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

Politechnika Wrocławska

Mateusz Gniewkowski

DD:MM:RR

Streszczenie

Streszczenie...

Spis treści

1 Wstęp **4**

2 Analiza problemu **5**
2.1 Śledzenie promieni 5
2.1.1 Podstawowy algorytm śledzenia promieni 5
2.1.2 Rekursywny algorytm śledzenia promieni 14
2.1.3 Równoległa wersja algorytmu śledzenia promieni 16
2.2 Optymalizacja algorytmu śledzenia promieni 17

3 Wybór technologii **18**
3.1 C++ 18
3.2 QT 18
3.3 Standard MPI 19

4 Projekt systemu **20**
4.1 Projekt klastra 20
4.1.1 Master 20
4.1.2 Slave 20
4.2 Opis programu 20

5 Implementacja **21**
5.1 Szczegółowy opis klas 21
5.2 Plik wejściowy 21
5.3 Warstwa prezentacji 21
5.4 Warstwa logiki biznesowej 21

6 Opis funkcjonalny **22**

7 Rezultaty **23**
7.1 Testy wydajnościowe 23
7.2 Omówienie wyników 23
7.2.1 Przyspieszenie obliczeń 23
7.2.2 Obliczenia w czasie rzeczywistym 23
7.3 Przykładowe obrazy 23

8 Podsumowanie **24**

9 Dodatek A - obliczenia równoległe **27**

Rozdział 1

Wstęp

Rozdział 2

Analiza problemu

2.1 Śledzenie promieni

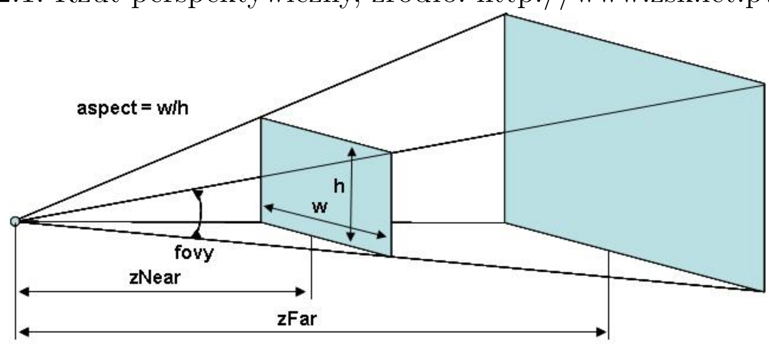
2.1.1 Podstawowy algorytm śledzenia promieni

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

Poniżej przedstawiono pseudokod podstawowego śledzenia promieni

piksele, obiekty
obj = null
dist = max

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



wybór środka rzutowania i rzutni

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

ustal kolor piksela na podstawie obj

Więcej na temat podstaw śledzenia promieni można przeczytać w [**?**, **?**, **?**]

Obliczenie przecięć

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

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

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

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

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 to mniejsze (bliższy punkt przecięcia). Podstawiając rozwiązanie do równania promienia otrzymamy punkt przecięcia zawierający w powierzchni kuli.

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

Dane są równanie promienia i równanie płaszczyzny:
$p = p_0 + tv$
$Ax + By + Cz + D = P \bullet N + D = 0$

gdzie
p_0 - punkt początkowy promienia (x_0, y_0, z_0)
v - wektor kierunkowy promienia o długości 1 (x_v, y_v, z_v)
t - parametr określający odległość danego punktu, należącego do promienia, od jego początku tego promienia
P - dowolny punkt płaszczyzny
N - wektor normalny do płaszczyzny

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

$(p_0 + tv) \bullet N + D = 0$
$t = -(p_0 \bullet N + D)/(v \bullet N)$

Podstawiając t do równania promienia otrzymujemy punkt przecięcia. Jeżeli $t < 0$ to płaszczyzna znajduje się za promieniem, w przeciwnym przypadku przed (gdy $t = 0$ punkt początkowy zawiera się w płaszczyźnie). Należy zwrócić uwagę, że $v \bullet N$ nie może być równe zero - jeżeli jest, znaczy to że promień nigdy nie przecina płaszczyzny (jest do niej równoległy).

