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

Politechnika Wrocławska

Mateusz Gniewkowski

DD:MM:RR

# Streszczenie

Streszczenie...

# Spis treści

# Rozdział 1

# Wstęp

## 1.1 Wykazu celów i zadań pracy

# Rozdział 2

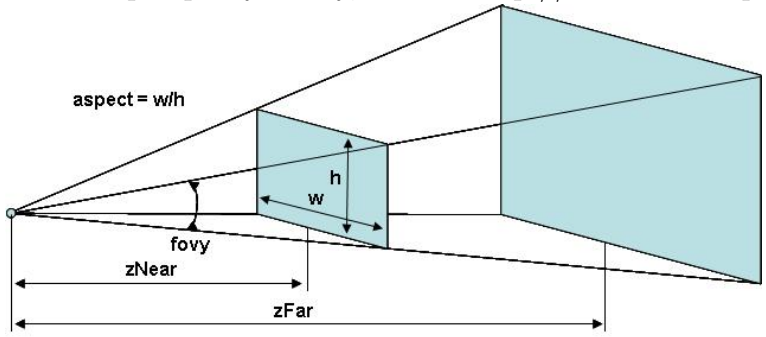
# Analiza problemu

## 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 ekranu 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.

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




Poniżej przedstawiono pseudokod podstawowego śledzenia promieni

```
piksele, obiekty
obj = null
dist = max

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]

### 2.1.2 Obliczanie przecięć

Kluczowym elementem metody śledzenia promieni jest obliczanie przecięć promieni z obiektami sceny - zajmuje on znakomitą większość czasu potrzebnego na wygenerowanie 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 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 równianie 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 ($\Delta \geqslant 0$) 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 mniejsze (bliższy punkt przecięcia). Podstawiając rozwiązanie do równania promienia otrzymamy punkt przecięcia zawierający się 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 następnie sprawdza się, czy zawiera się w nim wyliczony punkt przecięcia). Dodatkowo algorytm przecięcia promienia z płaszczyzną jest wykorzystywany w tworzeniu drzewa BSP (o którym więcej w rozdziale !TU WSTAW ROZDZIAŁ!).
Dane są równania 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 tego promienia
$P$ - 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).

### 1. Algorytm klasyczny

1. Wyznaczenie równania płaszczyzny z trójkąta:
   (a) Obliczenie wektora normalnego do trójkąta:

$$v = (b - a) \times (c - a)$$

$$n = v/|v|$$

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

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

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

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

### 2. Algorytm Möller – Trumbore

Algorytm „Möller – Trumbore” nazwany 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 $a$. Tworzymy 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 tu konieczne. Podstawiając za $P$ równianie promienia otrzymamy:

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

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

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

$$pvec = d \times (c-a)$$

$$qvec = (p-a) \times (b-a)$$

$$invDet = 1/((b-a) \bullet pvec)$$

$$u = ((p-a) \bullet pvec)) * invDet$$

$$v = (d \bullet qvec) * invDet$$

$$t = ((c-a) \bullet qvec) * invDet$$

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 point is on the line.
    float t = f * edge2.dotProduct(q);
    if (t > EPSILON) // ray intersection
    {
        outIntersectionPoint = rayOrigin + rayVector * t;
        return true;
    }
    // This means that there is a line
    // intersection but not a ray intersection.
    else
        return false;
}
```

### 2.1.3 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 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.

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

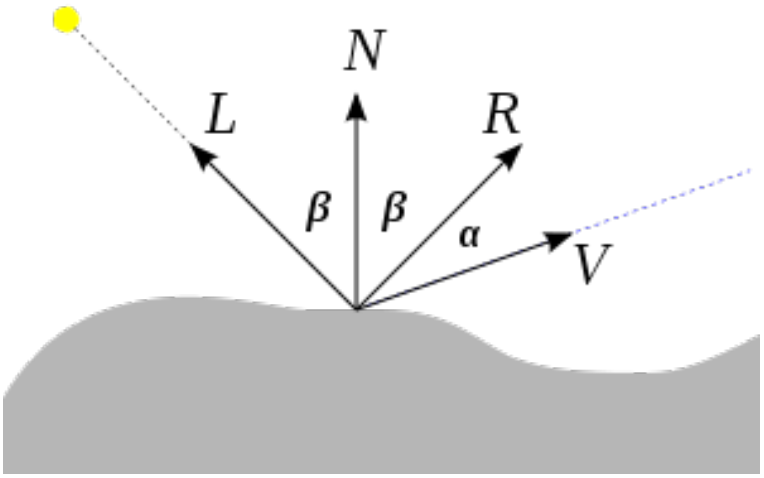
W powyższym wzorze $N$ i $L$ są kolejno: wektorem normalnym do powierzchni, wektorem wskazującym kierunek padania światła.

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

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

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

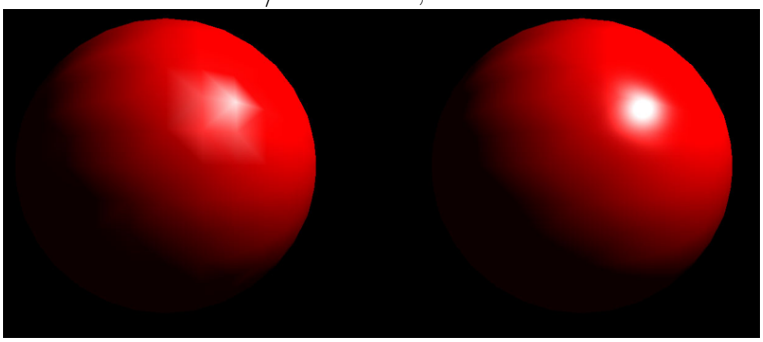
Rysunek 2.2: Model Phonga, źródło: https://pl.wikipedia.org/wiki/Cieniowanie_Phonga



### Cieniowanie

Cieniowanie ma na celu stworzenie złudzenia, w którym siatka trójkątów sprawia wrażenie gładkiej powierzchni. Najpopularniejszą metodą cieniowania jest „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) wektorów dla pozostałych punktów. Takie wektory są później wykorzystywane np. w modelu Phonga w miejsce prawdziwych wektorów normalnych. Jako że zarówno model Phonga jak i cieniowanie Phonga będą wykorzystywane w programie, którego dotyczy niniejsza praca, poniżej opisano metodę interpolacji wektorów.

