

Generowanie obrazu metodą śledzenia  
promieni w czasie rzeczywistym z  
wykorzystaniem obliczeń równoległych



Politechnika  
Wrocławska

Mateusz Gniewkowski

DD:MM:RR

# Streszczenie

Streszczenie...

# Spis treści

|          |                                                          |           |
|----------|----------------------------------------------------------|-----------|
| <b>1</b> | <b>Wstęp</b>                                             | <b>4</b>  |
| 1.1      | Wykazu celów i zadań pracy . . . . .                     | 4         |
| <b>2</b> | <b>Analiza problemu</b>                                  | <b>5</b>  |
| 2.1      | Śledzenie promieni . . . . .                             | 5         |
| 2.1.1    | Podstawowy algorytm śledzenia promieni . . . . .         | 5         |
| 2.1.2    | Rekursywny algorytm śledzenia promieni . . . . .         | 14        |
| 2.1.3    | Równoległa wersja algorytmu śledzenia promieni . . . . . | 16        |
| 2.2      | Optymalizacja algorytmu śledzenia promieni . . . . .     | 17        |
| 2.2.1    | Drzewa ósemkowe . . . . .                                | 18        |
| 2.2.2    | Drzewa K-d . . . . .                                     | 19        |
| 2.2.3    | Drzewa BSP . . . . .                                     | 19        |
| 2.2.4    | Wybór drzewa do implementacji . . . . .                  | 20        |
| <b>3</b> | <b>Wybór technologii</b>                                 | <b>24</b> |
| 3.1      | C++ . . . . .                                            | 24        |
| 3.2      | QT . . . . .                                             | 24        |
| 3.3      | Standard MPI . . . . .                                   | 25        |
| <b>4</b> | <b>Projekt systemu</b>                                   | <b>27</b> |
| 4.1      | Projekt klastra . . . . .                                | 27        |
| 4.1.1    | Master . . . . .                                         | 28        |
| 4.1.2    | Slave . . . . .                                          | 29        |
| 4.2      | Projekt programu . . . . .                               | 29        |
| 4.2.1    | Diagram klas . . . . .                                   | 30        |
| 4.2.2    | Opis klas . . . . .                                      | 31        |

|          |                                            |           |
|----------|--------------------------------------------|-----------|
| <b>5</b> | <b>Implementacja</b>                       | <b>42</b> |
| 5.1      | Szczegółowy opis klas . . . . .            | 42        |
| 5.2      | Plik wejściowy . . . . .                   | 42        |
| 5.3      | Warstwa prezentacji . . . . .              | 42        |
| 5.4      | Warstwa logiki biznesowej . . . . .        | 42        |
| <b>6</b> | <b>Opis funkcjonalny</b>                   | <b>43</b> |
| <b>7</b> | <b>Rezultaty</b>                           | <b>44</b> |
| 7.1      | Testy wydajnościowe . . . . .              | 44        |
| 7.2      | Omówienie wyników . . . . .                | 44        |
| 7.2.1    | Przyspieszenie obliczeń . . . . .          | 44        |
| 7.2.2    | Obliczenia w czasie rzeczywistym . . . . . | 44        |
| 7.3      | Przykładowe obrazy . . . . .               | 44        |
| <b>8</b> | <b>Podsumowanie</b>                        | <b>45</b> |

# Rozdział 1

## Wstęp

### 1.1 Wykazu celów i zadań pracy

# Rozdział 2

## Analiza problemu

### 2.1 Śledzenie promieni

#### 2.1.1 Podstawowy algorytm śledzenia promieni

Metoda śledzenia promieni pozwala określić widoczność obiektów znajdujących się na scenie (a tym samym na generowanie obrazu) na zasadzie śledzenia umownych promieni świetlnych biegnących od obserwatora w scenę. W perspektywicznym rozumieniu sceny (a takiego dotyczy algorytm zaimplementowany na potrzeby tej pracy), pierwszym krokiem algorytmu jest wybranie środka rzutowania (nazywanego okiem obserwatora) oraz rzutni (powierzchnia na której zostanie odwzorowana trójwymiarowa scena). Rzutnię (a właściwie interesujący nas wycinek rzutni - abstrakcyjne okno obserwatora) można podzielić na regularną siatkę, w której każde pole odpowiada jednemu pikselowi ekranie urządzenia (tzw. układ urządzenia). Kolejnym krokiem algorytmu jest wypuszczenie promienia wychodzącego z oka obserwatora, przechodzącego przez dany piksel ekranu i lecącego dalej - w scenę. Kolor piksela jest ustalany na podstawie barwy i oświetlenia najbliższego obiektu (więcej o metodach oświetlenia można przeczytać w rozdziale !TU WSTAW ROZDZIAŁ!), który został przecięty przez wysłany promień. W przypadku braku kolizji piksel przybiera barwę otoczenia.

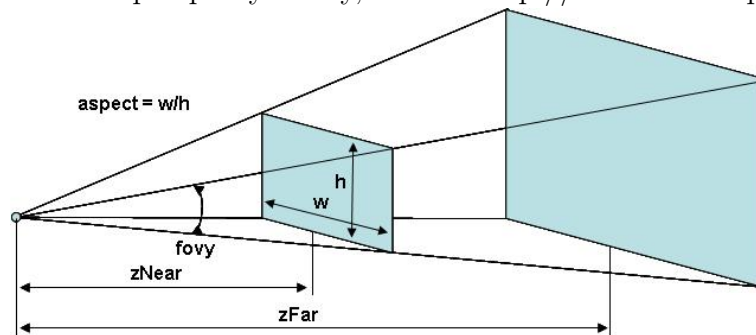
Poniżej przedstawiono pseudokod podstawowego śledzenia promieni

piksele, obiekty

obj = null

dist = max

Rysunek 2.1: Rzut perspektywiczny, źródło: <http://www.zsk.ict.pwr.wroc.pl>



wybór środka rzutowania i rzutni

```

for piksel in piksele do
    wyznacz promień
    for obiekt in obiekty do
        if promień przecina obiekt i dystans < dist then
            obj = obiekt
            dist = dystans
        end if
    end for
end for

```

ustal kolor piksela na podstawie obj

Więcej na temat podstaw śledzenia promieni można przeczytać w [1, 3, 4]

## Obliczenie przecięć

Kluczowym elementem metody śledzenia promieni jest obliczanie przecięć promieni z obiektami sceny - zajmuje one znakomitą większość czasu generowania sceny [3]. W związku z tym, chcąc optymalizować działanie programu największy wysiłek wkłada się w dwa poniższe elementy:

1. Zmniejszenie kosztu wyznaczenia punktu przecięcia promienia z obiektem (stosowanie optymalnych czasowo algorytmów badania przecięcia)

2. Zmniejszenie liczby obiektów, dla których należy zbadać, czy dany promień je przecina (np. poprzez zastosowanie metody brył otaczających, czy wprowadzenie hierarchii sceny)

Wyznaczenie przecięcia promienia z obiektem polega na rozwiązaniu szeregu układu równań zależnych od tego, z jakim obiektem szukamy przecięcia. Najczęściej scena składa się z wielu różnych wielokątów (poligonów), które w połączeniu ze sobą tworzą tzw. siatkę trójwymiarową (ang. mesh) reprezentującą dany obiekt - takie rozwiązanie daje możliwość tworzenia rozmaitych i skomplikowanych modeli 3D (podstawową składową takiego modelu nazywamy prymitywem). Najczęstszym rodzajem wykorzystywanych prymitywów (w grafice 3D) są trójkąty, gdyż da się z nich ułożyć dowolny inny wielokąt. Innym typem obiektów, z jakimi możemy szukać przecięcia, są wszelkiego rodzaju bryły dające zapisać się raczej w postaci prostego równania, niż zbioru punktów. Popularnym, w śledzeniu promieni, przykładem takiej bryły jest kula, lub torus. Poniżej przedstawiono metody badania przecięcia promieni z obiektami, które będą wykorzystywane w programie. Dokładny opis algorytmów oraz ich przykładową implementację można znaleźć w między innymi w [2, 4].

**Przecięcie promienia z kulą** Dane są równania promienia i kuli mające następującą postać:

$$p = p_0 + tv$$

$$(x - x_s)^2 + (y - y_s)^2 + (z - z_s)^2 - r^2 = 0$$

gdzie

$p_0$  - punkt początkowy promienia  $(x_0, y_0, z_0)$

$v$  - wektor kierunkowy promienia o długości 1  $(x_v, y_v, z_v)$

$t$  - parametr określający odległość danego punktu, należącego do promienia, od jego początku tego promienia

$(x_s, y_s, z_s)$  - współrzędne środka kuli

$r$  - promień kuli

Po podstawieniu równania promienia do równania kuli otrzymujemy równanie kwadratowe zależne od współczynnika  $t$ :

$$a = x_v^2 + y_v^2 + z_v^2 = 1$$

$$b = x_v(x_0 - x_s) + y_v(y_0 - y_s) + z_v(z_0 - z_s)$$

$$c = (x_0 - x_s)^2 + (y_0 - y_s)^2 + (z_0 - z_s)^2 - r^2$$



Jeżeli istnieją rozwiązania ( $\Delta \geq 0$ ) to  $t_{1,2} = -b \pm \sqrt{\Delta}$ . Najczęściej interesują nas tylko rozwiązania dodatnie (dla  $t < 0$  przecięcie znajduje się za promieniem). W przypadku dwóch rozwiązań dodatnich wybieramy to mniejsze (bliższy punkt przecięcia). Podstawiając rozwiązanie do równania promienia otrzymamy punkt przecięcia zawierający w powierzchni kuli.

**Przecięcie promienia z płaszczyzną** Płaszczyzna nie jest prymitywem, gdyż z definicji jest ona nieskończona, jednak wyliczenie przecięcia promienia z płaszczyzną jest najczęściej pierwszym krokiem znalezienia przecięcia z dowolnym poligonem (najpierw znajduje się przecięcie z płaszczyzną wyznaczoną przez dany wielokąt, a sprawdza się, czy punkt przecięcia się w nim zawiera). Dodatkowo algorytm przecięcia promienia z płaszczyzną jest wykorzystywany w tworzeniu drzewa BSP (o którym można dowiedzieć się więcej w rozdziale !TU WSTAW ROZDZIAŁ!).

