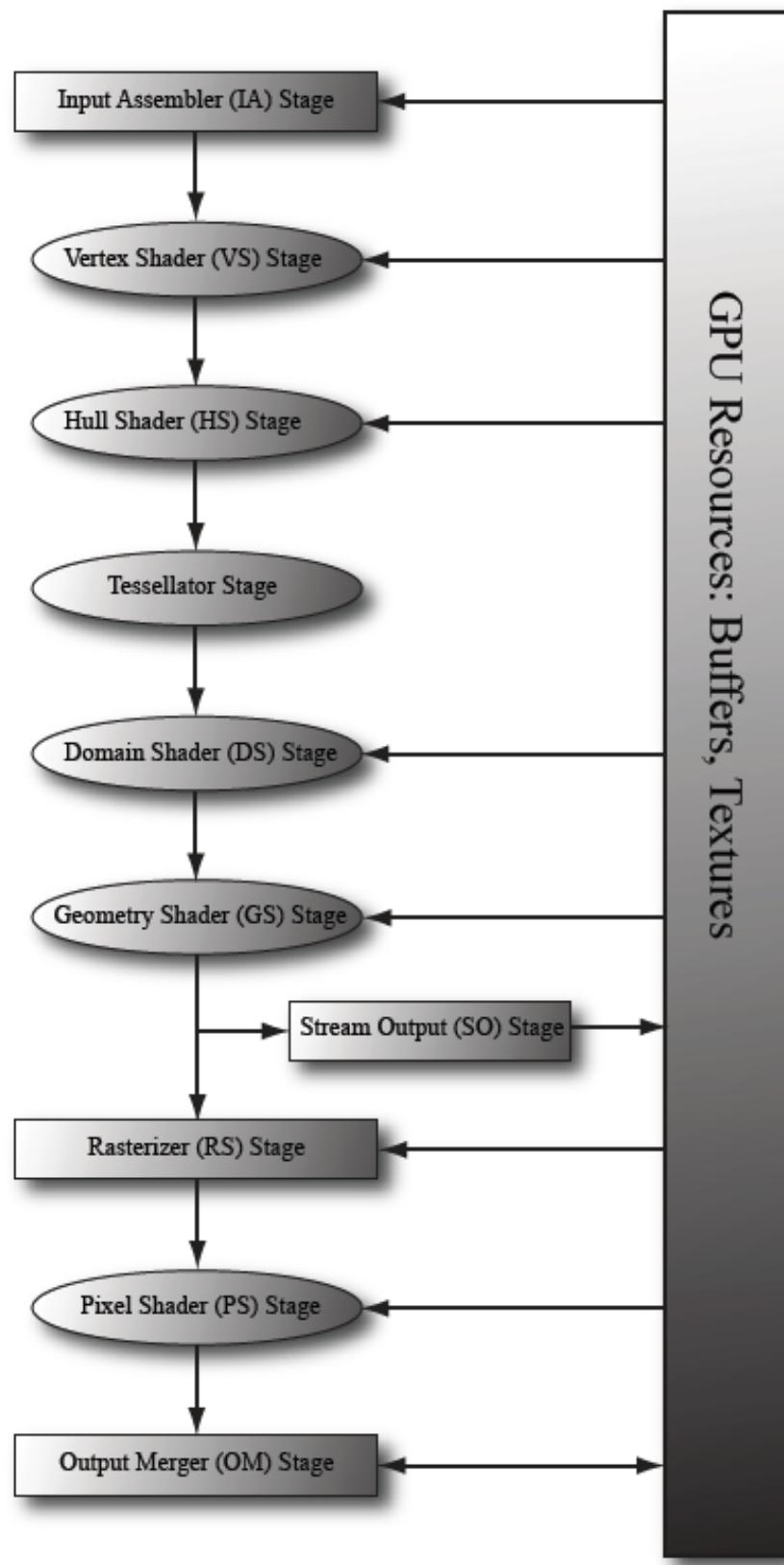


Rysunek 5.2.: Schemat silnika.

5.3. 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 5.3 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.

5. Prezentacja silnika



Rysunek 5.3.: Zdjęcie przedstawia potok graficzny w DirectX 11. [26]

5. Prezentacja silnika

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 [26].
- **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 [27, 28], 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).