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

Algorytm klasyczny 1. Wyznaczenie równania płaszczyzny z trójkąta:

a) Obliczenie wektora normalnego do trójkąta:

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

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

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

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

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

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

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. Weźmy teraz dwa wektory położone na krawędziach i zaczynające się w tym punkcie - $(c - a)$ i $(b - a)$. W ten sposób, startując z punktu a i przesuwając się zgodnie z wektorami (zgodnie z parametrami z zakresu od 0 do 1), możemy dostać się do dowolnego punktu należącego do trójkąta. Stąd bierze się równanie:

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

Należy zwrócić uwagę na dwa fakty. Po pierwsze jeżeli którаś ze zmiennych u i v jest mniejsza od zera lub większa od jedynki to jesteśmy poza trójkątem. Po drugie jeżeli $u + v > 1$ to przecięliśmy krawędź BC to również wyznaczony punkt znajduje się poza trójkątem.

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

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

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

$pvec = d \times (c - a)$
$qvec = (p - a) \times (b - a)$
$invDet = 1/((b - a) \bullet pvec)$
$u = ((p - a) \bullet pvec)) * invDet$
$v = (d \bullet qvec) * invDet$
$t = ((c - a) \bullet qvec) * invDet$

Poniżej przedstawiono przykładową implementację algorytm „Möller – Trumbore” zaczerpniętą z [**?**]:

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

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

Model światła

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

Najprostszym modelem światła jest tzw. oświetlenie bezkierunkowe. Wykorzystuje ono tzw. światło otoczenia (ambient light), które z definicji nie ma określonego źródła (wypełnia całą scenę) i dochodzi do każdego elementu z taką samą intensywnością.
$I = I_{amb} * k_{amb}$

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

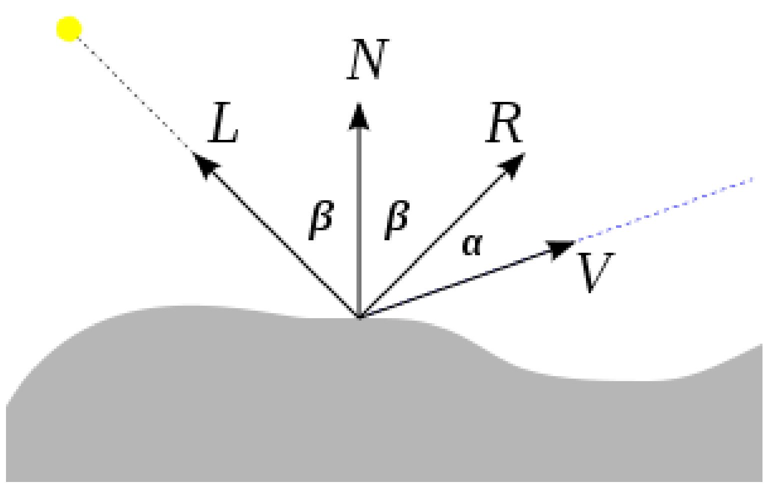
Bardziej zaawansowanym modelem, bo wykorzystującym światło rozproszone (diffuse light) jest tzw. model Lamberta - dodatkowo uwzględnia on punktowe źródła światła, których promienie padają pod pewnym kątem na daną powierzchnie. Model Lamberta zakłada, że oświetlane powierzchnie są idealnie matowe, zatem światło odbite od nich rozchodzi się tak samo we wszystkich kierunkach (odbicie lambertowskie). W związku z tym, nie ma możliwości otrzymania odblasków widocznych na powierzchniach błyszczących.

$I = I_{amb} * k_{amb} + I_{dif} * d_{amb} * (N \bullet L)$
W powyższym wzorze N i L są kolejno wektorem normalnym do powierzchni i wektorem wskazującym kierunek padania światła.