Dane są równanie promienia i równanie płaszczyzny:

$$p = p_0 + tv$$

$$Ax + By + Cz + D = P \bullet N + D = 0$$

gdzie

$p_0$  - punkt początkowy promienia ( $x_0, y_0, z_0$ )

$v$  - wektor kierunkowy promienia o długości 1 ( $x_v, y_v, z_v$ )

$t$  - parametr określający odległość danego punktu, należącego do promienia, od jego początku

$P$  - dowolny punkt płaszczyzny

$N$  - wektor normalny do płaszczyzny

Podstawiając równanie promienia (dowolny punkt promienia) za punkt płaszczyzny otrzymujemy:

$$(p_0 + tv) \bullet N + D = 0$$

$$t = -(p_0 \bullet N + D) / (v \bullet N)$$

Podstawiając  $t$  do równania promienia otrzymujemy punkt przecięcia. Jeżeli  $t < 0$  to płaszczyzna znajduje się za promieniem, w przeciwnym przypadku przed (gdy  $t = 0$  punkt początkowy zawiera się w płaszczyźnie). Należy

zwrócić uwagę, że  $v \bullet N$  nie może być równe zero - jeżeli jest, znaczy to że promień nigdy nie przecina płaszczyzny (jest do niej równoległy).

**Przecięcie promienia z trójkątem** Poniżej przedstawiono dwa sposoby na znalezienie punktu przecięcia promienia z trójkątem (trójkąt jest zdefiniowany poprzez trzy znane punkty - a, b, c).

**Algorytm klasyczny** 1. Wyznaczenie równania płaszczyzny z trójkąta:

a) Obliczenie wektora normalnego do trójkąta:

$$v = (b - a) \times (c - a) \quad n = v/|v|$$

b) Wyznaczenie płaszczyzny poprzez podstawienie do równania ogólnego dowolnego punktu będącego kątem trójkąta i wektora normalnego.

2. Znalezienie punktu przecięcia płaszczyzny z promieniem - punkt ten nazwijmy  $x$ .

3. Sprawdzenie, czy punkt przecięcia z płaszczyzną leży wewnątrz trójkąta:

Punkt leży wewnątrz trójkąta, jeżeli znajduje po tej samej stronie każdej krawędzi, co punkt nie należący do tej krawędzi:

$$(b - a) \times (x - a) \bullet n > 0 \quad (c - b) \times (x - b) \bullet n > 0 \quad (a - c) \times (x - c) \bullet n > 0$$

**Algorytm Möller – Trumbore** Algorytm „Möller – Trumbore” nazywany tak na cześć swoich twórców - Tomasa Möllera and Bena Trumbore’a - jest tzw. szybkim algorytmem badania przecięcia się promienia z trójkątem bez potrzeby wyznaczania płaszczyzny, na której leży trójkąt. Algorytm ten wykorzystuje współrzędne barycentryczne. Najpierw wybieramy dowolny róg trójkąta (jeden z punktów go definiujących) - będzie on naszym początkiem barycentrycznego układu współrzędnych. Powiedzmy, że tym punktem początkowym był punkt  $a$ . Weźmy teraz dwa wektory położone na krawędziach i zaczynające się w tym punkcie -  $(c - a)$  i  $(b - a)$ . W ten sposób, startując z punktu  $a$  i przesuwając się zgodnie z wektorami (zgodnie z parametrami z zakresu od 0 do 1), możemy dostać się do dowolnego punktu należącego do trójkąta. Stąd bierze się równanie:

$$P = a + u * (c - a) + v * (b - a)$$

Należy zwrócić uwagę na dwa fakty. Po pierwsze jeżeli któraś ze zmiennych  $u$  i  $v$  jest mniejsza od zera lub większa od jedynki to jesteśmy poza

trójkątem. Po drugie jeżeli  $u + v > 1$  to przecięliśmy krawędź BC to również wyznaczony punkt znajduje się poza trójkątem.

Na tym etapie, znając punkt przecięcia z płaszczyzną wyznaczoną przez trójkąt, możemy w prosty sposób sprawdzić, czy dany punkt należy do trójkąta, jednak liczenie punktu przecięcia z płaszczyzną nie jest konieczne. Podstawiając za  $P$  równanie promienia otrzymamy:

$$p + td = a + u * (c - a) + v * (b - a) \quad p - a = -td + u * (c - a) + v * (b - a)$$

Wartości parametrów  $t$ ,  $v$  i  $u$  można w prosty sposób wyliczyć stosując iloczyn skalarny i wektorowy:

$$\begin{aligned} pvec &= d \times (c - a) \\ qvec &= (p - a) \times (b - a) \\ invDet &= 1 / ((b - a) \bullet pvec) \\ u &= ((p - a) \bullet pvec) * invDet \\ v &= (d \bullet qvec) * invDet \\ t &= ((c - a) \bullet qvec) * invDet \end{aligned}$$

Poniżej przedstawiono przykładową implementację algorytmu „Möller – Trumbore” zaczerpniętą z [5]:

```
bool RayIntersectsTriangle(Vector3D rayOrigin ,
                           Vector3D rayVector ,
                           Triangle* inTriangle ,
                           Vector3D& outIntersectionPoint)
{
    const float EPSILON = 0.0000001;
    Vector3D vertex0 = inTriangle->vertex0;
    Vector3D vertex1 = inTriangle->vertex1;
    Vector3D vertex2 = inTriangle->vertex2;
    Vector3D edge1, edge2, h, s, q;
    float a, f, u, v;
    edge1 = vertex1 - vertex0;
    edge2 = vertex2 - vertex0;
    h = rayVector.crossProduct(edge2);
    a = edge1.dotProduct(h);
    if (a > -EPSILON && a < EPSILON)
        return false;
    f = 1/a;
```

```

s = rayOrigin - vertex0;
u = f * (s.dotProduct(h));
if (u < 0.0 || u > 1.0)
    return false;
q = s.crossProduct(edge1);
v = f * rayVector.dotProduct(q);
if (v < 0.0 || u + v > 1.0)
    return false;
// At this stage we can compute t to find out where the intersection
float t = f * edge2.dotProduct(q);
if (t > EPSILON) // ray intersection
{
    outIntersectionPoint = rayOrigin + rayVector * t;
    return true;
}
else // This means that there is a line intersection but not a ray
    return false;
}

```

## Model światła

Główny podział modeli światła stanowią modele empiryczne i fizyczne [3, 7]. W niniejszej pracy skupimy się na pierwszej grupie, gdyż jest ona mniej kosztowna obliczeniowo, a dająca zadowalające rezultaty - fizyczne modele światła są raczej wykorzystywane w badaniach niż w standardowych zastosowaniach grafiki komputerowej.

Najprostszym modelem światła jest tzw. oświetlenie bezkierunkowe. Wykorzystuje ono tzw. światło otoczenia (ambient light), które z definicji nie ma określonego źródła (wypełnia całą scenę) i dochodzi do każdego elementu z taką samą intensywnością.

$$I = I_{amb} * k_{amb}$$

W powyższym wzorze  $I_{amb}$  oznacza intensywność światła otoczenia, a  $k_{amb}$  to „albedo” powierzchni przedmiotu (stosunek ilości promienia odbitego do padającego). Wartość natężenia światła liczy się najczęściej dla trzech składowych RGB, zawierających się w przedziale od 0 do 1.

Bardziej zaawansowanym modelem, bo wykorzystującym światło rozproszone (diffuse light) jest tzw. model Lamberta - dodatkowo uwzględnia on

punktowe źródła światła, których promienie padają pod pewnym kątem na daną powierzchnię. Model Lamberta zakłada, że oświetlane powierzchnie są idealnie matowe, zatem światło odbite od nich rozchodzi się tak samo we wszystkich kierunkach (odbicie lambertowskie). W związku z tym, nie ma możliwości otrzymania odblasków widocznych na powierzchniach błyszczących.

$$I = I_{amb} * k_{amb} + I_{dif} * d_{amb} * (N \bullet L)$$

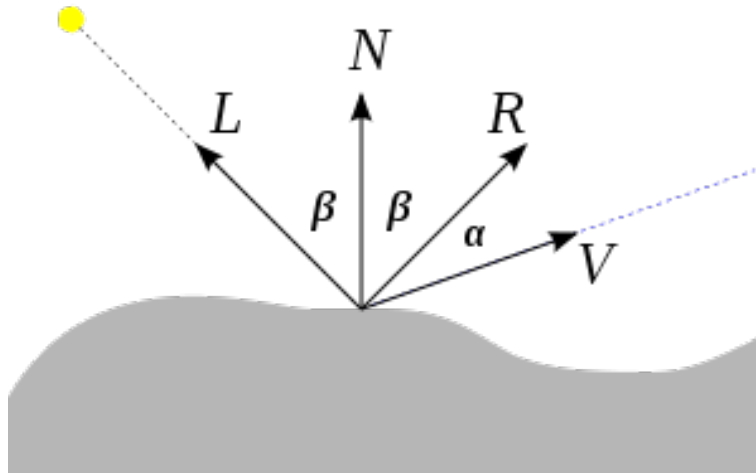
W powyższym wzorze  $N$  i  $L$  są kolejno wektorem normalnym do powierzchni i wektorem wskazującym kierunek padania światła.

Ostatnim, omawianym w tym dokumencie modelem światła, jest model Phong. Wprowadza on tzw. światło kierunkowe (specular light), które uwzględnia odblaski. Model Phong wyraża się wzorem:

$$I = I_{amb} * k_{amb} + 1/(a + bd + c^2d)(I_{dif} * d_{amb} * (N \bullet L) + k_{spec} * I_{spec} * (R \bullet V)^n)$$

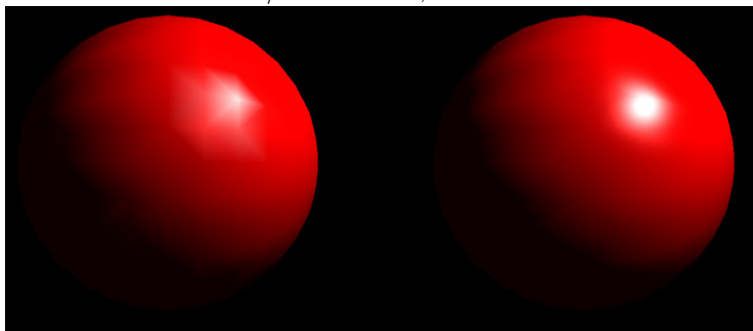
gdzie  $R$  i  $V$  oznaczają kolejno kierunek odbicia promienia i kierunek obserwacji. Ułamek  $1/(a + bd + c^2d)$  określa intensywność padającego światła w zależności od odległości od źródła ( $d$  to odległość od źródła światła, pozostałe parametry są dobierane empirycznie).