5.4. Implementacja algorytmów

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

5.4.1. Implementacja testu przecięcia promień-sfera

```
1      float SphereIntersection(Ray ray, Sphere sphere)
2  {
3      float3 spherePosition = float3(sphere.position.xyz);
4      float3 oc = ray.origin - spherePosition;
5
6      float a = 1; //dot(ray.direction, ray.direction);
7      float b = 2.0 * dot(ray.direction, oc);
8      float c = dot(oc, oc) - sphere.radius * sphere.radius;
9
10     float discriminant = b * b - 4.0f * a * c;
11
12     if (discriminant < 0.0f)
13     {
14         return -1.0f;
15     }
16
17     float t1 = (-b - sqrt(discriminant)) / (2.0f * a);
18     float t2 = (-b + sqrt(discriminant)) / (2.0f * a);
19
20     if (t1 > 0.0f && t2 > 0.0f)
21     {
22         return min(t1, t2);
23     }
24     else if (t1 > 0.0f)
25     {
26         return t1;
27     }
28     else if (t2 > 0.0f)
29     {
30         return t2;
31     }
32     else
33     {
34         return -1.0f;
35     }
36 }
37 }
```

5. Prezentacja silnika

Zmienna a ma wartość 1, ponieważ $ray.direction$ jest wektorem znormalizowanym, iloczyn skalarny dwóch tych samych znormalizowanych wektorów wynosi 1. Wybór mniejszej wartości z t_1 i t_2 jest ważny dla poprawnego renderowania wielu sfer w scenie, bez tego powstawałyby błędy „nachodzenia” się sfer.

5.4.2. 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)
```

5. Prezentacja silnika

```
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
46     return result;
}
```

W kodzie sprawdzane są wyniki pośrednie a, u, v , aby wykryć przypadki szczególne omówione w rozdziale 1.

5.4.3. Implementacja testu przecięcia promienia z AABB

```
1 float IntersectAABB(const Ray ray, const BVHNode node)
2 {
3     float3 invDir = 1 / ray.direction.xyz;
4     float3 t1 = (node.aabbMin.xyz - ray.origin.xyz) * invDir;
5     float3 t2 = (node.aabbMax.xyz - ray.origin.xyz) * invDir;
6
7     float3 tmin = min(t1, t2);
8     float3 tmax = max(t1, t2);
9
10    float tNear = max(max(tmin.x, tmin.y), tmin.z);
11    float tFar = min(min(tmax.x, tmax.y), tmax.z);
12
13    if (tFar <= tNear || tFar < 0.0f)
14    {
15        return -1.0f;
16    }
17
18    return tNear;
```

5. Prezentacja silnika

19 }

Zgodnie z opisanym algorytmem obliczane są poszczególne wartości. Zwarcaną zmienną jest tNear, oznacza ona punkt w którym promień wchodzi do pudełka AABB, tFar oznacza punkt w którym promień wychodzi z pudełka.

5.4.4. Konstrukcja BVH

```
1 struct BVHNode
2 {
3     aabb box;
4     BVHNode* left;
5     BVHNode* right;
6     bool isLeaf;
7     std::vector<Triangle> triangles;
8 };
```

Tak może wyglądać naiwna definicja wierzchołka BVH. Jest ona problematyczna do odtworzenia po stronie shadera, ponieważ w shaderach modelu 5.0 nie można używać wskaźników. Problemem jest również zajmowana pamięć której rozmiaru nie można przewidzieć wykorzystując `std::vector`. [29]

```
1 struct BVHNode
2 {
3     float4 aabbMin, aabbMax;
4     uint32_t leftFirst, triangleCount;
5 };
```

Zoptymalizowana struktura BVHNode. Informację czy dany wierzchołek jest liściem sprawdza się patrząc na wartość zmiennej `triangleCount`. Jeśli wynosi 0, to zmienna jest indeksem lewego potomka poprzedniego wierzchołka, jeśli jest większa od 0, to wierzchołek jest liściem.

Taką definicję wierzchołka BVH można łatwo odwzorować w shaderze.

