POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka, INF
SPECJALNOŚĆ: Inżynieria Internetowa, INT

PRACA DYPLOMOWA INŻYNIERSKA

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

Generating images with parallel real-time ray-tracing

AUTOR:
Mateusz Gniewkowski

PROWADZĄCY PRACĘ:
dr inż. Henryk Maciejewski

OCENA PRACY:

WROCŁAW 2017

Spis treści

Rozdział 1

Wstęp

Informatyka i jej wszystkie dziedziny pokrewne bez możliwości wizualizacji nie była by tym, czym są dzisiaj. Bez rozwoju grafiki komputerowej zakres, w jakim można wykorzystywać komputery, byłby dużo mniejszy. Nie istniałyby gry komputerowe, filmy pozbawiono by efektów specjalnych (aspekty artystyczne), a brak możliwości graficznej wizualizacji badań i projektów ograniczyłby rozwój technologiczny. Dlatego też metody generowania obrazów są nie tylko nieustannie doskonalone, ale i stały się przedmiotem badań licznej grupy specjalistów. Jedną z pierwszych metod pozwalających na renderowanie fotorealistycznych grafik jest tzw. *metoda śledzenia promieni* (ang. *ray tracing*. Jej początki zawdzięczamy Arthur'owi Appel'owi oraz Robert'owi Goldstein'owi i Roger'owi Nagel'owi, natomiast rekursywny algorytm po raz pierwszy wprowadził Turner Whitted - jego rozwiązanie uwzględniało również promienie odbite od powierzchni i takie, które uległy załamaniu.

Rysunek 1.1: Glasses - grafika wygenerowana przy użycie programy *POV-Ray*, autor: Gilles Tran, źródło: http://www.oyonale.com/modeles.php?lang=en&page=40

Metoda śledzenia promieni jest stosunkowo prostym algorytmem, który pozwala na generowanie bardzo złożonych i realistycznych grafik uwzględniających wiele zjawisk fizycznych. Wadą takiego rozwiązanie jest bardzo długi czas generowania obrazu, przez co nie jest ono wykorzystywane w aplikacjach interaktywnych. Zamiast tego wykorzystuje się tzw. *computer graphics pipeline* - skomplikowaną, wpieraną sprzętowo sekwencję kroków wykorzystywanych w bibliotekach graficznych takich jak *OpenGL* (dokładny opis działania można znaleźć w [21]). Pozwala ona na szybkie renderowanie obiektów, jednak obrazy wygenerowane tą metodą nie będą już tak realistyczne - wiele pracy wkłada się w poprawieniu ich jakości.

Podstawowym pytaniem, jakie jest stawiane w tej pracy, jest to, czy metoda śledzenia promieni ma szansę być wykorzystywana w aplikacjach interaktywnych tak, aby obliczenia związane z generowaniem obrazu były niewidoczne dla użytkownika? Jakie parametry sceny pozwalają na generowanie obrazu z zadowalającą prędkością? Czy zrównoleglenie obliczeń pozwoli zbliżyć się do minimalnego progu 24 klatek na sekundę tak, aby można było mówić o generowaniu obrazu w czasie rzeczywistym? W jaki sposób moglibyśmy przyspieszyć obliczenia? Wiele z tych pytań pozostanie otwartych, jednak badania takie jak te są próbą znalezienia na nie odpowiedzi.

1.1 Wykazu celów i zadań pracy

Celem niniejszej pracy jest:

1. Dokonanie analizy problemu na podstawie zebranej literatury.
2. Określenie wymagań jakie powinien spełniać system.
3. Dobór narzędzi programistycznych w celu zaimplementowania systemu realizującego algorytm śledzenia promieni.
4. Zaprojektowanie systemu.
5. Implementacja systemu.
6. Przeprowadzenie badań nad algorytmem metody śledzenia promieni.
7. Omówienie rezultatów.
8. Zaproponowanie dodatkowych rozwiązań mających na celu przyspieszenie obliczeń.

1.2 Budowa pracy

Niniejszy dokument składa się z ośmiu rozdziałów - ten, stanowiący wstęp, jest jednym z nich. W rozdziale drugim znajduje się dogłębna analiza problemu - zostały tam w sposób szczegółowy przedstawione wszystkie podejmowane zagadnienie. W rozdziale trzecim zostały zaproponowane technologie, które pozwolą na implementację programu mającego realizować rekursywną metodę śledzenia promienie z wykorzystaniem obliczeń równoległych. Na początku rozdziału czwartego zostały zdefiniowane wymagania i założenia jakie powinien realizować program. Kolejne podrozdziały w sposób bardziej formalny i szczegółowy pokazują architekturę aplikacji. W rozdziale piątym zostały zawarte niektóre szczegóły implementacyjne gotowego już programu. Rozdział szósty opisuje, w jaki sposób

program działa - przedstawiono tam podstawową instrukcje jego obsługi oraz budowę pliku wejściowego. Rozdział siódmy zawiera rezultaty przeprowadzonych testów, wraz z ich omówieniem. Znajdują tam się również przykładowe obrazy jakie zostały wygenerowane przez działającą aplikację. W rozdziale ósmym znajduje się podsumowanie pracy wraz z propozycjami kierunku dalszych badań.

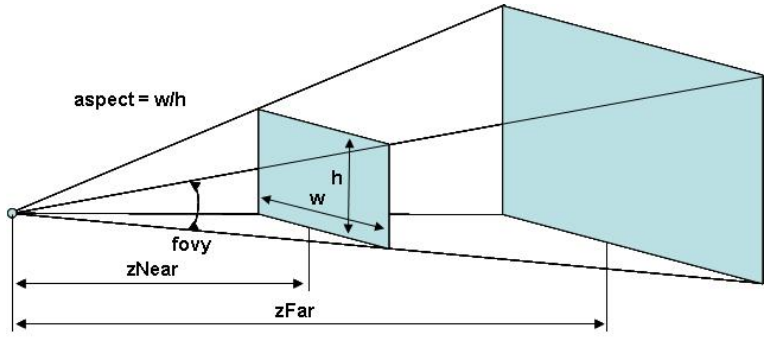
Rozdział 2

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 punkcie 2.1.3), który został przecięty przez wysłany promień. W przypadku braku kolizji piksel przybiera barwę otoczenia. Więcej na temat podstaw śledzenia promieni można przeczytać w [1, 3, 4].



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

Poniżej przedstawiono pseudokod podstawowego śledzenia promieni

Algorithm 1 Podstawowy algorytm metody śledzenia promieni

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

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

ustalenie kolor piksela na podstawie obj
```

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 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 punkcie 2.2.3).
Dane są równania promienia i równanie płaszczyzny:

$$p = p_0 + tv$$

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

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:

$$(p_0 + tv) \circ N + D = 0$$
$$t = \frac{-(p_0 \circ N + D)}{(v \circ N)}$$

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 $v \circ N$ 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:
   $$v = (b - a) \times (c - a)$$
   $$n = \frac{v}{|v|}$$
   (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:
   $$(b - a) \times (x - a) \circ n > 0$$
   $$(c - b) \times (x - b) \circ n > 0$$
   $$(a - c) \times (x - c) \circ n > 0$$

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:

$$pvec = d \times (c - a)$$
$$qvec = (p - a) \times (b - a)$$
$$invDet = \frac{1}{(b - a) \circ pvec}$$
$$u = ((p - a) \circ pvec)) * invDet$$
$$v = (d \circ qvec) * invDet$$
$$t = ((c - a) \circ qvec) * invDet$$

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;
}
```

Listing 2.1: Algorytm *Möller – Trumbore*

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.

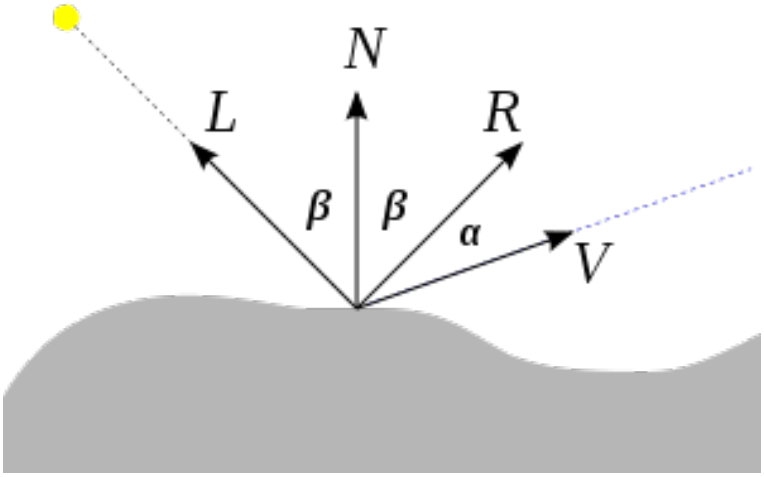
$$I = I_{amb} * k_{amb} + I_{dif} * k_{amb} * (N \circ L)$$

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:

$$I = I_{amb} * k_{amb} + \frac{1}{a + bd + cd^2} * k_{amb} * (N \circ L) + k_{spec} * I_{spec} * (R \circ V)^n)$$

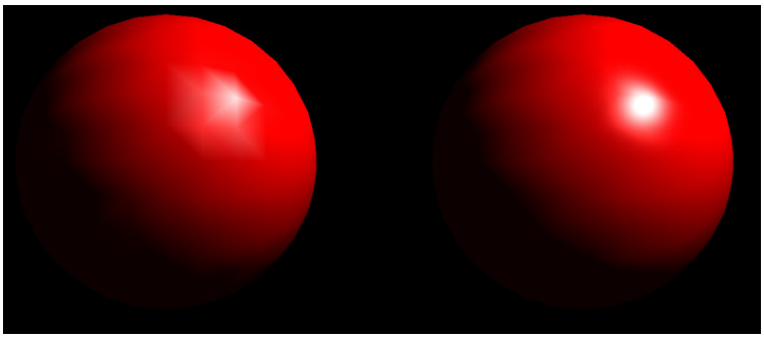
gdzie R i V oznaczają kolejno kierunek odbicia promienia i kierunek obserwacji. Ułamek $\frac{1}{a+bd+cd^2}$ określa intensywność padającego światła w zależności od odległości od źródła (d to odległość od źródła światła). Pozostałe parametry są dobieranie empirycznie.



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

Cieniowanie

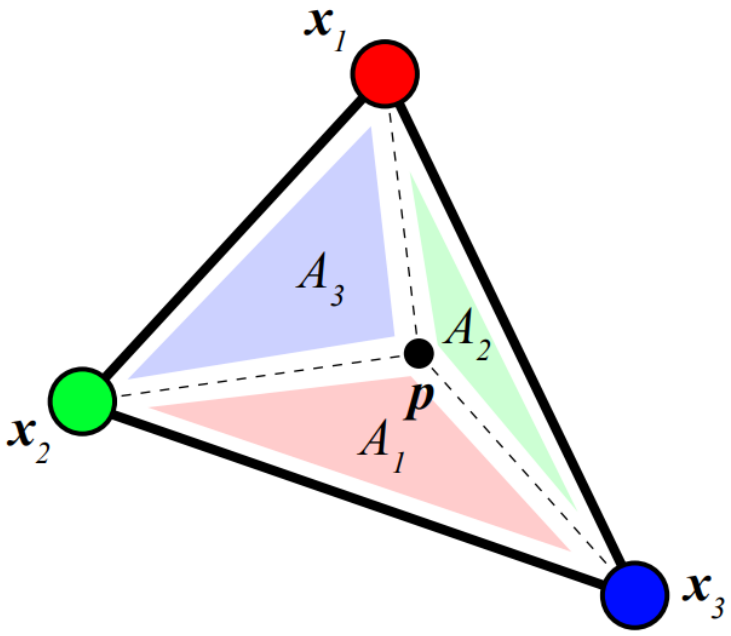
Cieniowanie ma na celu stworzenie złudzenia, w którym siatka trójkątów sprawia wrażenie gładkiej powierzchni. Najpopularniejszą metodą cieniowania jest „Cieniowanie Gourauda", które polega na wyliczeniu kolorów każdego wierzchołka prymitywu, a następnie interpolacji pozostałych. Takie rozwiązanie jest 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:



Rysunek 2.4: Interpolacja barycentryczna, źródło: nieznane

Na powyższym rysunku 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:

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

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

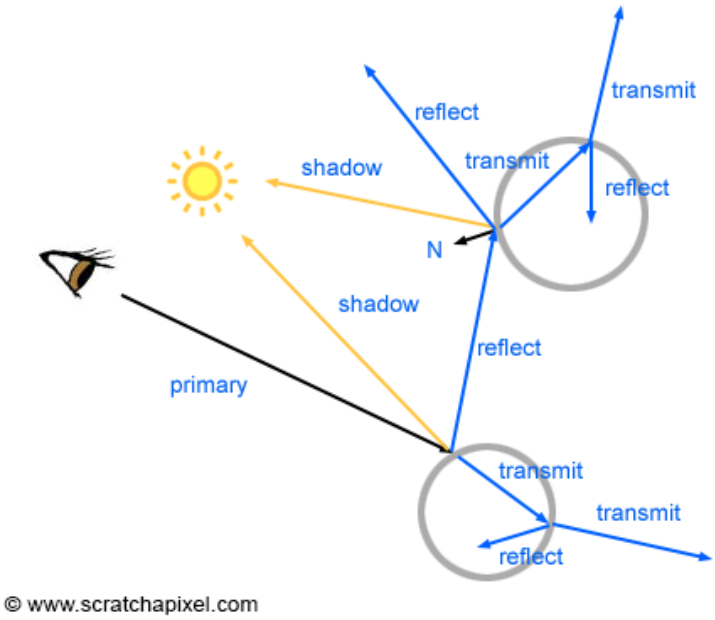
Rekursywny algorytm śledzenia promieni [1] jest rozwinięciem algorytmu podstawowego opisanego w poprzednim rozdziale. Uwzględnia on cienie, odbicia i załamania światła, dzięki śledzeniu dodatkowych promieni wysłanych z punktów przecięcia. W przypadku generowania cieni z każdego punktu przecięcia wysyła 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]:

$$\frac{sin(alpha)}{sin(beta)} = \frac{n_2}{n_1}$$

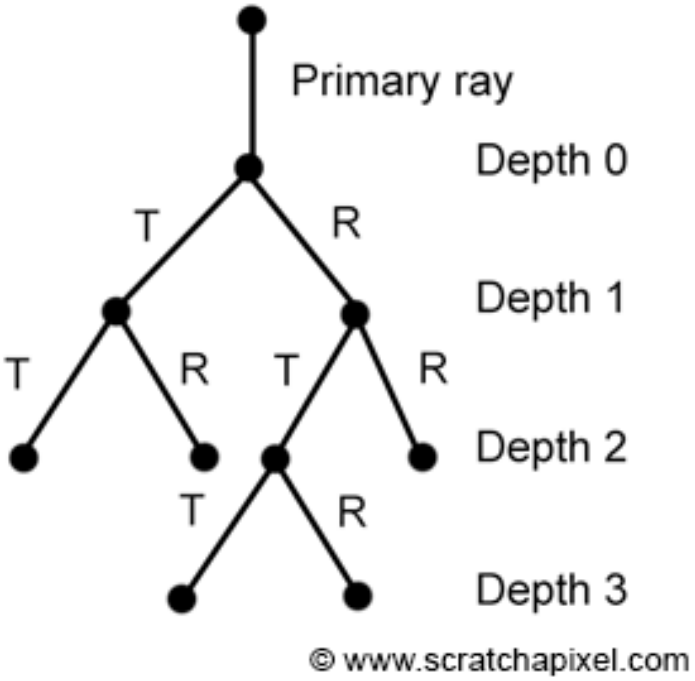
gdzie α - kąt padania, β - kąt załamania, n_1 - współczynnik załamania pierwszego materiału, n_2 współczynnik załamania drugiego materiału.

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 promieni. Ostatni z parametrów często jest wykorzystywany w warunku stopu - po osiągnięciu określonej głębokości funkcja zwraca osiągnięty kolor.



Rysunek 2.6: Wizualizacja algorytmu rekursywnego - drzewo promieni, źródło: [4]

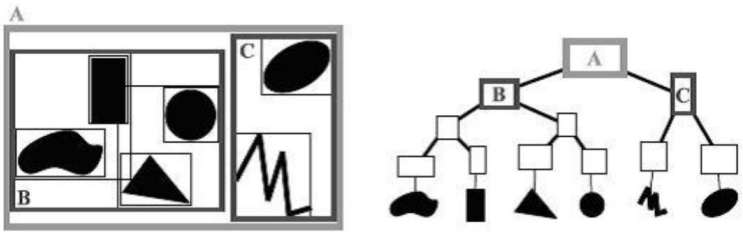
2.1.5 Równoległa wersja algorytmu śledzenia promieni

Algorytm śledzenie promieni jest bardzo kosztowny obliczeniowo. Jedną 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 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 rozdziale czwartym.

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 jest całą scena. Drzewo podziału przestrzeni, które zostało opisane w poprzednim akapicie, nazywamy drzewem brył ograniczających (lub otaczających), z angielskiego: bound volume hierarchy - BVH.