Rysunek 2.2: Model Phong, źródło: [https://pl.wikipedia.org/wiki/Cieniowanie\\_Phonga](https://pl.wikipedia.org/wiki/Cieniowanie_Phonga)



**Cieniowanie** Cieniowanie ma na celu stworzenie złudzenia, w którym siatka trójkątów sprawia wrażenie gładkiej powierzchni. Najpopularniejszymi metodami cieniowania są „Cieniowanie Gourauda”, które polega na wyliczeniu kolorów każdego wierzchołka prymitywu, a następnie interpolacji pozostałych. Takie rozwiązanie jest stosunkowo szybkie obliczeniowo, ale nie daje realistycznych rezultatów. Popularną alternatywą jest „Cieniowanie Phonga”. Polega ono na określeniu w każdym wierzchołku poligonu wektorów „normalnych” (niekoniecznie będących normalnymi do powierzchni), a następnie wyliczeniu (poprzez interpolację) takich wektorów dla pozostałych punktów. Takie wektory są później wykorzystywane np. w modelu Phong’a w miejsce prawdziwych wektorów normalnych. Jako, że zarówno model Phong’a jak i cieniowanie Phong’a będą wykorzystywane w programie, którego dotyczy niniejsza praca, poniżej opisano metodę interpolacji wektorów.

Rysunek 2.3: „Cieniowanie Gourauda” i „Cieniowanie Phong’a”, źródło: <http://www.csc.villanova.edu/mdamian>, 02.11.2017

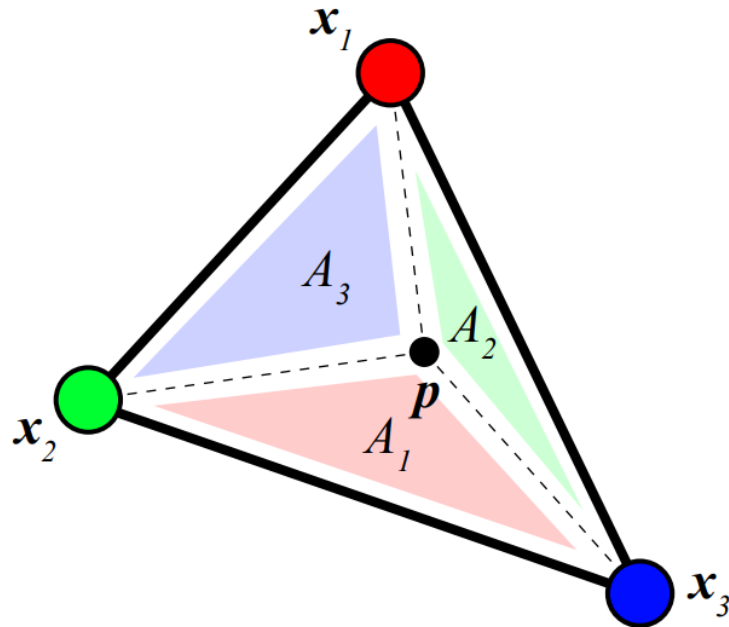


**Interpolacja barycentryczna** Do interpolacji wektorów może posłużyć nam tzw. barycentryczna. W przypadku trójkąta sytuacja prezentuje się następująco:

Na powyższym trójkącie zaznaczono punkt  $p$ , a następnie poprowadzono do niego proste z każdego wierzchołka. W ten sposób prymityw został podzielony na trzy mniejsze trójkąty ( $A_1$ ,  $A_2$ ,  $A_3$ ), których pola zależą od odległości od kolorystycznie odpowiadających punktów. Znając wektory normalne w tych trzech punktach i pola wszystkich trzech fragmentów możemy obliczyć jaki wpływ ma poszczególny wektor na wektor normalny w zaznaczonym punkcie:

$$N_p = N_{x_1} * P_{A_1} / P + N_{x_2} * P_{A_2} / P + N_{x_3} * P_{A_3} / P$$

Rysunek 2.4: Interpolacja barycentryczna, źródło: nieznane



gdzie :  $N_x$  - wektor normalny w odpowiadającym punkcie.  $P$  - pole całkowite  
 $P_A$  - pole fragmentu

Otrzymany w ten sposób wektor należy znormalizować, ponieważ jego długość nie koniecznie jest równa jeden.

### 2.1.2 Rekursywny algorytm śledzenia promieni

Rekursywny algorytm śledzenia promieni [1] jest rozwinięciem algorytmu podstawowego, opisanego w poprzednim rozdziale. Uwzględnia on cienie, odbicia i załamania światła, dzięki śledzeniu dodatkowych promieni wysłanych z punktów przecięcia. W przypadku generowania cieni, z każdego punktu przecięcia wysyła się promień w kierunku źródła światła. Jeżeli promień ten natrafi na przeszkodę, to znaczy, że punkt znajduje się w cieniu, a więc wyliczając barwę piksela (korzystając np. z modelu Phong'a omówionego powyżej), korzystamy tylko ze światła otoczenia. Dla odbić, do wysłania promienia wtórnego korzysta się z zasady, która mówi o tym, że kąt padania równy jest kątowi odbicia. W przypadku załamania światła, określa się współczynniki

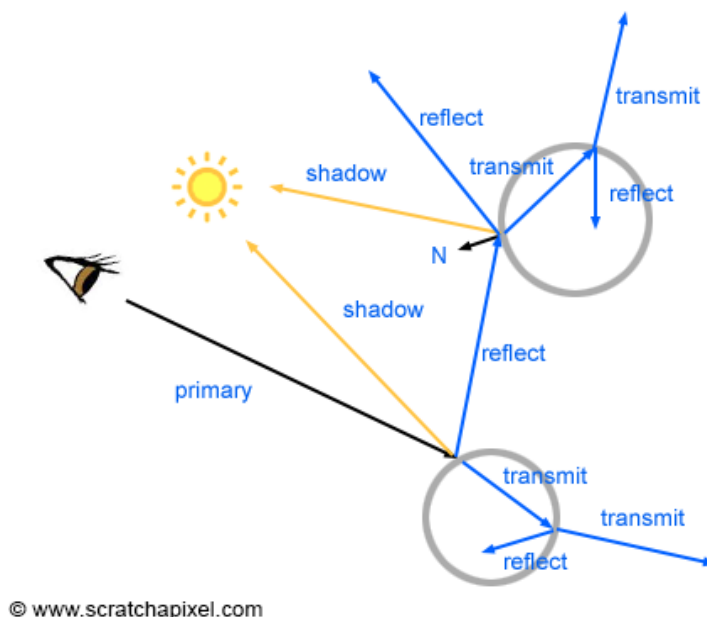
jego załamania (wartości dla różnych materiałów są tablicowane i ogólnie dostępne) i stosuje prawo Snelliusa [6]:

$$\sin(\alpha)/\sin(\beta) = n_2/n_1$$

gdzie  $\alpha$  - kąt padania,  $\beta$  - kąt załamania,  $n_1$  - współczynnik załamania pierwszego materiału,  $n_2$  współczynnik załamania drugiego materiału.

W tym miejscu należy zauważyć, że nie wszystkie materiały są przezroczyste i nie wszystkie materiały odbijają światło w podobny sposób co lustro (można w nich zobaczyć odbicia innych przedmiotów). Implementując algorytm śledzenia promieni należy wprowadzić mechanizm, który pozwala stwierdzić jaki ułamek ostatecznego koloru piksela będzie stanowić wynik śledzenia promieni odbitych/załamanych i czy takie promienie warto wysłać - każdy z nich znacząco wpływa na czas obliczeń, więc ich redukcja, która nie wpływa w zauważalnym stopniu na wygenerowany obraz, jest kluczowym elementem optymalizacji.

Rysunek 2.5: Wizualizacja algorytmu rekursywnego, źródło: [4]

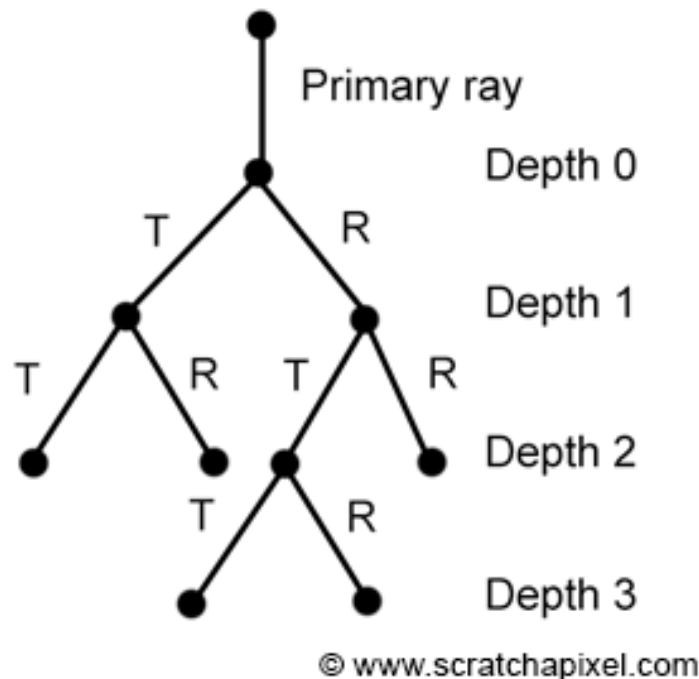


Rekursywny algorytm śledzenia promieni najczęściej implementuje się korzystając z funkcji rekurencyjnej [3], która przyjmuje punkt początkowy



promienia, jego kierunek, oraz aktualną głębokość drzewo rekurencyjne. Ostatni z parametrów często jest wykorzystywany w warunku stopu - po osiągnięciu określonej głębokości funkcja zwraca aktualnie osiągnięty kolor.