Rysunek 2.3: „Cieniowanie Gourauda" i „Cieniowanie Phonga", źródło: http://www.csc.villanova.edu/ mdamian, 02.11.2017



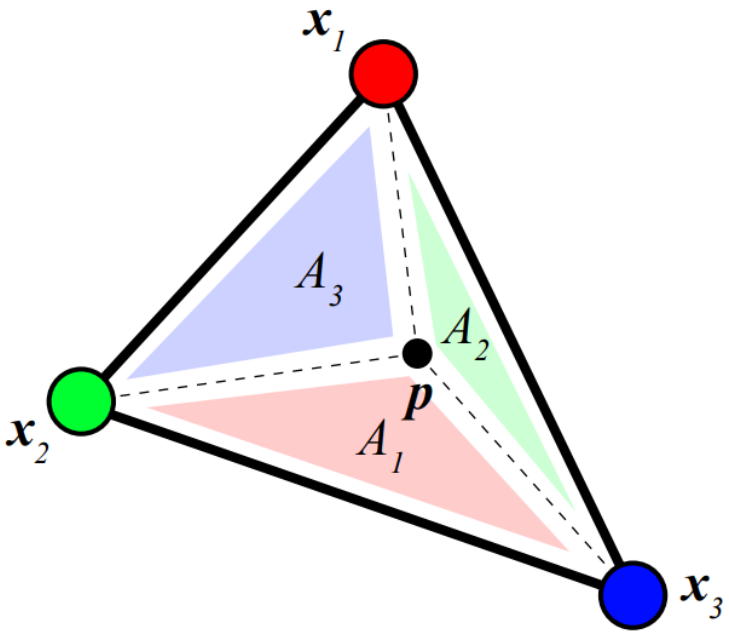
### Interpolacja barycentryczna

Do interpolacji wektorów może posłużyć nam tzw. interpolacja 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 im 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 punkcje:

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

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 niekoniecznie wynosi jeden.

### 2.1.4 Rekursywny algorytm śledzenia promieni

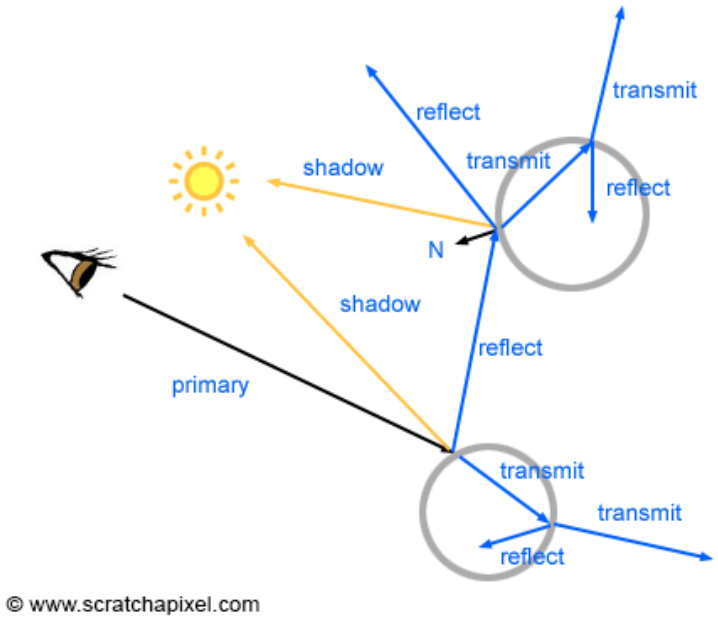
Rekursywny algorytm śledzenia promieni [1] jest rozwinięciem algorytmu podstawowego opisanego w poprzednim rozdziale. Uwzględnia on cienie, odbicia i załamania światła, dzięki śledzeniu dodatkowych promieni wysłanych z punktów przecięcia. W przypadku generowania cieni z każdego punktu przecięcia wysyła 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 Phonga omówionego powyżej), korzystamy tylko ze światła otoczenia. Dla odbić, do wysłania promienia wtórnego, korzysta się z zasady, która mówi, że kąt padania równy jest kątowi odbicia. W przypadku załamania światła, określa się współczynniki jego załamania (wartości dla różnych materiałów są stablicowane i ogólnodostępne) 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 jak 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 wysyłać - 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]

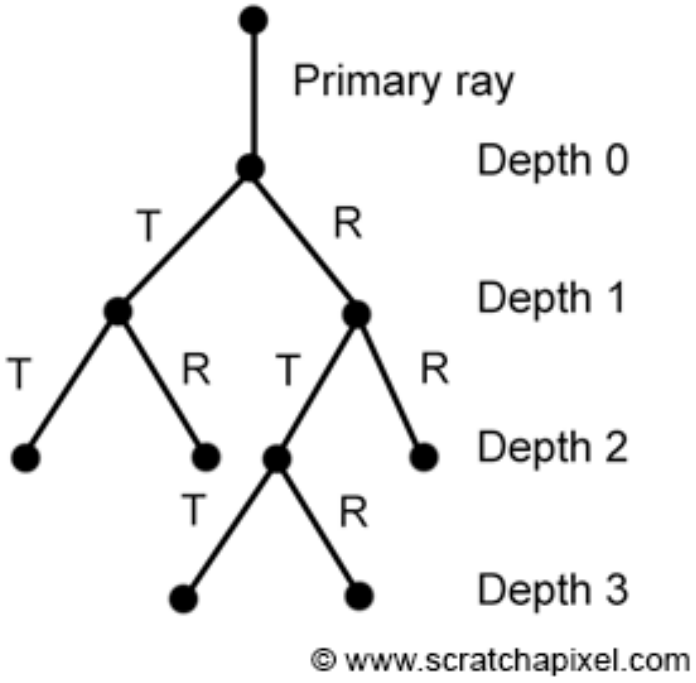



Rekursywny algorytm śledzenia promieni najczęściej implementuje się korzystając z funkcji rekurencyjnej [3], która przyjmuje punkt początkowy promienia, jego kierunek, oraz aktualną głębokość drzewa rekurencyjnego. Ostatni z parametrów często jest wykorzystywany w warunku stopu - po osiągnięciu określonej głębokości funkcja zwraca osiągnięty kolor.

### 2.1.5 Równoległa wersja algorytmu śledzenia promieni

Algorytm śledzenie promieni jest bardzo kosztowny obliczeniowo. Jedną ze skuteczniejszych metod przyspieszenia generowania obrazu jest jego zrówno-

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




leglenie, 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 na poziomie promieni pierwotnych - zbiór pikseli (pseudokod znajdujący się w punkcie 2.1.1) dzieli się na podzbiory, które można przeanalizować równolegle. Po obliczeniu kolorów wszystkich pikseli składa się je w jeden spójny obraz. Więcej o algorytmie równoległym można przeczytać w rozdziałach !ROZDZIAŁY!.