```
1 void Scene::BuildBVH(int numTriangles)
2 {
3     m_BVHNodes.resize(2 * numTriangles - 1);
4     m_TriIndexes.resize(numTriangles);
5
6     for (int i = 0; i < numTriangles; i++)
```

5. Prezentacja silnika

```
7     {
8         m_TriIndexes[i] = i;
9
10        Triangle& tri = m_Triangles[i];
11
12        XMVECTOR v1 = XMLoadFloat4(&tri.v1);
13        XMVECTOR v2 = XMLoadFloat4(&tri.v2);
14        XMVECTOR v3 = XMLoadFloat4(&tri.v3);
15
16        XMVECTOR sum = XMVectorAdd(XMVectorAdd(v1,
17                                       v2), v3);
18        XMVECTOR centroid = XMVectorScale(sum,
19                                         0.3333f);
20
21        XMStoreFloat4(&tri.centroid, centroid);
22    }
23
24    BVHNode& root = m_BVHNodes[rootNodeIndex];
25    root.leftFirst = 0;
26    root.triangleCount = numOfTriangles;
27
28    UpdateNodeBounds(rootNodeIndex);
29    SubDivide(rootNodeIndex);
30
31    m_RenderConfiguration.numberOfNodes = nodesUsed;
32}
```

Funkcja zaimplementowana jest w klasie Scene, która przechowuje m.in tablicę trójkątów modelu i tablicę wierzchołków drzewa BVH. Wiadomo, że maksymalną liczbą wierzchołków drzewa BVH jest wartość $2\text{numOfTriangles} - 1$ [29]. Dlatego przed rozpoczęciem obliczeń, w ramach małej optymalizacji, rozmiar tablicy wierzchołków (std::vector) jest odpowiednio zmieniany. Tablica Trilndexes zawiera w sobie indeksy trójkątów. Następnie w pętli przygotowywane są centroydy dla każdego trójkąta. Korzeń drzewa inicjalizowany jest w indeksie 0 w tablicy. Kolejnym krokiem jest wywołanie funkcji UpdateNodeBounds i SubDivide.

```
1 void Scene::UpdateNodeBounds(uint32_t nodeIndex)
2 {
3     BVHNode& node = m_BVHNodes[nodeIndex];
```

5. Prezentacja silnika

```
4     XMVECTOR aabbMin = XMVectorSet(1e30f, 1e30f, 1e30f,
5                                     1e30f);
6
7     XMVECTOR aabbMax = XMVectorSet(-1e30f, -1e30f, -1
8                                     e30f, -1e30f);
9
10    for (uint32_t first = node.leftFirst, i = 0; i <
11         node.triangleCount; i++)
12    {
13        int leafTriIdx = m_TriIndexes[first + i];
14        Triangle& leafTri = m_Triangles[leafTriIdx
15                                         ];
16
17        XMVECTOR v1 = XMLoadFloat4(&leafTri.v1);
18        XMVECTOR v2 = XMLoadFloat4(&leafTri.v2);
19        XMVECTOR v3 = XMLoadFloat4(&leafTri.v3);
20
21        aabbMin = XMVectorMin(aabbMin, v1);
22        aabbMin = XMVectorMin(aabbMin, v2);
23        aabbMin = XMVectorMin(aabbMin, v3);
24
25        aabbMax = XMVectorMax(aabbMax, v1);
26        aabbMax = XMVectorMax(aabbMax, v2);
27        aabbMax = XMVectorMax(aabbMax, v3);
28    }
29
30    XMStoreFloat4(&node.aabbMin, aabbMin);
31    XMStoreFloat4(&node.aabbMax, aabbMax);
32}
```

W tej funkcji dobierane są nowe pudełka dla poszczególnych wierzchołków drzewa BVH.

```
1 void Scene::SubDivide(uint32_t nodeIndex)
2 {
3     BVHNode& node = m_BVHNodes[nodeIndex];
4
5     if (node.triangleCount <= 2)
6     {
7         return;
```

5. Prezentacja silnika

```
8          }
9
10         XMVECTOR aabbMin = XMLoadFloat4(&node.aabbMin);
11         XMVECTOR aabbMax = XMLoadFloat4(&node.aabbMax);
12
13         XMVECTOR extent = XMVectorSubtract(aabbMax, aabbMin
14                                         );
15         int axis = 0;
16
17         XMFLOAT4 ex;
18         XMStoreFloat4(&ex, extent);
19
20         if (ex.y > ex.x)
21         {
22             axis = 1;
23         }
24
25         if (ex.z > ((axis == 0) ? ex.x : ex.y))
26         {
27             axis = 2;
28         }
29
30         XMFLOAT4 aabbMinf;
31         XMStoreFloat4(&aabbMinf, aabbMin);
32
33         float splitPos = 0.0f;
34         if (axis == 0)
35         {
36             splitPos = aabbMinf.x + ex.x * 0.5f;
37         }
38         else if (axis == 1)
39         {
40             splitPos = aabbMinf.y + ex.y * 0.5f;
41         }
42         else
43         {
44             splitPos = aabbMinf.z + ex.z * 0.5f;
45         }
```

5. Prezentacja silnika