Rysunek 2.6: Wizualizacja algorytmu rekursywnego, źródło: [4]



### 2.1.3 Równoległa wersja algorytmu śledzenia promieni

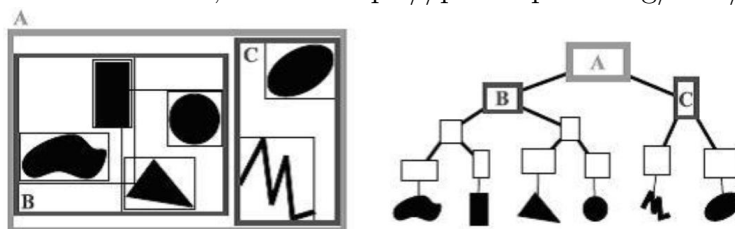
Algorytm śledzenia promieni jest bardzo kosztowny obliczeniowo. Jednym ze skuteczniejszych metod przyspieszenia generowania obrazu jest jego zrównoleglenie, które w przypadku tej metody jest bardzo proste, ponieważ wygenerowanie sceny składa się z wielu obliczeń mogących odbywać się niezależnie od siebie. Najczęściej zrównoleglenie następuje poziomo promieni pierwotnych - zbiór pikseli (pseudokodu znajdujący się w punkcie 2.1.1) dzieli się na podzbiory, mogące być przeanalizowane równolegle. Po obliczeniu kolorów wszystkich pikseli z wycinka, składa się je w jeden spójny obraz. Więcej o algorytmie równoległym można przeczytać w rozdziałach !ROZDZIAŁY!.

## 2.2 Optymalizacja algorytmu śledzenia promieni

Tak jak to było wspomniane wcześniej, metoda śledzenia promieni jest bardzo kosztowna obliczeniowo. W związku z tym powstały metody przyspieszające, które najczęściej opierają się na redukcji testów przecięcia promieni z obiektami. Otaczając pewien model (siatkę prymitywów) bryłą, wiemy że promień który potencjalnie się z nim przecina, musi najpierw przeciąć daną bryłę. W ten sposób zamiast badać setki przecięć z każdym trójkątem modelu z osobna, możemy przeprowadzić tylko jeden test na bryle otaczającej. Ponadto, każdą z takich brył możemy dzielić dalej na kolejne podzbiory prymitywów, a sama bryła może być fragmentem innego podziału. W ten sposób dochodzimy do czegoś co nazywa się hierarchą obiektów.

Hierarchia obiektów ma najczęściej postać drzewa, którego korzeniem najczęściej jest cała scena. Drzewo podziału przestrzeni, które zostało opisane w poprzednim akapicie nazywamy drzewem brył ograniczających (lub otaczających), z angielskiego, bound volume hierarchy - BVH.

Rysunek 2.7: drzewo BVH, źródło: [https://pl.wikipedia.org/wiki/Drzewo\\_BVH](https://pl.wikipedia.org/wiki/Drzewo_BVH)



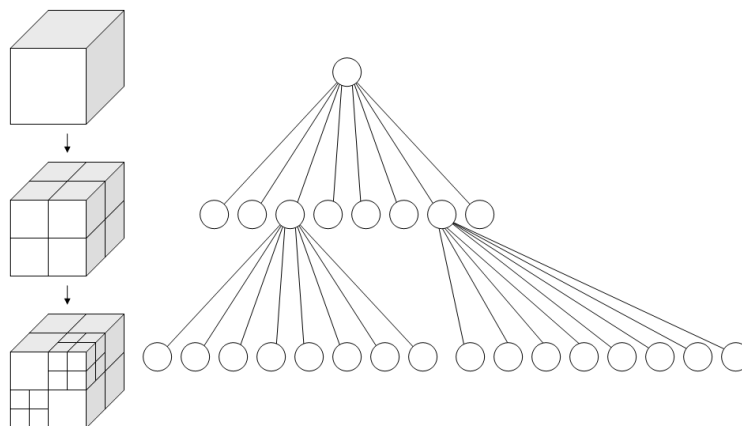
Podstawową zaletą drzewa BVH jest to, że w przypadku scen dynamicznych (takich w których obiekty poruszają się), nie trzeba przebudowywać całego drzewa od początku co klatkę animacji.

Poza drzewami BVH istnieje wiele innych metod podziału podprzestrzeni. Niżej zostaną opisane jeszcze trzy z nich - wiedza na ich temat pozwoli na wybór najlepszego rozwiązania. Więcej na temat każdego z opisanych tutaj drzew można przeczytać w [11, 2].

### 2.2.1 Drzewa ósemkowe

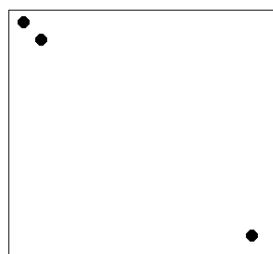
Drzewo ósemkowe (ang. octree) polega na rekurencyjnym podziale przestrzeni na mniejsze, regularne części, najczęściej sześciany.

Rysunek 2.8: octree, źródło: <https://en.wikipedia.org/wiki/Octree>



Takie rozwiązania ma znaczącą wadę w przypadku w której obiekty sceny są daleko od siebie, jednak prostota drzewa ósemkowego i krótki (w porównaniu do alternatyw) czas jego budowy sprawia, że jest ono często wykorzystywane w grafice komputerowej [12, 13].

Rysunek 2.9: Zły przykład dla drzewa ósemkowego

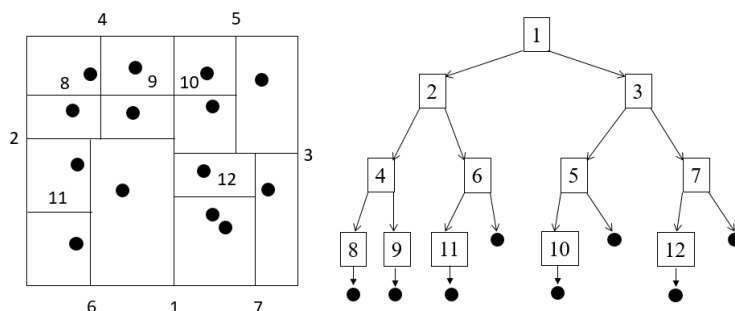


### 2.2.2 Drzewa K-d

Budowa drzewa K-d (skrót od *k-dimensional tree*) polega na podziale przestrzeni płaszczyznami równoległymi do osi układu (w przypadku trzech wymiarów są to osie x, y i z), w taki sposób aby po jednej i drugiej stronie „cięcia” była podobna liczba prymitywów, a sama płaszczyzna przecinała jak najmniej figur !przypis o SAH!. W ten sposób powstaje drzewo binarne.

Wybór płaszczyzny (w którym miejscu powinna ona przebiegać i do której osi powinna być równoległa) jest podstawowym elementem wpływającym na powstanie zrównoważonego drzewa. Najczęściej programiści uzależniają kierunek płaszczyzny od poziomu drzewa - w ten sposób „cięcia” można wyznaczyć dzięki prostym punktom.

Rysunek 2.10: Drzewo K-d dla dwóch wymiarów



Można powiedzieć, że drzewo k-d jest bardziej ogólnym przypadkiem drzewa ósemkowego, w którym przestrzeń nie musi być dzielona na takie same kształty - dzięki temu drzewo k-d jest często szybsze niż drzewo ósemkowe. Z drugiej strony czas budowania i takiego drzewa jest znacznie dłuższy (szukanie optymalnego „przecięcia”) niż w przypadku drzew ósemkowych.

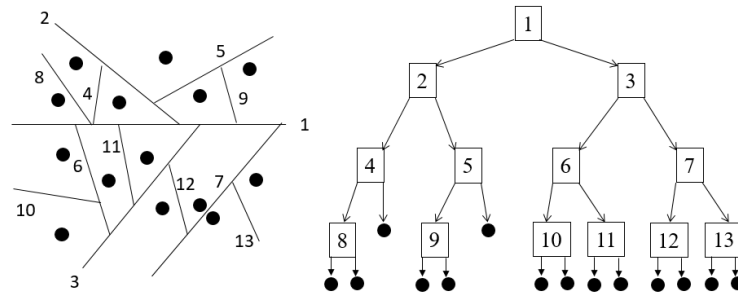
### 2.2.3 Drzewa BSP

Wadą drzew k-d jest możliwość cięć tylko pod trzema kątami (w przestrzeni 3D). W przypadku gdy dwa prymitywy mają wspólną krawędź będącą pod kątem do każdej z osi, nie mogą one zostać rozdzielone. Wadę tę eliminują drzewa BSP - ogólniejsza postać drzew K-d.

W każdym kolejnym kroku wybierana jest dowolna płaszczyzna (zgodnie z pewną strategią np. SAH), która dzieli przestrzeń na dwie inne podprzestrze-

nie. Drzewo BSP buduje się dłużej (głównie ze względu na trudność wyboru płaszczyzny podziału), ale często podział sceny jest jakościowo lepszy.

Rysunek 2.11: Drzewo K-d dla dwóch wymiarów



## 2.2.4 Wybór drzewa do implementacji

Spośród opisanych drzew, drzewa BSP i BVH intuicyjnie wydają się być najodpowiedniejszym wyborem dla metody śledzenia promieni. Na poparcie tej tezy warto zajrzeć do źródeł. Według [16] drzewo BVH jest wydajniejsze od drzewa ósemkowego (biorąc pod uwagę czas generowania sceny, a nie budowy drzewa). Jeżeli porównywać ze sobą drzewa BVH i KD warto zajrzeć do [?, 15]. Autorzy wskazują na przewagę drzewa BVH, jednak uzależniają wyniki od rodzaju promieni (pierwotne, wtóre) i od definicji sceny. Wygląda na to, że jeżeli promienie często nie trafiają w żaden obiekt, to drzewo BVH jest dużo skuteczniejsze (ma to związek ze sposobem przeglądania drzew K-d). Sytuacja nie jest już taka oczywista, przy scenach zamkniętych składających się z wielu trójkątów (dla mniejszych scen BVH jest najczęściej lepsze) - tendencja się odwraca. Zgodnie z intuicją i [14] dobrze zoptymalizowane drzewo BSP jest znacznie wydajniejsze od drzewa K-d, zwłaszcza jeżeli chodzi o czas spędzony na testach przecięć trójkątów z promieniami. W artykule zostały również przedstawione badania BVH - tutaj trudniej jest wykazać jednoznaczną wyższość jednego rozwiązania nad drugim. Podobnie jak było to opisane wyżej, drzewa BVH dużo lepiej działają w zamkniętych scenach (mało promieni, które nie trafiają w żaden obiekt).

Z powyższymi badaniami zdaje się zgadzać współczesna literatura i współczesne metody optymalizacji generowania grafiki. W [11] autor proponuje rozwiązania hybrydowe, w których duże otwarte przestrzenie i poruszające

się obiekty zamyka się w drzewach BVH (drzewa BVH przyspieszają wykrycie kolizji, a poruszające się modele nie wymuszają przebudowy drzewa), z kolei zamknięte, spójne i statyczne elementy hierarchizuje się stosując drzewa BSP.