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

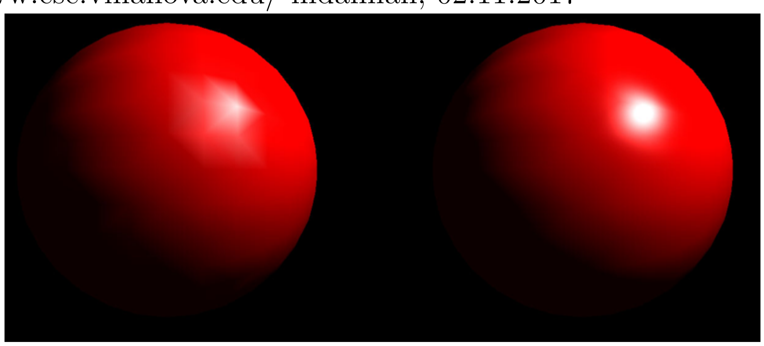
$I = I_{amb} * k_{amb} + 1/(a + bd + c^2 d)(I_{dif} * d_{amb} * (N \bullet L) + k_{spec} * I_{spec} * (R \bullet V)^n)$
gdzie R i V oznaczają kolejno kierunek odbicia promienia i kierunek obserwacji. Ułamek $1/(a + bd + c^2 d)$ określa intensywność padającego światła w zależności od odległości od źródła (d to odległość od źródła światła, pozostałe parametry są dobieranie empirycznie).

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



Cieniowanie Cieniowanie ma na celu stworzenie złudzenia, w którym siatka trójkątów sprawia wrażenie gładkiej powierzchni. Najpopularniejszymi metodami cieniowania są „Cieniowanie Gourauda", które polega na wyliczeniu kolorów każdego wierzchołka prymitywu, a następnie interpolacji pozostałych. Takie rozwiązanie jest stosunkowo szybkie obliczeniowo, ale nie daje realistycznych rezultatów. Popularną alternatywą jest „Cieniowanie Phonga". Polega ono na określeniu w każdym wierzchołku poligonu wektorów „normalnych" (niekoniecznie będących normalnymi do powierzchni), a następnie wyliczeniu (poprzez interpolację) takich wektorów dla pozostałych punktów. Takie wektory są później wykorzystywane np. w modelu 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

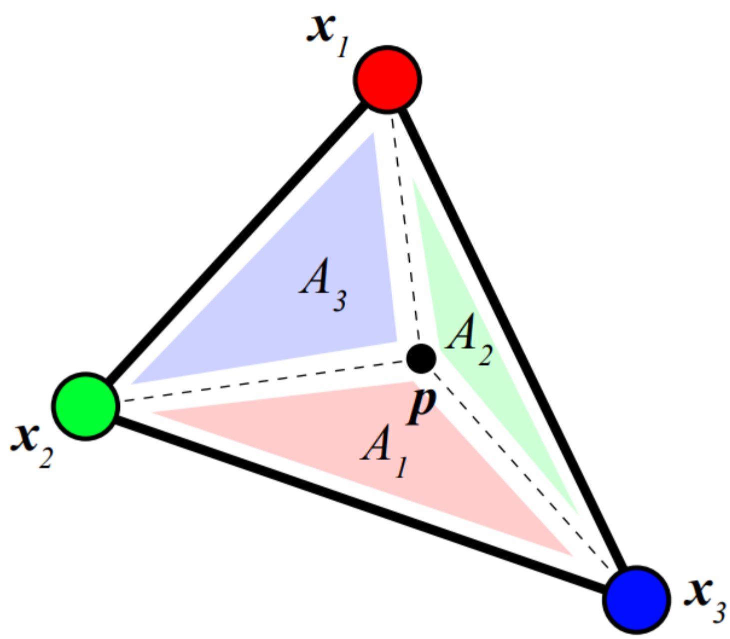


Interpolacja barycentryczna Do interpolacji wektorów może posłużyć nam tzw. barycentryczna. W przypadku trójkąta sytuacja prezentuje się następująco:

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

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

Rysunek 2.4: Interpolacja barycentryczna, źródło: nieznane



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

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

2.1.2 Rekursywny algorytm śledzenia promieni

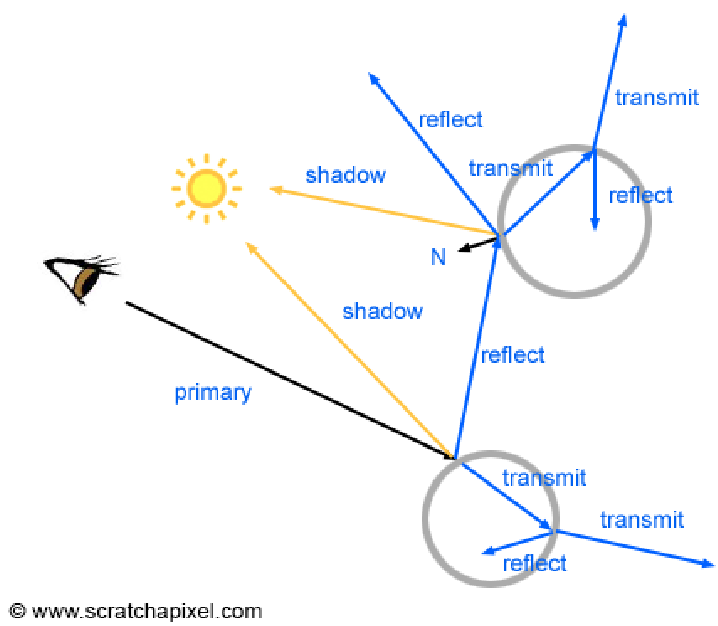
Rekursywny algorytm śledzenia promieni [?] jest rozwinięciem algorytmu podstawowego, opisanego w poprzednim rozdziale. Uwzględnia on cienie, odbicia i załamania światła, dzięki śledzeniu dodatkowych promieni wysłanych z punktów przecięcia. W przypadku generowania cieni, z każdego punktu przecięcia wysyła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

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

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

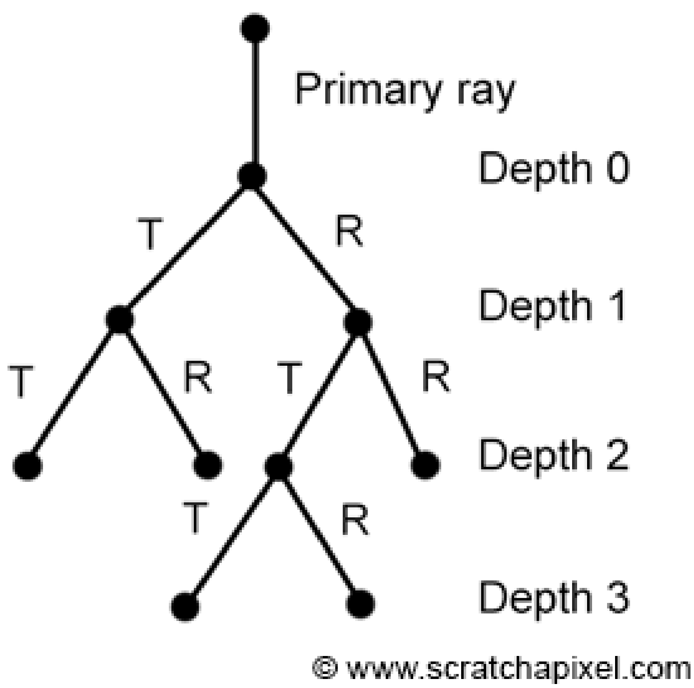
W tym miejscu należy zauważyć, że nie wszystkie materiały są przezroczyste i nie wszystkie materiały odbijają światło w podobny sposób co lustro (można w nich zobaczyć odbicia innych przedmiotów). Implementując algorytm śledzenia promieni należy wprowadzić mechanizm, który pozwala stwierdzić jaki ułamek ostatecznego koloru piksela będzie stanowić wynik śledzenia promieni odbitych/załamanych i czy takie promienie warto 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: [?]



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

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



2.1.3 Równoległa wersja algorytmu śledzenia promieni

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

2.2 Optymalizacja algorytmu śledzenia promieni

Rozdział 3

Wybór technologii

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

3.1 C++

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

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

3.2 QT

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

Qt zapewnia mechanizm sygnałów i slotów, automatyczne rozmieszczanie 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 [?] sprawiają, że Qt jest dobrym wyborem przy pisaniu aplikacji okienkowych.