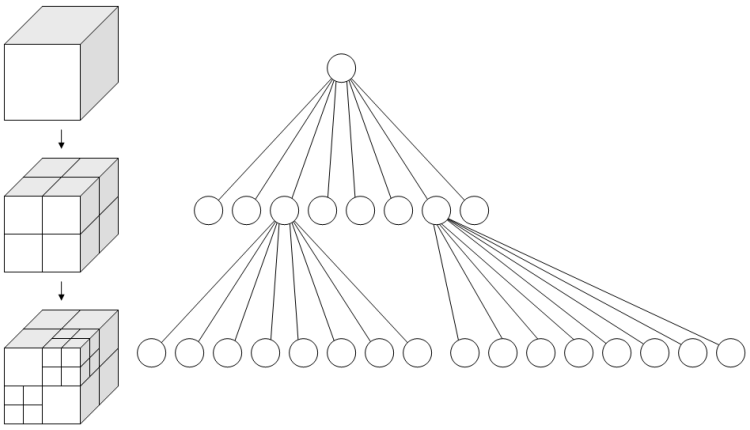
Rysunek 2.7: drzewo BVH, źródło: https://pl.wikipedia.org/wiki/Drzewo_BVH

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].

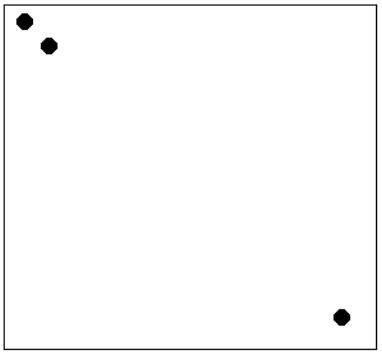
2.2.1 Drzewa ósemkowe

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



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

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].

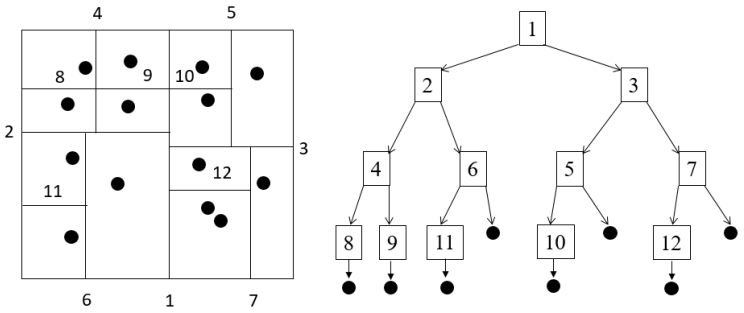


Rysunek 2.9: Zły przypadek dla drzewa ósemkowego

2.2.2 Drzewa K-d

Budowa drzewa K-d (skrót od *k-dimensional tree*) polega na podziale przestrzeni płaszczyznami równoległymi do osi układu 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 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.



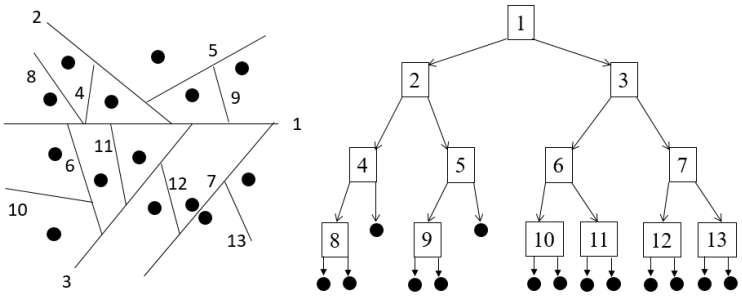
Rysunek 2.10: Drzewo K-d dla dwóch wymiarów

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 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.

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 BSP

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 jego budowy). Jeżeli porównywać ze sobą drzewa BVH i K-d warto zajrzeć do [14, 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.

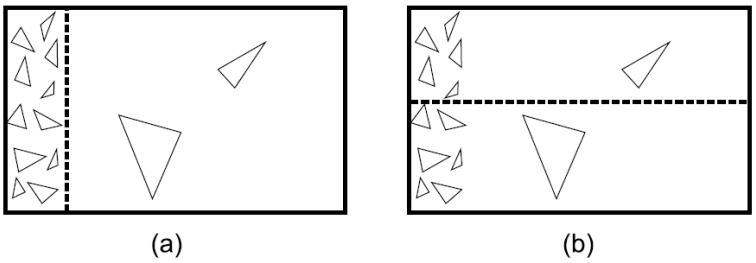
Budowa drzewa

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

Algorithm 2 Budowa drzewa BSP

```
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
```

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 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 głębokość drzewa.

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 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 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 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 pokazać 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 w kolejnych podrozdziałach. Niżej przedstawiono listę podstawowych zadań i założeń do zrealizowania w ramach implementacji systemu (została ona ustalona na podstawie analizy znajdującej się w rozdziale 2).

1. Aplikacja będzie realizowała rekursywny algorytm śledzenia promieni.
2. Aplikacja będzie aplikacją okienkową stworzoną przy użyciu biblioteki *QT*.
3. Dzięki wykorzystaniu technologii *MPI* obliczenia będą mogły wykonywać się równolegle (więcej szczegółów w kolejnym podrozdziale).
4. Zadanie realizowane przez pojedynczy węzeł wykonawczy będzie polegało na wyliczeniu kolorów pikseli pewnego fragmentu obrazu.
5. Musi zostać zaimplementowana możliwość definiowania parametrów sceny (pewna przestrzeń zawierająca obiekty) jak i obserwatora (określa sposób rzutowania obiektów sceny oraz miejsce z którego scena jest obserwowana oraz kierunek patrzenia).
6. Aplikacja będzie prezentowała animację budowaną w czasie rzeczywistym - obserwator (a więc i obraz) będzie się obracał względem pewnego ustalonego punktu. Kolejna klatka animacji będzie wyświetlana w momencie jej wygenerowania.
7. Aplikacja będzie wykorzystywała rzut perspektywiczny.
8. Zostaną stworzone dwa rodzaje obiektów mogących znajdować się na scenie - trójkąty i sfery.
9. Zostanie stworzona specjalna klasa definiująca punktowe źródła światła.
10. Wykorzystywanym modelem oświetlenia będzie *Model Phonga*.

11. Aplikacja ma uwzględniać promienie odbite od powierzchni, przechodzące przez nie i takie wysyłane w kierunku źródeł światła w celu określenia czy dany punkt znajduje się w cieniu.

12. Generowanie cieni będzie opcjonalne.

13. Aplikacja będzie dawała możliwość zdefiniowania własnej sceny poprzez plik wejściowy.

14. Aplikacja będzie dawała możliwość wczytywania modeli zdefiniowanych w formacie *obj*

15. Graficzny interfejs użytkownika powinien udostępniać informacje dotyczące poszczególnych węzłów klastra i parametrów związanych z generowanym obrazem.

16. Zostanie zaimplementowane drzewo BSP w celu badania w jakim stopniu może ono przyspieszyć obliczenia. Płaszczyzny dzielące będą wybierane spośród tych wyznaczanych przez trójkąty znajdujące się na scenie (takie, które zawierają w sobie wielokąt, takie które są do nich prostopadłe i styczne do krawędzi).

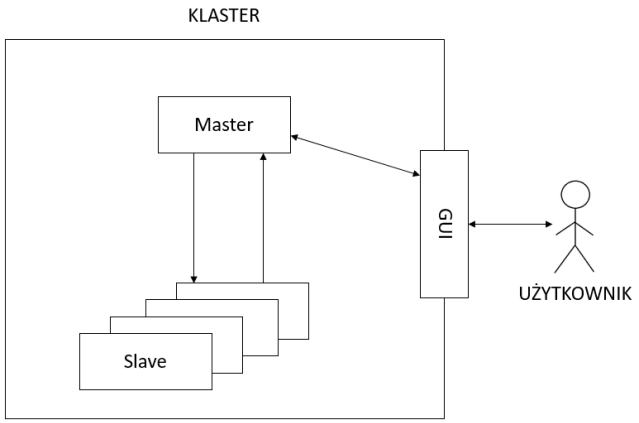
17. Funkcja oceny płaszczyzny dzielącej będzie wybierała taką płaszczyznę, która dzieli zbiór obiektów sceny na jak najrówniejsze podzbiory.

18. Algorytm dzielenia przestrzeni w drzewie BSP działa tak długo, aż dalszy podział stanie się niemożliwy.

19. Musi istnieć możliwość sterowania takimi parametrami jak: głębokość drzewa metody śledzenia promieni, rozmiary generowanego obrazu, ziarnistość zadania, sposób zrównoleglenia itp.

20. Korzystanie z drzewa BSP będzie opcjonalne - w przypadku, w którym nie będzie ono używane, algorytm będzie dokonywał przeglądu zupełnego obiektów (zarówno w sytuacji poszukiwania najbliższego obiektu, jak i ustalania czy dany punkt znajduje się w cieniu).

4.1 Projekt klastra



Rysunek 4.1: Schemat systemu

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.

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 sceny (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. 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.

Algorithm 3 Działanie *master'a*

```
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
```

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.

Algorithm 4 Działanie *slave'a*

```
recvScene()
recvCamera()

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

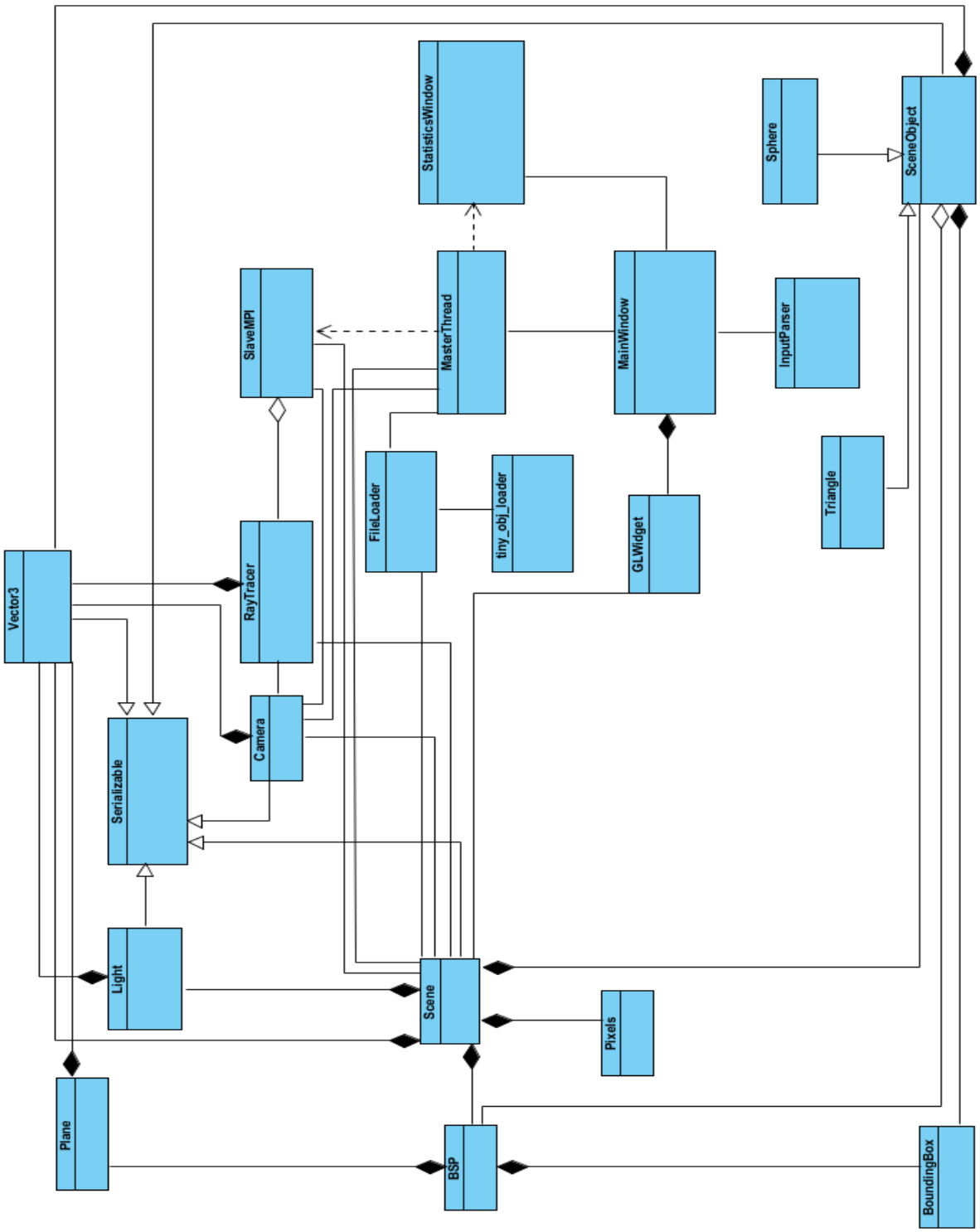
4.1.3 Definicja zadania i rodzaje komunikatów

TODO...

4.2 Projekt programu

4.2.1 Diagram klas

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



Rysunek 4.2: Diagram klas

4.2.2 Opis klas

BoundingBox

Tabela 4.1: BoundingBox

BoundingBox
+minX : float +maxX : float +minY : float +maxY : float +minZ : float +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* -polygons : SceneObject* -box : BoundingBox
+build(root : node*, polygons : SceneObject*, depth : int) -getBestPlane(polygons : list<SceneObject*>) : Plane +getClosest(cross : Vector3, start : Vector3, dir : Vector3) : SceneObject* +isInShadow(cross : Vector3, dir : Vector3, light : Vector3) : bool -getBoundingBox(polygons : list<SceneObject*>) : BoundingBox -intersect(root : node*, cross : Vector3, start : Vector3, dir : Vector3) : SceneObject* -deleteTree(root : node*) : void

Tabela 4.3: Node

Node
partitionPlane : Plane 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 -Far : float -pixWidth : int -pixHeight : int -povy : float -aspect : float -worldWidth : float -worldHeight : float -R : float -ver : float -hor : float -instance : Camera* -eye : Vector3¡float¿* -look : Vector3¡float¿* -up : Vector3¡float¿* -lookAt : Vector3¡float¿*
+setUp(pixWidth : int, pixHeight : int) : void +getInstance() : Camera * +getWorldPosOfPixel(x : int, y : int) : Vector3¡float¿ +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 #resizeGL(int w, int h) : void #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 -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.

InputParser

Tabela 4.7: InputParser

InputParser
-tokens : vector<string>
+getCmdOption(option : string const) : string const) +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* -amb : Vector3* -dif : Vector3* -spec : Vector3*

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* -statisticWindow : StatisticsWindow* -masterThread : MasterThread* -statusLabel : QLabel*
-createMaster() -ShowStats()

-setSpeed(double time) -on_actionStatistics_triggered() -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 -camera : Camera* -scene : Scene* -processSpeed : double** -worldSize : int -status : MPI_Status -names : vector<string> -pending : int -numChunks : int -queue : queue<Chunk> -test : int
run() -splitToChunks(int num) -clearQueue(queue<Chunk> q) -sendCameraBcast() -sendCameraPointToPoint() -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() +getNumOfChunks() : int +workIsReady() : void +setTime(double time) : void +processInfo(double **speeds) : void +close() : void +setName(int num, QString name) : void

Klasa MasterThread jest główną klasą 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 stopx : int starty : int 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* +x : int +y : int +startx : int +starty : int
+setStartXY(x : int, y : int) : void +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 +b : float +c : float +d : float
+classifyObject(obj : SceneObject*) : int +classifyPoint(point : Vector3*) : int +getDistToPoint(point : Vector3*) : float +rayIntersectPlane(start : Vector3, dir : Vector3) : bool +getNormal() : Vector3 +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* -scene : Scene*
+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 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 -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

Obiekt klasy Scene zawiera definicję całej sceny. Jest on napisany według wzorca Singleton. 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 #transparency : float #mirror : float #local : float #density : float #amb : Vector3* #dif : Vector3* #spec : Vector3*
+getLocalColor(normal : Vector3, cross : Vector3, observation : Vector3) : Vector3 +trace(cross : Vector3, start : Vector3, dir : Vector3, dist : float) : bool +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 +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 rodziale 6.

SlaveMPI

Tabela 4.18: 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.

StatisticsWindow

Tabela 4.19: StatisticsWindow

StatisticsWindow
-Ui::StatisticsWindow *ui; -int worldSize;
+resizeEvent(QResizeEvent *event) : void +setTime(double time) : void +setChunks(int i) : void +setXY(int x, int y) : void +setObj(int i) : void +setLights(int i) : void +setProccessName(int num, QString str) : void +setProccessSpeed(double **speed) : void -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 -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 takie jak *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));
    }
}
}
```

Listing 5.1: Fragment klasy *RayTracer*

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 wysyłany 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();

}
```

Listing 5.2: Fragment klasy *RayTracer*

Powyższa funkcja 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 stwierdzenie, 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 drzewa rekurencji) 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 kierunkiem 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;
        }
    }
```

Listing 5.3: Fragment klasy *BSP* - budowa drzewa

Powyższy fragment 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 kolei jeżeli płaszczyzna go przecina, to jest on dzielony na dwa (w sytuacji powstania trójkąta i czworokąta, czworokąt jest dzielony na dwa trójkąty) - każda z połówek trafi do odpowiedniego „dziecka". Program uwzględnia sfery przechowywane w postaci równania (a więc prosty podział takiego obiektu nie jest możliwy). W momencie, w którym płaszczyzna przetnie 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 osiągnięta maksymalna głębokość drzewa (określana przez parametr przekazywany do funkcji przy pierwszym wywołaniu), lub w sytuacji, w której dalszy podział jest niemożliwy. Wtedy wierzchołek jest zamieniany w liść (wskaźniki na potomstwo są puste), a wejściowa lista obiektów zostaje do niego przypisana. Nie biorąc pod uwagę sfer 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;
}
```

Listing 5.4: Fragment klasy *BSP* - przeglądanie drzewa

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 wielokątem - 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 ona żadnego obiektu, a promień przecina płaszczyznę 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 workIsReady();
    camera->rotate();
    sendCameraPointToPoint();
}
}
```

Listing 5.5: Fragment klasy *MasterThread*

Powyższy kod przedstawia główną pętlę programu węzła zarządzającego. Zanim zostanie ona wywołana, *MasterThread* rozsyła informacje o wczytanej scenie, kamerze i głębokości drzewa śledzenia promieni do każdego z węzłów wykorzystując komunikację typu *broadcast* (dzieje się to w konstruktorze klasy).