Na potrzebę tej pracy zostało zaimplementowane drzewo BSP. W związku z tym zostanie one dokładniej omówione niż miało to miejsce wyżej.

**Budowa drzewa BSP** Drzewo BSP zostało w szczególności opisane w [11], jednak warto również polecić adres serwera ftp [17], na którym można znaleźć wiele użytecznych informacji na temat drzewa (między innymi przykładową implementację jego elementów), poniższa treść w dużej mierze bazuje na ostatnio wspomnianej pozycji.

**Budowa drzewa** Podstawową wersję algorytmu budowania drzewa BSP można przedstawić przy użyciu pseudokodu:

```
function BUILD(*node, list<polygon>)

    node.plane = getBestPlane()
    frontlist<polygon>, backlist<polygon>

    for polygon in list<polygons> do
        if polygon.inFrontOf(node.plane) then frontlist.add(polygon)
        else if polygon.inBackOf(node.plane) then backlist.add(polygon)
        else...
        end if
    end for

    Build(node.front, frontlist)
    Build(node.back, backlist)

end function
```

Pierwszym ważnym krokiem, którego powyższy algorytm nie wyjaśnia jest wybór płaszczyzny podziału. Najczęściej kandydatami są płaszczyzny wyznaczone przez prymitywy w wierzchołku (zgodnie z nimi, lub prostopadłe do nich i styczne do krawędzi). W najprostszym modelu wybiera się płaszczyznę, która dzieli zbiór trójkątów na jak najrówniejsze części - w ten sposób po-

wstaje dobrze zbilansowane drzewo binarne. Popularną alternatywą takiego postępowania jest heurystyka SAH [18, 19, 20] (ang. Surface Area Heuristic), która faworyzuje podziały na podprzestrzenie, z których jedna jest duża i zawiera niewielką liczbę prymitywów, a druga jest mała i zawiera ich dużo. Takie podejście jest uzasadnione, biorąc pod uwagę prawdopodobieństwo trafienia promienia w taką przestrzeń - może się okazać, że mniej zbilansowane drzewo będzie przeglądane krócej (mimo tego, że najgorszy przypadek jest jest dużo poważniejszy). Zastosowanie funkcji SAH zostało zaproponowane w [19] i wydaje się najlepszym rozwiązaniem, jednak znacząco wydłuża ono czas budowania drzewa i jest trudniejsze programistycznie ze względu na potrzebę liczenia objętości w drzewie BSP, co nie jest tak proste jak w przypadku drzew K-d i wymaga dodatkowo zamknięcia całej sceny w bryle otaczającej (co pozytywnie wpływa na czas wykonania programu). Najczęściej objętości podprzestrzeni w drzewach BSP są aproksymowane graniastopami.

Kolejnym pytaniem jest co zrobić gdy prymityw leży na płaszczyźnie dzielącej przestrzeń lub jest przez nią przecinany. W przypadku figury leżącej na płaszczyźnie jest ona albo przypisywana do obu podprzestrzeni, albo przechowywana w danym wierzchołku. Jeżeli chodzi o poligony, które zostały przecięte to zazwyczaj dzieli się je na dwie części i przypisuje do odpowiednich potomków wierzchołka (można ich również nie dzielić i przypisać do danego wierzchołka).

Ostatnim elementem do omówienia jest warunek stopu rekurencji. Jeżeli zastosowano funkcję SAH, to jest to moment w którym dalszy podział jest nieopłacalny. W przeciwnym przypadku najczęściej stosuje się technikę, w której wierzchołki są zamieniane w liście, jeżeli zbiór ich prymitywów jest odpowiednio mały. Można również ograniczyć drzewo co do głębokości.

Bardzo łatwo popełnić błąd skutkujący nieskończoną rekurencją. Wybierając płaszczyznę podziału wg. prymitywów może się zdarzyć, że któryś z otrzymanych zbiorów będzie wypukły. W takiej sytuacji podział może nie być możliwy - wszystkie prymitywy mogą znaleźć się albo przed, albo za płaszczyzną dzielącą. Należy wprowadzić mechanizmy zabezpieczające.

**Przeglądanie drzewa** Przeglądanie drzewa polega na rekurencyjnym sprawdzaniu po której stronie płaszczyzny danego wierzchołka znajduje się początek promienia - od tej strony zaczniemy. Jeżeli nie znaleziono przecięcia z żadną figurą po danej stronie, a promień przecina płaszczyznę dzielącą

cą, należy sprawdzić drugą stronę. Najgorszy przypadek to taki w którym nie znaleziono przecięcia - algorytm trawersowania drzewa odwiedzi większość wierzchołków, co może wydłużyć czas działania programu w stopniu większym niż ma to miejsce w przeglądzie zupełnym prymitywów. Więcej informacji o przeglądzie drzewa można znaleźć w [17, 11].



# Rozdział 3

## Wybór technologii

W tym rozdziale przedstawiono technologie (wraz z uzasadnieniem wyboru) jakie zostały użyte do zaimplementowania programu, którego dotyczy praca.

### 3.1 C++

Język C++ jest ustandaryzowanym językiem programowania ogólnego przeznaczenia, który został zaprojektowany przez Bjarne Stroustrupa. Umożliwia on stosowanie kilku paradygmatów programowania, w tym programowania obiektowego, które, w przypadku śledzenia promieni, jest rozwiązaniem wskazanym. Programowanie obiektowe, w którym program definiuje się za pomocą obiektów, pasuje do problematyki problemu (program składać się będzie ze sceny, jej elementów, kamery itd.). Mechanizmy abstrakcji takie jak dziedziczenie, enkapsulacja, czy polimorfizm pozwolą na wygodne zaprogramowanie obsługi różnego typu obiektów sceny.

Dodatkowo język C++ słynie z wydajności i pozwala na bezpośrednie zarządzanie pamięcią - te właściwości pozwalają na napisanie zoptymalizowanego (pod względem czasu wykonania i zajętości pamięci) programu, co jest kluczowym elementem tematu niniejszej pracy.

### 3.2 QT

Qt jest zestawem bibliotek i narzędzi do tworzenia graficznego interfejsu użytkownika w językach takich jak C++, Java, QML, C#, Python i wielu innych.

Qt zapewnia mechanizm sygnałów i slotów, automatyczne rozmieszczanie widgetów i system obsługi zdarzeń. Środowisko jest dostępne między innymi dla systemów Windows, Linux, Solaris, Symbian i Android. Popularność rozwiązania, elastyczność, duża społeczność i wsparcie ze strony producenta [8] sprawiają, że Qt jest dobrym wyborem przy pisaniu aplikacji okienkowych.

### 3.3 Standard MPI

Wybór sposobu zrównoleglenia jest podyktowany nie tylko rodzajem problemu, którego dotyczy praca, ale również dostępnym sprzętem. Najbardziej elastyczną technologią pozwalającą na obliczenia równoległe są klastry - grupa połączonych ze sobą niezależnych komputerów mogących różnić się podzespołami. Minusem takiego rozwiązania jest to, że przeciwieństwie do systemów wieloprocesorowych, procesory nie są podłączone magistralą ze wspólną pamięcią, co z kolei oznacza wolniejszą i trudniejszą programistycznie komunikację między nimi. Wiele problemów mogłoby być rozwiązana efektywniej, w taki, alternatywny sposób. Kolejnym problemem jest trudność rozłożenia obliczeń pomiędzy stacjami wykonawczymi, ponieważ czas obliczeń (i czas przesyłu danych przez sieć) może być znacząco różny dla poszczególnych komputerów. W metodzie śledzenia promieni narzut komunikacyjny jest relatywnie niski, a sugerowany w punkcie 2.1.3 sposób zrównoleglenia obliczeń nie powinien stanowić dużego problemu w ich rozłożeniu, więc klastr obliczeniowy jest dobrym rozwiązaniem, zwłaszcza że jest to rozwiązanie tanie, dostępne i łatwe w rozbudowie. Pomijając dodawanie nowych węzłów, stacje nie muszą ograniczać się do jednego rodzaju podzespołów - wykorzystując koprocesory takie jak „Xeon Phi”, różnego rodzaju karty graficzne, FPGA, czy inne dedykowane układy, można zyskać znaczną moc obliczeniową, ale (tak jak to jest napisane wyżej) nie każdy zrównolegalny problem będzie efektywnie rozwiązywany taką technologią [9].

MPI (Message Passing Interface) jest standardem przesyłania komunikatów pomiędzy procesami znajdującymi się na jednym lub wielu komputerach. Standard ten operuje na na architekturze MIMD (Multiple Instructions Multiple Data) - każdy proces wykonuje się we własnej przestrzeni adresowej, pracuje na różnych danych i może wykonywać różne instrukcje. MPI udostępnia bogaty interfejs pozwalający zarówno na komunikację typu punkt - punkt, jak i komunikację zbiorową. Jedną z implementacji standardu jest MPICH. Na stronie producenta można znaleźć bogatą dokumentację i poradniki dot.

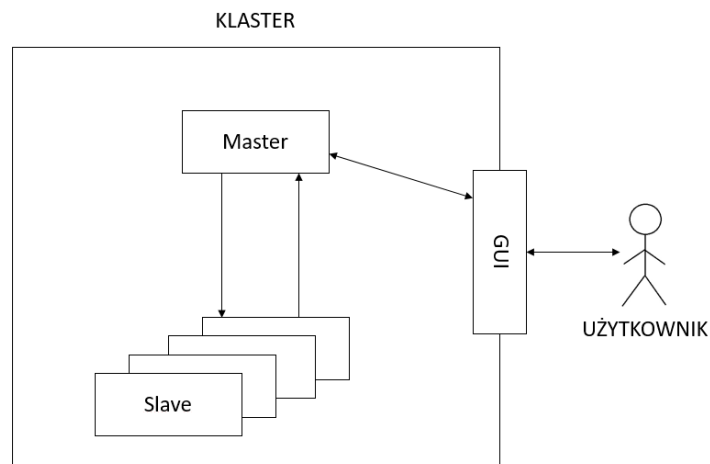
tecnologii [10].