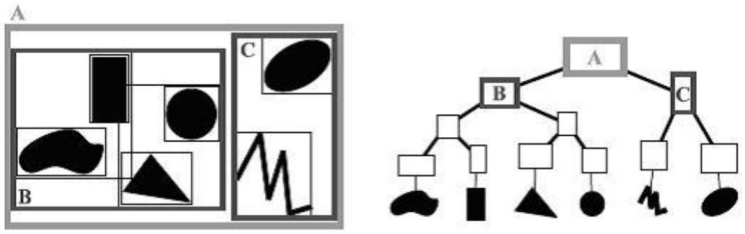
## 2.2 Optymalizacja algorytmu śledzenia promieni

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 mo-

delu 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 jest całą 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




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

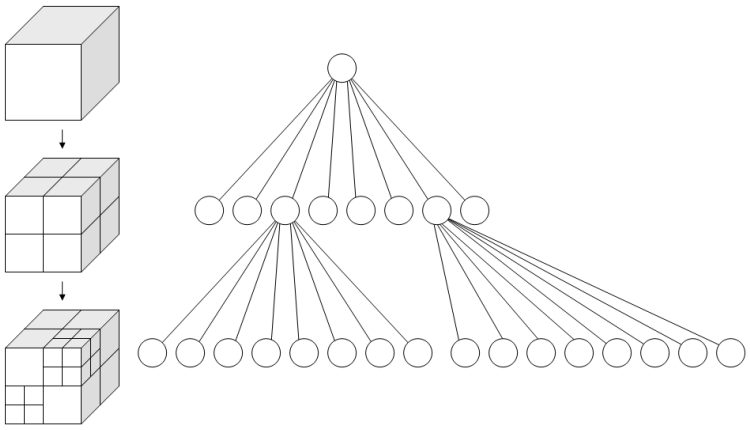
Budowa drzewa ósemkowego (ang. octree) polega na rekurencyjnym podziale przestrzeni na mniejsze regularne części - najczęściej sześciany.

Takie rozwiązanie ma znaczącą wadę w przypadku, kiedy 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].

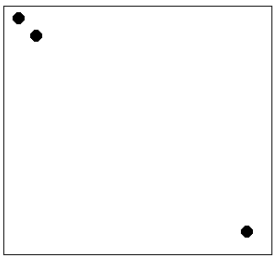
### 2.2.2 Drzewa K-d

Budowa drzewa K-d (skrót od *k-dimensional tree*) polega na podziale przestrzeni płaszczyznami równoległymi do osi układu współrzędnych (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 (jest to jedna z

Rysunek 2.8: octree, źródło: https://en.wikipedia.org/wiki/Octreee



Rysunek 2.9: Zły przypadek dla drzewa ósemkowego

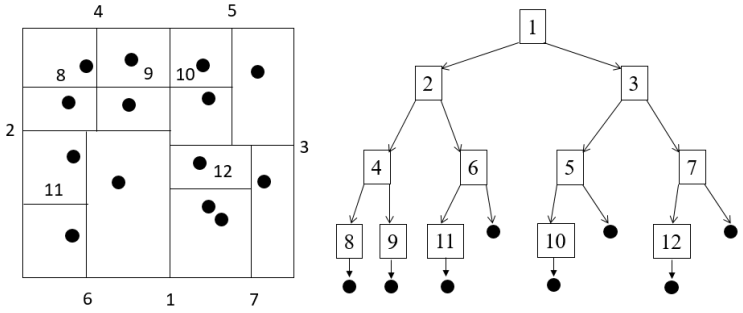


popularniejszych metod), a sama płaszczyzna przecinała jak najmniej figur. W ten sposób powstaje drzewo binarne.

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 „cięcie” można wyznaczyć dzięki prostym punktom.

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 obliczenia z zastosowaniem drzewa k-d są często szybsze niż przy zastosowaniu drzewa ósemkowego. Z drugiej jednak strony czas budowania takiego drzewa jest znacznie dłuższy (szukanie optymalnego „przecięcia”) niż w przypadku drzew ósemkowych.

Rysunek 2.10: Drzewo K-d dla dwóch wymiarów

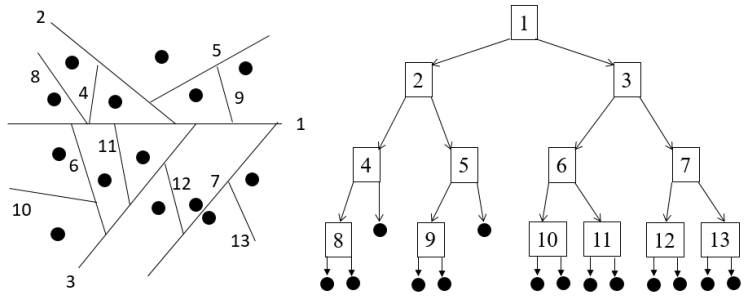



### 2.2.3 Drzewa BSP

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 podprzestrzenie. 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 hipotezy 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 K-d 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). Przy scenach zamkniętych składających się z wielu trójkątów (dla mniejszych scen BVH jest najczęściej lepsze) sytuacje nie jest już taka oczywista - tendencja się odwraca. Zgodnie z przewidywaniami, potwierdzonymi przez [14], dobrze zoptymalizowane drzewo BSP jest znacznie wydajniejsze od drzewa K-d, zwłaszcza jeżeli chodzi o czas zużyty na testy badające przecięcia trójkątów z promieniami. We wspomnianym artykule zostały również przedstawione badania na temat drzewa BVH - w tym przypadku 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).

Powyższe badania potwierdza współczesna literatura fachowa i obecnie stosowane metody optymalizacji generowania grafiki. W [11] autor proponuje rozwiązania hybrydowe, w których duże otwarte przestrzenie i poruszające się obiekty zamykane są 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 potrzeby 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 szczegółowo opisane w [11], jednak warto również polecić adres strony[17], na której można znaleźć wiele użytecznych informacji na powyższy temat (między innymi przykładową implementację jego elementów), poniższa treść w dużej mierze bazuje na tej pozycji.