Zgodnie z założeniami, rozpoczyna się ona od podziału generowanego obrazu na fragmenty, których definicje trafiają na kolejkę oczekujących zadań. Następnie algorytm zdejmuje zadania z kolejki i wysyła je do kolejnych węzłów wykonawczych (zostaje tutaj ustalona liczba zadań aktualnie wykonywanych). W momencie, w którym master otrzyma wyniki działań (tablicę pikseli) któregokolwiek węzła, następuje sprawdzenie, czy istnieją jeszcze jakieś zadania do wykonania. Jeżeli tak, to jedno z nich zostanie wysłane do węzła, od którego otrzymaliśmy właśnie fragment obrazu, jeżeli nie, to liczba aktualnie wykonywanych zadań zmniejsza się.

W chwili, w której liczba wykonywanych zadań spadnie do zera, zostaje wysłany sygnał do wątku GUI informujący go o tym, że generowana klatka animacji jest gotowa. Zostaje również uaktualniona pozycja kamery, która następnie jest wysyłana do każdego z węzłów wykonawczych.

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;
}
```

Listing 5.6: Fragment klasy *SlaveMPI*

Powyższa metoda pokazuje, jak działa pętla główna węzłów wykonawczych. Zanim zostanie ona uruchomiona, wszelkie niezbędne informacje dot. sceny zostają odebrane przez każdy z węzłów (dzieje się to w konstruktorze).

Zgodnie z opisem w rozdziale (!ROZDZIAŁ!) program czeka na polecenia mogące nadejść od węzła zarządzającego - może być to żądanie zakończenia programu, definicja fragmentu obrazu, który należy wyznaczyć, czy definicja kamery (jest ona aktualizowana co klatkę animacji). Do wyznaczenia tablicy pikseli program wykorzystuje klasę *RayTracer*, która dokładniej została opisana wyżej.

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
```

Listing 5.7: Interfejs *Serializable*

Kluczowym elementem programu jest potrzeba wysyłania obiektów pomiędzy węzłami. W tym celu potrzebny jest zestaw metod umożliwiających ich serializację i deserializację do strumienia bajtów (tak aby było możliwe wysyłanie obiektów funkcjami udostępnianymi przez *MPI*). Jest on zdefiniowany poprzez interfejs *Serializable*, który wymusza w klasach dziedziczących zdefiniowanie metod *serialize* (serializacja obiektu do wektora bajtów), *deserialize* (deserializacja obiektu ze strumienia bajtów) i *getType* (metoda umożliwiająca ustalenie z jakim typem obiektu mamy do czynienia).

Rozdział 6

Opis funkcjonalny

6.1 Uruchomienie programu

Do uruchomienia aplikacji niezbędne jest zainstalowanie biblioteki *MPICH* na każdym komputerze będącym elementem klastra. Prawidłowe działanie programu (w przypadku, w którym chcemy, aby działał on na kilku maszynach) wymaga skonfigurowania połączenia *ssh* , ponieważ ta implementacja standardu *MPI* korzysta z niego jako ze sposobu komunikacji. Zarówno bibliotekę jak i instrukcje konfiguracji można znaleźć na stronie producenta [10]. Program uruchamia się z linii komend w następujący sposób:

```
mpiexec [−n liczba procesów] −H [adresy komputerów w klastrze oddzielone przecinkami]
./ParallelRayTracing [−s] [−b] [−f nazwa pliku wejściowego] [−w szer. w pikselach]
[−h wys. w pikselach] [−c liczba fragmentów] [−d głębokość drzewa] [−t liczba testów]
```

Komenda *mpiexec* jest zdefiniowana w standardzie *MPI* i posiada ona znacznie więcej argumentów opcjonalnych - te wymienione stanowią niezbędne minimum. Liczba procesów musi być ustawiona na więcej niż jeden. Opcja *-H* nie jest konieczna w przypadku uruchamiania aplikacji na jednym komputerze. Jako ostatni parametr komendy *mpiexec* podaje się nazwę programu do uruchomienia wraz z jego parametrami, które zostały niżej omówione.

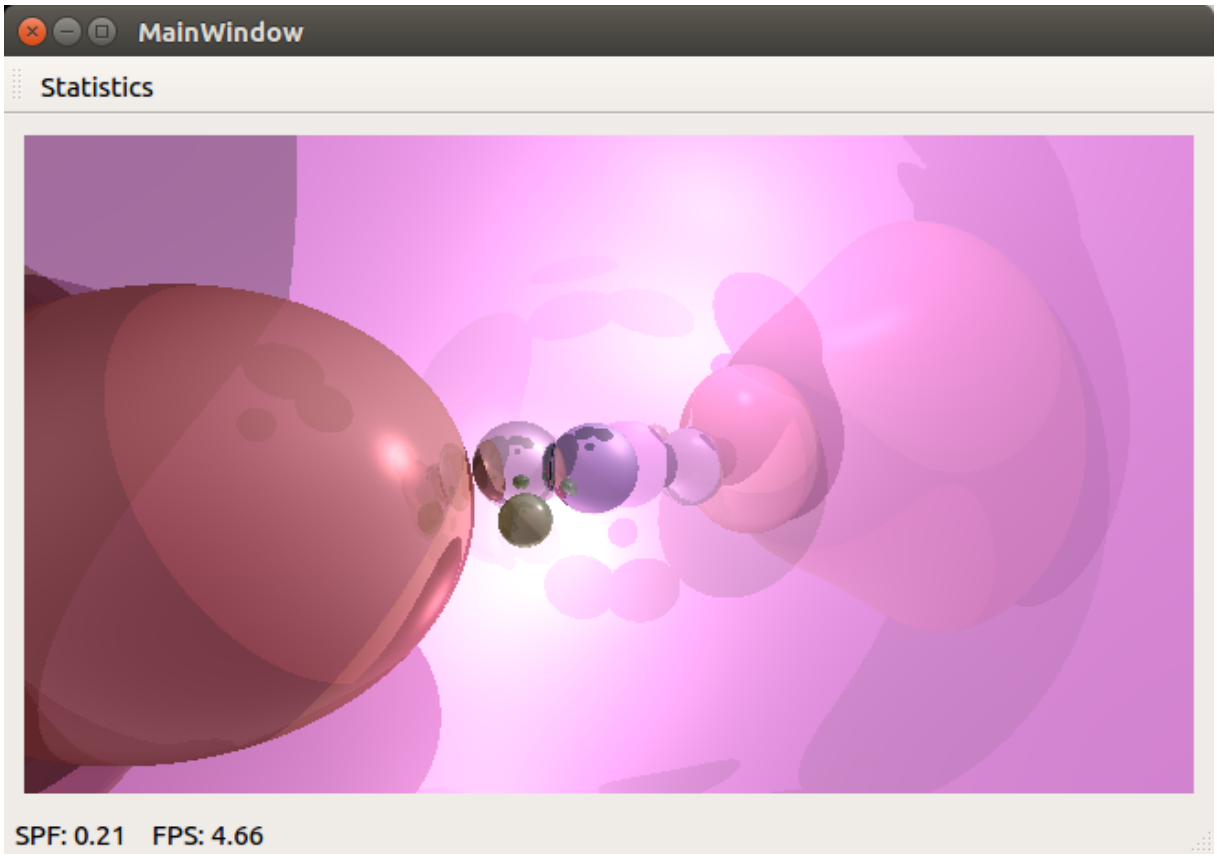
- **-s** - obecność tej flagi włącza śledzenie promieni od punktów przecięcia do źródeł światła - odpowiada ze efekt cieni,
- **-b** - obecność tej flagi sprawia, że algorytm korzysta z drzewa BSP,
- **-f** - po tej fladze należy podać nazwę pliku z definicją sceny (sposób opisu został podany niżej). Jeżeli nie zostanie ona zdefiniowana program domyślnie szuka pliku „scene.txt”,
- **-w** - szerokość generowanego obrazu w pikselach. Domyślną wartością jest 700,
- **-h** - wysokość generowanego obrazu w pikselach. Domyślną wartością jest 500,
- **-c** - liczba fragmentów (zadań) podniesiona do kwadratu, na które zostanie podzielony obraz,
- **-d** - głębokość drzewa śledzenia promieni. Domyślna wartość to 3,

- **-t** - liczba testów jakie mają wykonać się w ramach uruchomienia programu. Domyślna wartość to 0 - w takiej sytuacji program będzie dział bez końca i nie zwróci wyników działania na konsolę. Jeżeli parametr zostanie ustawiony na wyższą wartość to program zakończy swoje działanie po wygenerowaniu zadanej liczby klatek - czas odpowiedzi danego węzła, jak i czas generowania klatki zostanie uśredniony.

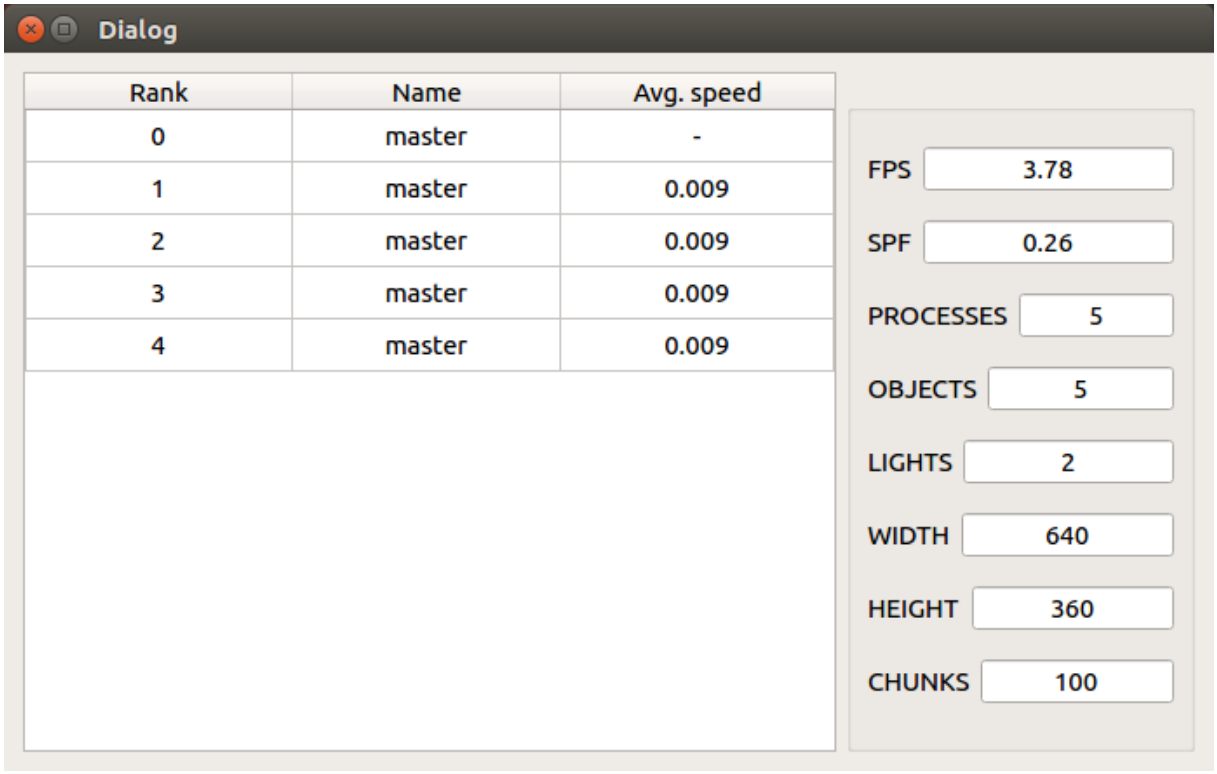
```
mpiexec -n 5 ./ParallelRayTracing -s -f spheres.txt -w 640 -h 360
```

.

Po uruchomieniu programu na ekranie pojawi się okno widoczne na obrazie (!numer obrazka!). W centralnej części interfejsu widzimy podgląd generowanej animacji - obraz obraca się według punktu zdefiniowanego w pliku wejściowym co każdą klatkę animacji (opis pliku wejściowego znajduje się niżej). W dolnej części ekranu widzimy podstawowe informacje związane z czasem generowania obrazu: *SPF* (ang. *second per frame* - czas generowania klatki) i *FPS* (ang. *frames per seconds* - liczba klatek na sekundę). W górnej części interfejsu znajduje klawisz, który otwiera okno z dodatkowymi statystykami (!numer obrazka!). Tabela widoczna w lewej części dodatkowego interfejsu przedstawia średni czas, jaki danemu węzłowi zajmuje liczenie zadania (generowanie fragmentu obrazu). Czas podawany jest w sekundach. W przykładzie pokazanym na obrazku wszystkie procesy liczące znajdują się na komputerze *master*. W prawej części okna podano dodatkowe parametry związane ze sceną.



Rysunek 6.1: GUI



Rysunek 6.2: GUI

6.2 Opis pliku wejściowego

Plik wejściowy składa się z szeregu linii definiujących zadane obiekty. Dwa z nich są obowiązkowe i mogą wystąpić w pliku tylko raz (w przypadku podania więcej niż jednej definicji program skorzysta z ostatniej) - *Scene* i *Camera*. Niżej opisano jakie typu obiektów użytkownik może definiować oraz podano przykładowy plik wejściowy. Jeżeli pierwsze słowo w linii (słowo kluczowe) nie zostanie rozpoznane to zostanie on zignorowana. Jeżeli w którymś z parametrów znajduje się błąd obiekt również nie będzie wczytany, ale użytkownik dostanie informację zwrotną o tym, w którym miejscu w pliku pojawił się problem. Parametry są podawane wg. słów kluczowych - ich wartości zawierają się w nawiasach trójkątnych. Jeśli definiowany parametr jest wektorem, kolejne wartości zostają oddzielone przecinkami.

- *scene* - zawiera parametry światła globalnego i kolor tła.
    1. *background* - wektor definiujący kolor tła podany w modelu *RGB*. Wartości powinny zawierać się w przedziale od 0 do 1,
    2. *global* - wektor określający natężenie światła globalnego (patrz rozdział (RODZIAŁ), *Model Phonga*. Wartości powinny zawierać się w przedziale od 0 do 1,
- *camera* - definiuje kamerę (obserwatora).
    1. *eye* - wektor określający pozycję kamery (oka obserwatora),
    2. *lookAt* - wektor określający punkt, na który patrzy obserwator (to wokół niego będzie się on obracał),

3. *zNear* - wartość skalarna określająca odległość obserwatora od rzutni (patrz (!rozdział!)
4. *zFar* - wartość skalarna określająca maksymalną odległość, na jaką widzi obserwator (patrz (!ROZDZIAŁ!)
5. *povy* - wartość skalarna określająca pionowy kąt widzenia obserwatora (patrz (!ROZDZIAŁ!). Jest ona podawana w stopniach.

- *light* - definiuje właściwości i położenie punktowego źródła światła

1. *pos* - wektor określający pozycję światła.
2. *amb* - wektor określający, w jaki sposób źródło światła wpływa na oświetlenie otoczenia (ang. *ambient light* - patrz (!ROZDZIAŁ!))
3. *dif* - wektor określający natężenie światła rozproszonego (ang. *diffuse light* - patrz (!ROZDZIAŁ!))
4. *spec* - wektor określający natężenie światła kierunkowego (ang. *specular light* - patrz (!ROZDZIAŁ!))

- *triangle*

1. *pointA* - wektor określający położenie jednego z wierzchołków trójkąta.
2. *pointB* - wektor określający położenie jednego z wierzchołków trójkąta.
3. *pointC* - wektor określający położenie jednego z wierzchołków trójkąta.
4. *amb* - wektor określający procentowy wpływ światła otoczenia na powierzchnię (ang. *ambient light* - patrz (!ROZDZIAŁ!)). Wartości powinny zawierać się w przedziale od 0 do 1,
5. *dif* - wektor określający procentowy wpływ światła rozproszonego na powierzchnię (ang. *diffuse light* - patrz (!ROZDZIAŁ!). Wartości powinny zawierać się w przedziale od 0 do 1,
6. *spec* - wektor określający procentowy wpływ światła kierunkowego na powierzchnię (ang. *specular light* - patrz (!ROZDZIAŁ!)). Wartości powinny zawierać się w przedziale od 0 do 1,
7. *specShin* - skalar, który wpływa na wygląd odblasków na powierzchni (przyjmuje wartości od kliku do kilkuset).
8. *trans* - skalar określający procentowy wpływ przezroczystości przedmiotu na kolor powierzchni (w przypadku zera nie jest wysyłany promień przechodzący przez powierzchnię). Wartości zawierać się w przedziale od 0 do 1,
9. *mirror* - skalar określający procentowy wpływ promieni odbitych na ostateczny kolor piksela (w przypadku zera nie jest wysyłany żaden promień odbity).
10. *local* - parametr skalarny określający procentowy wpływ właściwych parametrów powierzchni na kolor piksela.
11. *density* - skalar określający gęstość przedmiotu (współczynniki gęstości są stabelaryzowane i dostępne w Internecie), która wpływa na złamanie światła. Gęstość ośrodka sceny wynosi 1.