# Rozdział 4

## Projekt systemu

### 4.1 Projekt klastra

Rysunek 4.1: Schemat systemu



Na powyższym schemacie przedstawiono konceptualny schemat działania systemu. W założeniach, użytkownik ma komunikować się z klastrem poprzez graficzny interfejs użytkownika (zbudowany z wykorzystaniem biblioteki Qt), który udostępni mu podgląd dynamicznie budowanej animacji i wszelkich statystyk z nią związanych. Program master'a (a więc głównego węzła klastra) ma wykonywać się na tej samej maszynie, która udostępnia interfejs - główny

proces zostanie podzielony na dwa wątki: jeden zajmujący się obsługą klastra (wątek master'a) i drugi związany z użytkownikiem (wątek GUI). Węzeł zarządzający komunikuje się z każdym z węzłów wykonawczych, zlecając im zadania i zbierając od nich wyniki. W chwili, w której zostanie wygenerowana cała klatka, master informuje wątek GUI, o tym że wygenerowany obraz jest gotowy do wyświetlenia. W poniższych punktach zostanie zaprezentowany proponowany algorytm zachowania węzłów.

#### 4.1.1 Master

Pierwszym i najważniejszym zadaniem węzła zarządzającego jest wczytanie pliku zawierającego definicję generowanego obrazu. Na podstawie pliku wejściowego ma on stworzyć scenę, kamerę, obiekty i światła. Następnie musi on rozesłać informacje na temat obiektów do wszystkich węzłów wykonawczych, tak aby każdy z nich posiadał tę samą definicję obrazu (broadcast). Gdy już każdy z węzłów zasygnalizuje gotowość, węzeł zarządzający rozsyła zadania policzenia danego wycinka obrazu do węzłów wykonawczych. Poniżej przedstawiono przykładowy pseudokod działania master'a. Warto zwrócić uwagę, że zadania umieszczane są na kolejce oczekującej - takie rozwiązanie powinno dawać lepsze rezultaty niż rozesłanie wszystkim węzłom wszystkich zadań od razu, ponieważ różne fragmenty obrazu mogą być generowane z różną prędkością. Mogłoby więc dochodzić do sytuacji, w której część węzłów zrealizowało już swoje zadania (a więc ich moc obliczeniowa nie jest wykorzystywana), a część (która dostała bardziej wymagające obliczenia) ciągle liczy.

```
readFile()  
sendScene()  
sendCamera()
```

```
while true do
```

```
    queue = splitImageToChunks()  
    //pending - liczba zleconych, niewykonanych zadań  
    pending = sendChunkToEveryNode()
```

```
    while pending > 0 do  
        msg = recvMessage()
```

```

    if msg == EXIT then exit()
    else if msg = PIXELS then recvPixels()
        if queue is not empty then sendChunkToSlave()
        else pending-
        end if
    end if
end while
informGUI()
updateCameraPos()
end while

```

#### 4.1.2 Slave

Nawiązując do poprzedniego punktu, pierwszą czynnością, którą powinien wykonać każdy ze slave'ów jest odebranie definicji obiektów wykorzystywanych przy generowaniu sceny. Następnie w pętli, może on czekać na zadanie, realizować je (z wykorzystaniem klasy RayTracer) i odsyłać z powrotem do węzła zarządzającego.

```

recvScene()
recvCamera()

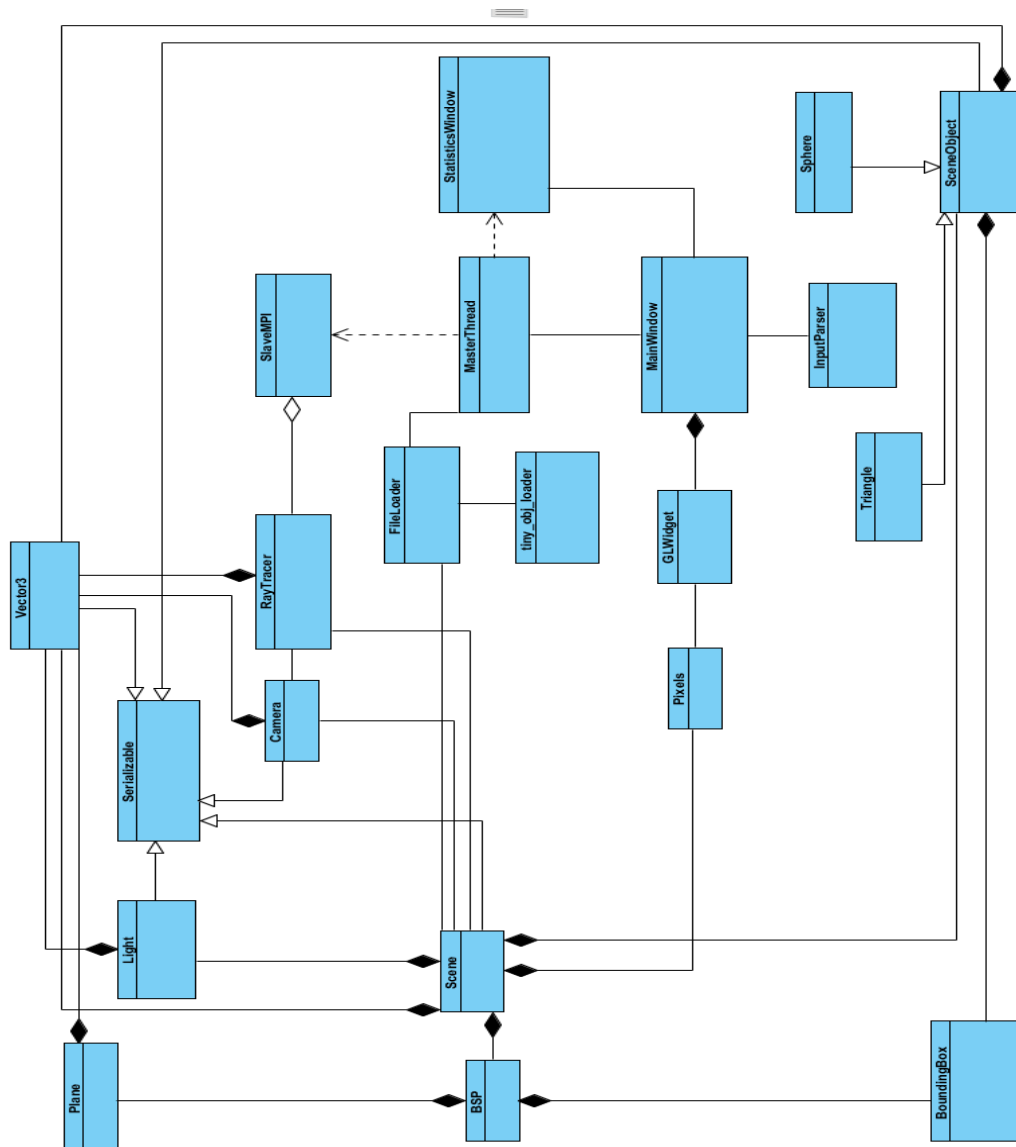
while true do
    msg = recvMessage()
    if msg == EXIT then exit()
    else if msg == CHUNK then recvChunk() pixels = recursiveRayTracer() sendPixels()
    else if msg == CAMERA then recvCamera() //aktualizacja pozycji kamery
    end if
end while

```

## 4.2 Projekt programu

Tu napisz co to ma robić

### 4.2.1 Diagram klas



Rysunek 4.2: Diagram klas

Powyżej przedstawiono uproszczony diagram klas. Opis poszczególnych z nich (wraz ze spisem atrybutów i metod) znajduje się w kolejnym podrozdziale.

## 4.2.2 Opis klas

Tabela 4.1: BoundingBox

| BoundingBox                                                                                        |
|----------------------------------------------------------------------------------------------------|
| +minX : float<br>+maxX : float<br>+minY : float<br>+maxY : float<br>+minZ : float<br>+maxZ : float |
| +intersect(start: Vector3, dir: Vector3)                                                           |

Klasa BoundingBox reprezentuje prostopadłościany, które w założeniu mają otaczać inne obiekty (bryła otaczająca). Ma ona umożliwić przyspieszenie badania przecięcia promieni z elementami sceny (stąd metoda intersect). W programie będzie wykorzystywana przez drzewo BSP (otoczenie całej sceny) i SceneObject.

Tabela 4.2: BSP

| BSP                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| +tree : node*<br>-polygons : SceneObject*<br>-box : BoundingBox                                                                                                                                                                                                                                                                                                                                                                                                          |
| +build(root : node*, polygons : SceneObject*, depth : int)<br>-getBestPlane(polygons : list<SceneObject*>) : Plane<br>+getClosest(cross : Vector3, start : Vector3, dir : Vector3) : SceneObject*<br>+isInShadow(cross : Vector3, dir : Vector3, light : Vector3) : bool<br>-getBoundingBox(polygons : list<SceneObject*>) : BoundingBox<br>-intersect(root : node*, cross : Vector3, start : Vector3, dir : Vector3) : SceneObject*<br>-deleteTree(root : node*) : void |

Tabela 4.3: Node

| Node                                                    |
|---------------------------------------------------------|
| partitionPlane : Plane<br>polygons : list<SceneObject*> |



|                               |
|-------------------------------|
| front : node*<br>back : node* |
|-------------------------------|

Klasa BSP implementuje drzewo opisane w rozdziale (!RODZIAŁ!). Poza metodami związanymi z budową drzewa, zawiera ona metody analogiczne do Klasy Scene (z tą różnicą, że metody Scene wykorzystują przegląd zupełny obiektów) umożliwiające przeglądanie sceny w poszukiwaniu przeciętych obiektów i badania, czy dany punkt znajduje się w cieniu. Drzewo BSP jest składową klasy Scene, która wywołuje jego metody (jeżeli jest ustawiona flaga mówiąca o wykorzystaniu drzewa).

Podstawowym elementem drzewa jest struktura Node, która zawiera pola takie jak płaszczyzna podziału, wskaźniki na kolejne wierzchołki drzewa (reprezentujące przestrzeń przed i za płaszczyzną) i listę obiektów należących do danego wierzchołka. Obiekt może należeć do wierzchołka np. w sytuacji gdy wierzchołek jest liściem lub obiekt leży na płaszczyźnie podziału. Klasa BSP zostanie dokładniej opisana w rozdziale 6.

Tabela 4.4: GLwidget

| GLwidget                      |
|-------------------------------|
| scene : Scene*                |
| initializeGL() : void         |
| resizeGL(int w, int h) : void |
| paintGL() : void              |

Widżety są składowymi elementami graficznego interfejsu użytkownika i nie wszystkie z nich muszą być reprezentowane poprzez obiekty. Klasa GLwidget odpowiada za prezentowanie kolejnych klatek użytkownikowi. W przypadku kiedy główne okno aplikacji (MainWindow, „rodzic” GLwidget) otrzyma informacje o wygenerowaniu kolejnej klatki od MasterThread wywołuje odpowiednią metodę GLwidget - GLwidget odwołuje się do Klasy Scene od której otrzymuje tablicę pikseli (Pixels) do wyświetlenia.