```
46         int i = node.leftFirst;
47         int j = i + node.triangleCount - 1;
48
49         while (i <= j)
50         {
51             XMVECTOR cen = XMLoadFloat4(&m_Triangles[
52                                         m_TriIndexes[i]].centroid);
53
53             float centroidAxis = 0.0f;
54             if (axis == 0)
55             {
56                 centroidAxis = XMVectorGetX(cen);
57             }
58             else if (axis == 1)
59             {
60                 centroidAxis = XMVectorGetY(cen);
61             }
62             else
63             {
64                 centroidAxis = XMVectorGetZ(cen);
65             }
66
67             if (centroidAxis < splitPos)
68             {
69                 i++;
70             }
71             else
72             {
73                 std::swap(m_TriIndexes[i],
74                           m_TriIndexes[j]);
75                 j--;
76             }
77
78             int leftCount = i - node.leftFirst;
79
80             if (leftCount == 0 || leftCount == node.
81                 triangleCount)
82             {
```

5. Prezentacja silnika

```
82                     return;
83     }
84
85     int leftChildIndex = nodesUsed++;
86     int rightChildIndex = nodesUsed++;
87
88
89     m_BVHNodes[leftChildIndex].leftFirst = node.
90         leftFirst;
91     m_BVHNodes[leftChildIndex].triangleCount =
92         leftCount;
93     m_BVHNodes[rightChildIndex].leftFirst = i;
94     m_BVHNodes[rightChildIndex].triangleCount = node.
95         triangleCount - leftCount;
96     node.leftFirst = leftChildIndex;
97     node.triangleCount = 0;
98
99     UpdateNodeBounds(leftChildIndex);
100    UpdateNodeBounds(rightChildIndex);
101 }
```

Funkcja dzieląca węzły drzewa na mniejsze według najdłuższej osi. Jest to funkcja rekurencyjna (wywoływana na CPU), jej warunkiem stopu jest węzeł zawierający maksymalnie 2 trójkąty.

5.4.5. 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 }
```

5. Prezentacja silnika

```
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             if (hit.t < 0.0f)  
27             {  
28                 continue;  
29             }  
30  
31             if (hit.t < info.hitDistance)  
32             {  
33                 info.hitDistance = hit.t;  
34                 info.t = hit.t;  
35                 info.hitPoint = RayAt(ray, hit.t);  
36  
37                 float w = 1.0f - hit.u - hit.v;  
38  
39                 info.normal = normalize(tri.n1.xyz * w + tri.n2.xyz  
40                     * hit.u + tri.n3.xyz * hit.v);  
41                 info.objectIndex = triIdx;  
42                 info.materialIndex = 3;  
43             }  
44         }  
45     }  
46     else  
47     {
```

5. Prezentacja silnika

```
48     int leftChild = node.leftFirst;
49     int rightChild = node.leftFirst + 1;
50
51     if (rightChild < numNodes)
52     {
53         stack[stackPtr++] = rightChild;
54     }
55     if (leftChild < numNodes)
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 [30]), dlatego funkcja imituje rekurencje używając "stos". Po sprawdzeniu czy wierzchołek jest liściem wykonuje się standardowy kod sprawdzający przecięcie z trójkątem. Jeśli wierzchołek nie jest liściem, to jego potomków dodaje się do stosu.

5.4.6. 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 Fdielectrics = float3(0.04f, 0.04f, 0.04f);
10 float3 F0 = lerp(Fdielectrics, material.albedo.xyz, material.
11 metalness);
12
13 float cosTheta = dot(info.normal, V);
14 float3 F = FresnelSchlick(max(cosTheta, 0.0f), F0);
```

5. Prezentacja silnika

```
14
15     float reflectionChance = max(F.r, max(F.g, F.b));
16     float randomValue = RandomFloat(seed);
17     float2 xi = float2(RandomFloat(seed), RandomFloat(seed));
18
19     if (randomValue < reflectionChance)
20     {
21
22         float3 H = SampleGGX(xi, info.normal, material.roughness);
23         float3 L = reflect(-V, H);
24
25         float NdotV = saturate(dot(info.normal, V));
26         float NdotL = saturate(dot(info.normal, L));
27         float NdotH = saturate(dot(info.normal, H));
28         float VdotH = saturate(dot(V, H));
29
30         if (NdotL > 0.0f)
31         {
32             float G = G_Smith(material.roughness, NdotV, NdotL);
33             float3 weight = (F * G * VdotH) / max(NdotH * NdotV,
34                                         EPSILON);
35
36             contribution *= weight / reflectionChance;
37             ray.direction = L;
38         }
39         else
40         {
41             return float3(0, 0, 0);
42         }
43     }
44     else
45     {
46         if (material.metalness >= 1.0f)
47         {
48             return light;
49         }
50         float3 L = CosineWeightedSample(seed, info.normal);
51
52         float3 kd = (1.0f - F) * (1.0f - material.metalness);
```