### Budowa drzewa

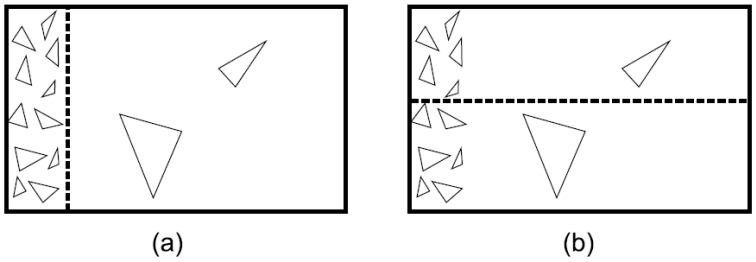
Podstawową wersję algorytmu budowania drzewa BSP można przedstawić przy użyciu pseudokodu:

Pierwszym ważnym krokiem, którego powyższy algorytm nie wyjaśnia, jest wybór płaszczyzny podziału. Najczęściej kandydatami są płaszczyzny wyznaczane przez prymitywy w wierzchołku drzewa (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 powstaje dobrze zbilansowane drzewo binarne. Popularną alternatywą takiego postępowania jest heurystyka SAH [18, 19, 20] (ang. Surface

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

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

    for polygon in list<plygons> 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
```

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 budowy 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 prostopadłościanami.

Rysunek 2.12: Płaszczyzny podziału: a - SAH, b - klasyczny podział; źródło: [18]



Kolejnym problem pojawia się w sytuacji 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 (metoda SAH decyduje o dalszym podziale biorąc pod uwagę przewidywany koszt przeglądania podprzestrzeni powstałych w wyniku podziału i koszt przeglądania prymitywów, jeżeli taki podział nie został dokonany). 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 płaszczyzny 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 więc wprowadzić mechanizmy zabezpieczające przed taką ewentualnością.

### 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, 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 zużycia 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 widżetów i system obsługi zdarzeń. Środowisko jest dostępne między innymi dla systemów Windows, Linux, Solaris, Symbian i Android. Popularność roz-

wiązania, 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 rodzaju dostępnego sprzętu. 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 w przeciwieństwie do systemów wieloprocesorowych, procesory nie są podłączone magistralą ze wspólną pamięcią, co z kolei oznacza wolniejszą i trudniejszą programistycznie komunikację między nimi. W taki, alternatywny sposób, wiele problemów mogłoby być rozwiązane efektywniej. 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 klaster obliczeniowy jest dobrym rozwiązaniem, zwłaszcza że jest to rozwiązania 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 zrównoleglalny 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. tej technologii [10].

# Rozdział 4

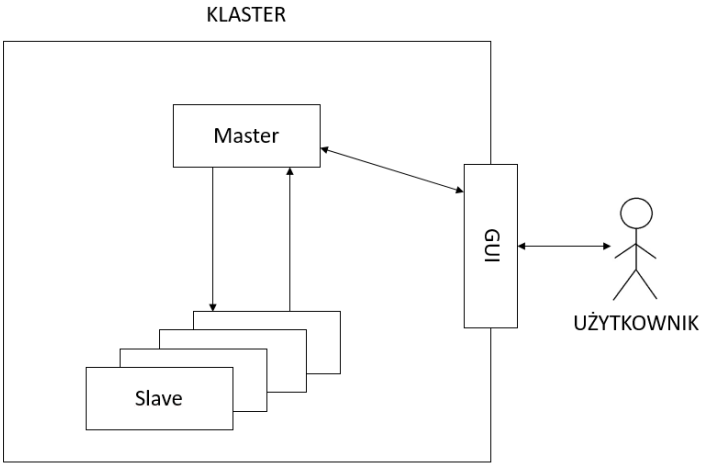
# Projekt systemu

W tym rozdziale przedstawiono projekt systemu, który ma zostać zaimplementowany na potrzeby tej pracy. Projektowany system powinien umożliwiać uruchomienie aplikacji na dowolnie dużym klastrze obliczeniowym składającym się z maszyn o różnej specyfikacji. Na komputerze, na którym aplikacja jest uruchamiana (a więc tym wykorzystywanym przez użytkownika) powinno uruchomić się okno dające podgląd na generowaną animację. Aplikacja powinna dawać możliwość załadowania pliku opisującego scenę, która następnie będzie rozesłana do wszystkich węzłów klastra. Opis działania klastra i planowanych klas programu (powiązania między nimi i rola w systemie) został umieszczony poniżej.

## 4.1 Projekt klastra

Na poniższym 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.

Rysunek 4.1: Schemat systemu




### 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 w kolejce oczekującej - takie rozwiązanie powinno dawać lepsze rezultaty niż rozesłanie do węzłów wszystkich zadań od razu, ponieważ różne fragmenty obrazu mogą być generowane z różną prędkością. Mogłoby więc dochodzić do sytuacji, w której część węzłów zrealizowała już swoje zadania (a więc ich moc obliczeniowa nie jest wykorzystywana), a część (która dostała bardziej wymagające obliczenia) ciągle liczy.

### 4.1.2 Slave

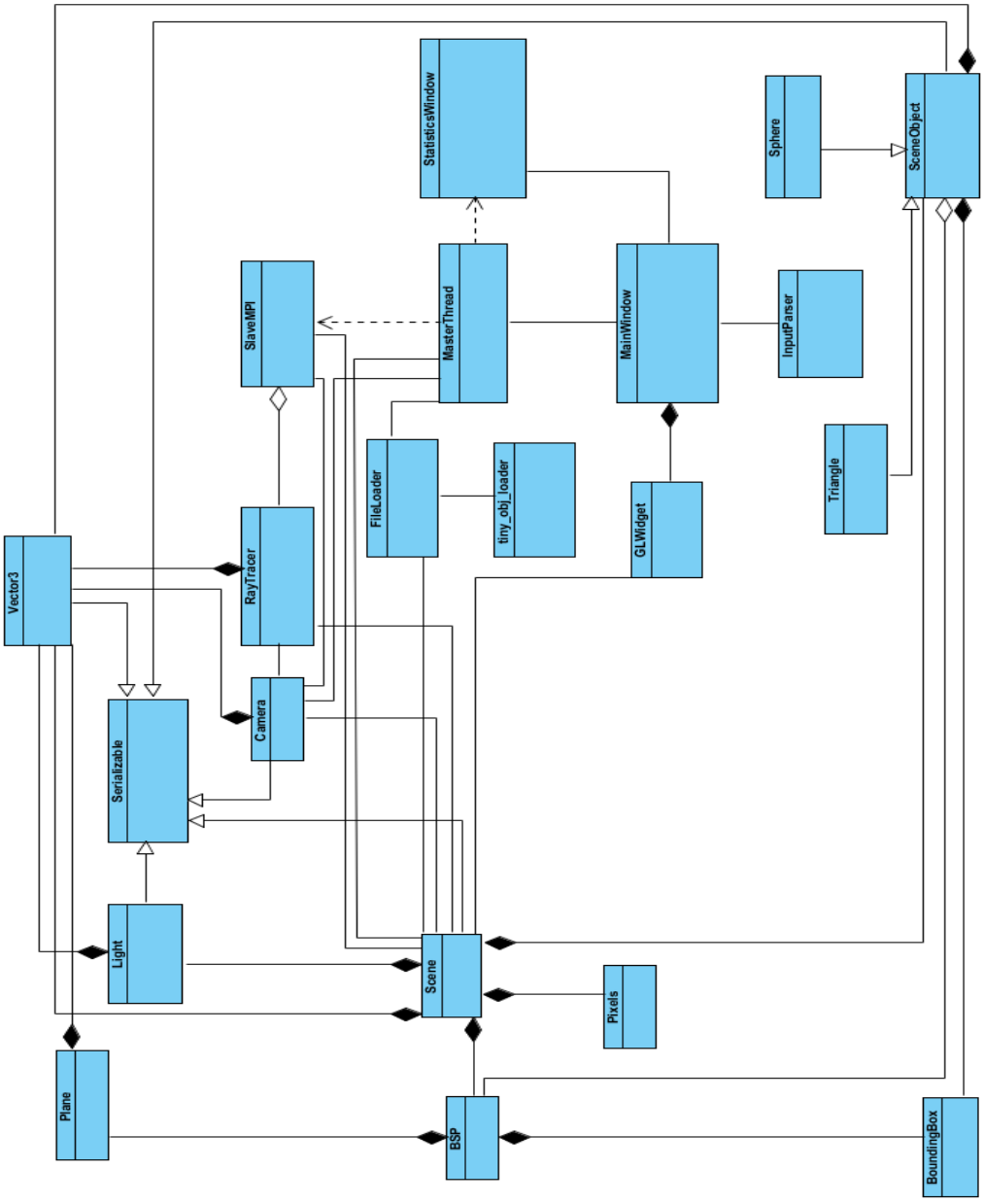
Nawiązując do poprzedniego punktu, pierwszą czynnością, którą powinien wykonać każdy ze slave'ów, jest odebranie definicji obiektów wykorzystywanych przy generowaniu sceny. Następnie w pętli, może on czekać na zadanie, realizować je (z wykorzystaniem klasy RayTracer) i odsyłać z powrotem do węzła zarządzającego.