Tabela 4.5: FileLoader

| FileLoader                                     |
|------------------------------------------------|
| -readCameraSettings(line : char const*) : bool |
| -readSceneSettings(line : char const*) : bool  |
| -readSphere(line : char const*) : bool         |

```

-readLight(line : char const*) : bool
-readTriangle(line : char const*) : bool
-readObj(line : char const*) : bool
+ReadFile(fname : char const*) : bool

```

FileLoader jest klasą odpowiedzialną za wczytanie definicji sceny z pliku (stąd powiązania z klasami Scene i Camera). Jest ona wykorzystywana przez klasę MasterThread. Klasa korzysta z biblioteki tinyobjloader dzięki której możliwe jest wczytywanie modeli zapisanych w formacie „obj”. Bibliotekę można znaleźć pod adresem <https://github.com/syoyo/tinyobjloader>. Jest ona udostępniona na licencji MIT.

Tabela 4.6: InputParser

| InputParser                                          |
|------------------------------------------------------|
| tokens : vector<string>                              |
| +getCmdOption(option : string const) : string const) |
| +cmdOptionExists(option : string const) : bool       |

Klasa InputParser jest prostą klasą pozwalającą na obróbkę danych wejściowych (dokładniej opisane w rozdziale !WSTAW RODZIAŁ!). Klasa MainWindow, która w założeniach ma tworzyć wątek MasterThread, wykorzystuje ją do przekazania mu parametrów.

Tabela 4.7: Light

| Light                                           |
|-------------------------------------------------|
| +pos : Vector3*                                 |
| +amb : Vector3*                                 |
| +dif : Vector3*                                 |
| +spec : Vector3*                                |
| +serialize(bytes : vector<char>*) : void        |
| +deserialize(bytes : vector<char> const) : void |
| +getType() : char                               |

Klasa Light określa obiekty światła. Więcej o świetle i jego znaczeniu na scenie można przeczytać w rozdziale !ROZDZIAŁ!.

Tabela 4.8: MainWindow

| MainWindow                                                                                                       |
|------------------------------------------------------------------------------------------------------------------|
| ui : MainWindow*<br>statisticWindow : StatisticsWindow*<br>masterThread : MasterThread*    statusLabel : QLabel* |
| createMaster()<br>ShowStats()<br>setSpeed(double time)<br>on_actionStatistics_triggered()<br>onQuit();           |

Klasa MainWindow reprezentuje główne okno aplikacji. Ze względów projektowych jest ona odpowiedzialna za tworzenie obiektu MasterThread (ma to związek z mechanizmem przypisania sygnałów do slotów - mechanizm komunikacji międzywątkowej w Qt) jak i potrzebie komunikacji między jedną i drugą klasą. Klasa MainWindow (wraz z GLwidget i StatisticWidnow) reprezentuje warstwę prezentacji tworzonej aplikacji.

Tabela 4.9: MasterThread

| MasterThread                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| isAlive : bool<br>camera : Camera*<br>scene : Scene*<br>processSpeed : double**<br>worldSize : int<br>status : MPI_Status<br>names : vector<string><br>pending : int<br>numChunks : int<br>queue : queue<Chunk><br>test : int |
| run()<br>splitToChunks(int num)<br>clearQueue(queue<Chunk> q)<br>sendCameraBcast()<br>sendCameraPointToPoint()<br>sendScene()                                                                                                 |

```

sendDepth(int depth)
sendNextChunk(int dest)
sendExitSignal()
recvPixels(MPI_Status stat) : int
recvMessage() : int
finishPending()
updateProcessSpeed()
waitUntillRdy()
printResult(double spf, double bsp)
getNames()
emitNames()

```

Główna klasa aplikacji, której obiekt tworzony jest w tym samym procesie co MainWindow, jednak działający w osobnym wątku. Takie rozwiązanie pozwala zachować responsywność aplikacji. Wątek okna głównego jest odpowiedzialny za przetwarzanie zdarzeń wygenerowanych przez użytkownika czy program, a wątek MasterThread, niezależnie od niego, zajmuje się w tym czasie generowaniem kolejnej klatki animacji. Rozdzielamy w ten sposób warstwę prezentacji od warstwy biznesowej tworząc tym samym aplikację przyjaźniejszą użytkownikowi.

Głównym zadaniem MasterThread jest zarządzanie węzłami wykonawczymi (sami stanowią serce węzła nadzorującego). Posiada on szereg metod umożliwiających komunikację z innymi procesami tworzącymi aplikację. Ma on dostęp do lokalnych kopii obiektów Scene i Camera, ponieważ musi je rozsyłać po klastrze i mieć możliwość modyfikacji ich parametrów (przesunięcie kamery co klatkę, aktualizacja obiektu Pixels). Więcej o zadaniach i sposobie działania MasterThread można przeczytać (!rozdział z wyżej!, !rozdział z implementacji!).

Tabela 4.10: Chunk

| Chunk                                                      |
|------------------------------------------------------------|
| startx : int<br>stopx : int<br>starty : int<br>stopy : int |

Chunk jest prostą strukturą wykorzystywaną w komunikacji Master/Slave

ve (MasterThread, SlaveMPI). Zawiera on w sobie informacje o tym jaki wycinek obrazu powinien być wyznaczany przez dany węzeł wykonawczy.

Tabela 4.11: Pixels

| Pixels                                                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| +data : unsigned char*<br>+x : int<br>+y : int<br>+startx : int<br>+starty : int                                                                                                                                    |
| +serialize(bytes : vector<char>*) : void<br>+deserialize(bytes : vector<char> const) : void<br>+getType() : char<br>+setStartXY(x : int, y : int) : void<br>+setPixel(posX : int, posY : int, vec : Vector3) : void |

Klasa Pixels przechowuje tablice pikseli w formie ciągu bajtów unsigned char\* (taka reprezentacja jest wykorzystywana przez funkcje rysujące, które zapewniają przyspieszenie sprzętowe). Pozwala ona stworzyć wygodny interfejs czytania i pisania do tablicy.

Tabela 4.12: Plane

| Plane                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| +a : float<br>+b : float<br>+c : float<br>+d : float                                                                                                                                                                                                       |
| Spis metod<br>+classifyObject(obj : SceneObject*) : int<br>+classifyPoint(point : Vector3*) : int<br>+getDistToPoint(point : Vector3*) : float<br>+rayIntersectPlane(start : Vector3, dir : Vector3) : bool<br>+getNormal() : Vector3<br>+isValid() : bool |

Klasa reprezentuje obiekty płaszczyzn, wykorzystywane przy podziale podprzestrzeni przez drzewo BSP. Zawiera metody pozwalające określić po

której stronie płaszczyzny znajduje się dany obiekt.

Tabela 4.13: RayTracer

| RayTracer                                                                 |
|---------------------------------------------------------------------------|
| +camera : Camera*                                                         |
| +scene : Scene*                                                           |
| +buffer : Vector3***                                                      |
| +basicRayTracer() : void                                                  |
| +recursiveRayTracer(depth : int) : void                                   |
| +getColorRecursive(start : Vector3, dir : Vector3, depth : int) : Vector3 |

Klasa RayTracer zawiera w sobie zestaw metod, które pozwalają na realizację algorytmu śledzenia promieni. W tym celu odwołuje się ona do pól klasy Scene i Camera. Każdy obiekt SlaveMPI (zawierający w sobie algorytm działający na węzłach wykonawczych) tworzy własny obiekt RayTracer'a. Klasa ta zostanie dokładniej opisana w rozdziale 6.

Tabela 4.14: Scene

| Scene                                                                        |
|------------------------------------------------------------------------------|
| +numOfLights : int                                                           |
| +numOfObjects : int                                                          |
| +useShadows : bool                                                           |
| +useBSP : bool                                                               |
| +instance : Scene*                                                           |
| +lights : Light**                                                            |
| +sceneObjects : SceneObject**                                                |
| +pixels : Pixels*                                                            |
| +backgroundColor : Vector3*                                                  |
| +globalAmbient : Vector3*                                                    |
| +bsp : BSP*                                                                  |
| +getInstance() : Scene *                                                     |
| +buildBSP(depth : int) : void                                                |
| +addObject(sceneObject : SceneObject*) : void                                |
| +addLight(light : Light*) : void                                             |
| +setUpPixels(x : int, y : int) : void                                        |
| +getClosest(cross : Vector3, start : Vector3, dir : Vector3) : SceneObject * |
| +isInShadow(cross : Vector3, dir : Vector3, lightPos : Vector3) : bool       |

```

+setPixelColor(x : int, y : int, color : Vector3) : void
+serialize(bytes : vector<char>*) : void
+deserialize(bytes : vector<char> const) : void
+getType() : char

```

Obiekt zawierający definicję całej sceny. Jest on napisany według wzorca Singleton (może istnieć tylko jeden obiekt takiej klasy). Wywołując statyczną metodę getInstance() otrzymamy na niego wskaźnik. Obiekt Scene jest jednym z częściej wykorzystywanych obiektów, gdyż jest centralnym elementem aplikacji - zawiera pola istotne dla wielu klas. W związku z tym udostępnia on interfejs pozwalający na pobieranie interesujących daną klasę danych (np. metoda isInShadow, która bada czy dany punkt znajduje się w cieniu. Klasa, zgodnie z flagą useBSP, dokonuje przeglądu zupełnego, lub wykorzystuje drzewo).

Tabela 4.15: SceneObject

| SceneObject                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| #specShin : float<br>#transparency : float<br>#mirror : float<br>#local : float<br>#density : float<br>#amb : Vector3*<br>#dif : Vector3*<br>#spec : Vector3*                                                                                        |
| +getLocalColor(normal : Vector3, cross : Vector3, observation : Vector3) : Vector3<br>+trace(cross : Vector3, start : Vector3, dir : Vector3, dist : float) : bool<br>+getNormalVector(cross : Vector3) : Vector3<br>+getBoundingBox() : BoundingBox |

SceneObject jest wirtualną klasą po której powinny dziedziczyć wszystkie obiekty sceny. Pozwala on na wykorzystywanie wielopostaciowości (widoczne np. w klasie Scene), wymuszając na klasach potomnych implementacje metod pozwalających określić przecięcie z promieniem (trace()), obliczenie wektora normalnego w punkcie (getNormalVector()), czy pobranie koloru w punkcie (getLocalColor())

Tabela 4.16: Serializable

| Serializable                                    |
|-------------------------------------------------|
| +serializedSize : int                           |
| +serialize(bytes : vector<char>*) : void        |
| +deserialize(bytes : vector<char> const) : void |
| +getType() : char                               |

Klasa Serializable jest właściwie Interfejsem - dziedziczą po niej wszystkie klasy, które muszą mieć możliwość serializacji (zamiana obiektu na ciąg bajtów) w związku z potrzebą wysłania ich do innych węzłów klastra. Bardziej szczegółowe informacje o serialializacji można znaleźć w rozdziale 6.