- *sphere*

1. *amb* - wektor określający procentowy wpływ światła otoczenia na powierzchnię (ang. *ambient light* - patrz (!ROZDZIAŁ!)). Wartości powinny zawierać się w przedziale od 0 do 1,
2. *dif* - wektor określający procentowy wpływ światła rozproszonego na powierzchnię (ang. *diffuse light* - patrz (!ROZDZIAŁ!). Wartości powinny zawierać się w przedziale od 0 do 1,
3. *spec* - wektor określający procentowy wpływ światła kierunkowego na powierzchnię (ang. *specular light* - patrz (!ROZDZIAŁ!)). Wartości powinny zawierać się w przedziale od 0 do 1,
4. *specShin* - skalar, który wpływa na wygląd odblasków na powierzchni (przyjmuje wartości od kliku do kilkuset).
5. *trans* - skalar określający procentowy wpływ przezroczystości przedmiotu na kolor powierzchni (w przypadku zera nie jest wysyłany promień przechodzący przez powierzchnię).
6. *pos* - wektor określający położenie środka sfery.
7. *radius* - skalar określający promień sfery.
8. *trans* - skalar określający procentowy wpływ przezroczystości przedmiotu na kolor powierzchni (w przypadku zera nie jest wysyłany promień przechodzący przez powierzchnię). Wartości zawierać się w przedziale od 0 do 1,
9. *mirror* - skalar określający procentowy wpływ promieni odbitych na ostateczny kolor piksela (w przypadku zera nie jest wysyłany żaden promień odbity).
10. *local* - parametr skalarny określający procentowy wpływ właściwych parametrów powierzchni na kolor piksela.
11. *density* - skalar określający gęstość przedmiotu (współczynniki gęstości są stabelaryzowane i dostępne w Internecie), która wpływa na złamanie światła. Gęstość ośrodka sceny wynosi 1.

- *obj* - po tym słowie kluczowym należy podać nazwę pliku *obj* (wraz z rozszerzeniem). Program wczyta ten plik konwertując wielokąty w nim zdefiniowane na trójkąty. Pliki *.obj* mogą wskazywać na pliki *.mtl* zawierające definicję materiałów z nimi związanych. Jeżeli taki plik znajduje się w katalogu z programem, to również zostanie on wczytany. W przeciwnym wypadku zostanie zastosowany materiał domyślny.

```
camera eye<0; 10; -14> lookAt<0; 0; 0> zNear<1> zFar<30> povy<76>
scene background<1; 0.5; 0.5> global<0.4; 0.4; 0.4>
triangle pointA<-1; 0; 0> pointB<1; 0; 0> pointC<0; 2; 0> amb<0.7;
    0.3; 0.5> dif<0.6; 0.7; 0.8> spec<0.8; 0.8; 0.8> specShin<33> trans
    <0.8> mirror<0.5> local<0.2> density<1>
light pos<0; 7; -2> amb<0.3; 0.3; 0.3> dif<0.5; 0.5; 0.5> spec<0.9;
    0.8; 0.8>
sphere amb<0.7; 0.7; 0.4> dif<0.6; 0.7; 0.8> spec<0.8; 0.8; 0.8>
    specShin<70> pos<-4; 3; 2> radius<3> trans<1.0> mirror<0.0> local
    <0.1> density<1.5>
obj glass.obj
```

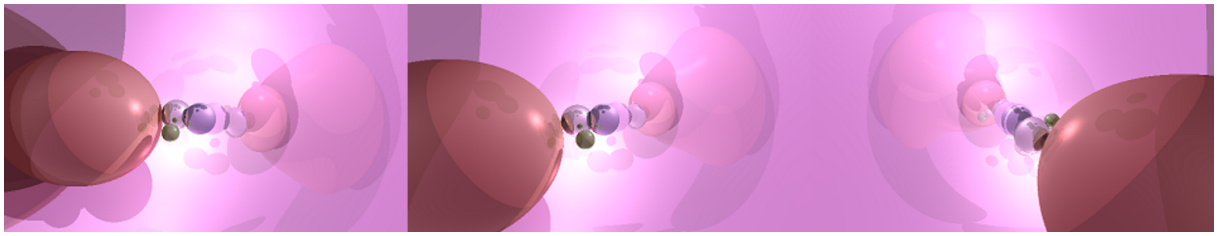
Rozdział 7

Rezultaty

7.1 Testy wydajnościowe i wnioski

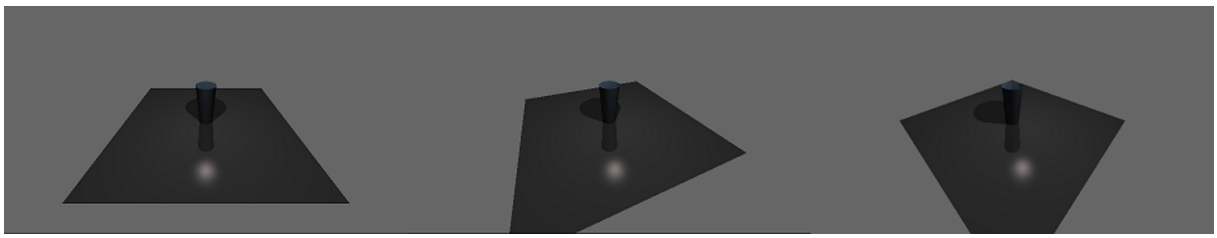
W niniejszym rozdziale zostały zaprezentowane wyniki badań nad efektywnością zaimplementowanego algorytmu. Przedstawiono tutaj zależność czasową między definicją sceny, na podstawie której generowany jest obraz, a zastosowanym podejściem do problemu. Testy były przeprowadzane dla czterech różnych scen (specyficzne ich cechy, jak i ustawienia programu, zostały przedstawione na początku każdego podrozdziału).

Rysunek 7.1: Spheres



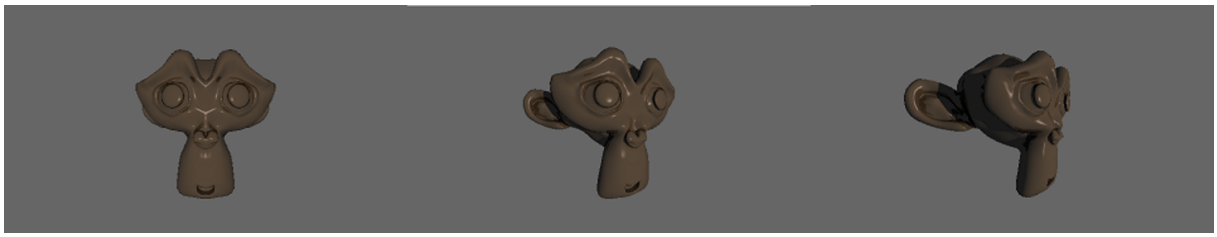
Powyższa scena składa się z pięciu sfer o różnych parametrach powierzchniowych - jedna z nich jest półprzezroczysta, inne odbijają światło w podobny sposób jak lustro. Największa z nich otacza pozostałe, dzięki czemu każdy promień wysłany w scenę trafia zawszę w jakąś kulę.

Rysunek 7.2: Glass, źródło:



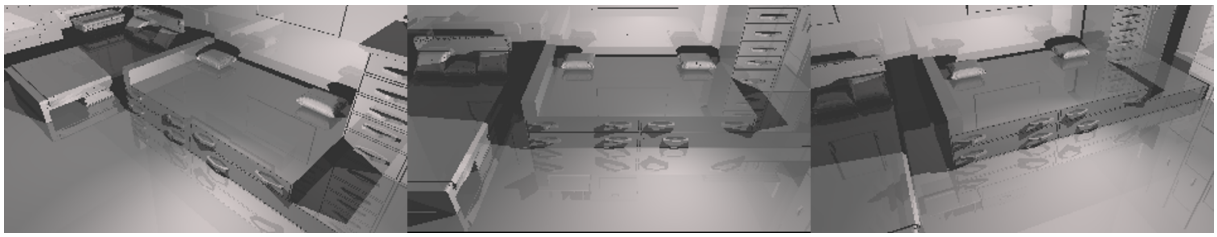
Scena autorstwa (!autor!). Składa się ona z 366 trójkątów. Obserwator ustawiony jest pod takim kątem, aby dobrze widzieć całą scenę. Większość promieni pierwotnych nie trafi w żaden obiekt, a szklanka będąca centralną częścią sceny, nie jest przezroczysta (definicje materiałów nie zawierają informacji o przezroczystości).

Rysunek 7.3: Suzanne, źródło:



Suzanne jest popularnym modelem wykorzystywanym w badaniach dotyczących grafiki komputerowej. Składa się ona z 15488 trójkątów. Definicja materiału została dostarczona przez autorów (MIT) i nie zawiera informacji o przezroczystości. Jak na obiekt takich rozmiarów jest on bardzo skomplikowany.

Rysunek 7.4: Kid, źródło



Scena została pobrana ze strony (strona). Składa się ona z 27946 trójkątów. Autor nie dostarcza definicji materiałów (w założeniu obiekty mają być teksturowane), więc każdy obiekt będzie miał domyślne parametry. To czego nie widać na powyższym obrazku, to brak ściany mającej znajdować się za obserwatorem - w związku z tym większość promieni wtórnych w nic nie trafi.

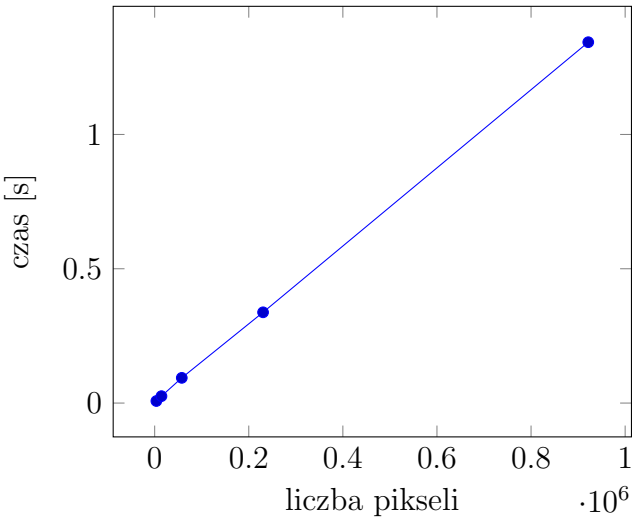
7.1.1 Zależność czasowa od liczby pikseli

W niniejszym podrozdziale pokazano, w jaki sposób czas generowania obrazu zależy od jego rozmiarów. Testy zostały przeprowadzone dla kilku różnych ujęć scen zaprezentowanych wyżej (kilka takich samych ujęć dla różnej liczby pikseli). Obliczenia dotyczące tego punktu nie zostały zrównoleglone. Parametrom niebędącym przedmiotami tego badania zostały arbitralnie przypisane niniejsze wartości:

- liczba świateł - 1
- wyznaczenie cieni - nie
- głębokość drzewa śledzenia promieni - 3
- użycie drzewa BSP - nie
- zrównoleglenie - nie

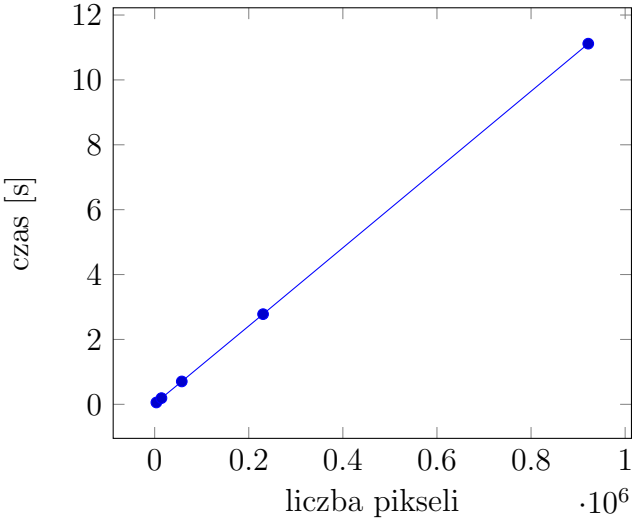
Podstawowym wnioskiem, jaki nasuwa się na podstawie otrzymanych wykresów, jest to, że zależność między czasem generowania obrazu a liczbą pikseli jest liniowa, mimo że czas generowania koloru poszczególnych pikseli może być różny. W momencie, w którym promień nie trafi w żaden obiekt, nie zostanie wysłany żaden promień wtórny, co znacząco

Spheres - zależność od liczby pikseli



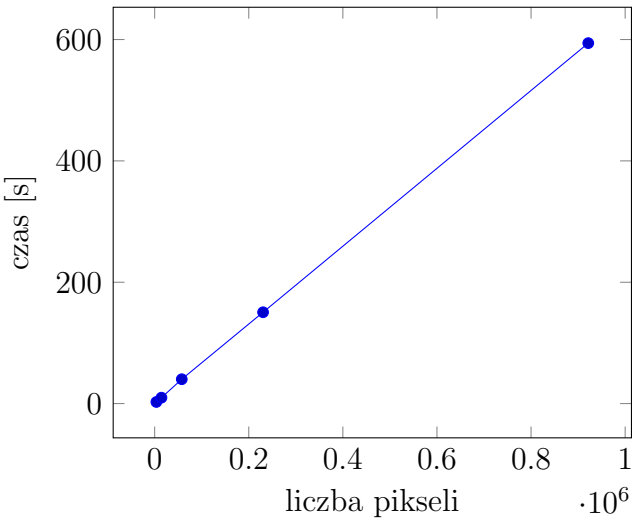
liczba pikseli	spf [s]
3600 (80x45)	0,008
14400 (160x90)	0,026
57600 (320x180)	0,094
230400 (640x360)	0,338
921600 (1280x720)	1,343

Glass - zależność od liczby pikseli



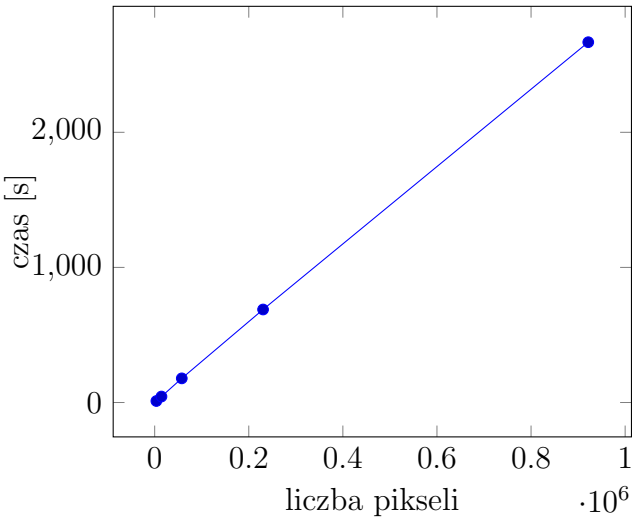
liczba pikseli	spf [s]
3600 (80x45)	0,057
14400 (160x90)	0,191
57600 (320x180)	0,705
230400 (640x360)	2,779
921600 (1280x720)	11,117

Suzanne - zależność od liczby pikseli



liczba pikseli	spf [s]
3600 (80x45)	2,678
14400 (160x90)	9,739
57600 (320x180)	40,162
230400 (640x360)	150,539
921600 (1280x720)	594,091

Kid - zależność od liczby pikseli



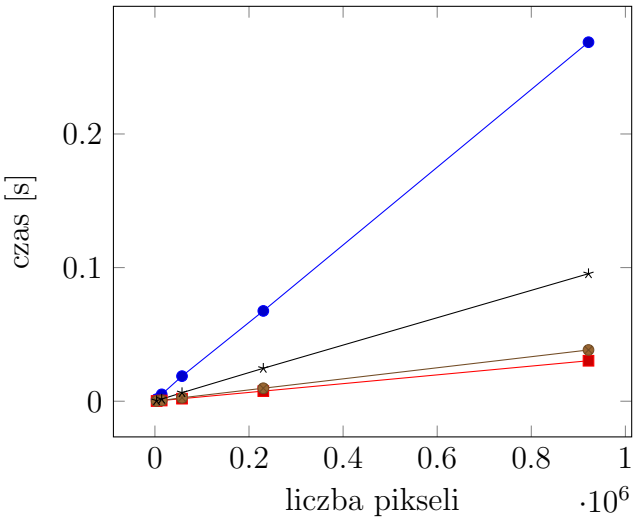
liczba pikseli	spf [s]
3600 (80x45)	11,074
14400 (160x90)	44,164
57600 (320x180)	178,688
230400 (640x360)	688,478
921600 (1280x720	2666,950

skraca czas działania algorytmu. Gdy zwiększamy rozmiar obrazu liczba promieni, które trafią w obiekt i takich, które nie trafią nic, rośnie proporcjonalnie do skali powiększenia.