5. Prezentacja silnika

```
52     contribution *= (material.albedo.xyz * kd) / (1.0f -
53         reflectionChance);
54     ray.direction = L;
55 }
```

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. Zmienna inicjalizowana jest wartością 1.

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

Podsumowanie

5.1. Testy

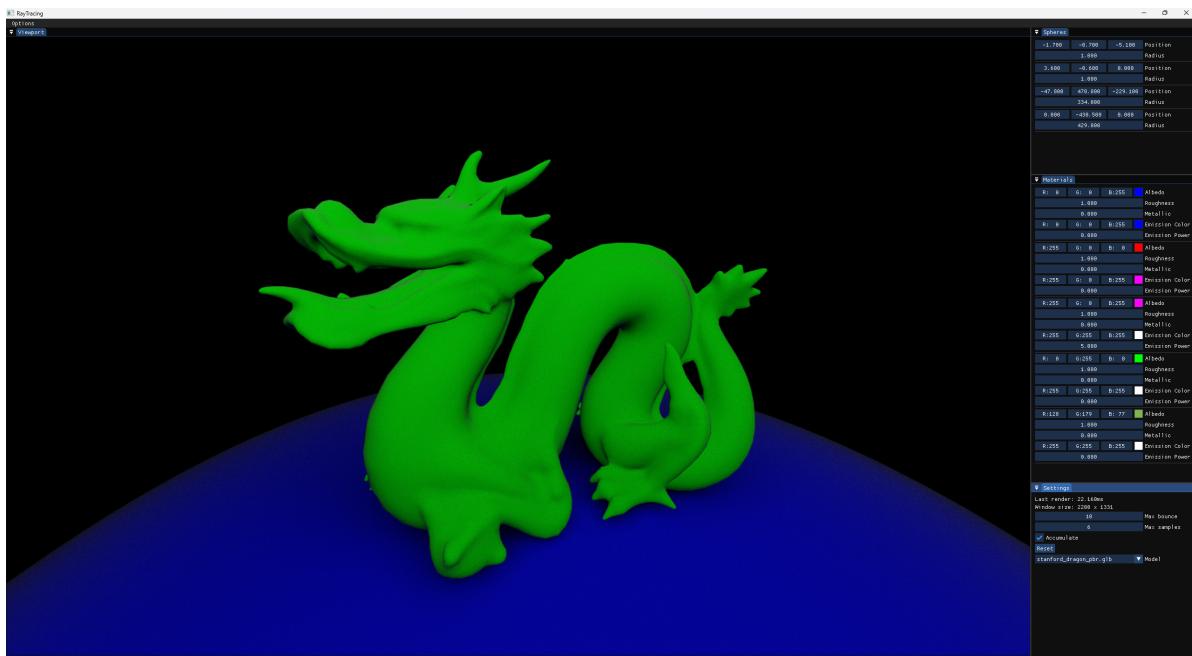
Po zaimplementowaniu drzewa BVH zostały wykonane testy wydajności silnika. Testy opierały się mierzeniu czasu jaki zajęło zrenderowanie jednej klatki w rozdzielczości 2200 x 1331, przy 10 odbiciach i 6 promieniach na piksel. Test zapisywał czasy renderowania sceny dla 3000 klatek, następnie na podstawie zgromadzonych danych wyznaczono średnią arytmetyczną czasu potrzebnego na wygenerowanie obrazu. Renderowano model Stanford dragon zbudowany z 19 332 trójkątów. Testy odbywały się w konfiguracji release złączoną optymalizacją /Ox favor speed w Visual studio 2022 na karcie graficznej AMD Radeon RX 9070 xt.

Tabela 5.1.: 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.

5. Podsumowanie



Rysunek 5.1.: Zdjęcie testowanego modelu

Przeprowadzono również testy dla mniejszego modelu zbudowanego z 264 trójkątów.