Tabela 4.17: SlaveMPI

| SlaveMPI                         |
|----------------------------------|
| +x : int                         |
| +y : int                         |
| +depth : int                     |
| +status : MPI_Status             |
| +pixels : Vector3***             |
| +camera : Camera*                |
| +scene : Scene*                  |
| +exec() : int                    |
| +recvCameraBcast() : void        |
| +recvCameraPointToPoint() : void |
| +recvScene() : void              |
| +recvDepth() : void              |
| +recvChunk() : void              |
| +recvMessage() : int             |
| +sendPixels() : void             |
| +sendName() : void               |
| +sendRdy() : void                |

SlaveMPI jest klasą analogiczną do klasy MasterThraed, jednak jej obiekt jest tworzony na każdym węźle wykonawczym. Zawiera ona w sobie metody implementujące mechanizmy komunikacji z resztą węzłów i takie pozwalające na wykonywanie zadań zleconych przez mastera Metoda exec() implementuje algorytm opisany w rozdziale (!wstaw rozdział!) - główną pętlę programu



węzłów wykonawczych. Więcej o implementacji można znaleźć w rozdziale 6.

Tabela 4.18: StatisticsWindow

| StatisticsWindow                                                                                                                                                                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ui::StatisticsWindow *ui;<br>int worldSize;<br>resizeEvent(QResizeEvent *event) : void<br>setTime(double time) : void<br>setChunks(int i) : void<br>setXY(int x, int y) : void<br>setObj(int i) : void<br>setLights(int i) : void<br>setProccessName(int num, QString str) : void<br>setProccessSpeed(double **speed) : void<br>setUpList() : void |

StatisticWindow jest oknem, które (jak sama nazwa wskazuje) ma prezentować statystyki dot. programu - czas generowania jednej klatki, średni czas pracy danego węzła, liczbę obiektów na scenie itd.

Tabela 4.19: Sphere

| Sphere                             |
|------------------------------------|
| +radius : float<br>+pos : Vector3* |

Klasa Sphere jest klasą reprezentującą sferę. Dziedziczy ona po SceneObject, a więc implementuje metody specyficzne dla tego typu obiektu (np. badanie przecięcia z obiektem).

Tabela 4.20: Triangle

| Triangle                                                                                                     |
|--------------------------------------------------------------------------------------------------------------|
| +pointA : Vector3*<br>+pointB : Vector3*<br>+pointC : Vector3*<br>+normalA : Vector3*<br>+normalB : Vector3* |

|                                                                               |
|-------------------------------------------------------------------------------|
| +normalC : Vector3*                                                           |
| +split(plane : Plane, front : list<Triangle*>, back : list<Triangle*>) : void |
| +getPointbyNum(a : int) : Vector3 *                                           |
| +getPlanes() : list<Plane>                                                    |
| +getPerpendicularPlane(i : int) : Plane                                       |
| +getPlane() : Plane                                                           |
| +Area(a : Vector3, b : Vector3) : float                                       |
| +getBoundingBox() : BoundingBox                                               |

Triangle jest klasą reprezentującą obiekty trójkąta. Triangle podobnie jak Sphere dziedziczy po SceneObject i implementuje metody specyficzne dla tego typu obiektu.

Tabela 4.21: Vector3

| Vector3                                                          |
|------------------------------------------------------------------|
| +x : type                                                        |
| +y : type                                                        |
| +z : type                                                        |
| +normalize() : Vector3                                           |
| +scalarProduct(v : Vector3) : float                              |
| +vectorProduct(v : Vector3) : Vector3                            |
| +rotateX(alpha : float) : void                                   |
| +rotateY(alpha : float) : void                                   |
| +rotateZ(alpha : float) : void                                   |
| +distanceFrom(v : Vector3) : float                               |
| +powDistanceFrom(v : Vector3) : float                            |
| +reflect(n : Vector3) : Vector3                                  |
| +refract(normalVector : Vector3, a : float, b : float) : Vector3 |
| +isZeroVector() : bool                                           |
| +length() : float                                                |

Klasa Vector3 jest klasą pozwalającą definiować obiekty takie jak punkt, czy wektor w przestrzeni 3D. Zapewnia ona szereg metod implementujących różne operacje matematyczne na tego typu obiektach. Poza wszystkimi metodami wymienionymi wyżej zostanie również przeciążonych wiele operatorów arytmetycznych.

# Rozdział 5

## Implementacja

### 5.1 Szczegółowy opis klas

#### 5.1.1 RayTracer

#### 5.1.2 BSP

#### 5.1.3 MasterThread

#### 5.1.4 SlaveMPI

### 5.2 Serializacja

## Rozdział 6

### Opis funkcjonalny

# Rozdział 7

## Rezultaty

### 7.1 Testy wydajnościowe

### 7.2 Omówienie wyników

#### 7.2.1 Przyspieszenie obliczeń

#### 7.2.2 Obliczenia w czasie rzeczywistym

### 7.3 Przykładowe obrazy

## Rozdział 8

## Podsumowanie

# Bibliografia

- [1] J. D. Foley i in. *Wprowadzenie do Grafiki Komputerowej*. tłum. J. Zabrodzki, WNT, Warszawa 1995.
- [2] F. Dunn, I. Parberry, *3D Math Primer for Graphics and Game Development*. Wordware Publishing, Inc. 2002.
- [3] K. Suffern *Ray Tracing from the Ground Up*. A K Peters, Ltd. Wellesley, Massachusetts, 2007.
- [4] StrachPixel 2.0: <https://www.scratchapixel.com>, 27.09.2017
- [5] Wikipedia, Wolna Encyklopedia:  
[https://en.wikipedia.org/wiki/Moller-Trumbore\\_intersection\\_algorithm](https://en.wikipedia.org/wiki/Moller-Trumbore_intersection_algorithm), 15.09.2017
- [6] Wikipedia, Wolna Encyklopedia:  
[https://pl.wikipedia.org/wiki/Prawo\\_Snelliusa](https://pl.wikipedia.org/wiki/Prawo_Snelliusa), 02.11.2017
- [7] M. Falski *Przegląd modeli oświetlenia w grafice komputerowej* Praca magisterska pod kierunkiem dr A. Łukaszewskiego, Uniwersytet Wrocławski, 2004
- [8] Qt: <https://www.qt.io/>
- [9] Wikipedia, Wolna Encyklopedia:  
[https://en.wikipedia.org/wiki/Parallel\\_computing](https://en.wikipedia.org/wiki/Parallel_computing), 15.08.2017
- [10] MPICH — High-Performance Portable MPI:  
<https://www.mpich.org/>, 03.07.2017
- [11] Christer Ericson, *Real-Time Collision Detection*, CRC Press, 2005

- [12] A. S. Glassner, *Space Subdivision for Fast Ray Tracing*, University of North Carolina at Chapel Hill, 1984
- [13] H. Samet, *Implementing Ray Tracing with Octrees and Neighbor Finding*, University of Maryland, Collage Park, 1989
- [14] T. Vinkler, V. Havran, J. Bittner, *Bounding Volume Hierarchies versus Kd-trees on Contemporary Many-Core Architectures*, Marsyk University, Czech Technical University in Prague
- [15] M. Zlatuska, V. Havran *Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms*, Czech Technical University in Prague
- [16] T. Dievald, <http://thomasdiewald.com/blog/?p=1488>, 11.08.2017
- [17] BSP FAQ: <ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html>, 03.11.2017
- [18] R. Żukowski, *Automatyczna dekompozycja sceny 3D na portale i sektory* Praca magisterska pod kierunkiem dr A. Łukaszewskiego, Uniwersytet Wrocławski, 2008 ‘
- [19] R. P. Kammaje, B. Mora, *A Study of Restricted BSP Trees for Ray Tracing* University of Wales Swansea
- [20] I. Wald, *Realtime Ray Tracing and Interactive Global Illumination*, Computer Graphics Group, Saarland University Saarbrücken, Germany, 2004



# Spis rysunków

|      |                                                                                                                                                          |    |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1  | Rzut perspektywiczny, źródło: <a href="http://www.zsk.ict.pwr.wroc.pl">http://www.zsk.ict.pwr.wroc.pl</a>                                                | 6  |
| 2.2  | Model Phong'a, źródło: <a href="https://pl.wikipedia.org/wiki/Cieniowanie_Phonga">https://pl.wikipedia.org/wiki/Cieniowanie_Phonga</a>                   | 12 |
| 2.3  | „Cieniowanie Gourauda” i „Cieniowanie Phong'a”, źródło: <a href="http://www.csc.villanova.edu/~mian">http://www.csc.villanova.edu/~mian</a> , 02.11.2017 | 13 |
| 2.4  | Interpolacja barycentryczna, źródło: nieznane                                                                                                            | 14 |
| 2.5  | Wizualizacja algorytmu rekursywnego, źródło: [4]                                                                                                         | 15 |
| 2.6  | Wizualizacja algorytmu rekursywnego, źródło: [4]                                                                                                         | 16 |
| 2.7  | drzewo BVH, źródło: <a href="https://pl.wikipedia.org/wiki/Drzewo_BVH">https://pl.wikipedia.org/wiki/Drzewo_BVH</a>                                      | 17 |
| 2.8  | octree, źródło: <a href="https://en.wikipedia.org/wiki/Octree">https://en.wikipedia.org/wiki/Octree</a>                                                  | 18 |
| 2.9  | Zły przypadek dla drzewa ósemkowego                                                                                                                      | 18 |
| 2.10 | Drzewo K-d dla dwóch wymiarów                                                                                                                            | 19 |
| 2.11 | Drzewo K-d dla dwóch wymiarów                                                                                                                            | 20 |
| 4.1  | Schemat systemu                                                                                                                                          | 27 |
| 4.2  | Diagram klas                                                                                                                                             | 30 |