Wiedząc jak zbudowane są sceny, należy zwrócić uwagę na wzrost czasu generowania obrazu w zależności od liczby obiektów. Wykres (!wykres ze stosunkami!) pokazuje, że sceny *Glass* i *Suzzane*, które są podobne do siebie ze względu na budowę (zbudowane są wyłącznie z trójkątów, a generowany obraz jest obrazem zamkniętym - żaden z trójkątów nie wychodzi poza kadr), mają zbliżony stosunek liczby obiektów do czasu generowania klatki. Oznacza to, że ze względu na swoje podobieństwo wzrost liczby obiektów powoduje proporcjonalny wzrost czasu generowania obrazu - jak pokazują pozostałe proste, nie jest to jednak regułą. Dla sceny *Spheres* stosunek liczby obiektów do czasu generowania jest najmniej opłacalny. Jest to spowodowane równomiernym rozrostem drzewa śledzenia promieni - każdy promień trafia w jakiś obiekt, przez co liczba promieni wtórnych rośnie równomiernie. Podobna sytuacja ma miejsce w przypadku sceny *Kid*, jednak tutaj rozrost drzewa jest hamowany przez brak ściany znajdującej się za obserwatorem (część promieni trafia w przestrzeń, co powoduje zmniejszenie liczby promienie wtórnych).

Widzimy, że wzrost czasu generowania obrazu jest w oczywisty sposób zależny od liczby obiektów znajdujących się na scenie, jednak w dużej mierze zależy on również od ich wzajemnego położenia i od położenia obserwatora na scenie (umiejscowienie obserwatora również wpływa na rozrost drzewa, ponieważ to od niego wysyłane są promienie pierwotne). W związku z powyższym przewidzenie ile czasu zajmie generowanie obrazu jest zadaniem trudnym, ale nie niemożliwym - znając wszystkie parametry sceny i algorytmu możemy przewidzieć wersję wydarzeń, w której każdy z promieni napotka na przeszkodę - pesymistyczna złożoność obliczeniowa proponowanego algorytmu (uwzględniającego cienie, dwa rodzaje promieni wtórnych - odbite i załamane) przedstawia się funkcją $T(d, n, o) = (2^d - 1) * o^2 * n$ gdzie:
d - maksymalna głębokość drzewa
o - liczba obiektów
n - liczba promieni pierwotnych

Kid - zależność od liczby pikseli

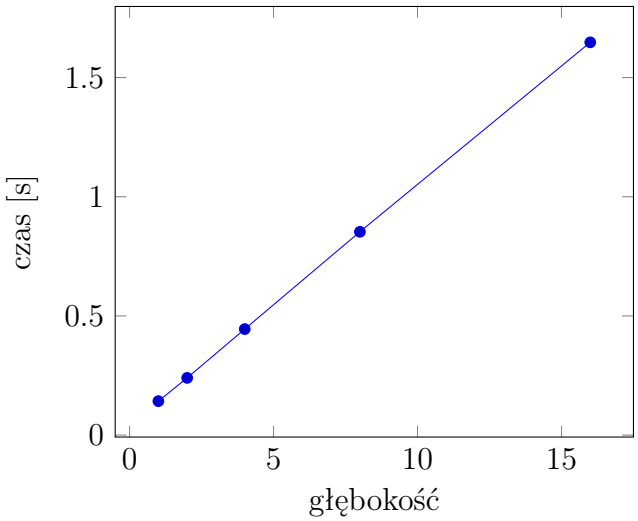


7.1.2 Zależność czasowa od głębokości drzewa

W niniejszym podrozdziale zaprezentowano, w jaki sposób zależy czas generowania obrazu od głębokości drzewa śledzenia promieni. Testy zostały przeprowadzone dla kilku różnych ujęć scen zaprezentowanych wyżej (kilka takich samych ujęć dla różnej głębokości drzewa). Obliczenia dotyczące tego punktu nie zostały zrównoleglone. Parametrom niebędącym przedmiotami tego badania zostały arbitralnie przypisane niniejsze wartości:

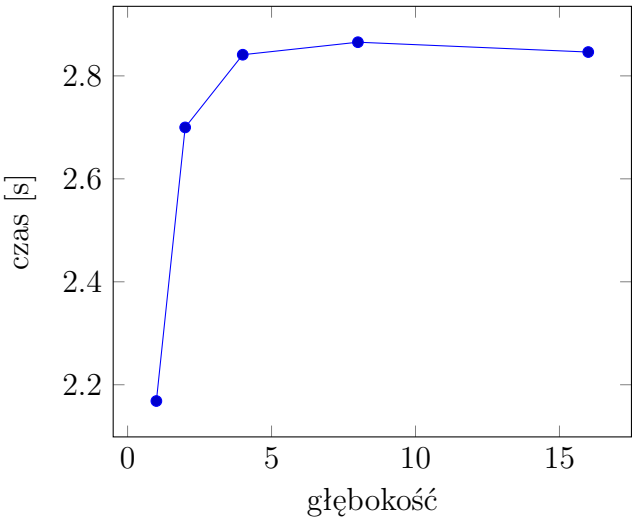
- liczba świateł - 1
- cieniowanie - nie
- liczba pikseli - 640x360 (230400)
- użycie drzewa BSP - nie
- zrównoleglenie - nie

Spheres - zależność od głębokości drzewa



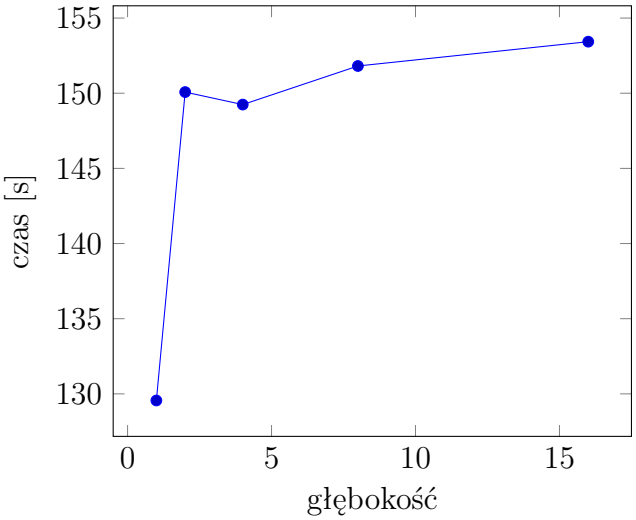
głębokość	spf [s]
1	0,142
2	0,240
4	0,445
8	0,853
16	1,648

Glass - zależność od głębokości drzewa



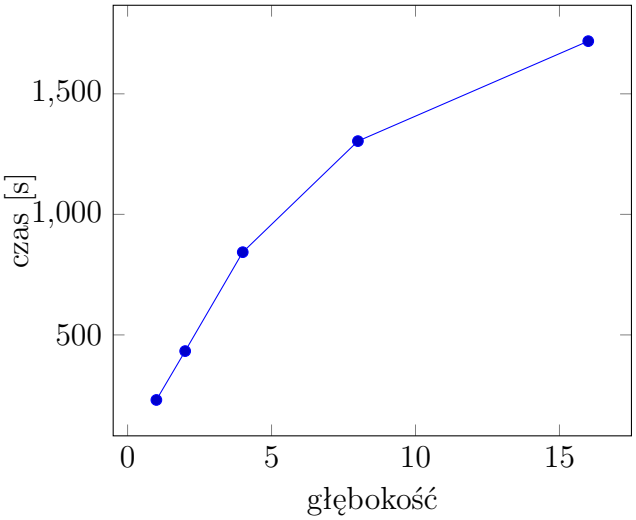
głębokość	spf [s]
1	2,168
2	2,670
4	2,841
8	2,865
16	2,846

Suzanne - zależność od głębokości drzewa



głębokość	spf [s]
1	129,559
2	159,077
4	149,244
8	151,812
16	153,429

Kid - zależność od głębokości drzewa



głębokość	spf [s]
1	230,193
2	432,880
4	843,196
8	1303,880
16	1718,510

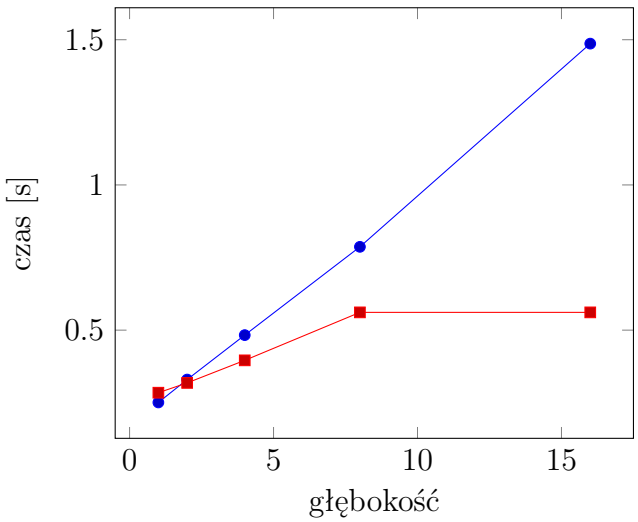
Na powyższych wykresach widzimy, że wzrost czasu w zależności od głębokości jest logarytmiczny - jest to spowodowane spadkiem liczby promieni na każdym kolejnym poziomie drzewa (promienie trafiają w próżnię, więc nie są generowane promienie wtórne). Najbardziej wyróżnia się zależność dla sceny *Spheres*, która sprawia wrażenie liniowej. Jest to spowodowane tym, że na tej scenie każdy promień trafia w jakiś obiekt. Teoretycznie funkcje mogłyby rosnąć wykładniczo, ponieważ algorytm dopuszcza generowanie dwóch promieni wtórnych z danego punktu - dzieje się tak, gdy powierzchnia obiektu jest półprzezroczysta - zostaje wysłany promień odbity i promień załamany. W definicji sceny *Spheres* występuje jeden taki obiekt, jednak (jak widać na wykresie) promienie trafiają w niego na tyle rzadko (związane jest to z jego rozmiarami i umiejscowieniem), że nie widzimy tego na wykresie. W przypadku sceny *Kid* każdy obiekt jest w pewnym stopniu przezroczysty, jednak brak ściany pomieszczenia za obserwatorem i tak powoduje spadek promieni wtórnych (jednak jest on wolniejszy niż w przypadku scen *Glass* i *Suzanne*).

7.1.3 Zależność czasowa od światła

W niniejszym podrozdziale zaprezentowano, w jaki sposób zależy czas generowania obrazu od głębokości drzewa śledzenia promieni. Testy zostały przeprowadzone dla kilku różnych ujęć scen zaprezentowanych wyżej, na których losowo rozrzucono zadaną liczbę źródeł światła. Obliczenia dotyczące tego punktu nie zostały zrównoleglone. Parametrom niebędącym przedmiotami tego badania zostały arbitralnie przypisane poniższe wartości:

- głębokość drzewa śledzenia promieni - 3
- liczba pikseli - 640x360 (230400)
- użycie drzewa BSP - nie
- zrównoleglenie - nie

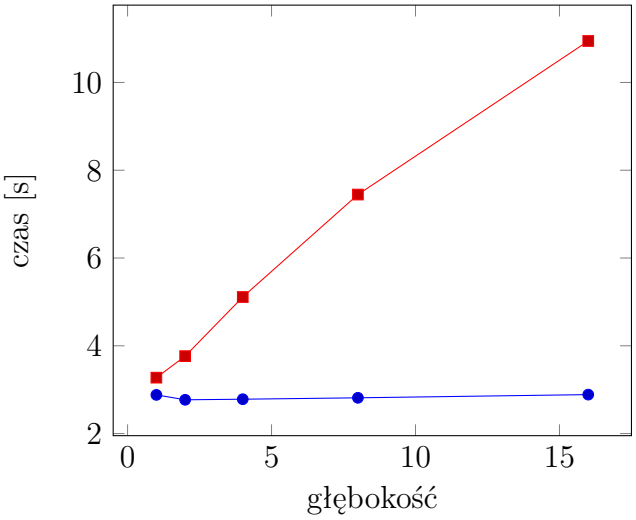
Spheres - zależność od światła



liczba świateł	spf [s]	
	bez cieni	z cieniami
1	0,251	0,285
2	0,330	0,318
4	0,483	0,396
8	0,787	0,561
16	1,487	1,202

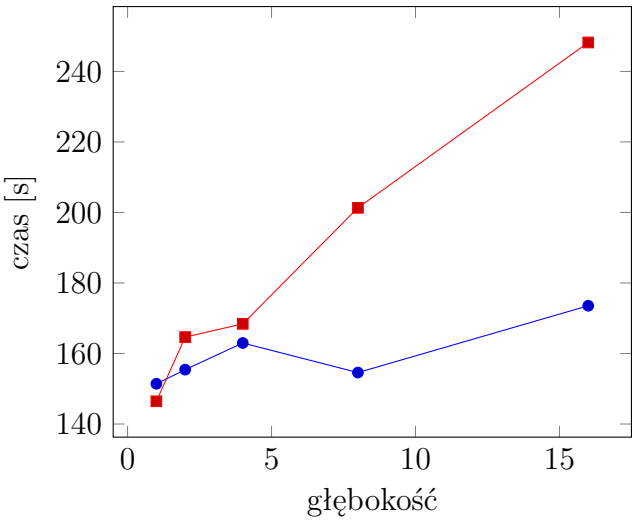
Z powyższych wykresów wynika, że wpływ liczby źródeł światła na czas generowania obrazu jest znaczny, zwłaszcza w przypadku, w którym śledzimy promień od punktu przecięcia promienia z obiektem do źródła światła (w celu sprawdzenia czy punkt znajduje

Glass - zależność od światła



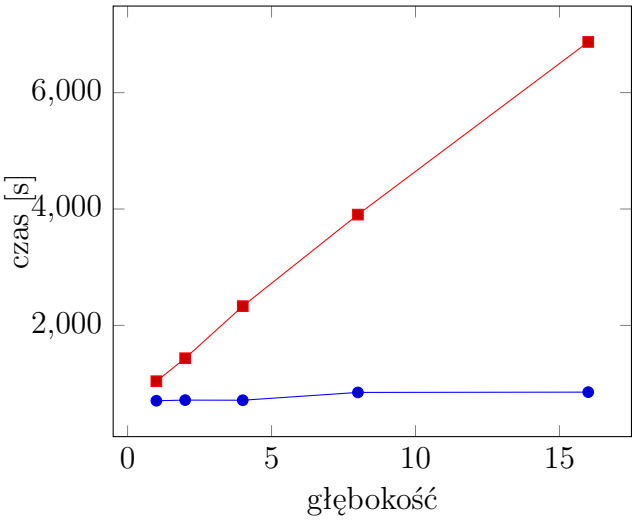
liczba świateł	spf [s]	
	bez cieni	z cieniami
1	2,881	3,274
2	2,770	3,764
4	2,783	5,011
8	2,815	7,446
16	2,888	10,944

Suzanne - zależność od światła



liczba świateł	spf [s]	
	bez cieni	z cieniami
1	151,408	146,452
2	155,419	164,650
4	162,980	168,411
8	154,590	201,321
16	173,540	248,212

Kid - zależność od światła



liczba świateł	spf [s]	
	bez cieni	z cieniami
1	705,337	1040,560
2	716,45	1435,520
4	714,296	2330,470
8	848,262	3904,240
16	853,408	6871,040

się w cieniu). Uwzględnienie cieni wymaga przeglądu wszystkich obiektów tak długo, aż nie znajdzie się jakiś między punktem a źródłem światła. W najgorszym przypadku, czyli takim, w którym nic nie rzuca cienia, musimy przejrzeć wszystkie obiekty. W przypadku skomplikowanych scen znacznie wydłuża to czas generowania obrazu (zwłaszcza, że należy to zrobić dla każdego ze źródeł światła - stąd liniowy wzrost czasu generowania obrazu). Wyjątek stanowi scena *Spheres* zawierająca niewielką liczbę obiektów. W sytuacji, w której jakiś obiekt znajduje się w cieniu, nie musimy dla danego punktu wyliczać koloru korzystając z *Modelu Phonha* ¡!ROZDZIAŁ!¿ - liczy się tylko światło otoczenia. W związku z tym przy niewielkiej liczbie obiektów koszt jego szukania jest niewielki. Oszczędzamy czas procesora dzięki temu, że nie musimy korzystać ze skomplikowanego modelu światła - paradoksalnie bardziej realistyczny obraz generuje się szybciej.