```
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()
            elsepending–
            end if
        end if
    end while
    informGUI()
    updateCameraPos()
end while
```

```
recvScene()
recvCamera()

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

## 4.2 Projekt programu

### 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

#### BoundingBox

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.

#### BSP

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* |
| 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 wykrzystaniu 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.

### Camera

Tabela 4.4: Camera

| Camera |
|---|
| +zNear : float<br>+zFar : float<br>+pixWidth : int<br>+pixHeight : int<br>+povy : float<br>+aspect : float<br>+worldWidth : float<br>+worldHeight : float<br>+R : float<br>+ver : float<br>+hor : float<br>+instance : Camera*<br>+eye : Vector3¡float¿*<br>+look : Vector3¡float¿*<br>+up : Vector3¡float¿*<br>+lookAt : Vector3¡float¿* |
| +setUp(pixWidth : int, pixHeight : int) : void<br>+getInstance() : Camera *<br>+getWorldPosOfPixel(x : int, y : int) : Vector3¡float¿<br>+rotate() : void |

Podobnie jak obiekt klasy Scene, obiekt klasy Camera jest zbudowany według wzorca Singleton (może istnieć tylko jeden obiekt takiej klasy). Klasa ta

definiuje obiekt wirtualnej kamery, którą możemy umiejscowić w dowolnym miejscu i (z jej perspektywy) obserwować dowolny skrawek sceny. To ona zapewnia definicję rzutni i pozwala na translację współrzędnych świata (okno obserwatora) na współrzędne urządzenia (piksele ekranu).

### GLwidget

Tabela 4.5: GLwidget

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

Widgety 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.

### FileLoader

Tabela 4.6: FileLoader

| FileLoader |
|---|
| -readCameraSettings(line : char const*) : bool<br>-readSceneSettings(line : char const*) : bool<br>-readSphere(line : char const*) : bool<br>-readLight(line : char const*) : bool<br>-readTriangle(line : char const*) : bool<br>-readObj(line : char const*) : bool<br>+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.

### InputParser

Tabela 4.7: InputParser

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

Klasa InputParser jest prostą klasą pozwalająca 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.

### Light

Tabela 4.8: Light

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

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

### MainWindow

Tabela 4.9: 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.

### MasterThread

Tabela 4.10: 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()<br>sendDepth(int depth)<br>sendNextChunk(int dest)<br>sendExitSignal()<br>recvPixels(MPI_Status stat) : int<br>recvMessage() : int<br>finishPending()<br>updateProcessSpeed()<br>waitUntillRdy()<br>printResult(double spf, double bsp)<br>getNames()<br>emitNames() |

Klasa MasterThread jest główną klasa aplikacji, której obiekt tworzony jest w tym samym procesie co obiekt MainWindow, jednak działa 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 (sam stanowi on 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 rozesłać po klastrze i mieć możliwość modyfikacji ich parametrów (przesunięcie kamery co klatkę, aktualizacja obiektu Pixels). Więcej o zadaniach i sposobie działania MasterThread można przeczytać (!rodział z wyżej!, !rodział z implementacji!).

Tabela 4.11: Chunk

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

Chunk jest prostą strukturą wykorzystywaną w komunikacji Master/Slave (MasterThread, SlaveMPI). Zawiera on w sobie informacje o tym, jaki wycinek obrazu powinien być wyznaczany przez dany węzeł wykonawczy.

### Pixels

Tabela 4.12: 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.

### Plane

Tabela 4.13: Plane

| Plane |
|---|
| +a : float<br>+b : float<br>+c : float<br>+d : float |
| +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.

### RayTracer

Tabela 4.14: RayTracer

| RayTracer |
|---|
| +camera : Camera*<br>+scene : Scene* |
| +basicRayTracer() : void<br>+recursiveRayTracer(depth : int) : void<br>+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 klas 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.