Tabela 5.2.: 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, D3D_DRIVER_TYPE_HARDWARE,
8           nullptr,
9           flags,
10          featureLevelArray, 2, D3D11_SDK_VERSION,
```

5. Podsumowanie

```
10     &result.device, &result.featureLevel, &result.  
           deviceContext));
```

Flaga D3D11_CREATE_DEVICE_DEBUG aktywuje warstwę debugowania w DX11. Tryb ten rozszerza kontrolę błędów wynikających z: przesłania 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 [31].

Wykonano również testy jednostkowe dla funkcji implementującej BVH. Interfejs użytkownika przetestowano manualnie.

5.2. 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 Krona. *Fizyka w doświadczeniach*. Dostęp 24.01.2026. URL: https://www.fuw.edu.pl/~kkorona/wwwykl/skrypt_w13.pdf.
- [4] *Anatomia i fizjologia narządu wzroku*. Dostęp 31.01.2026. URL: https://sound.eti.pg.gda.pl/student/pp/oko-budowa_i_wlasnosci.pdf.
- [5] *Wikipedia*. Dostęp 31.01.2026. URL: https://en.wikipedia.org/wiki/Cone_cell.
- [6] Tomas Möller i Ben Trumbore. "Fast Minimum Storage Ray-Triangle Intersection". W: *Journal of Graphics Tools* (1997).
- [7] Doug Baldwin i Michael Weber. "Fast Ray-Triangle Intersections by Coordinate Transformation". W: *Journal of Computer Graphics Techniques* (2016).
- [8] Matt Pharr, Jakob Wenzel i Greg Humphreys. *Physically Based Rendering. From Theory To Implementation*. 2023.
- [9] *Wikipedia*. Dostęp 5.01.2026. URL: <https://en.wikipedia.org/wiki/Centroid>.
- [10] *tavianator*. Dostęp 4.02.2026. URL: https://tavianator.com/2022/ray_box_boundary.html.
- [11] *Rodolphe Vaillant's homepage*. Dostęp 17.01.2026. URL: <https://rodolphe-vaillant.fr/entry/85/phong-illumination-model-cheat-sheet>.
- [12] *Wikipedia*. Dostęp 5.02.2026. URL: https://en.wikipedia.org/wiki/Phong_reflection_model#.
- [13] *Learn OpenGL*. Dostęp 5.02.2026. URL: <https://learnopengl.com/Advanced-Lighting/Advanced-Lighting>.

Bibliografia

- [14] Robert Cook i Kenneth Torrance. "A Reflectance Model for Computer Graphics". W: (1982).
- [15] *Real Shading in Unreal Engine 4*. Dostęp 24.01.2026. URL: <https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf>.
- [16] Eric Heitz. "Sampling the GGX Distribution of Visible Normals". W: *Journal of Computer Graphics Techniques* 7.4 (2018).
- [17] *Wikipedia*. Dostęp 24.01.2026. URL: https://en.wikipedia.org/wiki/Schlick%27s_approximation.
- [18] *Physically Based Rendering in Filament*. Dostęp 5.02.2026. URL: <https://google.github.io/filament/Filament.md.html>.
- [19] *DirectX 11*. Dostęp 5.02.2026. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11>.
- [20] *Win32*. Dostęp 5.02.2026. URL: <https://learn.microsoft.com/en-us/windows/win32/>.
- [21] *ImGui*. Dostęp 5.02.2026. URL: <https://github.com/ocornut/imgui>.
- [22] *Assimp*. Dostęp 5.02.2026. URL: <https://github.com/assimp/assimp>.
- [23] *Stb_IMG*. Dostęp 5.02.2026. URL: <https://github.com/nothings/stb>.
- [24] *Visual Studio*. Dostęp 5.02.2026. URL: <https://visualstudio.microsoft.com/pl/downloads/>.
- [25] *Premake*. Dostęp 5.02.2026. URL: <https://premake.github.io/>.
- [26] Frank Luna. *Introduction to 3D Game Programming with DirectX 11*. 2012.
- [27] 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/>.
- [28] *Optimizing Triangles for a Full-screen Pass*. Dostęp 22.01.2026. URL: <https://wallisc.github.io/rendering/2021/04/18/Fullscreen-Pass.html>.
- [29] Jacco Bikker. *How to build a BVH Part 1: Basics*. Dostęp 24.01.2026. URL: <https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/>.

Bibliografia

- [30] *DirectX Raytracing (DXR) Functional Spec.* Dostęp 23.01.2026. URL: <https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#ray-recursion-limit>.
- [31] *Software Layers.* Dostęp 24.01.2026. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-devices-layers>.