7.1.4 Zależność czasowa od zrównoleglenia

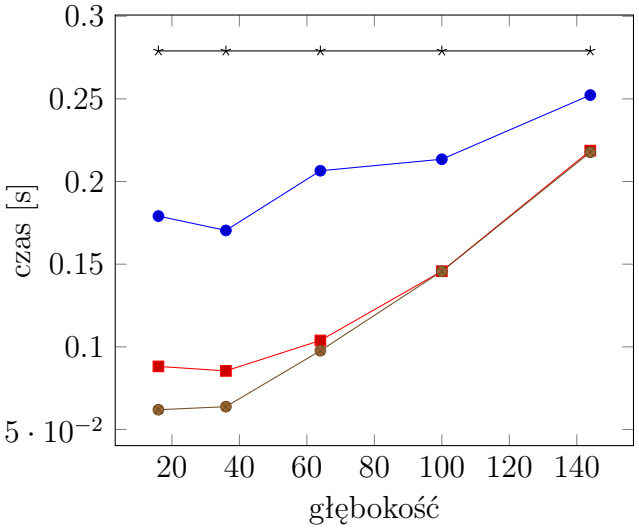
W niniejszym podrozdziale pokazano, w jakim stopniu zrównoleglenie pozwala na przyspieszenie generowania obrazu. Testy zostały przeprowadzone dla pięciu różnych ziarnistości podziału obrazu (ziarnistość zadania) i dla trzech różnych ilości zaangażowanych procesorów. Testy dla pięciu procesorów zostały przeprowadzone na jednej maszynie, dla dziesięciu na dwóch i dla piętnastu na trzech (wartość podana w tabelach uwzględnia tylko węzły wykonawcze). Każdy z komputerów był maszyną wirtualną, której udostępniono 6 procesorów wirtualnych. Połączenia pomiędzy komputerami były realizowane poprzez technologię *Fast Ethernet*. W dolnej części tabel z wynikami podano ilokrotnie maksymalnie przyspieszyły obliczenia względem podejścia niezrównoleglonego (stosunek czasu obliczeń algorytmu wykonującego się na jednym procesorze do minimalnego czasy otrzymanego w teście). Czas generowania obrazu z wykorzystaniem pojedynczego procesora został wyliczony na nowo i nie ma nic wspólnego z czasami podanymi uprzednio. Powodem takiego podejścia jest to, że testy omówione wyżej były wykonywane na fizycznej maszynie o innej specyfikacji. Parametrom niebędącym przedmiotami tego badania zostały arbitralnie przypisane niniejsze wartości:

- liczba świateł - 1
- cieniowanie - tak
- głębokość drzewa śledzenia promieni - 3
- użycie drzewa BSP - nie

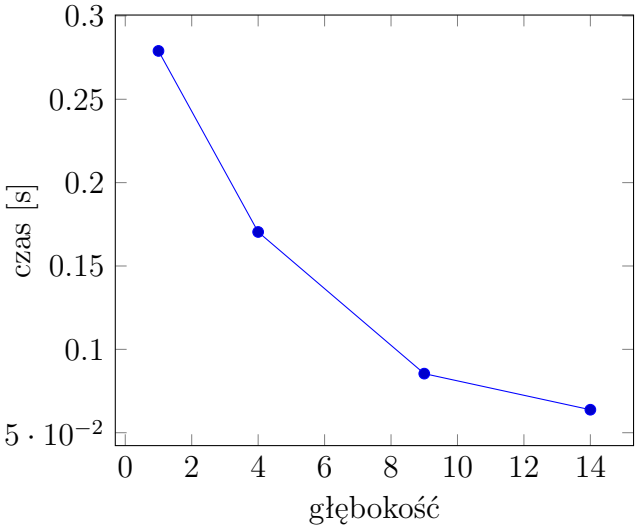
Najważniejszym wnioskiem płynącym z powyższych badań jest to, że wraz ze wzrostem liczby węzłów czas generowania maleje logarytmicznie. Oznacza to, że w pewnym momencie zwiększania rozmiarów klastra, nie uzyskamy żadnego przyspieszenia obliczeń. Jest to wniosek zgodny z oczekiwaniami - maksymalna liczba komputerów, której moc obliczeniową można by wykorzystać, nie będzie większa od liczby pikseli obrazu (przynajmniej dla tej metody zrównoleglenia). Górne ograniczenie rozmiarów klastra (dla zadanej wielkości obrazu) jest oczywiście niższe ze względu na czas potrzebny do przesłania informacji pomiędzy węzłami.

Zastanówmy się teraz nad optymalną ziarnistością zadania (liczbą wycinków obrazu). Najciekawiej prezentuje się wykres dotyczący sceny *Spheres* - wraz ze wzrostem liczby wycinków, drastycznie spada wydajność klastra. Dzieje się tak dlatego, ponieważ więcej czasu zajmuje przesyłanie danych pomiędzy węzłami niż rzeczywiste obliczenia. Na pozostałych wykresach liczba wycinków zdaje się nie mieć takiego znaczenia, jednak jest to

Spheres - zależność od zrównoleglenia (640x360)

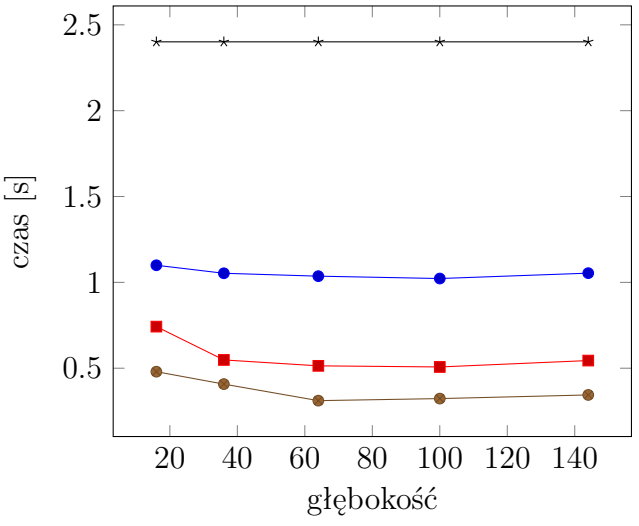


Spheres - zależność od zrównoleglenia (640x360)

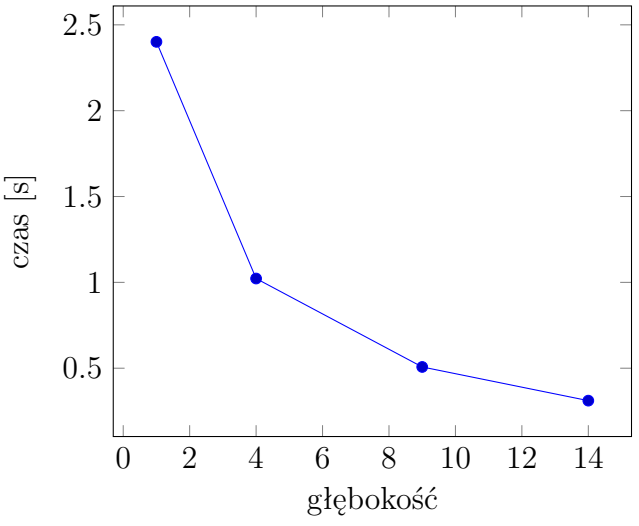


liczba fragmentów	spf [s]		
	4 proc.	9 proc.	14 proc.
16	0,179	0,088	0,062
36	0,170	0,086	0,064
64	0,207	0,104	0,098
100	0,213	0,146	0,146
144	0,252	0,219	0,218
max przysp.	1,640	3,270	4,500

Glass - zależność od zrównoleglenia (640x360)



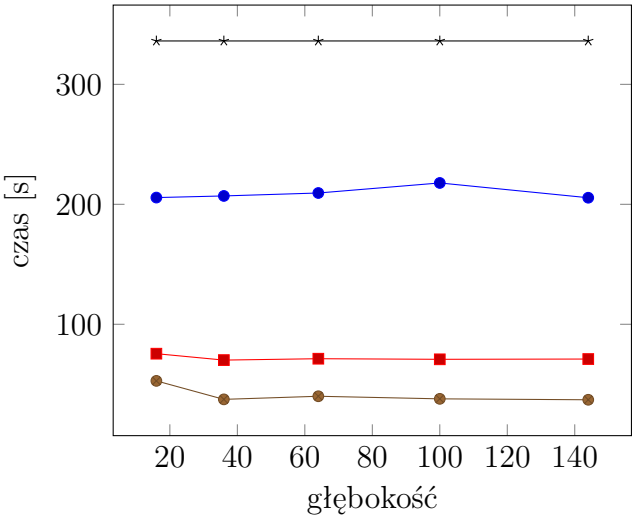
Spheres - zależność od zrównoleglenia (640x360)



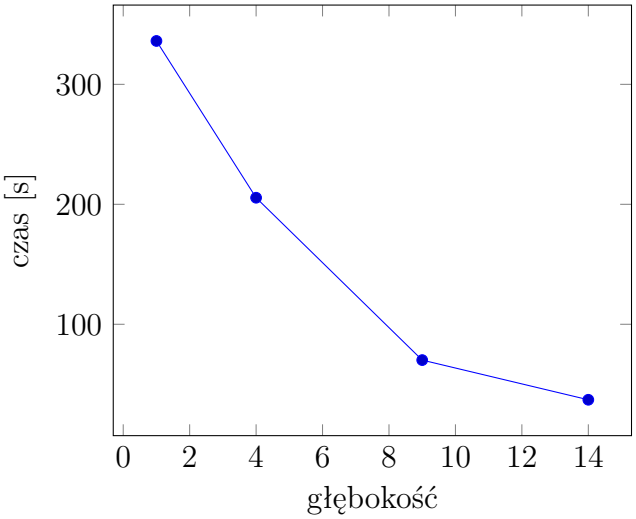
spowodowane zastosowaną skalą. Wg. danych z tabeli ¡!numer tabeli!¿, optymalną liczba wycinków dla 14 procesów oscyluje wokół wartości 64. Dla pozostałych scen czas generowania klatki spada wraz ze wzrostem ziarnistości. Warto zwrócić uwagę, że dla 14 procesów czas działania algorytmu dla 16 fragmentów jest najdłuższy - jest to spowodowane tym, że po policzeniu pierwszej partii zadań wiele węzłów pozostaje bezczynnych. Dane pokazują również, że optymalna ziarnistość zmienia się wraz ze zmianą rozmiaru klastra.

liczba fragmentów	spf [s]		
	4 proc.	9 proc.	14 proc.
16	1,100	0,742	0,480
36	1,053	0,548	0,407
64	1,036	0,514	0,311
100	1,022	0,507	0,323
144	1,053	0,545	0,344
max przysp.	2,350	4,730	7,720

Suzanne - zależność od zrównoleglenia (640x360)



Spheres - zależność od zrównoleglenia (640x360)



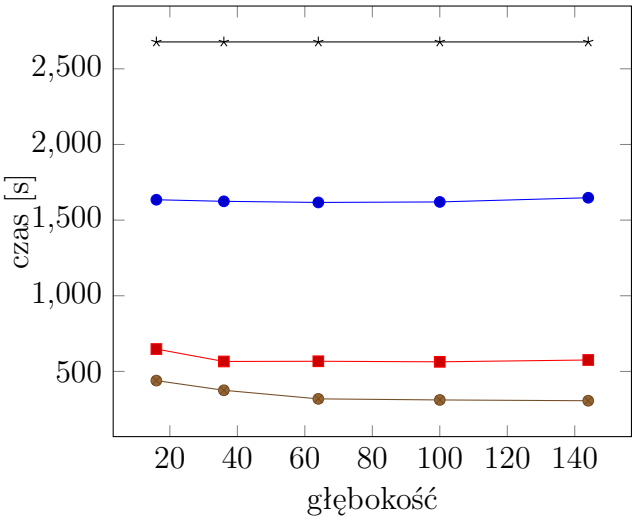
liczba fragmentów	spf [s]		
	4 proc.	9 proc.	14 proc.
16	205,605	75,473	52,809
36	206,972	70,136	37,398
64	209,420	71,284	39,900
100	217,794	70,788	37,383
144	205,472	71,007	37,061
max przysp.	1,640	4,790	9,070

7.1.5 Wykorzystanie drzewa BSP

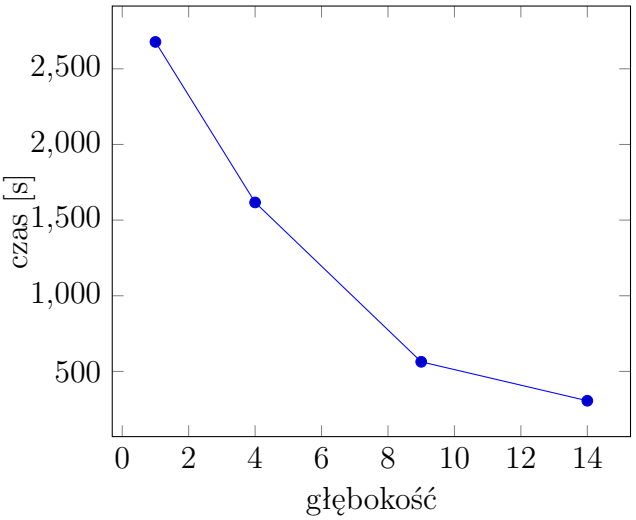
W niniejszym podrozdziale pokazano, w jakim stopniu wykorzystanie drzewa BSP pozwala na przyspieszenie generowania obrazu. W dolnej części tabel z wynikami podano ilokrotnie maksymalnie przyspieszyły obliczenia względem podejścia niezrównoleglonego (stosunek czasu obliczeń algorytmu wykonującego się na jednym procesorze do minimalnego czasy otrzymanego w teście) oraz czas budowy drzewa BSP. Parametrom niebędącym przedmiotami tego badania zostały arbitralnie przypisane poniższe wartości:

- głębokość drzewa śledzenia promieni - 3
- liczba pikseli - 640x360 (230400)
- użycie drzewa BSP - nie

Kid - zależność od zrównoleglenia (640x360)



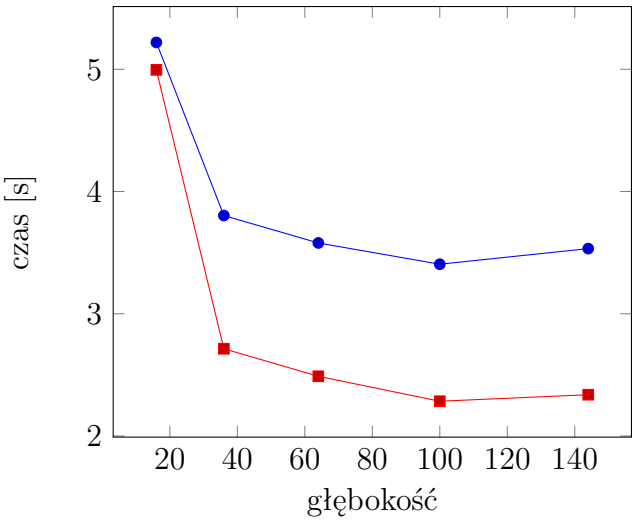
Spheres - zależność od zrównoleglenia (640x360)



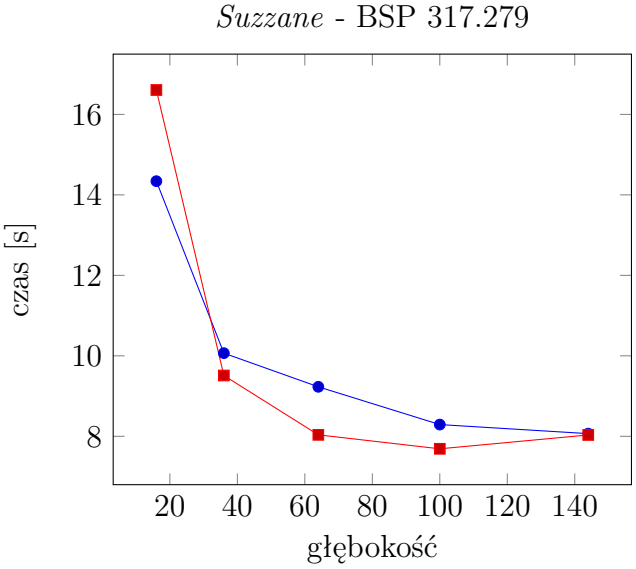
liczba fragmentów	spf [s]		
	4 proc.	9 proc.	14 proc.
16	1634,560	647,817	439,232
36	1624,210	565,670	375,321
64	1616,771	567,067	318,538
100	1620,153	563,411	311,411
144	1639,121	575,474	306,143
max przysp.	1,660	4,750	8,750

- liczba świateł - 1
- cieniowanie - tak
- liczba węzłów wykonawczych - 14

Glass - BSP 0.107058



liczba fragmentów	spf [s]	
	brak bryły ot.	z bryłą ot.
16	5,219	4,995
36	3,803	2,714
64	3,579	2,488
100	3,405	2,284
144	3,533	2,338
max przysp.	0,700	1,050
czas gen. drzewa [s]	0,107	



liczba fragmentów	spf [s]	
	brak bryły ot.	z bryłą ot.
16	14,340	16,606
36	10,067	9,511
64	9,230	8,036
100	8,292	7,689
144	8,067	8,033
max przysp.	41,670	43,710
czas gen. drzewa [s]	317,279	

Metoda śledzenia promieni wykorzystująca drzewo BSP pozwoliła na zmniejszenie czasu generowania obliczeń sceny *Suzanne* do kilku sekund. Początkowy (i co ważne jednorazowy) koszt budowy drzewa przyniósł pożądane rezultaty. Inaczej ma się sytuacja w przypadku sceny *Glass*, która, jak wynika z powyższych danych, jest przypadkiem sceny, dla której korzystanie z drzewa BSP jest nieopłacalne - nie dość, że musieliśmy ponieść koszt związany z jego budową, to jeszcze generowanie kolejnych klatek trwa dłużej niż w przypadku algorytmu niezrównoleglonego. Jakie cechy musi posiadać scena, żeby korzystanie z drzewa BSP było opłacalne i w jaki sposób możemy poprawić jego działanie?