### Scene

Tabela 4.15: Scene

| Scene |
|---|
| +numOfLights : int<br>+numOfObjects : int<br>+useShadows : bool |

| |
|---|
| +useBSP : bool<br>+instance : Scene*<br>+lights : Light**<br>+sceneObjects : SceneObject**<br>+pixels : Pixels*<br>+backgroundColor : Vector3*<br>+globalAmbient : Vector3*<br>+bsp : BSP* |
| +getInstance() : Scene *<br>+buildBSP(depth : int) : void<br>+addObject(sceneObject : SceneObject*) : void<br>+addLight(light : Light*) : void<br>+setUpPixels(x : int, y : int) : void<br>+getClosest(cross : Vector3, start : Vector3, dir : Vector3) : SceneObject *<br>+isInShadow(cross : Vector3, dir : Vector3, lightPos : Vector3) : bool<br>+setPixelColor(x : int, y : int, color : Vector3) : void<br>+serialize(bytes : vector<char>*) : void<br>+deserialize(bytes : vector<char> const) : void<br>+getType() : char |

Obiekt klasy Scene zawiera 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).

### SceneObject

Tabela 4.16: 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 |

| +getBoundingBox() : BoundingBox |
|---|

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

### Serializable

Tabela 4.17: Serializable

| Serializable |
|---|
| +serializedSize : int |
| +serialize(bytes : vector<char>*) : void<br>+deserialize(bytes : vector<char> const) : void<br>+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 rodziale 6.

### SlaveMPI

Tabela 4.18: SlaveMPI

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

### StatisticsWindow

Tabela 4.19: StatisticsWindow

| StatisticsWindow |
|---|
| Ui::StatisticsWindow *ui;<br>int worldSize; |
| 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 statystki dotyczące programu - czas generowania jednej klatki, średni czas pracy danego węzła, liczbę obiektów na scenie itd.

### Sphere

Tabela 4.20: 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 promieniem).

### Triangle

Tabela 4.21: Triangle

| Triangle |
|---|
| +pointA : Vector3* |
| +pointB : Vector3* |
| +pointC : Vector3* |
| +normalA : Vector3* |
| +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.

### Vector3

Tabela 4.22: 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 implementacją wszystkich metod zawartych w tabeli wyżej, zostanie również przeciążona

część operatorów arytmetycznych - ułatwi to korzystanie z klasy.

# Rozdział 5

# Implementacja

## 5.1 Szczegółowy opis wybranych fragmentów kodu

### 5.1.1 RayTracer

Klasa *RayTracer* implementuje zestaw metod realizujących algorytm śledzenia promieni. Korzysta ona z interfejsu udostępnianego przez klasy tj. *Scene*, czy *Camera* aby generować kolejne promienie do wysłania. Poniżej zostały omówione dwa najważniejsze fragmenty kodu zawarte w tej klasie

```
void RayTracer::recursiveRayTracer(int depth) {

Vector3<float> worldPosOfPixel;
Vector3<float> directionVector;

for(int i = 0; i < scene->getWidth(); i++) {
    for(int j = 0; j < scene->getHeight(); j++) {
        worldPosOfPixel = camera->getWorldPosOfPixel(i + scene->
            getStartX(),j + scene->getStartY());
        directionVector = worldPosOfPixel - *camera->getEye();
        directionVector.normalize();
        scene->setPixelColor(i, j, getColorRecursive(
            worldPosOfPixel, directionVector, depth));
    }
}
}
```

Powyższy fragment kodu implementuje algorytm, który można znaleźć w rozdziale !TU WSTAW RODZIAŁ!. Dla każdego piksela sceny zostaje wygenerowany promień pierwotny (*directionVector*), który następnie jest wysy-

łany w scenę - funkcja *getColorRecursive* (opisana niżej) zwraca kolor jaki należy przypisać danemu punktowi ekranu. Każdy z węzłów wykonawczych posiada swój egzemplarz obiektu klasy *Scene* (wykorzystywany w powyższym kodzie) zmodyfikowany w taki sposób, aby przechowywał on jedynie fragment obrazu (*Pixels)* - początek wycinka jest określany zmiennymi *startX* i *startY*, a zmodyfikowane wymiary obrazu pozwalają określić jego koniec. Więcej o komunikacji Master/Slave można przeczytać w rozdziałach !TU WSTAW RODZIAŁY!.

---

```
Vector3<float> RayTracer::getColorRecursive(Vector3<float>
    startPoint, Vector3<float> directionVector, int depth)
{

SceneObject* sceneObject;
Vector3<float> crossPoint;
Vector3<float> reflectedRay;
Vector3<float> localColor;
Vector3<float> reflectedColor;

//refraction
Vector3<float> transparencyColor;
Vector3<float> transparencyRay;

if (depth == 0)
    return Vector3<float>();

depth--;

sceneObject = scene->getClosest(crossPoint, startPoint,
    directionVector);

if (sceneObject == nullptr)
    return Vector3<float>(*scene->backgroundColor);

Vector3<float> normalVector = sceneObject->getNormalVector(
    crossPoint);
Vector3<float> observationVector = directionVector*-1;


if (observationVector.scalarProduct(normalVector) < 0) {
    normalVector = normalVector*-1;
}

if(sceneObject->getTransparency()>0) {
    transparencyRay = directionVector.refract(normalVector,
        sceneObject->getDensity(), 1);
    transparencyColor = getColorRecursive(crossPoint,
```

```
        transparencyRay, depth);
}

if (sceneObject->getLocal()>0) {
    localColor.setValues(sceneObject->getLocalColor(normalVector
        , crossPoint, observationVector));
}

if (sceneObject->getMirror()>0) {
    reflectedRay = directionVector.reflect(normalVector);
    reflectedColor = getColorRecursive(crossPoint, reflectedRay,
        depth);
}

return localColor*sceneObject->getLocal() + reflectedColor*
    sceneObject->getMirror() + transparencyColor*sceneObject->
    getTransparency();

}
```