3.3 Standard MPI

Wybór sposobu zrównoleglenia jest podyktowany nie tylko rodzajem problemu, którego dotyczy praca, ale również dostępnym sprzętem. Najbardziej elastyczną technologią pozwalającą na obliczenia równoległe są klastry - grupa połączonych ze sobą niezależnych komputerów mogących różnić się podzespołami. Minusem takiego rozwiązania jest to, że przeciwieństwie do systemów wieloprocesorowych, procesory nie są podłączone magistralą ze wspólną pamięcią, co z kolei oznacza wolniejszą i trudniejszą programistycznie komunikację między nimi. Wiele problemów mogłoby być rozwiązana efektywniej, w taki, alternatywny sposób. Kolejnym problemem jest trudność rozłożenia obliczeń pomiędzy stacjami wykonawczymi, ponieważ czas obliczeń (i czas przesyłu danych przez sieć) może być znacząco różny dla poszczególnych komputerów. W metodzie śledzenia promieni narzut komunikacyjny jest relatywnie niski, a sugerowany w punkcie 2.1.3 sposób zrównoleglenia obliczeń nie powinien stanowić dużego problemu w ich rozłożeniu, więc 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.

MPI (Message Passing Interface) jest standardem przesyłania komunikatów pomiędzy procesami znajdującymi się w 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. W związku z tym, jednym z pierwszych kroków projektowanego algorytmu będzie rozesłanie definicji sceny do wszystkich komputerów w klastrze (przesłanie całej sceny będzie musiało nastąpić tylko raz - w momencie startu programu).

Rozdział 4

Projekt systemu

4.1 Projekt klastra

4.1.1 Master

4.1.2 Slave

4.2 Opis programu

Rozdział 5

Implementacja

5.1 Szczegółowy opis klas

5.2 Plik wejściowy

5.3 Warstwa prezentacji

5.4 Warstwa logiki biznesowej

Rozdział 6

Opis funkcjonalny

Rozdział 7

Rezultaty

7.1 Testy wydajnościowe

7.2 Omówienie wyników

7.2.1 Przyspieszenie obliczeń

7.2.2 Obliczenia w czasie rzeczywistym

7.3 Przykładowe obrazy

Rozdział 8

Podsumowanie

Bibliografia

[1] J. D. Foley i in. *Wprowadzenie do Grafiki Komputerowej*. tłum. J. Zabrodzki, WNT, Warszawa 1995.

[2] F. Dunn, I. Parberry, *3D Math Primer for Graphics and Game Development*. Wordware Publishing, Inc. 2002.

[3] K. Suffern *Ray Tracing from the Ground Up*. A K Peters, Ltd. Wellesley, Massachusetts, 2007.

[4] StrachPixel 2.0: https://www.scratchapixel.com, 27.09.2017

[5] Wikipedia, Wolna Encyklopedia: https://en.wikipedia.org/wiki/Moller–Trumbore_intersection_algorithm, 15.09.2017

[6] Wikipedia, Wolna Encyklopedia: https://pl.wikipedia.org/wiki/Prawo_Snelliusa, 02.11.2017

[7] M. Falski *Przegląd modeli oświetlenia w grafice komputerowej* Praca magisterska pod kierunkiem dr A. Łukaszewskiego, Uniwersytet Wrocławski, 2004

[8] Qt: https://www.qt.io/

Spis rysunków

2.1 Rzut perspektywiczny, źródło: http://www.zsk.ict.pwr.wroc.pl 6
2.2 Model Phonga, źródło: https://pl.wikipedia.org/wiki/Cieniowanie_Phonga 12
2.3 „Cieniowanie Gourauda" i „Cieniowanie Phonga", źródło: http://www.csc.villanova.edu/ m
mian, 02.11.2017 13
2.4 Interpolacja barycentryczna, źródło: nieznane 14
2.5 Wizualizacja algorytmu rekursywnego, źródło: [?] 15
2.6 Wizualizacja algorytmu rekursywnego, źródło: [?] 16

Rozdział 9

Dodatek A - obliczenia równoległe