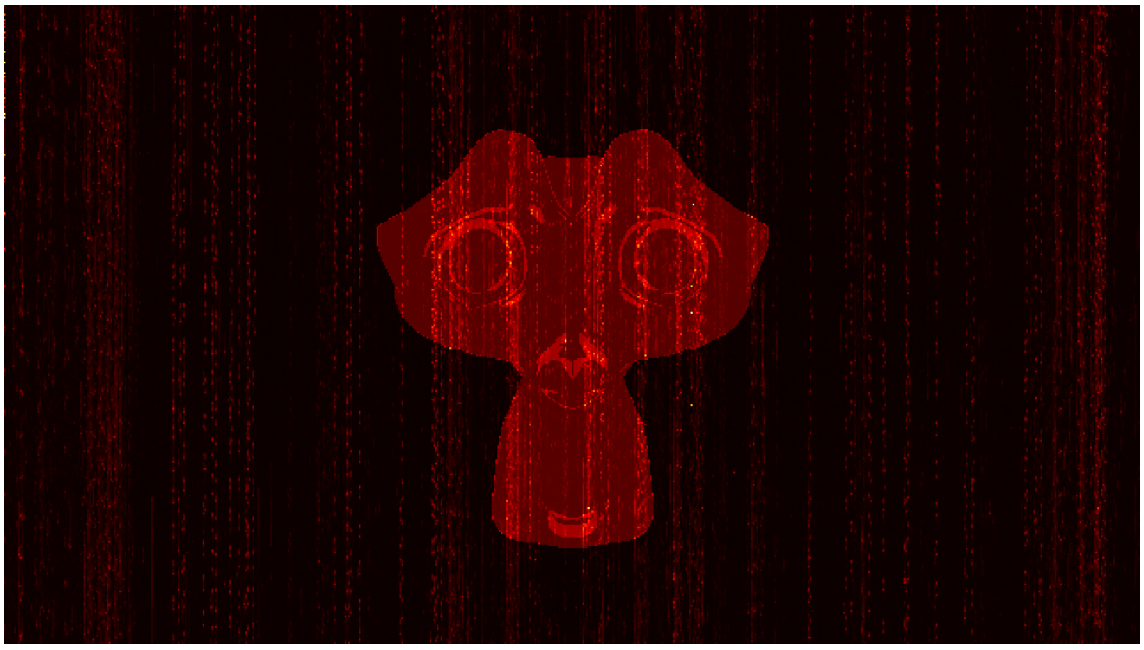
Podstawową problemem sceny *Glass* jest duża liczba promieni pierwotnych, które w nic nie trafiają. Biorąc pod uwagę, że obserwator widzi całą scenę, to większość płaszczyzn dzielących jest przez te promienie przecinane (w zależności od ich nachylenia). W takiej sytuacji algorytm szukania najbliższego obiektu więcej czasu spędza na trawersowaniu drzewa niż miałoby to miejsce w przypadku przeglądu zupełnego - odwiedzi on większość węzłów drzewa. Kolejnym czynnikiem mogącym powodować problem jest wzajemne ułożenie wielokątów na scenie. W sytuacji, w której tworzą one zbiór wypukły, wybieranie płaszczyzny dzielącej spośród tych wyznaczanych przez te wielokąty, sprawia, że żadna z nich nie podzieli przestrzeni. Częściowym rozwiązaniem tego problemu jest szukanie kandydatów na płaszczyzny dzielące nie tylko wśród takich, które pokrywają się w wielokątami sceny, ale też wśród płaszczyzn prostopadłych do tych wieloboków - takie podejście zostało zaimplementowane w testowanym algorytmie.

W celu polepszenia wydajności drzew BSP należy się skupić na czterech elementach: wprowadzeniu brył otaczających (redukcja liczby promieni nietrafiających w żaden obiekt, dla których trzeba by było przeprowadzać testy z wykorzystaniem drzewa BSP), poszukiwaniu lepszej metody wyboru kandydatów na płaszczyzny dzielące, ulepszeniu funkcji oceny płaszczyzny (np. zastosowanie funkcji SAH - !ROZDZIAŁ!), lub poprawie algorytmu trawersowania drzewa.

W celu pokazania, w jakim stopniu wprowadzenie brył otaczających może poprawić rezultaty, testowane sceny zostały zamknięte w prostopadłościanach. Jak widać na wykresie (!wykres szklanka!) zysk czasowy wynikający z zastosowania bryły otaczającej jest znaczący, jednak czas generowania obrazu ciągle jest lepszy dla przeglądu zupełnego. W przypadku *Suzanne* różnica zdaje się być niewielka. Wynika to z położenia obserwatora - w sytuacji, w której znajduje się on blisko obiektu (stosunkowo mało promieni pierwotnych nie trafia w nic), zastosowanie bryły otaczającej w kształcie prostopadłościanu

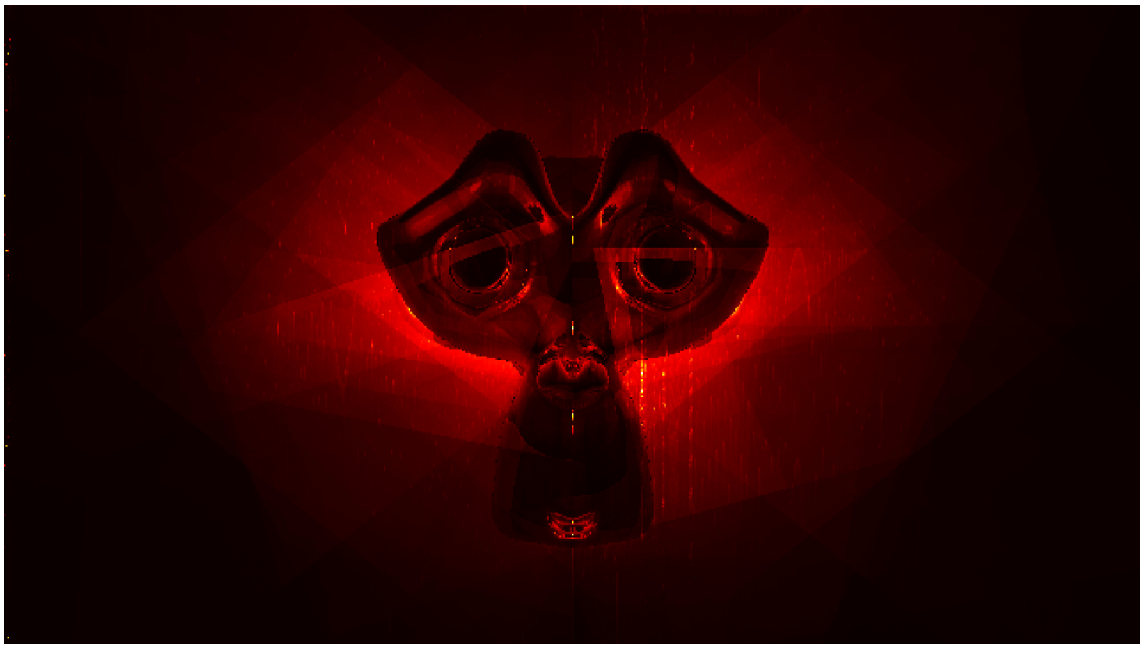
redukuje niewielką liczbę testów. Gdyby obserwator zaczął się oddalać od obiektu, to czas generowania obrazu z wykorzystaniem drzewa BSP by rósł, jednak dzięki wykorzystaniu bryły otaczającej tendencja się odwróci. Niżej przedstawiono mapy cieplne obrazujące które piksele liczyły się najdłużej. Im jaśniejszy obszar tym jego wygenerowanie zajęło więcej czasu. Szum pojawiający się na obrazach jest najprawdopodobniej spowodowany przełączaniem się procesów.

Rysunek 7.23: Przegląd zupełny



Na powyższej mapie cieplnej widzimy, że najwięcej czasu algorytm spędził na generowaniu pikseli zawierających model.

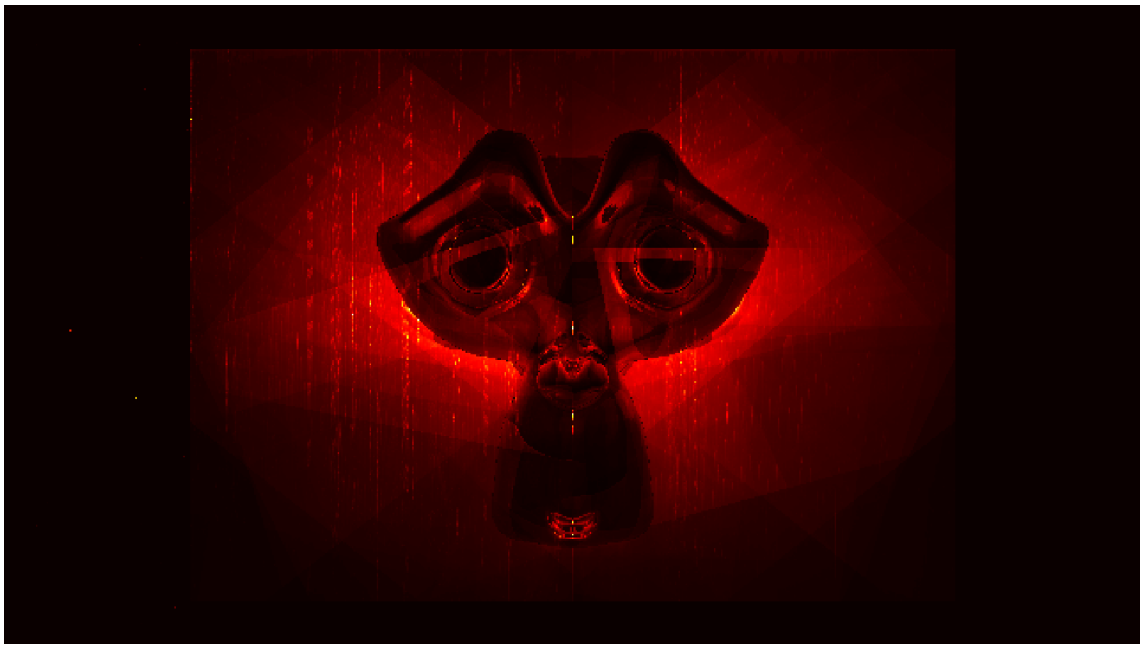
Rysunek 7.24: Brak wykorzystanie bryły otaczającej



W przypadku zastosowania drzewa BSP ciężar obliczeń spadł na otoczenie modelu a

nie na sam model. Powyższy obraz w interesujący sposób przedstawia płaszczyzny dzielące przestrzeń. Elementy znajdujące się za większą liczbą płaszczyzn są jaśniejsze.

Rysunek 7.25: Z wykorzystaniem bryły otaczającej

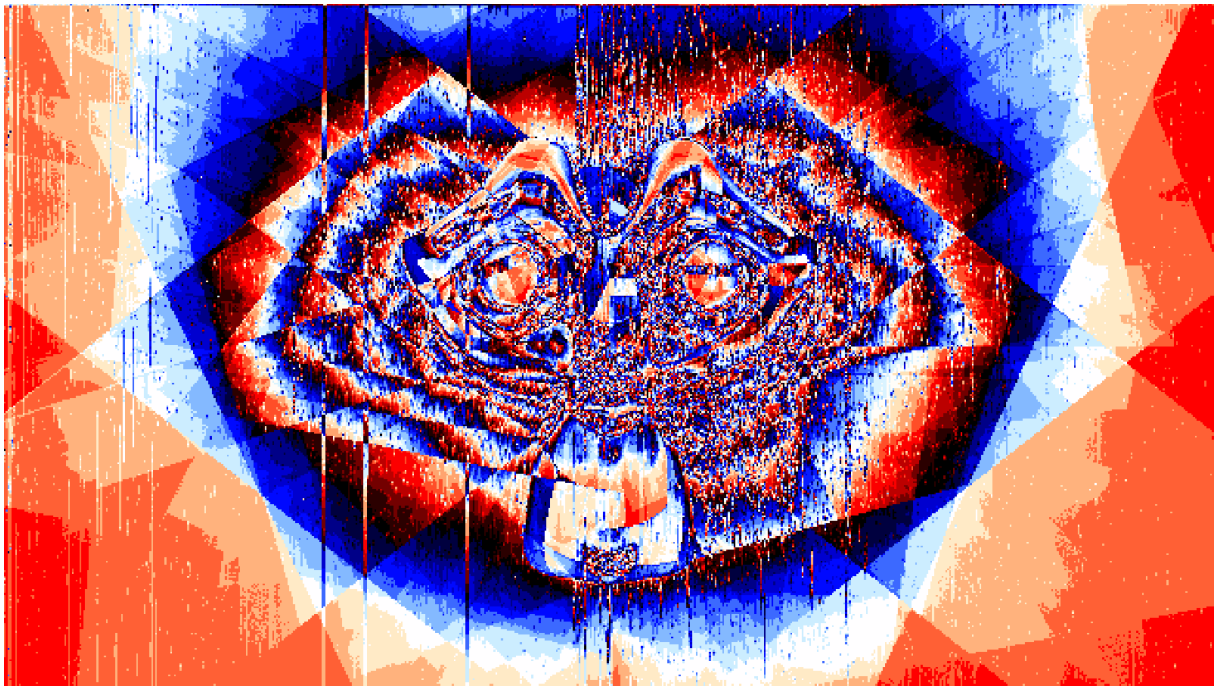


Zastosowanie bryły otaczającej sprawia, że czas liczenia promieni nie trafiających w żaden obiekt maleje właściwie do zera (koszt badania przecięcia promienia z bryłą otaczającą). Wbrew temu, co może sugerować powyższy obraz, każda ze ścian bryły otaczającej jest styczna do modelu - wrażenie, że jest inaczej powoduje rzut perspektywiczny (elementy, które są bliżej nas, zdają się być większe). Nie byłoby, tak gdyby został zastosowany rzut prosty.

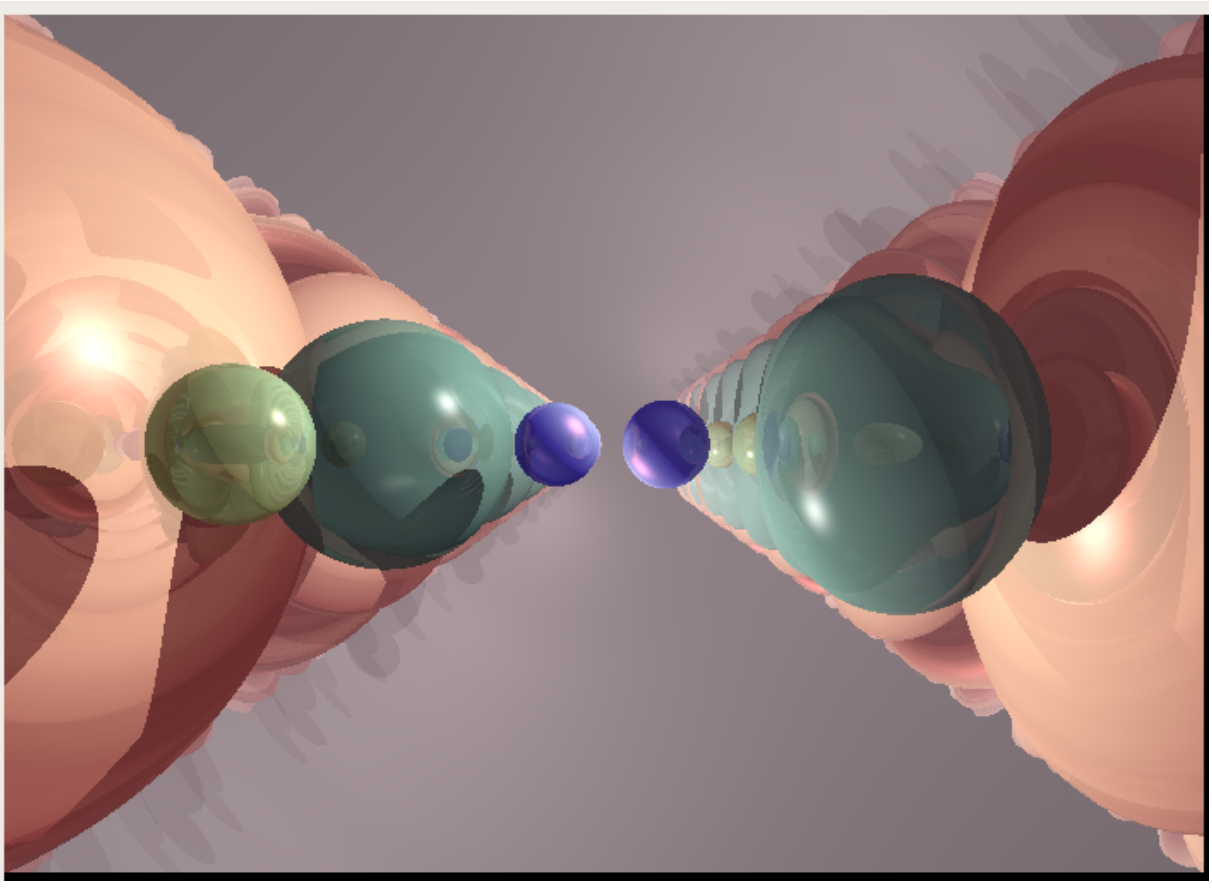
Na podstawie powyższych informacji można wywnioskować, że zastosowanie brył otaczających, które lepiej przylegałyby do modeli, zmniejszy czas wykonywania obliczeń. Dodatkowo, chcąc wprowadzić funkcję SAH niezbędne, jest zamknięcie całej sceny w bryle, ponieważ ta wymaga liczenia objętości podprzestrzeni (musi więc istnieć jakieś ograniczenie). Naturalnym rozwinięciem wydaje się również wprowadzenie hierarchii brył otaczających - na podstawie powyższej analizy zdaje się, że każdy integralny i nieruchomy obiekt powinien być najpierw zamknięty w bryle, a następnie dzielony z wykorzystaniem drzewa BSP. Rozwiązania hybrydowe są szeroko stosowane w grafice, zwłaszcza, że drzewo BVH ma tą wyższość nad drzewem BSP, że przemieszczanie się obiektów nie wymaga przebudowania go całego.