Powyższa metoda jest rekurencyjnie wywoływaną metodą pozwalającą określić ostateczny kolor piksela, z którego został wysłany promień pierwotny (wysłanie promienia pierwotnego następuje w funkcji *recursiveRayTracer*. Przyjmuje ona promień (w postaci punktu początkowego i wektora kierunku) oraz zmienną określającą głębokość drzewa - jest ona dekrementowana z każdym kolejnym rekurencyjnym wywołaniem funkcji, a kiedy osiągnie zero, rekurencja jest przerywana. Pierwszym krokiem algorytmu jest określenie, czy promień przeciął się z jakimś obiektem (jest tutaj wykorzystywany albo przegląd zupełny, albo drzewo BSP). Jeżeli nie to ostateczny kolor piksela (lub jego składowa na danym poziomie) przyjmuje wartość koloru tła. Jeżeli tak, to algorytm wybiera obiekt będący najbliżej początku promienia (obiekt widoczny z perspektywy tego punktu) i (w zależności od modelu *Phonga* i parametrów powierzchni omówionych w rozdziałach !TU WSTAW ROZDZIAŁ!) ustala lokalną barwę obiektu oraz wysyła dwa kolejne promienie mające wpływ na barwę ostateczną - promień odbity od powierzchni i promień przez nią przechodzący (jest tutaj uwzględnianie złamanie światła). Ostateczny kolor piksela jest sumą kolorów lokalnych osiągniętych przez wszystkie promienie powstałe w wyniku wysłania promienia pierwotnego. Niezrozumiały może wydawać się następujący fragment:

```
if (observationVector.scalarProduct(normalVector) < 0) {
    normalVector = normalVector*-1;
}
```

Biorąc pod uwagę, że kierunek wektora normalnego ma wpływ na otrzymany kolor lokalny powierzchni (jeżeli jego kierunek jest niezgodny z kie-

runkiem światła to znaczy, że powierzchnia nie jest oświetlona) należy go odwrócić tak aby był on zgodny z kierunkiem obserwacji - np w sytuacji w której obserwator znajdowałby się w kuli (wraz ze światłem oświetlającym scenę), a wektor normalny do powierzchni kuli skierowany byłby na zewnątrz, oświetlenie i tak nie miałoby na nią wpływu.

### 5.1.2 BSP

W tym punkcie zostanie przedstawione w jaki sposób zaimplementowano budowę drzewa BSP oraz jego przeglądanie.

---

```
root->partitionPlane = getBestPlane(polygons);
while(!polygons.empty()) {
    object = polygons.back();
    polygons.pop_back();
    result = root->partitionPlane.classifyObject(object);
    switch (result) {
        case FRONT:
            frontList.push_back(object);
        break;

        case BACK:
            backList.push_back(object);
        break;

        case COINCIDENT:
            backList.push_back(object);
            frontList.push_back(object);
        break;

        case SPANNING: {
            if (object->getType() == 's') {
                root->polygons.push_back(object);
            } else {
                Triangle *triangle = static_cast<Triangle*>(
                    object);
                std::list<Triangle*> tempFrontList, tempBackList
                    ;
                triangle->split(root->partitionPlane,
                    tempFrontList, tempBackList);
                while (!tempBackList.empty()) {
                    backList.push_back(tempBackList.back());
                    tempBackList.pop_back();
                }
                while (!tempFrontList.empty()) {
                    frontList.push_back(tempFrontList.back());
                    tempFrontList.pop_back();
```

```
            }
            }
        }
        break;

        default:
            break;
        }
    }
```

Powyższy fragment kodu jest fragmentem kodu funkcji budującej drzewo, który nie został do końca uwzględniony w rozdziale (!TU WSTAW ROZDZIAŁ Z PSEUDOKODEM!). Pierwszym krokiem każdej kolejnej rekurencji budowy drzewa jest ustalenie płaszczyzny podziału - brane są pod uwagę wszystkie te, które są wyznaczane przez trójkąty zawarte w danym wierzchołku i te które są do tych trójkątów prostopadłe (styczne z krawędziami). Poprzez najlepszą płaszczyznę rozumie się taką, która dzieli trójkąty na równe ilościowo grupy. Następnie, w pętli, algorytm sprawdza, po której stronie wybranej płaszczyzny znajduje się dany obiekt z listy - w zależności od sytuacji trafia on do listy, która zostanie przekazana kolejnym dzieciom („przedniemu" i „tylnemu"). W przypadku gdy trójkąt leży na płaszczyźnie podziału jest on umieszczany w obu listach, z kolejki jeżeli płaszczyzna przecina trójkąt to jest on dzielony na dwa (w przypadku powstania trójkąta i czworokąta, czworokąt jest dzielony na dwa trójkąty) - każda z połówek trafi do odpowiedniego „dziecka". Biorąc pod uwagę, że program uwzględnia sfery przechowywane w postaci równania (a więc prosty podział takiego obiektu nie jest możliwy), w przypadku w którym płaszczyzna przecina sferę, trafia ona tylko do listy obiektów rozpatrywanego wierzchołka. Zaimplementowanie sfer wymagało takiej niekonwencjonalnej modyfikacji algorytmu, która będzie miała wpływ na sposób przeglądania drzewa. Rekurencja kończy się, kiedy zostanie osiągnięty maksymalna wysokość drzewa (określana przez parametr przekazywany do funkcji przy pierwszym wywołaniu), lub w sytuacji, w której dalszy podział jest niemożliwy (wierzchołek otrzymuje jeden trójkąt lub dalszy podział nie przynosi efektów). Wtedy wierzchołek jest zamieniany w liść (wskaźniki na potomstwo są puste), a wejściowa lista obiektów zostaje do niego przypisana. Pomijając sfery wszystkie wierzchołki niebędące liśćmi są puste.

```
SceneObject *BSP::intersect(BSP::node *root, Vector3<float> &
    crossPoint, Vector3<float> &startingPoint, Vector3<float> &
    directionVector) {

if (root->back == nullptr && root->front == nullptr) {
```

```
    return getClosestInNode(root->polygons, crossPoint,
        startingPoint, directionVector);
}

SceneObject *thisNodeHit = nullptr;
Vector3<float> tempCross;

if(!root->polygons.empty()) {
    thisNodeHit = getClosestInNode(root->polygons, tempCross,
        startingPoint, directionVector);
}

node *near;
node *far;
SceneObject *hit = nullptr;;


switch (root->partitionPlane.classifyPoint(&startingPoint)) {
    case FRONT:
        near = root->front;
        far = root->back;
    break;

    case BACK:
        near = root->back;
        far = root->front;
    break;

    case COINCIDENT: {

        Vector3<float> point = startingPoint + directionVector;
        if (root->partitionPlane.classifyPoint(&point) == FRONT)
            {
            near = root->front;
            far = root->back;
        }
        else {
            near = root->back;
            far = root->front;
        }
    }
    break;

    default:
        return nullptr;
        break;
}

hit = intersect(near, crossPoint, startingPoint, directionVector
```

```
    );

if (hit == nullptr && root->partitionPlane.rayIntersectPlane(
    startingPoint,directionVector)) {
     hit = intersect(far, crossPoint, startingPoint,
         directionVector);
}

if (thisNodeHit != nullptr) {
     if (hit != nullptr) {
          if (tempCross.distanceFrom(startingPoint) < crossPoint.
              distanceFrom(startingPoint)) {
               hit = thisNodeHit;
               crossPoint = tempCross;
          }
     }
     else {
          hit = thisNodeHit;
          crossPoint = tempCross;
     }
}
return hit;
}
```

Powyżej przedstawiono rekurencyjny algorytm przeszukiwania drzewa. Funkcja ta przyjmuje wskaźnik na sprawdzany wierzchołek drzewa i promień, a zwraca wskaźnik na znaleziony obiekt i punkt przecięcia (zmienna *crossPoint* widoczna w liście parametrów).

Pierwszym krokiem algorytmu jest sprawdzenie czy dany wierzchołek jest liściem Jeżeli tak to zostaje przeprowadzony test przecięcia na każdym obiekcie znajdującym się w wierzchołku - wybieramy najbliższy trafiony i zwracamy go do funkcji wywołującej. Następnie należy sprawdzić, czy dany wierzchołek rzeczywiście jest pusty (komplikacje spowodowane nietypowym obiektem nie będącym poligonem - sferą). Jeżeli nie jest, to ponownie zostaną przeprowadzone testy przecięć dla każdego obiektu znajdującego się w wierzchołku - znaleziony obiekt przechowujemy w zmiennej *thisNodeHit*.

Następnie, w zależności od tego, po której części płaszczyzny znajduje się punkt początkowy promienia, zostają ustawione zmienne „near” (połowa, w której znajduje się punkt początkowy) i „far” (alternatywa). W przypadku, w którym punkt startowy zawiera się w płaszczyźnie dzielącej, sprawdzamy czy wektor kierunku nie jest równoległy do płaszczyzny - jeżeli jest „near” i „far” nie ma znaczenia; jeżeli nie jest, to jako „near” wybieramy tę połowę wskazywaną przez wektor.

Kolejnym krokiem jest rekurencyjne wywołanie tej funkcji dla potomka „near”. Jeżeli nie zwróci ono żadnego obiektu, a promień przecina płaszczy-

znę dzielącą to rozwiązanie może znajdować się jeszcze w drugim potomku („far"). Ostatni fragment kodu sprawdza, czy jeżeli znaleziono obiekt w danym węźle i obiekt w jednym z dzieci to który z nich jest bliżej - ten zostanie zwrócony jako wynik.

### 5.1.3 MasterThread

```
void MasterThread::run() {

while (true) {

    splitToChunks(numChunks);

    pending = 0;
    for (int i=1; i<worldSize; i++) {
        if (!sendNextChunk(i)) break;
        pending++;
        }

    int dest;
    while(pending>0) {
        switch(recvMessage()) {
            case EXIT: return; break;
            case PIXELS:
                dest = recvPixels(status);
                if (!sendNextChunk(dest))
                    pending--;
                break;
            default: break;
        }
        emit processInfo(processSpeed);
    }

    camera->rotate();
    sendCameraPointToPoint();

    emit workIsReady();

}
}
```

### 5.1.4 SlaveMPI

```
int SlaveMPI::exec() {

RayTracer rayTracer;
while(true) {

    switch(recvMessage()) {
        case EXIT:
            return EXIT; break;
        case CHUNK:
            recvChunk();
            rayTracer.recursiveRayTracer(depth);
            sendPixels(); break;
        case CAMERA:
            recvCameraPointToPoint(); break;
        default: break;
    }
}
return 0;
}
```

## 5.2 Serializacja

```
#ifndef SERIALIZABLE_H
#define SERIALIZABLE_H

#include "vector"

class Serializable
{
public:

    virtual void serialize(std::vector<char> *bytes) = 0;
    virtual void deserialize(const std::vector<char>& bytes) =
       0;
    virtual char getType() = 0;
    virtual ~Serializable();
    int serializedSize;

};

#endif // SERIALIZABLE_H
```

```
void Sphere::serialize(std::vector<char> *bytes)
{
    bytes->resize(serializedSize);
```

```
    char *ptr = bytes->data();
    std::vector<char> vec;

    amb->serialize(&vec);
    memcpy(ptr, vec.data(), vec.size()); ptr += vec.size();
    dif->serialize(&vec);
    memcpy(ptr, vec.data(), vec.size()); ptr += vec.size();
    spec->serialize(&vec);
    memcpy(ptr, vec.data(), vec.size()); ptr += vec.size();

    memcpy(ptr, &specShin, sizeof(specShin)); ptr += sizeof(
        specShin);

    pos->serialize(&vec);
    memcpy(ptr, vec.data(), vec.size()); ptr += vec.size();

    memcpy(ptr, &radius, sizeof(radius)); ptr += sizeof(radius);
    memcpy(ptr, &transparency, sizeof(transparency)); ptr +=
        sizeof(transparency);
    memcpy(ptr, &mirror, sizeof(mirror)); ptr += sizeof(mirror);
    memcpy(ptr, &local, sizeof(local)); ptr += sizeof(local);
    memcpy(ptr, &density, sizeof(density));
}
```

```
void Sphere::deserialize(const std::vector<char> &bytes)
{
    const char* ptr = bytes.data();
    std::vector<char> vec;
    vec.resize(Vector3<float>::serializedSize);
    memcpy(vec.data(), ptr, vec.size()); ptr += vec.size();
    amb->deserialize(vec);
    memcpy(vec.data(), ptr, vec.size()); ptr += vec.size();
    dif->deserialize(vec);
    memcpy(vec.data(), ptr, vec.size()); ptr += vec.size();
    spec->deserialize(vec);

    memcpy(&specShin, ptr, sizeof(specShin)); ptr += sizeof(
        specShin);

    memcpy(vec.data(), ptr, vec.size()); ptr += vec.size();
    pos->deserialize(vec);

    memcpy(&radius, ptr, sizeof(radius)); ptr += sizeof(radius);
    memcpy(&transparency, ptr, sizeof(transparency)); ptr +=
        sizeof(transparency);
    memcpy(&mirror, ptr, sizeof(mirror)); ptr += sizeof(mirror);
    memcpy(&local, ptr, sizeof(local)); ptr += sizeof(local);
    memcpy(&density, ptr, sizeof(density));
}
```

# 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, 15.09.2017

[6] Wikipedia, Wolna Encyklopedia: 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, 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 Praque

[15] M. Zlatuska, V. Havran *Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms*, Czech Technical University in Praque

[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 Saarbrucken, Germany, 2004

# Spis rysunków