7.2 Przykładowe obrazy

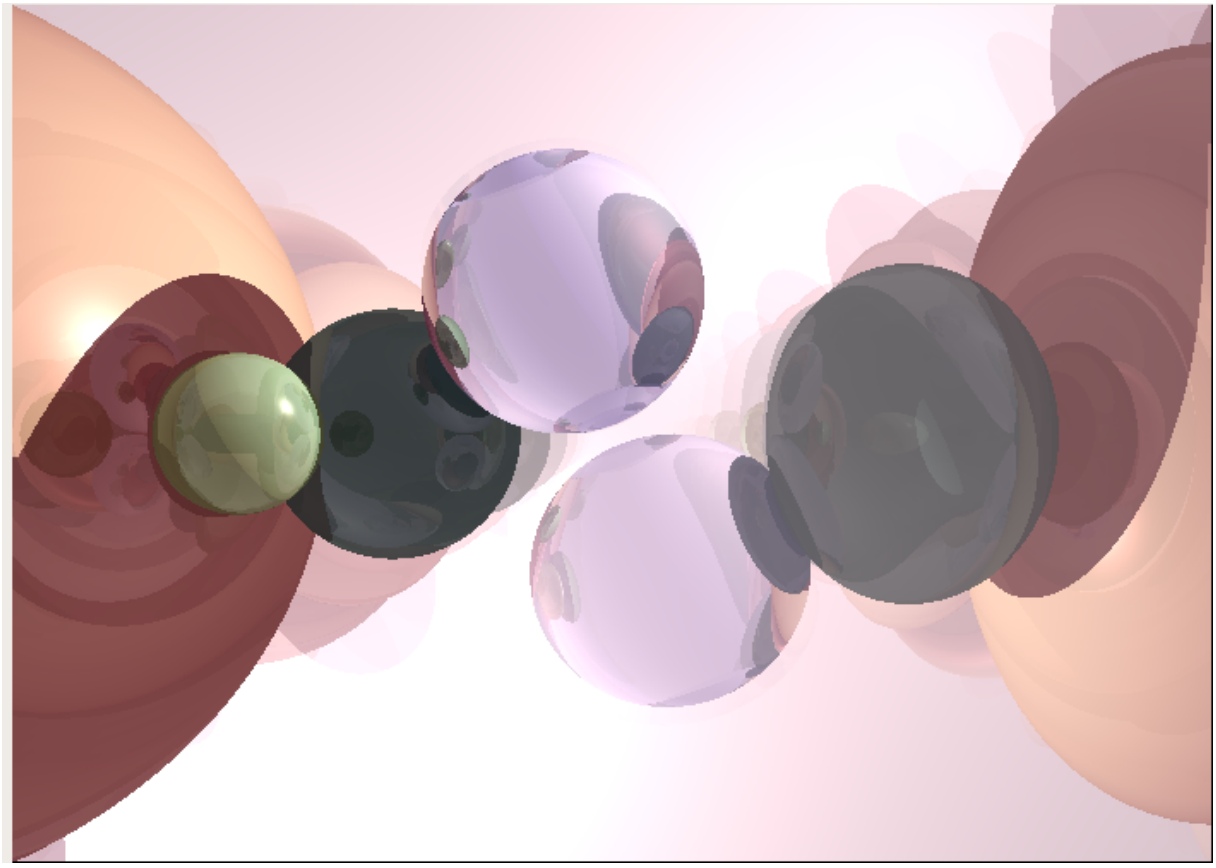
Rysunek 7.26: Spheres



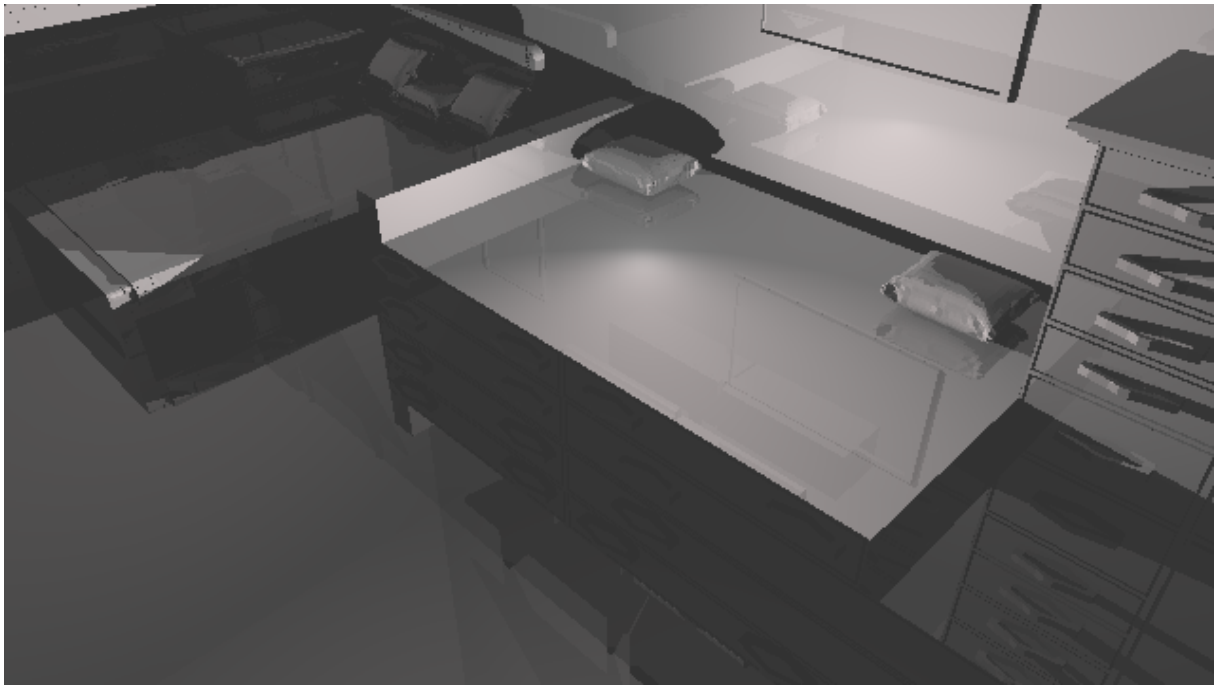
Rysunek 7.27: Spheres



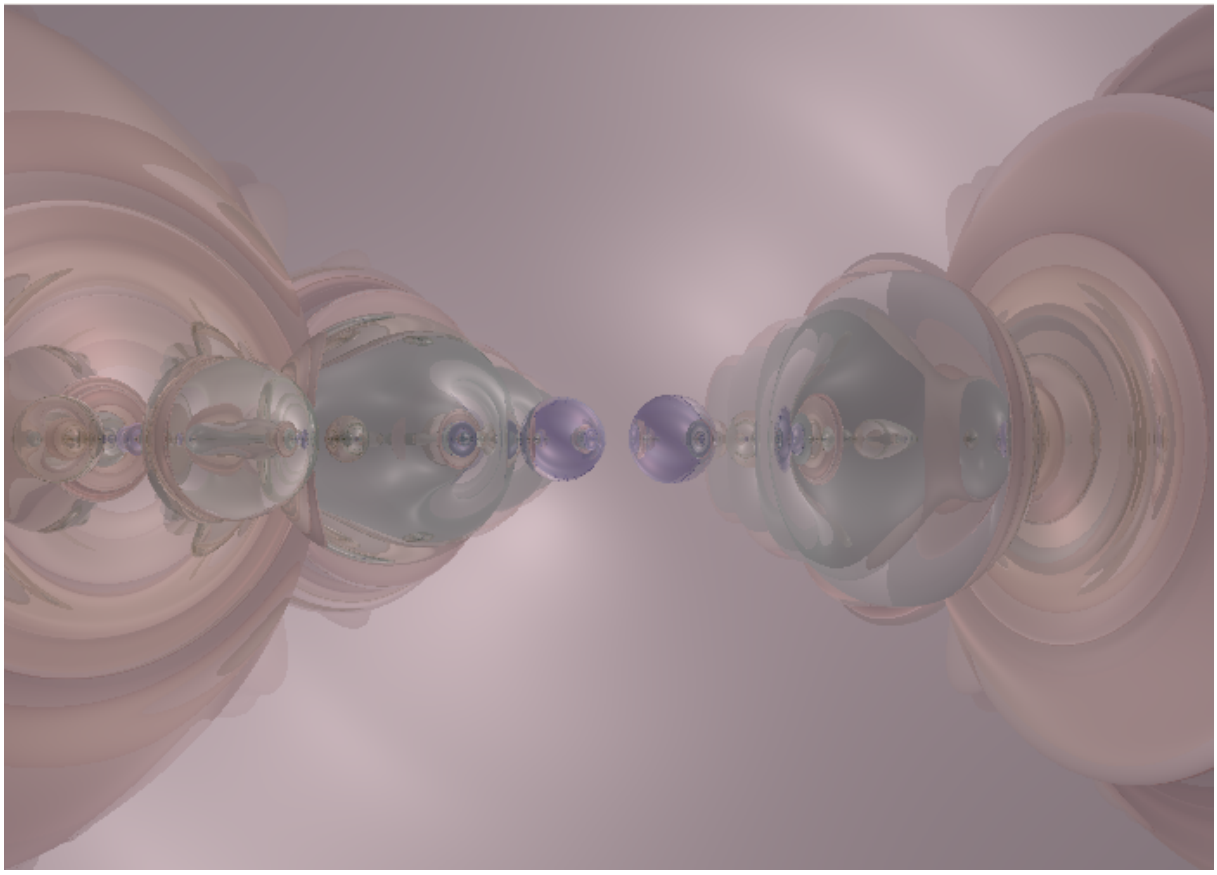
Rysunek 7.28: Spheres



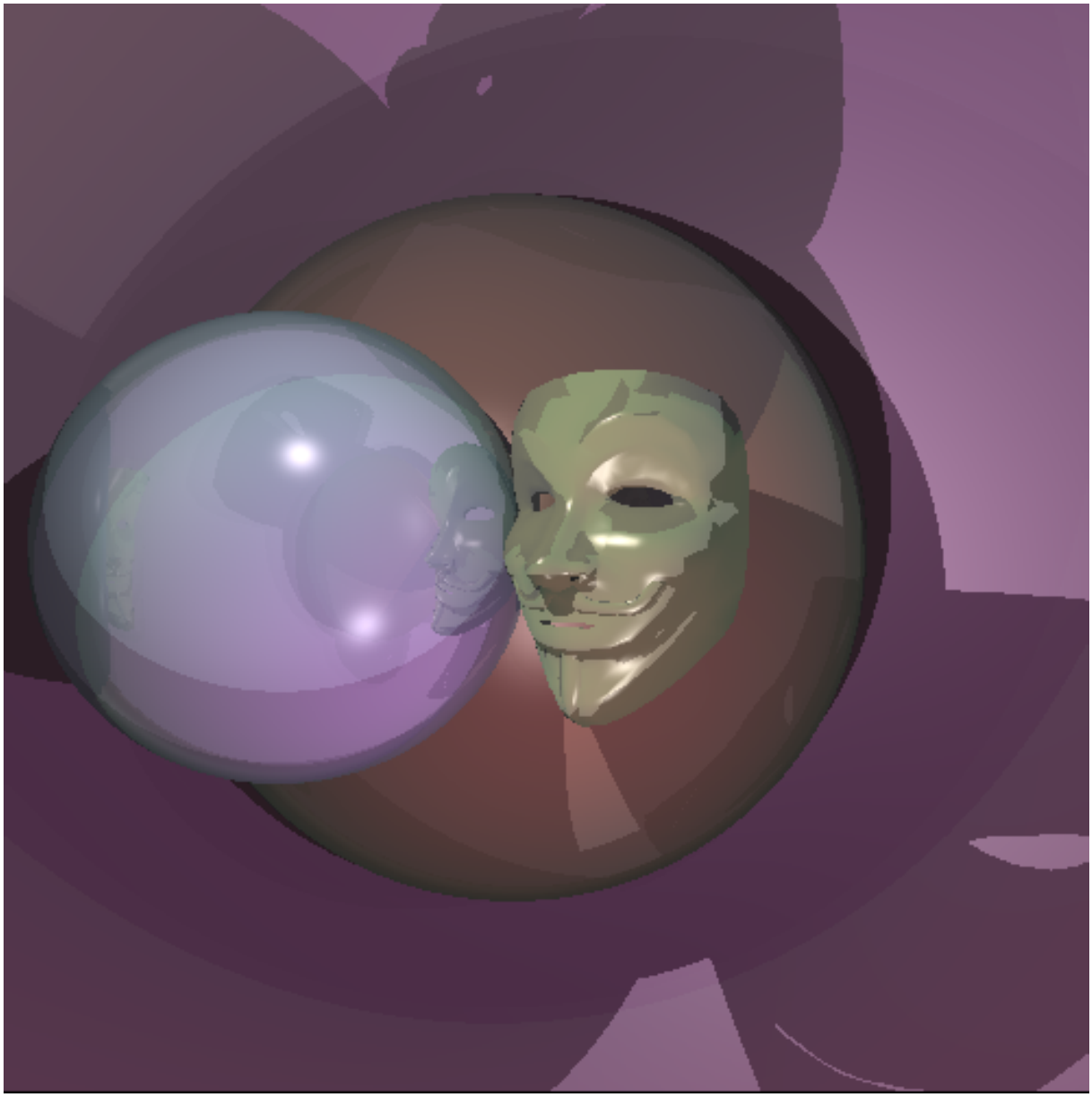
Rysunek 7.29: Spheres



Rysunek 7.30: Spheres



Rysunek 7.31: Spheres



Rozdział 8

Podsumowanie

Podstawowym celem pracy było zbadanie czy metoda śledzenia promieni ma rację bytu w interaktywnych aplikacjach graficznych. By uzyskać odpowiedź na to pytanie przeprowadzono szereg testów pokazujących między innymi, w jaki sposób definicja sceny, której dwuwymiarowa wizualizacja ma się pojawić przed oczami użytkownika, wpływa na czas generowania obrazu. Dzięki nim dowiedzieliśmy się, w jaki sposób czynniki takie jak rozmiar obrazu, liczba świateł, głębokość drzewa wpływają na szybkość obliczeń. Taka wiedza pozwala na umiejętne minimalizowanie czasu potrzebnego na generowanie realistycznych grafik, przy jak niewielkim wpływie na jakość obrazu.

Przejdźmy do omówienia wyników zrównoleglenia algorytmu. Dzięki niemu możliwe było uzyskanie ok. 16 klatek na sekundę przy najmniej skomplikowanej scenie (*Spheres*), która mimo swojej prostoty i tak była bardzo efektowna. Dla większej liczby obiektów znajdujących się na scenie rezultaty nie były wystarczające (mimo maksymalnego przyspieszenia sięgającego 900%) - klatka dla sceny *Glass* generowała się ok. 0,3 sekundy, dla *Suzanne* 37 sekund, a dla *Kid* ok. 5 min. Takie czasy są niedopuszczalne. W ramach badań nad możliwością przyspieszenia obliczeń generowania obrazu zostało przetestowane drzewo BSP. Okazuje się, że pozwala ono w niektórych sytuacjach (dokładnie w jakich zostało opisane w punkcie 7.1.5) na znaczne przyspieszenie obliczeń (czas generowania obrazu na podstawie sceny *Suzanne* spadł do 8 sekund), jednak, jak pokazuje alternatywny, „zły" przykład *Glass*, potrafi również pogorszyć wydajność. Walką z tego typu sytuacjami może być zmiana strategi podziału przestrzeni na *SAH* (punkt 2.2.4), lub zastosowanie brył otaczających. Istnieje możliwość podejścia hybrydowego, które wykorzystywałoby zalety zarówno drzew BSP jak i drzew BVH - w celu implementacji takiego podejścia należy dokładnie zbadać zachowanie drzew BVH, a następnie próbować łączyć ze sobą oba rozwiązania (prawdopodobnie modyfikując książkowe zachowanie jednego i drugiego). Propozycją, która nie jest uzasadniona żadnymi badaniami a czystą intuicją, jest zamykanie integralnych, nieruchomych (drzewa BVH nie potrzebują przebudowy w przypadku przemieszczenia się obiektów) modeli w bryłach otaczających, a następnie podział tych brył drzewem BSP - takie rozwiązanie powinno zmniejszyć negatywny wpływ wad drzewa BSP.

Czy metoda śledzenia promieni może być wykorzystywana w aplikacjach interaktywnych? Uważam, że tak - badania jakie zostały przeprowadzone pozwalają przypuszczać, że gdyby do zrównoleglenia obliczeń był wykorzystywany koprocesor graficzny (*GPU*), to stopień przyspieszania powinien być wystarczający do budowania np. prostych gier komputerowych opartych na tym algorytmie. Możliwość dalszych optymalizacji, które powinny poprawić rezultaty, zdaje się potwierdzać tę hipotezę. Ciekawa wydaje się również opcja zbudowania klastra obliczeniowego opartego o karty graficzne. Niniejsza praca to

dopiero szczyt góry lodowej - ogrom zagadnień i zależności, które należy zgłębić jest przytłaczający i zarazem fascynujący.

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] Z. J. Czech *Wprowadzenie do obliczeń równoległych*, Wydawnictwo Naukowe PWN, 2013

[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

[21] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview