POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: **INFORMATYKA (INF)**
SPECJALNOŚĆ: Systemy i sieci komputerowe (ISK)

PRACA DYPLOMOWA MAGISTERSKA

Metody śledzenia promieni w syntezie realistycznych obrazów cyfrowych

AUTOR:
Krzysztof Dragan

PROWADZĄCY PRACĘ:
mgr inż. Wojciech Tarnawski, Z0402

OPIEKUN:
dr inż. Michał Woźniak, Z0402

OCENA PRACY:

Wrocław 2000

Spis Treści

Spis ważniejszych terminów

raytracing	Ogólny algorytm tworzenia cyfrowych obrazów przy pomocy śledzenia promieni. W szczególności odnosi się do raytracingu wstecznego, polegającego na śledzeniu promieni od obserwatora do źródeł światła.
renderowanie	Generowanie cyfrowego obrazu przy pomocy określonych technik, np. metodą śledzenia promieni.
scena	Obszar przestrzeni symulujący rzeczywistość, której obraz generujemy. Zawiera obiekty geometryczne (zwykle zamknięte) oraz obiekty będące źródłami światła.
promień	Symulacja drogi kwantu światła. Jest to półprosta mająca swój początek (typowo oznaczany O – *ang*. *origin*) oraz wektor kierunku (typowo oznaczany D – *ang. direction*).
promień pierwotny	(*ang. primary ray*) Promień wygenerowany dla konkretnego piksela obrazu, pochodzący z wirtualnego oka obserwatora.
promień wtórny	(*ang. secondary ray*) Promień będący kontynuacją innego promienia, w szczególności promienia pierwotnego. Jeden promień może dać początek wielu innym. Początkiem promienia wtórnego jest punkt na powierzchni obiektu trafiony poprzednim promieniem.
promień przecięcia	(*ang. intersection ray*) Promień dla którego szuka się najbliższego przecinanego obiektu. Wszystkie promienie pierwotne są promieniami przecięć.
promień cienia	(*ang. shadow ray*) Promień służący sprawdzeniu, czy źródło światła jest zasłonięte przez jakikolwiek obiekt. Promieni cieni używa się podczas generowania cieni w fazie oświetlenia. Dla promienia cienia szuka się dowolnego przecinanego obiektu.
piksel	(*ang. picture element)* Pojedynczy element dyskretnego obrazu. Obraz jest prostokątną siatką pikseli, a każdy piksel jest zwykle kwadratem, przyjmującym kolor opisany trzema składowymi – czerwoną, zieloną i niebieską.
tekstura	Obraz symulujący zróżnicowaną kolorystycznie powierzchnię obiektu.
teksel	(*ang. texture element*) Pojedynczy element, piksel obrazu tekstury.
materiał	Zespół właściwości powierzchni obiektu, determinujących sposób interakcji powierzchni ze światłem.

1 Wstęp

W dzisiejszych czasach komputery stały się na tyle mocne, by sprostać bardzo wymagającym zadaniom, takim jak tworzenie złożonych, realistycznych obrazów. Spośród różnych metod syntezy obrazu, najpopularniejszą jest śledzenie promieni, czyli raytracing. Umożliwia on uzyskanie wysokiego poziomu realizmu. Jest podstawą najpopularniejszych pakietów graficznych, takich jak 3D Studio MAX czy Maya. Najbardziej spektakularnym zastosowaniem raytracingu jest kinematografia, gdzie używa się go do generowania pojedynczych postaci (jak np. dinozaury w Parku Jurajskim) a nawet całych filmów (Final Fantasy z roku 2001). Niestety aby uzyskać wysoki poziom realizmu, należy dokładniej symulować efekty fizyko-optyczne, co powoduje wzrost zapotrzebowania na moc obliczeniową.

Obecnie największe moce obliczeniowe możliwe są do uzyskania w rozproszonych systemach komputerów połączonych siecią. Tysiące tanich komputerów osobistych zdolne są do wykonywania ilości obliczeń jaka jeszcze niedawno nie była dostępna nawet na superkomputerach.

Celem niniejszej pracy jest zaprojektowanie i zaimplementowanie raytracera – programu do generowania obrazów. Program ma działać na komputerach klasy PC pod kontrolą systemu operacyjnego Windows, a jednocześnie musi być przenaszalny na inne komputery i platformy. Musi oferować podstawowe efekty wizualne, będące domeną raytracingu, a jednocześnie musi umożliwiać rozszerzanie funkcjonalności, ulepszanie i dodawanie nowych komponentów programu. Ponadto program musi wspierać równoległe, rozproszone generowanie jednego obrazu przez wiele komputerów w celu przyspieszenia obliczeń.

Przyjęty układ pracy odzwierciedla kolejne etapy syntezy obrazu. Rozdział 2 zawiera ogólne wprowadzenie w problematykę śledzenia promieni. Opisuje on proponowany sposób podziału całego algorytmu syntezy obrazu na mniejsze etapy dla zapewnienia lepszej modularności. Rozdział 3 opisuje początkowy etap tworzenia obrazu – sposób obliczania promieni pierwotnych w wirtualnej kamerze. Tworzenie promieni pierwotnych jest kluczowym elementem syntezy obrazu i wpływa na „kształt” obrazu. Rozdział 4 opisuje metody przecięcia poszczególnych typów obiektów z promieniami. W zależności od zestawu wspieranych typów obiektów program można dostosować do różnych formatów danych wejściowych. Rozdział 5 opisuje wybrany algorytm redukcji ilości obliczeń. Etap ten jest istotny z punktu widzenia efektywności algorytmu i ma największy wpływ na czas obliczeń. Rozdziały 6, 7 i 8 opisują kolejne etapy polepszające realizm generowanych obrazów. Rozdział 6 opisuje sposoby „powlekania” obiektów teksturami. Tekstury symulują właściwości kolorystyczne powierzchni obiektów (np. trawa, drewno, itp.). Rozdział 7 opisuje algorytmy wyznaczania interakcji obiektów ze światłem. Rozdział 9 opisuje dodatkowe algorytmy użyte przy generowaniu obrazów: promienie wtórne symulujące odbicia światła od powierzchni obiektów (efekt lustra) i sposób w jaki tworzony jest ostateczny obraz widziany przez użytkownika, a także związany z nim algorytm podziału zadań umożliwiający przetwarzanie rozproszone, co jest jednym z głównych czynników skracających czas obliczeń. Rozdział 9 zawiera schematy obiektowe struktury i sposobu interakcji elementów programu między sobą, a także instrukcję użytkownika. Rozdział 10 stanowi podsumowanie pracy.

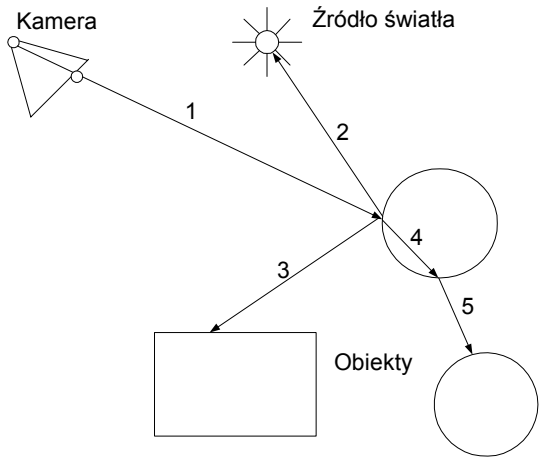
2 Wprowadzenie do raytracingu. Ogólny zarys

2.1 Omówienie idei raytracingu

Tworzenie obrazu metodami śledzenia promieni polega na symulacji efektów fizyko-optycznych. Symulowanie promieni światła biegnących od źródeł światła do oka obserwatora nie jest możliwe ze względu na zbyt dużą złożoność obliczeniową takiego podejścia. A zatem symuluje się promienie wsteczne, tzn. biegnące od oka obserwatora do źródeł światła.

Aby utworzyć obraz, trzeba zasymulować tyle promieni wychodzących z oka obserwatora, ile jest pikseli na obrazie. Dla każdego piksela generuje się co najmniej jeden promień.

Podczas symulacji pojedynczego promienia zachodzi szereg efektów pseudo-fizycznych. Zaliczają się do nich bezpośrednie odbicia światła od powierzchni, odbicia światła wtórnego pochodzącego od powierzchni innych obiektów oraz załamania promieni przy przejściu przez powierzchnie obiektów.



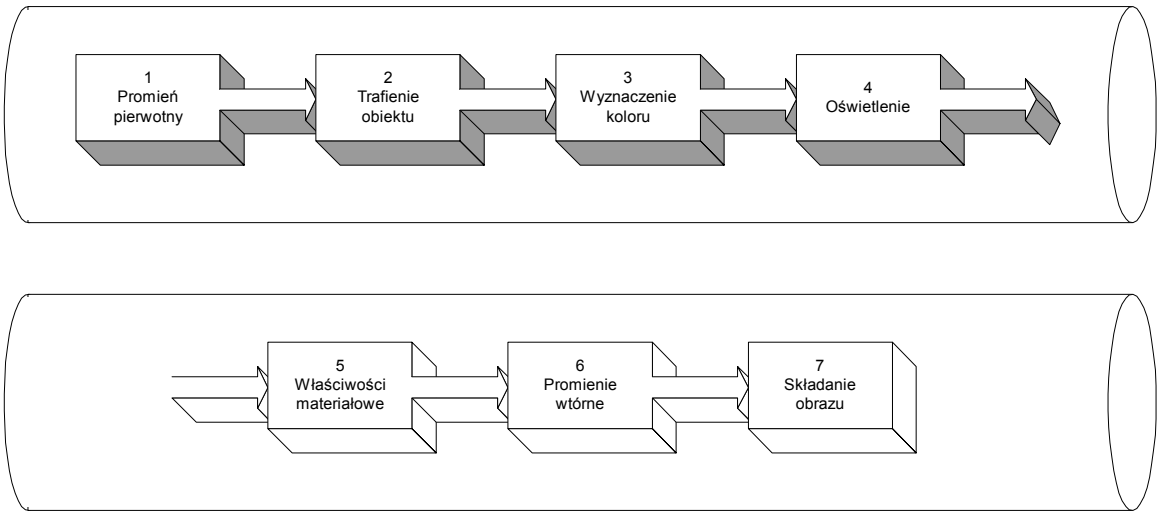
Rys. 2.1 – Droga promienia

Powyższy rysunek obrazuje symulację pojedynczego promienia. Promień 1 jest promieniem pierwotnym i wychodzi z kamery. Jest on wygenerowany dla pojedynczego piksela obrazu. Promień ten trafia w kulę, na której powierzchni zachodzi powielenie promienia. Pikselowi przypisywany jest kolor powierzchni w trafionym punkcie, modyfikowany przez źródło światła (promień 2) oraz dodawane są kolory promieni wtórnych 3-5. Promień 3 jest promieniem odbitym a promień 4 przenika przez powierzchnię kuli i załamuje się. Dla każdego promienia wtórnego 3-5 zachodzą takie same obliczenia jak dla promienia pierwotnego 1. Wynikowy kolor piksela jest sumą kolorów pochodzących od kolejnych promieni.

2.2 Etapy syntezy obrazu metodą śledzenia promieni

Tworzenie obrazu metodą raytracingu opiera się na generowaniu promieni wstecznych, wychodzących z oka wirtualnego obserwatora. Promienie te trafiają w obiekty a pikselom obrazu dla których wygenerowano promienie przypisywane są kolory trafionych obiektów.

Bazując na informacjach zawartych w [1], [8], dokonano podziału algorytmu syntezy obrazu na etapy tak, by każdy z nich był niezależny od pozostałych i miał ściśle zdefiniowane wejście i wyjście.



Rys. 2.2 – Potok obróbki promienia

Powyższy schemat pokazuje kolejne etapy algorytmu obliczania koloru pojedynczego piksela. Na wejściu podawane są współrzędne piksela na obrazie. Pomiędzy kolejnymi etapami przekazywane są dane opisujące promień i jego właściwości wyliczone w poprzednich etapach. Etapy te są scharakteryzowane następująco:

1. Utworzenie promieni pierwotnych. Każdy promień pierwotny wychodzi z wirtualnego oka obserwatora i przechodzi przez wybrany piksel obrazu. Promienie pierwotne generuje się dla odpowiadających im pikseli. Tworzenie ich uzależnione jest od geometrii kamery. (Rozdział 3).
2. Znajdowanie obiektów przecinanych przez promienie pierwotne. Podczas realizacji tego etapu bierze się pod uwagę geometrię różnych obiektów (Rozdział 4). Ponieważ obiektów w scenie może być bardzo dużo, stosuje się metody redukcji ilości przecięć, to jest zawężania obszaru poszukiwań najbliższego przecinanego obiektu (Rozdział 5).
3. Wyznaczanie koloru trafionego punktu. Po trafieniu obiektu promieniem dysponujemy punktem na powierzchni obiektu. Obiekt może mieć zróżnicowaną kolorystycznie powierzchnię symulowaną teksturą. Musimy więc znaleźć współrzędne teksela na teksturze, którym odpowiada trafiony przez nas punkt (Rodział 6).
4. Oświetlanie trafionego punktu. Wpływ na trafiony punkt mają różnorakie światła o różnych właściwościach. Niektóre światła są zasłonięte, przez co widoczne są cienie (Rozdział 7).
5. Uwzględnienie wpływu świateł i właściwości powierzchni obiektu. Zależnie od tych właściwości modyfikowany jest wyznaczony kolor (Rozdział 7).
6. Utworzenie promieni wtórnych – efekty odbicia i przezroczystości powierzchni obiektów. Każdy wtórny promień przecięcia przechodzi drogę od punktu 2 do 5. Wygenerowany kolor mnożony jest przez współczynnik wkładu promienia i dodawany do finalnego koloru piksela.

7. Składanie obrazu. Na obraz złożony z pikseli o wyznaczonych kolorach nakłada się różne filtry w celu wizualnej poprawy obrazu i wyeliminowania niepożądanych artefaktów.

W ostatnim etapie pikselowi przypisywany jest ostateczny kolor. Po wykonaniu całego algorytmu dla każdego piksela otrzymujemy gotowy obraz. Całość tworzy tzw. potok obróbki promienia który posiada kilka ciekawych właściwości.

2.3 Zalety przetwarzania potokowego

Potok jest to liniowy algorytm podzielony na rozłączne etapy. Każdy z etapów może być przetwarzany niezależnie od innych, a jego danymi wejściowymi są dane wyjściowe z poprzedniego etapu.

W przypadku raytracingu, przetwarzanie potokowe ma kilka istotnych zalet z punktu widzenia możliwości optymalizacji algorytmu. Po pierwsze każdy etap syntezy obrazu może zostać wymieniony na inny spełniający te same kryteria. Na przykład optymalizację ilości przecięć można zastąpić innym algorytmem, zależnie od potrzeb i od konfiguracji sceny.

Podział algorytmu na etapy – potok – daje też możliwość zrównoleglenia obliczeń; w szczególnych zastosowaniach dla części lub dla wszystkich etapów można zastosować wyspecjalizowane jednostki obliczeniowe.

Potok zwiększa lokalność czasową danych, ponieważ w każdym z etapów potrzebny jest inny, specyficzny zestaw danych. Dzięki potokowi można je obrabiać grupowo.

2.4 Organizacja programu pod kątem czynników satelitarnych

Oprogramowanie zostało zaprojektowane tak, aby jak najlepiej wykorzystać potok raytracingu. Każdy z etapów może być wymieniony na np. inny lub lepiej zoptymalizowany. Struktura programu zapewnia duże możliwości rozbudowy o inne moduły, inną funkcjonalność.

Program powstał dla komputerów PC, jednak dzięki przenośności języka C++ może on być z powodzeniem uruchomiony na innych platformach. Całkowita niezależność od interfejsu użytkownika i funkcjonalności związanej z systemem operacyjnym również znakomicie wspiera przenaszalność.

Przeniesienie programu na inny system operacyjny lub platformę sprzętową wymaga dopisania kilku podstawowych modułów udostępniających funkcjonalność systemu operacyjnego, taką jak wątki czy pliki.

Program jest również przygotowany na przetwarzanie równoległe. Zarówno potokowy sposób przetwarzania, jak i niezależność obliczeń dla różnych pikseli sprawiają, że zrównoleglenie obliczeń na komputerze o wielu jednostkach lub na wielu komputerach jest bardzo proste.

Struktura programu nie wymusza konkretnych danych wejściowych czy wyjściowych. Dzięki temu istnieje możliwość wprowadzania scen pochodzących z różnych programów do modelowania – można to uzyskać przez dopisanie odpowiedniego modułu zdolnego do odczytania konkretnego formatu sceny. Generowany obraz można przesłać przez sieć, zapisać do pliku lub utworzyć animację z całej serii obrazów. Możliwe to jest poprzez dodanie odpowiednich modułów.

Schemat struktury programu został przedstawiony i dokładniej opisany w Rozdziale 9.

3 Synteza promieni pierwotnych

3.1 Wprowadzenie

W pierwszym etapie syntezy obrazu oblicza się promienie pierwotne. Kalkulacje te bazują na tzw. geometrii kamery, czyli kształcie obiektu kamery oraz na rozkładzie pikseli w kamerze. Kamera jest obiektem nierenderowalnym, czyli takim, którego nie widać na obrazie końcowym.

Istnieje wiele rodzajów geometrii kamer, w szczególności perspektywiczna (promienie rzutowane są na prostokąt), cylindryczna (promienie rzutowane są na powierzchnię cylindra), kulista (promienie rzutowane są na powierzchnię kuli) oraz inne, głównie wariacje powyższych. Wybrano implementację kamery perspektywicznej ponieważ jest ona najprostsza i najbardziej intuicyjna, jest stosowana we wszystkich programach do modelowania obiektów jako domyślna a inne rodzaje kamer są stosowane jedynie w szczególnych, rzadkich przypadkach.

3.2 Geometria kamery perspektywicznej a widok

Perspektywa, na której bazuje kamera perspektywiczna, opiera się na założeniu, że jeśli połączymy oko obserwatora z wybranym punktem obrazu prostą, to prosta ta wyznaczy nam przecięcie z najbliższym obiektem sceny, którego kolor stanie się kolorem punktu naszego obrazu. W raytracingu zamiast prostej mamy półprostą, której końcem jest punkt oka obserwatora, a półprostą tą nazywamy promieniem pierwotnym.

Głównym elementem kamery jest widok, czyli wirtualny ekran. Widok odzwierciedla punkty generowanego obrazu. Geometria widoku, czyli jego kształt, jest tym samym geometrią kamery. Rozmiary widoku determinują sposób w jaki obserwator widzi obiekty w scenie – im większy widok, tym renderowane obiekty wydają się mniejsze.

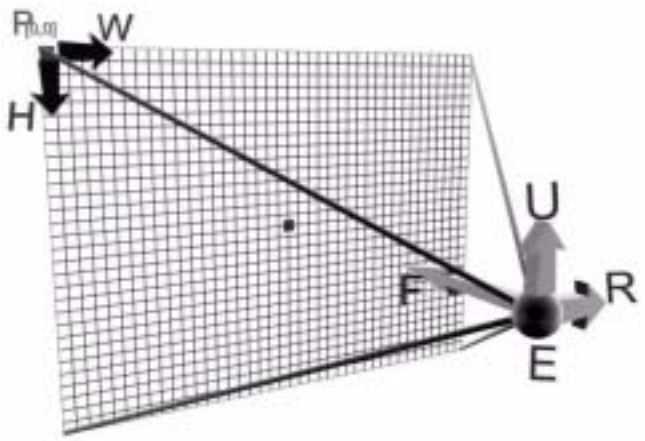
Widok składa się z dyskretnej siatki punktów odpowiadających poszczególnym pikselom generowanego obrazu. W kamerze perspektywicznej widok jest prostokątem, czyli przekształcenie punktów widoku w piksele jest prostą operacją skalowania, gdyż obraz również z definicji jest prostokątem. Skalowanie to może być niejednorodne, gdyż stosunek szerokości do wysokości obrazu nie musi odpowiadać stosunkowi widoku. Punkty należące do widoku będziemy odtąd oznaczali jako $P_{[x,y]}$, gdzie x i y są współrzędnymi odpowiedniego piksela na generowanym obrazie ($x \in [0,w-1]$, $y \in [0,h-1]$, a w i h to szerokość i wysokość obrazu).

Drugim elementem kamery jest oko obserwatora, czyli punkt z którego wychodzą promienie. Punkt ten będziemy oznaczali jako E.

Podczas generowania obrazu dla każdego piksela musimy wyliczyć promień pierwotny. Ponieważ promień jest półprostą określoną przez punkt i wektor, dla każdego piksela musimy zatem znaleźć te dwie wartości z widoku. Dla kamery perspektywicznej punkt zaczepienia O jest punktem wirtualnego oka obserwatora, a wektor kierunku D wektorem z punktu oka obserwatora E do punktu widoku $P_{[x,y]}$ odpowiadającego pikselowi.

$$O = E \qquad \text{(Równ. 3.1)}$$
$$D = \text{Norm}(P_{[x,y]} - E)$$

Wektor D normalizujemy, gdyż później uprości to wiele obliczeń, jak na przykład określanie odległości punktu przecięcia od obserwatora.



Rys. 3.1 – Struktura kamery perspektywicznej

Aby znaleźć punkt $P_{[x,y]}$ widoku odpowiadający pikselowi o współrzędnych [x,y], wystarczy znać punkt w lewym górnym rogu widoku $P_{[0,0]}$ oraz wektory osi widoku W i H. Kierunki wektorów W i H odpowiadają osiom X i Y widoku, natomiast ich długości odległościom odpowiednio w poziomie i w pionie pomiędzy punktami widoku. Tym sposobem nietrudno znaleźć położenia punktów odpowiadających pikselom – wektory W i H tworzą bazę płaszczyzny widoku, a ich długości są jednostkami odpowiadającymi pikselom z obrazu.

$$P_{[x,y]} = P_{[0,0]} + xW + yH \qquad \text{(Równ. 3.2)}$$

Zatem podstawowymi wartościami określającymi kamerę są punkty E i $P_{[0,0]}$ oraz wektory W i H. Dane wejściowe otrzymywane podczas tworzenia kamery nie zawierają tych wartości, z wyjątkiem punktu E. Popularny opis kamery uwzględnia więc punkt obserwatora E, kąt widzenia w poziomie obrazu ϕ, szerokość i wysokość widoku w i h, oraz wektory F i U wytyczające kierunki „przód” i „góra” kamery.

Po pierwsze chcemy przekształcić wektor U w U’ tak, aby był on ortogonalny z wektorem F. Algebra wektorowa podpowiada nam, że możemy uzyskać wektor R wskazujący kierunek „w prawo” kamery, mnożąc wektorowo U przez F:

$$R = \text{Norm}(U \times F) \qquad \text{(Równ. 3.3)}$$

Kolejność mnożenia w powyższym wzorze jest istotna i wynika z lewoskrętności przyjętego układu współrzędnych. Kolejno znajdujemy wektor U’ wskazujący rzeczywistą „górę” kamery i ortogonalny z F i z R:

$$U' = \text{Norm}(F \times R) \qquad \text{(Równ. 3.4)}$$

Normalizacja w powyższych wzorach jest konieczna ze względu na sinus kąta pomiędzy mnożonymi wektorami uwikłany w wynik mnożenia. Kolejno wyliczamy wektory W

i H. Odległość punktów widoku w poziomie jest równa szerokości widoku podzielonej przez szerokość generowanego obrazu w_i. Odległość punktów widoku w pionie wyliczamy analogicznie. W poniższych wzorach w_i i h_i stanowią rozdzielczość generowanego obrazu.

$$W = R\,\frac{w}{w_i} \qquad \text{(Równ. 3.5)}$$
$$H = -U'\,\frac{h}{h_i}$$

Minus we wzorze na H wynika z faktu, iż oś Y w generowanym obrazie jest skierowana w dół, przeciwnie niż w standardowym układzie kartezjańskim.

Aby znaleźć punkt $P_{[0,0]}$, musimy poznać odległość widoku od oka obserwatora. Odległość tą liczymy z cotangensa połowy kąta widzenia ϕ:

$$d = \frac{w}{2}\,\mathrm{arcctg}(\frac{\phi}{2}) \qquad \text{(Równ. 3.6)}$$

Odległość ta pozwala nam znaleźć punkt leżący pośrodku widoku. Wektory $W*w_i/2$ i $H*h_i/2$ wytyczają połowę szerokości i wysokości widoku, zatem odejmując je od punktu środkowego uzyskujemy punkt $P_{[0,0]}$.

$$P_{[0,0]} = E + d\,\mathrm{Norm}(F) - W\,\frac{w_i}{2} - H\,\frac{h_i}{2} \qquad \text{(Równ. 3.7)}$$

3.3 Ogólna implementacja kamery

Przewodnią ideą kamery jest wyliczanie promieni pierwotnych, czyli par [O,D] (punkt zaczepienia promienia i wektor kierunku). Danymi wejściowymi do funkcji wyliczającej promień są współrzędne piksela dla którego promień jest generowany. Bazowa klasa kamery może więc wyglądać tak:

```
class Ray {
public:
     Point O;
     Vector D;
};
class Camera {
public:
     virtual void SetResolution( int x, int y ) = 0;
     virtual void GenerateRay( int x, int y, Ray* p_ray ) = 0;
};
```

Docelowa kamera (np. perspektywiczna) może zatem generować promienie w dowolny sposób. Zawsze elementem wejściowym są współrzędne piksela. Kamera uwzględnia również zmianę rozdzielczości generowanego obrazu.

Wzór wyliczający promień może być dowolny. W szczególności można dodać mapę perturbacji generowanych wektorów D – mapa ta może np. symulować niedokładność wykonania soczewek (dla punktów na brzegu odchylenia mogą być duże a dla punktów centralnych niewielkie).

Konstruktory docelowej klasy kamery mogą uwzględniać różne sposoby tworzenia kamery na podstawie różnych opisów.

4 Metody znajdowania przecięć promieni z obiektami

4.1 Definicja przecięcia i wyliczane wartości

Na potrzeby niniejszej pracy wyselekcjonowano szereg typów obiektów – brył. Wyboru dokonano pod względem zapotrzebowania (dwa rodzaje geometrii: CSG - Rozdział 4.2 i geometria bazująca na trójkątach – Rozdział 4.3), a także pod względem prostoty algorytmu przecięć – wybrane bryły mają powierzchnie co najwyżej drugiego stopnia. Z powodu trudności w dostępie do odpowiednich źródeł, wszystkie wzory wyprowadzono na potrzeby pracy.

Istotą znalezienia przecięcia promienia z obiektem jest znalezienie punktu leżącego zarówno na półprostej promienia jak i na powierzchni obiektu. Jeśli powierzchnia obiektu jest opisana równaniem, znalezienie punktu przecięcia sprowadza się do rozwiązania układu dwóch równań – równania prostej i równania powierzchni.

Równanie wektorowe prostej promienia ma następującą postać:

$$P = O + Dt \qquad \text{(Równ. 4.1)}$$

gdzie P jest punktem leżącym na prostej, O jest punktem zaczepienia promienia (również leżącym na tej prostej), D jest wektorem kierunkowym promienia i jego prostej a t jest zmienną. Ogólnie do opisania dowolnego punktu leżącego na prostej promienia wystarczy skalar t. Ponieważ w Punkcie 3.1 znormalizowaliśmy D (|D|=1), t jest tym samym odległością punktu P od punktu zaczepienia O. Jeśli punkt t leży na półprostej promienia, t jest dodatnie, jeśli na drugiej półprostej – ujemne. Oznacza to, że dla promieni pierwotnych punkty P o wartościach ujemnych t nie są widziane przez obserwatora (są z drugiej strony kamery).

Równanie wektorowe powierzchni obiektu ma następującą postać:

$$P = C + f(t) \qquad \text{(Równ. 4.2)}$$

gdzie C jest punktem centralnym obiektu (punktem na którym zbudowany jest obiekt), a f(t) opisuje geometrię powierzchni – wartością tej funkcji są wektory ($f:[t] \rightarrow [x,y,z]$).

Aby rozwiązań układ dwóch powyższych równań dla szukanego t, możemy napisać równość ich prawych stron:

$$O + Dt = C + f(t) \qquad \text{(Równ. 4.3)}$$

A zatem pozostaje nam do rozwiązania następujące równanie:

$$Dt - f(t) + X = 0 \qquad (X = O - C) \qquad \text{(Równ. 4.4)}$$

Przyjęliśmy X = O – C, co znacznie upraszcza dalsze równania konkretnych obiektów. Dla każdego typu obiektu musimy zatem rozwiązać każde takie równanie (4.4) ze względu na t, a następnie wyliczyć P z Równania 4.1.

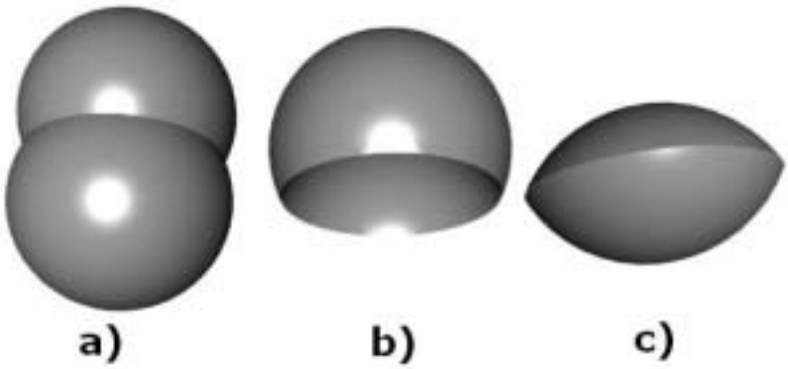
Oprócz znalezienia punktu przecięcia P, musimy jeszcze znaleźć wektor normalny N. Wektor normalny jest to wektor jednostkowy (|N|=1) prostopadły do powierzchni obiektu lokalnie w punkcie P. Wektor ten jest wyliczany inaczej dla każdego typu obiektu, gdyż jego wartość zależy od geometrii powierzchni.

W kolejnych sekcjach wyprowadzono rozwiązania Równania 4.4 dla konkretnych typów obiektów. W literaturze można znaleźć jedynie skąpe, ogólnikowe opisy tych algorytmów, a na potrzeby programu konieczne było ich rozwinięcie. Docelowymi rozwiązaniami równań są wartości t określające punkty przecięcia promienia z powierzchniami obiektów oraz wektory normalne N w tych punktach.

4.2 Znajdowanie przecięć z obiektami CSG

4.2.1 Omówienie idei CSG

Założeniem CSG, czyli konstrukcyjnej geometrii bryłowej (*ang. Constructive Solid Geometry*), jest budowanie obiektów z mniejszych, ściśle określonych „klocków". Do dyspozycji mamy szereg prostych brył, takich jak kula, cylinder czy sześcian. Obiekty te możemy ze sobą składać przy pomocy tzw. operacji boolowskich, które odpowiadają typowym działaniom na zbiorach – suma, różnica i przecięcie. W rezultacie operacji boolowskich uzyskujemy powierzchnię pokrywającą kształt który jest rezultatem działań na objętości obiektów.



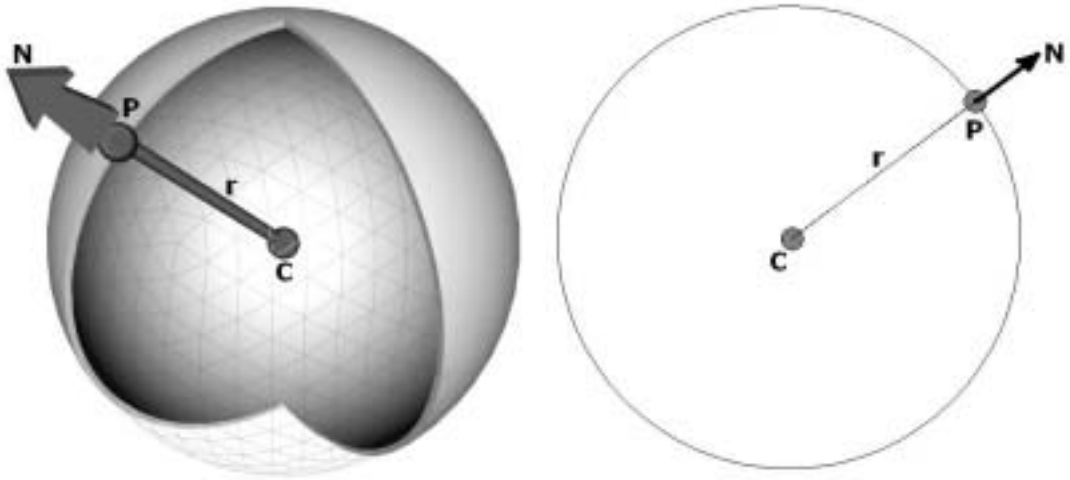
Rys. 4.1 – Przykład obiektów CSG złożonych z dwóch kul:
a) suma, b) różnica, c) przecięcie

Niewątpliwą zaletą CSG jest prostota definiowania obiektów. Wiele obiektów ze świata rzeczywistego można opisać przy pomocy złożenia prostych brył. Możemy również powiększać dostępny zestaw brył bazowych o bardziej skomplikowane, takie jak np. torus (obiekt o kształcie obwarzanka) czy blob (płynne „stopienie" kilku brył, np. kul). Oczywiście im bardziej skomplikowany obiekt, tym bardziej skomplikowane jego równanie a co za tym idzie rozwiązanie równania przecięcia staje się trudniejsze.

Wprowadzanie obiektów CSG do programu renderującego wymaga zdefiniowania specjalnego formatu wejściowego ze względu na specyfikę tego rozwiązania. Scena jest więc hierarchiczną listą obiektów prostych i złożonych. Obiektem prostym jest każda bryła prosta. Obiektami złożonymi są zbiory obiektów (prostych lub złożonych) związanych operacjami boolowskimi.

4.2.2 Kula

Kula jest jedną z najprostszych brył prostych, ponieważ ma równanie powierzchni zaledwie drugiego stopnia. Obiekt ten równie dobrze można by nazwać sferą, jako że definiujemy jedynie jego dwuwymiarową powierzchnię.



Rys. 4.2 – Kula w trzech i dwóch wymiarach

Powierzchnię kuli można opisać jako kształt, którego każdy punkt leży w takiej samej odległości od centrum kuli, a odległość ta jest równa promieniowi. Warunek ten można zapisać w formie tzw. równania kuli:

$$\forall_{P \in \Pi} |P - C| = r \qquad \text{(Równ. 4.5)}$$

gdzie P jest punktem na powierzchni kuli Π, C jest centrum kuli a r promieniem. Szukając punktu przecięcia promienia z kulą zauważamy, że szukany punkt P z Równania 4.1 leży właśnie na powierzchni kuli opisanej Równaniem 4.5. Równanie 4.5 możemy przekształcić do bardziej przydatnej formy, podnosząc obie strony równania do kwadratu. Moduł do kwadratu z wektora jest iloczynem skalarnym wektora przez siebie, zatem:

$$(P - C) \circ (P - C) = r^2$$
$$(P - C)^2 = r^2$$

Do powyższego równania podstawiamy P z Równania 4.1:

$$(O + Dt - C)^2 = r^2$$
$$(Dt + X)^2 = r^2 \qquad (X = O - C)$$

Ostatecznie korzystając z algebry wektorów otrzymujemy równanie kwadratowe dla t:

$$(D \circ D)t^2 + 2(D \circ X)t + (X \circ X) - r^2 = 0 \qquad \text{(Równ. 4.6)}$$

Jak widzimy współczynniki Równania kwadratowego 4.6 wynoszą odpowiednio:

$$\begin{cases} a = D \circ D \\ \frac{b}{2} = D \circ X \\ c = X \circ X - r^2 \end{cases} \qquad \text{(Równ. 4.7)}$$

Dla uproszczenia możemy przyjąć a=1, ponieważ założyliśmy, że normalizujemy wektor kierunkowy promienia D (D◦D=1). Kolejnym poczynionym uproszczeniem jest wyliczenie b/2 – dzięki temu unikamy dodatkowych mnożeń w ostatecznym algorytmie przecięcia. Rozwiązanie trójmianu jest zatem następujące:

$$\frac{\Delta^2}{4} = (\frac{b}{2})^2 - ac \qquad \text{gdzie } \frac{\Delta^2}{4} \geq 0$$

$$\begin{cases} t_1 = \dfrac{-\frac{b}{2} - \sqrt{\frac{\Delta^2}{4}}}{a} \\ t_2 = \dfrac{-\frac{b}{2} + \sqrt{\frac{\Delta^2}{4}}}{a} \end{cases} \qquad \text{(Równ. 4.8)}$$

Ostateczne równania dla t wymagają wykonania kosztownej obliczeniowo operacji pierwiastkowania. Możemy jednak zawczasu określić, czy w ogóle trafiliśmy w kulę, poprzez sprawdzenie znaku wyliczonego współczynnika $\Delta^2/4$. Jeśli jest on ujemny, promień ominął kulę (jego prosta nie przecina kuli).

Ponieważ sfera jest powierzchnią kwadratową, jeśli promień przecina ją, najczęściej istnieją dwa punkty przecięcia, dlatego wyliczamy dwa pierwiastki Równania 4.6 t_1 i t_2, przy czym ze sposobu ich wyliczania w Równaniu 4.8 wynika $t_1<t_2$.

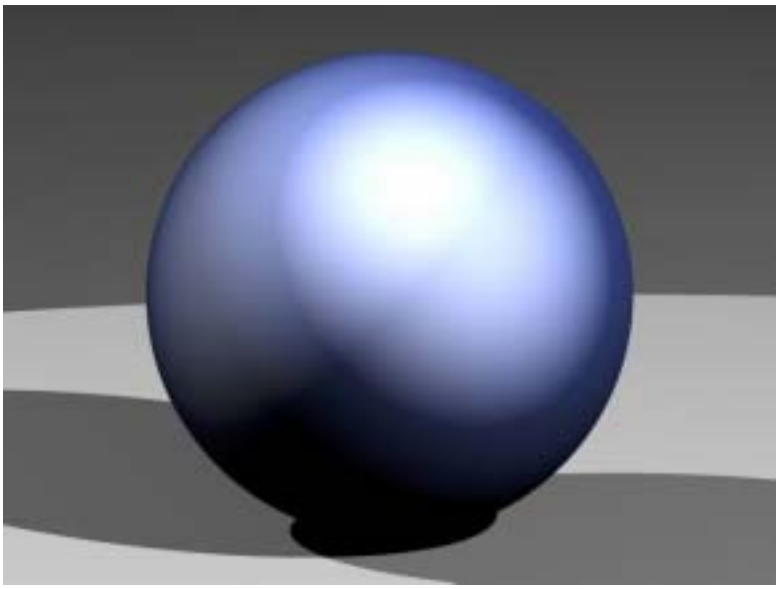
Musimy jeszcze sprawdzić czy trafiona kula znajduje się po właściwej stronie kamery – tu wystarczy warunek $t_2>\varepsilon$, gdzie $\varepsilon \rightarrow 0^+$. Jeśli t_1 znajduje się po przeciwnej stronie kamery, sprawdzimy to później. Taki sposób postępowania upraszcza algorytm w przypadku operacji boolowskich, co zobaczymy w sekcjach 4.2.8-4.2.10.

Wektor normalny kuli też nie jest trudny do wyliczenia. Zakładając że wyliczyliśmy już trafiony punkt P podstawiając t do Równania 4.1, wystarczy zauważyć, że wektor normalny jest niejako przedłużeniem promienia. A zatem:

$$N = \text{Norm}(P - C)$$

ale ponieważ długość wektora P–C jest równa promieniowi zgodnie z Równaniem 4.5, możemy zapisać:

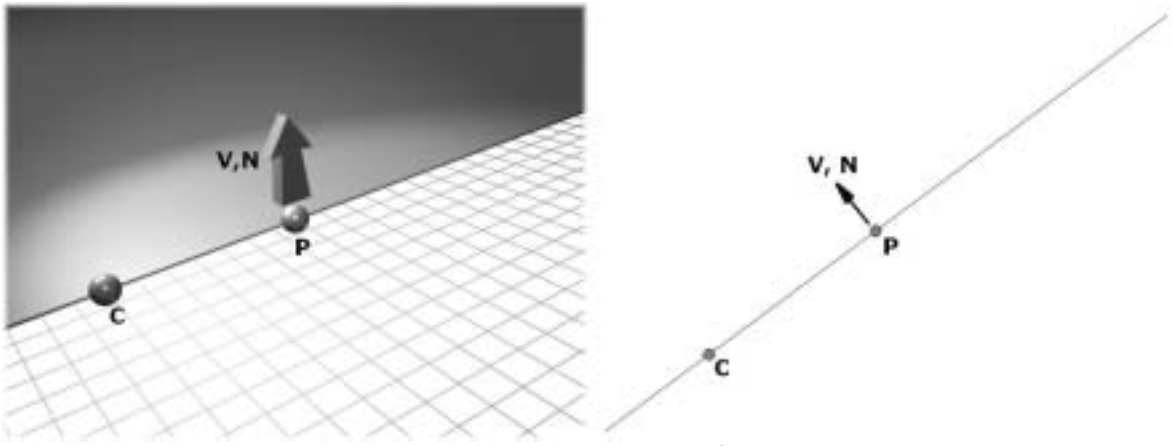
$$N = \frac{P - C}{r} \qquad \text{(Równ. 4.9)}$$



Rys. 4.3 – Wyrenderowana kula

4.2.3 Płaszczyzna

Płaszczyzna nie jest obiektem bezpośrednio używanym w CSG, ponieważ nie jest zamknięta i nie okala objętości. Jest jednak używana w innych prostych obiektach, takich jak sześcian czy obcięty cylinder.



Rys. 4.4 – Płaszczyzna w trzech i dwóch wymiarach

Płaszczyznę można opisać za pomocą jednego punktu leżącego na niej i jej wektora normalnego V, czyli wektora jednostkowego ortogonalnego do niej. Wybrany punkt jest wtedy punktem zaczepienia płaszczyzny, czyli punktem centralnym C. Jeśli zatem weźmiemy dowolny punkt z płaszczyzny i przesuniemy go tak, jakby płaszczyzna była zaczepiona w punkcie [0,0,0], utworzymy wektor P-C, który jest ortogonalny do V.

$$\bigvee_{P \in \Pi} (P - C) \circ V = 0 \qquad \text{(Równ. 4.10)}$$

Równanie 4.10 definiuje zatem punkty należące do płaszczyzny – warunek na iloczyn skalarny równy 0 jest warunkiem prostopadłości wektorów. Jeśli do Równania 4.10 wstawimy P z Równania 4.1 i uprościmy, otrzymamy równanie liniowe:

$$(O + Dt - C) \circ V = 0$$
$$(Dt + X) \circ V = 0$$
$$(D \circ V)t + (X \circ V) = 0 \qquad \text{(Równ. 4.11)}$$

Po wykonaniu prostych przekształceń ostatecznie otrzymujemy wzór na t:

$$t = -\frac{X \circ V}{D \circ V} \qquad \text{gdzie } D \circ V \neq 0 \qquad \text{(Równ. 4.12)}$$

Zauważmy, że faktycznie prosta promienia przecina się z płaszczyzną zawsze gdy wektor kierunkowy promienia D nie jest równoległy do płaszczyzny, czyli nie jest ortogonalny do wektora normalnego płaszczyzny V – stąd warunek D◦V≠0 zapobiegający jednocześnie dzieleniu przez zero.

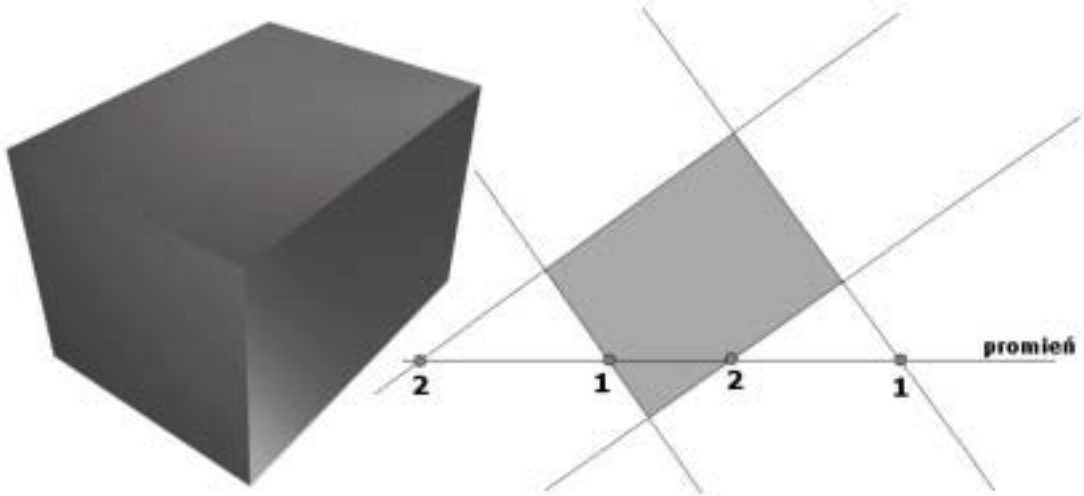
Ponieważ chcemy aby znalezione przecięcie znajdowało się na półprostej promienia, czyli po widocznej stronie obserwatora, musimy upewnić się, że t>0. Warunek ten możemy spełnić wykonując jedno porównanie zanim jeszcze wykonamy dzielenie:

$$\text{sgn}(X \circ V) \neq \text{sgn}(D \circ V) \qquad \text{(Równ. 4.13)}$$

Wektor normalny N jest taki sam w każdym punkcie płaszczyzny i jest równy V.

$$N = V \qquad \text{(Równ. 4.14)}$$

4.2.4 Prostopadłościan



Rys. 4.5 – Przecięcie z prostopadłościanem (1,2 – numery płyt)

Analogicznie do algorytmu przedstawionego w [9] zaproponowano algorytm przecięcia promienia z dowolnym prostopadłościanem.

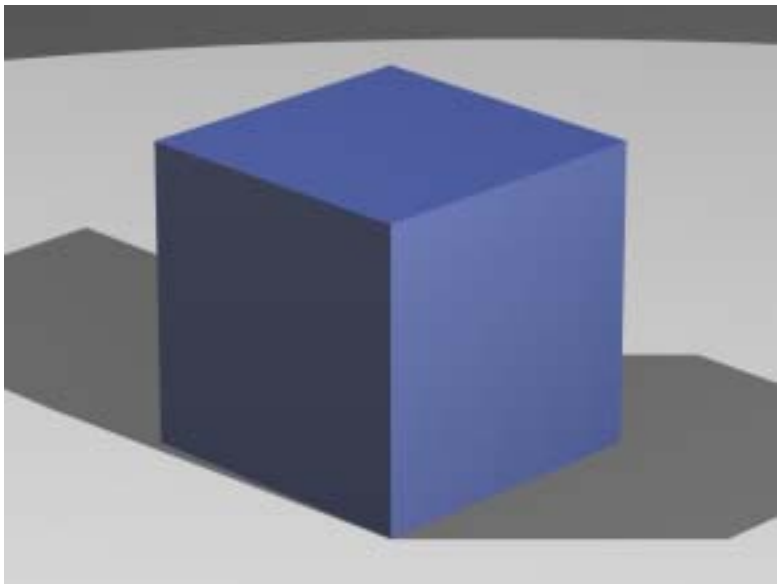
Prostopadłościan otrzymujemy z przecięcia trzech płyt (*ang. slabs*), z których każda ograniczona jest dwoma równoległymi płaszczyznami. W pierwszej kolejności wyliczamy

punkty przecięcia z kolejnymi płytami. Znalezione odcinki leżące na prostej promienia muszą się wzajemnie przecinać, w przeciwnym razie nie trafiliśmy w prostopadłościan.

Podczas wyliczania przecięć z kolejnymi płytami, sprawdzamy czy płyta nie leży za obserwatorem ($t_1<t_2\leq\varepsilon$). Jeśli kolejna płyta rzeczywiście jest niewidoczna dla obserwatora, oznacza to że na pewno prostopadłościan również jest niewidoczny i możemy zaniechać dalszych obliczeń.

Poniższa funkcja *IntersectBox()* demonstruje w jaki sposób znajdujemy przecięcia z prostopadłościanem. W celu optymalizacji wykorzystywane są funkcje *fmin* i *fmax*, znajdujące odpowiednio minimum i maksimum z dwóch wartości. Funkcja zwraca *false* jeśli nie istnieje przecięcie z prostopadłościanem. Ponadto funkcja wykorzystuje mniejszą funkcję *Slab::Intersect()* która znajduje dwa przecięcia z płytą.

```
// Płyty
Slab s1, s2, s3;

bool Intersect( Point O, Vector D, double* p_t1, double* p_t2 )
{
     // Przecięcie z pierwszą płytą
     double t1, t2;
     s1.Intersect( O, D, &t1, &t2 );
     if ( t2 < epsilon ) return false;

     // Przecięcie z drugą płytą
     double u1, u2;
     s2.Intersect( O, D, &u1, &u2 );
     if ( u2 < epsilon ) return false;
     t1 = fmax( t1, u1 );
     t2 = fmin( t2, u2 );
     if ( t1 >= t2 ) return false;

     // Przecięcie z trzecią płytą
     s3.Intersect( O, D, &u1, &u2 );
     if ( u2 < epsilon ) return false;
     t1 = fmax( t1, u1 );
     t2 = fmin( t2, u2 );
     if ( t1 >= t2 ) return false;

     // Zwrócenie wyniku
     *p_t1 = t1, *p_t2 = t2;
     return true;
}
```



Rys. 4.6 – Wyrenderowany sześcian

4.2.5 Cylinder

Cylinder ma powierzchnię drugiego stopnia, tak jak kula. Obiekt ten jest definiowany za pomocą punktu początkowego C (punktu podstawy cylindra), wysokości h, wektora osi V (|V|=1) oraz promienia r.

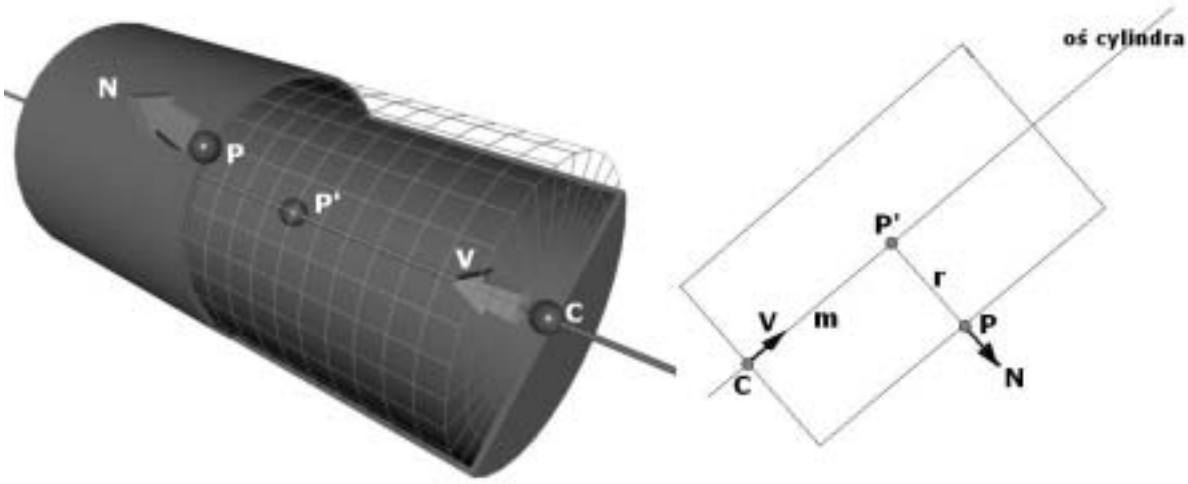
Dla cylindra nieskończonego dowolny punkt P leży na jego powierzchni wtedy i tylko wtedy, gdy jest odległy od osi cylindra o jego promień r. Innymi słowy, jeśli rzutujemy punkt P na oś cylindra zdefiniowaną parą (C,V), odległość P od jego rzutu P' będzie równa promieniowi r.

$$\bigvee_{P \in \Pi} | P - P' | = r \qquad \text{(Równ. 4.15)}$$

$$P' = C + Vm \qquad \text{(Równ. 4.16)}$$

gdzie P jest punktem leżącym na powierzchni cylindra Π, P' jest rzutem punktu P na oś cylindra, r jest promieniem cylindra, C jest punktem podstawy cylindra i jednocześnie punktem zaczepienia jego osi, V wektorem osi a m jest skalarem określającym odległość P' od C.

Dodatkowym plusem z zastosowania takiej formy w Równaniu 4.15 jest późniejsza możliwość obcięcia cylindra dwoma płaszczyznami, ponieważ punkt P leży na obciętym cylindrze wtedy i tylko wtedy gdy 0<m<h.



Rys. 4.7 – Cylinder w trzech i dwóch wymiarach

Jeśli chcemy znaleźć przecięcie promienia z cylindrem, musimy zacząć od wyliczenia skalara m. Zauważmy, że ponieważ punkt P’ wjest rzutem punktu P na prostą (oś cylindra), to wektor P-P’ jest prostopadły do wektora osi V.

$$(P - P') \circ V = 0 \qquad \text{(Równ. 4.17)}$$

Podstawmy P z Równania 4.1 i P’ z Równania 4.16:

$$(O + Dt - C - Vm) \circ V = 0$$

Teraz rozdzielmy sumę w nawiasie względem mnożenia przez V:

$$(O + Dt - C) \circ V - (V \circ V)m = 0$$
$$(V \circ V)m = (Dt + O - C) \circ V$$

Zgodnie z założeniem Równania 4.4, że X=O-C, oraz z faktem że V jest wektorem jednostkowym mamy ostatecznie:

$$m = (D \circ V)t + (X \circ V) \qquad \text{(Równ. 4.18)}$$

Oznacza to, że m jest rozwiązaniem równania liniowego ze względu na t.

Podstawmy uzyskane m do Równania 4.15:

$$| P - P' | = r$$
$$| O + Dt - C - Vm | = r$$
$$| Dt + O - C - V((D \circ V)t + (X \circ V)) | = r$$
$$| (D - V(D \circ V))t + X - V(X \circ V) | = r$$

Jeśli podstawimy p=D-V(D◦V) i q=X-V(X◦V) oraz podniesiemy obie strony równania do kwadratu, otrzymamy:

$$(pt - q)^2 = r^2$$
$$p^2t^2 - 2pqt + q^2 - r^2 = 0 \qquad \text{(Równ. 4.19)}$$

A zatem ostatecznie mamy równanie kwadratowe. Po podstawieniu odpowiednio p i q i wykonaniu elementarnych przekształceń otrzymamy równanie kwadratowe o następujących współczynnikach:

$$\begin{cases} a = D \circ D - (D \circ V)^2 \\ \frac{b}{2} = D \circ X - (X \circ V)(D \circ V) \\ c = X \circ X - (X \circ V)^2 - r^2 \end{cases} \qquad \text{(Równ. 4.20)}$$

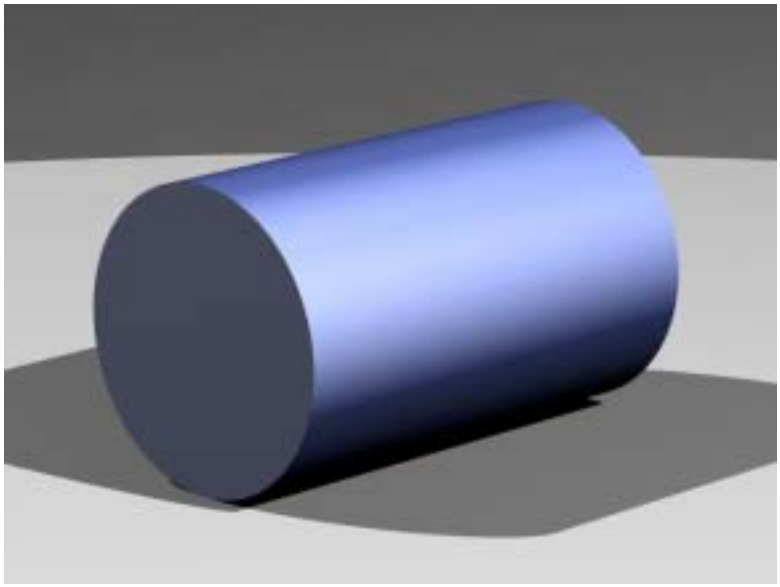
Pozostałe obliczenia wykonujemy zgodnie z Równaniem 4.8, tak jak dla kuli.

Mając wyliczone t_1 i t_2 z Równania 4.19, wyliczamy odpowiednio m_1 i m_2. Wartości te są konieczne do znalezienia wektora normalnego N w trafionym punkcie P oraz do ograniczenia cylindra.

Jeśli cylinder jest ograniczony, wartości m muszą znajdować się w przedziale od 0 (podstawa cylindra) do h (wysokość cylindra). Jeśli $m_1<0$ i $m_2<0$ albo $m_1>h$ i $m_2>h$, obcięty cylinder nie został trafiony. W przeciwnym razie, jeśli m_1 lub m_2 jest poza zakresem, odpowiadające mu t zastępujemy wartością dla płaszczyzny ograniczającej, tzn. wyliczamy z Równania 4.12, jako że D◦V i X◦V już mamy obliczone.

Wektor normalny N jest równy Norm(P-P'), ale biorąc pod uwagę Równania 4.15 i 4.16, mamy:

$$N = \frac{P - C - Vm}{r} \qquad \text{(Równ. 4.21)}$$



Rys. 4.8 – Wyrenderowany cylinder

4.2.6 Stożek

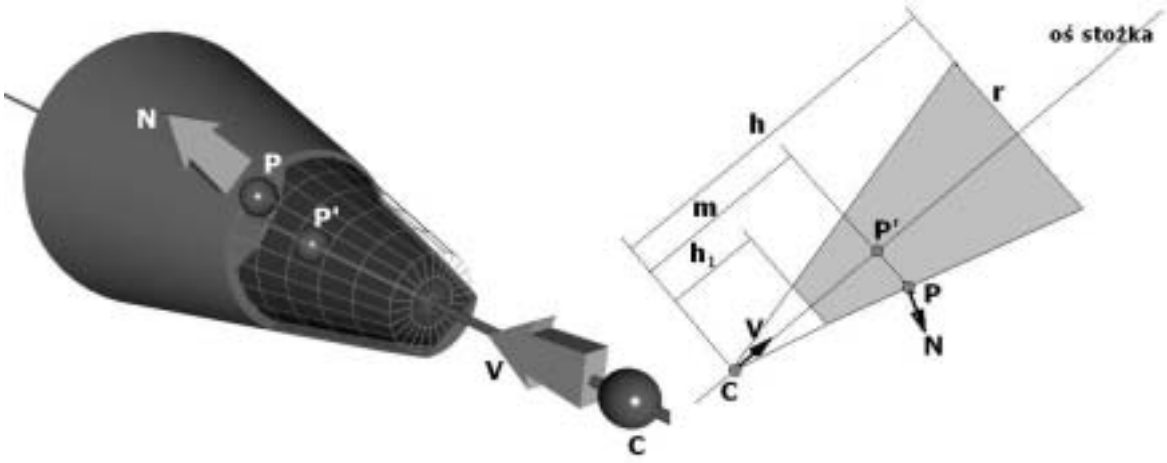
Stożek jest pewną uogólnioną formą cylindra. Przyjmując, że punkt zaczepienia stożka C jest jednocześnie jego wierzchołkiem, a h i r są odpowiednio jego wysokością i promieniem podstawy, oraz V jest wektorem wskazującym od C do podstawy i prostopadłym do podstawy (|V|=1), nietrudno zauważyć następującą zależność:

$$\forall_{P \in \Pi} \frac{|P - P'|}{m} = k \qquad \text{gdzie } k = \frac{r}{h} \qquad \text{(Równ. 4.22)}$$

co oznacza, że stosunek lokalnego promienia po rzucie punktu P do odległości rzutu od punktu zaczepienia stożka jest równy stosunkowi promienia podstawy do wysokości stożka (z twierdzenia Talesa). Ze względu na podobieństwo stożka do cylindra, P' opisane jest Równaniem 4.16, oraz m znajdujemy z Równania 4.18. Dla uproszczenia działań przyjmujemy stałą k=r/h.

Po pomnożeniu obu stron Równania 4.22 przez m, oraz podstawieniu P' z Równania 4.16 mamy:

$$|O + Dt - C - Vm| = km$$
$$(Dt + X - Vm)^2 = k^2 m^2$$



Rys. 4.9 – Stożek w trzech i dwóch wymiarach

Nietrudno udowodnić, że po podstawieniu m z Równania 4.18 i wykonaniu elementarnych przekształceń otrzymamy równanie kwadratowe ze względu na t o następujących współczynnikach:

$$\begin{cases} a = (D \circ D) - (1 + k^2)(D \circ V)^2 \\ \frac{b}{2} = (D \circ X) - (1 + k^2)(D \circ V)(X \circ V) \\ c = (X \circ X) - (1 + k^2)(X \circ V)^2 \end{cases} \qquad \text{(Równ. 4.24)}$$

Równanie kwadratowe o powyższych współczynnikach rozwiązujemy analogicznie jak dla kuli czy cylindra.

Stożek, tak jak cylinder, może być albo otwarty (nieskończony), albo zamknięty (obcięty płaszczyznami prostopadłymi do osi obiektu). Ograniczenie stożka wykonujemy identycznie jak dla cylindra, przy czym możemy przyjąć stożek ścięty, taki dla którego $h_1<m<h$, gdzie h jest właściwą wysokością stożka, a h_1 jest wysokością na której został ścięty.

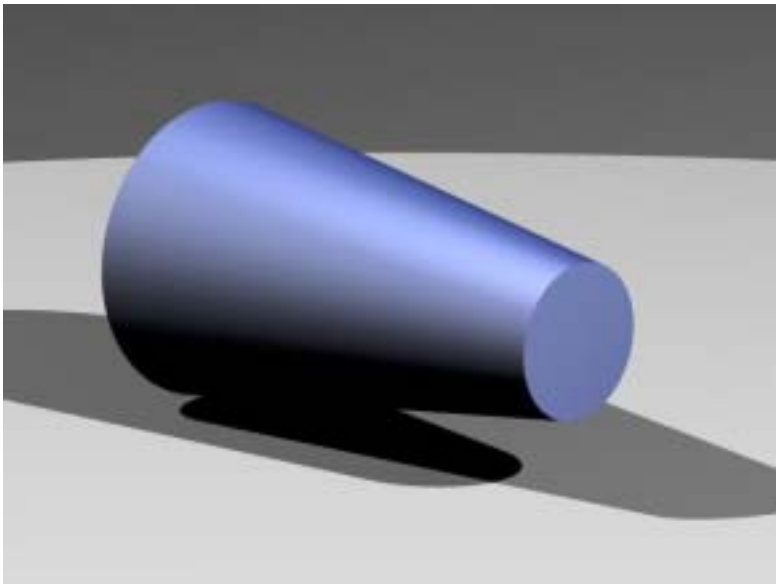
Wektor normalny jest wektorem prostopadłym do powierzchni stożka w punkcie P, czyli wektora P-C, oraz leży na płaszczyźnie wyznaczonej punktami P, P' i C. Nietrudno

wykazać, że do wektora P-P’ wystarczy dodać –a*V, gdzie stosunek długości P-P’ do a musi być taki jak odwrotność stosunku promienia podstawy do wysokości, czyli 1/k. A zatem:

$$\frac{|P - P'|}{a} = \frac{1}{k} \Rightarrow a = mk^2$$

Ostatecznie otrzymujemy:

$$N = \text{Norm}(P - C - (1 + k^2)Vm) \qquad \text{(Równ. 4.24)}$$

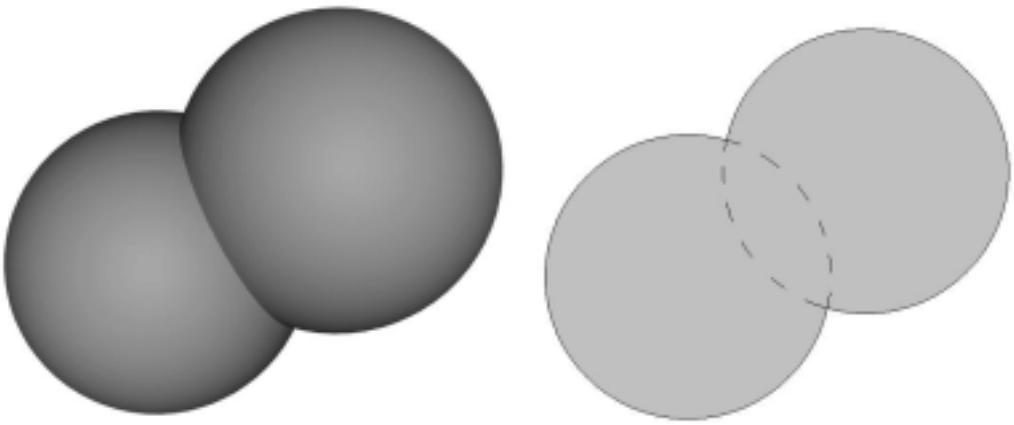


Rys. 4.10 – Wyrenderowany stożek

4.2.7 Suma Boolowska

Obiekty Boolowskie w przeciwieństwie do obiektów prostych wymagają podejścia algorytmicznego zamiast matematycznego. Składają się one bowiem z pod-obiektów będących zarówno obiektami prostymi jak i Boolowskimi.

Dla uproszczenia przyjmujemy, że jeden obiekt Boolowski składa się z dokładnie dwóch pod-obiektów. Aby znaleźć przecięcie promienia z obiektem Boolowskim najpierw znajdujemy wszystkie przecięcia z jego pod-obiektami. Każdy z pod-obiektów musi być zamknięty, czyli generuje parzystą liczbę przecięć. Przecięcia o parzystych indeksach (przecięcia są numerowane od 0) są otwarciami obiektu a przecięcia o indeksach nieparzystych są zamknięciami obiektu. Innymi słowy, przecięcia możemy pogrupować w pary opisujące odcinki, z których każdy ma otwarcie i zamknięcie obiektu, czyli właściwa objętość obiektu znajduje się pomiędzy przecięciami. W każdej liście przecięć może być dowolna liczba odcinków, przy czym przecięcia są posortowane rosnąco. Wszystkie przecięcia w liście przecięć, oprócz pierwszego (o indeksie 0), są większe od ε.



Rys. 4.11 – Suma Boolowska dwóch kul w trzech i dwóch wymiarach

W przypadku sumy, mając do dyspozycji dwa ciągi przecięć z pod-obiektami, musimy wygenerować sumaryczny ciąg. Ciąg wyjściowy musi spełniać takie same warunki jak ciągi wejściowe – musi być posortowany rosnąco. Odcinki nie przecinające się z innymi po prostu kopiujemy, natomiast odcinki które się przecinają musimy złączyć.

Następujący pseudokod prezentuje sposób postępowania w przypadku generowania sumy Boolowskiej przecięć. Zmienne a i b są wskaźnikami na bieżące przecięcia (na początku wskazują na początki tablic) a zmienne offa i offb wskazują na pierwsze elementy za końcami tablic. Zmienna dest wskazuje na tablicę wynikową.

```
// Przeszukujemy tablice do skutku
while ( a < offa && b < offb ) {

     // Skopiowanie odcinków B poprzedzających bieżący A
     while ( *(b+1) < *a ) {
          skopiuj b[0] i b[1] do dest
          if ( b >= offb ) skopiuj pozostałe a do dest i zakończ
     }

     // Skopiowanie odcinków A poprzedzających bieżący B
     while ( *(a+1) < *b ) {
          skopiuj a[0] i a[1] do dest
          if ( a >= offa ) skopiuj pozostałe b do dest i zakończ
     }

     // Łączenie odcinków przecinających się
     if ( odcinek( a[0], a[1] ) przecina odcinek( b[0], b[1] ) ) {

          // Otwarcie złączonego odcinka
          skopiuj mniejszy( a[0], b[0] ) do dest

          // Znalezienie końca złączonego odcinka
          for (;;) {

               // Pierwszy kończy się odcinek B
               if ( *(b+1) < *(a+1) ) {
                    jeśli następny odcinek B nie przecina się a A
                         skopiuj a[1] do dest i przerwij pętlę
               }
```

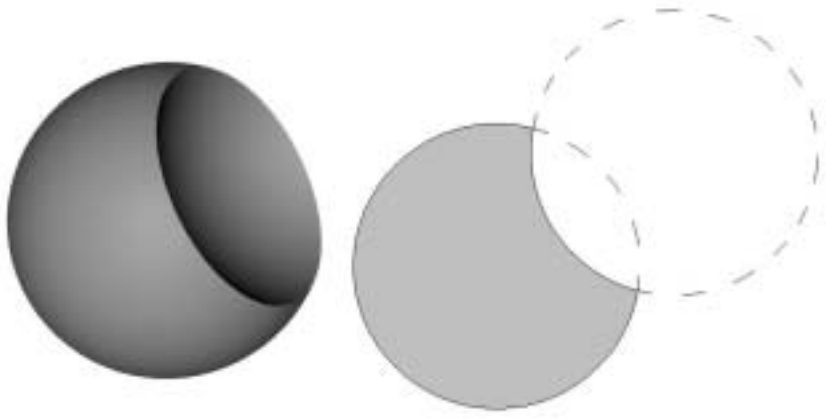
```
                    // Pierwszy kończy się odcinek A
                    else if ( *(a+1) < *(b+1) ) {
                         jeśli następny odcinek A nie przecina się z B
                              skopiuj b[1] do dest i przerwij pętlę
                    }

                    // Brak przecięcia
                    else {
                         skopiuj a[1] do dest
                         pomiń bieżący b
                         przerwij pętlę
                    }
               }
          }
     }

     skopiuj pozostałe a lub b do dest i zakończ
```

W powyższym uproszczonym pseudokodzie dla uproszczenia pominięto kilka warunków końcowych algorytmu. Jak widać najbardziej skomplikowane jest łączenie długich ciągów przecinających się odcinków.

Warto zauważyć, że do tablicy wynikowej musimy kopiować po dwa przecięcia – pierwsze otwierające i drugie zamykające.

4.2.8 Różnica Boolowska

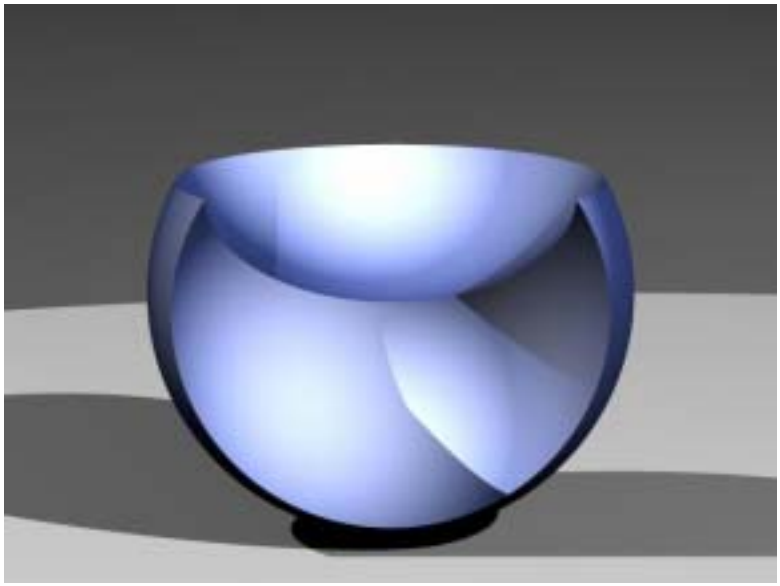
Różnica Boolowska opiera się na tych samych założeniach co suma, jednak jest ideowo nieco prostsza. Wszystkie warunki dla różnicy są takie same jak dla sumy w Punkcie 4.2.7.



Rys. 4.12 – Różnica Boolowska dwóch kul w trzech i dwóch wymiarach

W różnicy Boolowskiej przyjmujemy pierwszy z pod-obiektów (A) za „dodatni”, a drugi (B) za „ujemny”. Musimy skrócić wszystkie odcinki promienia pochodzące od obiektu A tak, aby nie pokrywały się z odcinkami od obiektu B. W algebrze zbiorów różnicę definiuje się jako zbiór który zawiera wszystkie elementy ze zbioru A takie, których nie ma w zbiorze B. I dokładnie tym kierujemy się tworząc algorytm przecięcia.

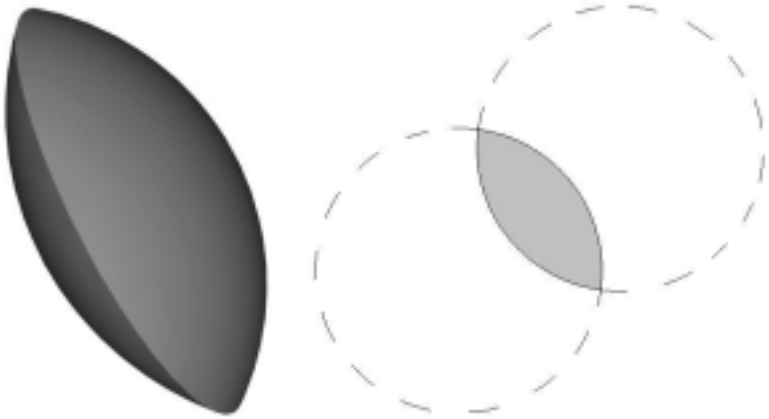
```
// Przeglądamy sekwencję A
for ( ; a < offa; a += 2 ) {

     // Pomijamy odcinki B nie mające wpływu na A
     while ( *(b+1) <= *a ) {
          b += 2;
          jeśli nie ma więcej B – skopiuj resztę A i zakończ
     }

     // Nie ma przecięcia
     if ( *(a+1) < *b )
          skopiuj a[0] i a[1] do dest

     // Przecięcie B z A wystąpiło
     else {

          // Został odcięty koniec A
          if ( *a < *b && *b > ε )
               skopiuj a[0] i b[0] do dest

          // Skopiowanie części A z pominięciem B
          while ( *(a+1) > *(b+1) ) {
               skopiuj b[1] do dest
               skopiuj mniejszy z (a[1], b[0]) do dest
          }
     }
}
```



Rys. 4.13 – Wyrenderowana różnica kul

4.2.9 Przecięcie Boolowskie

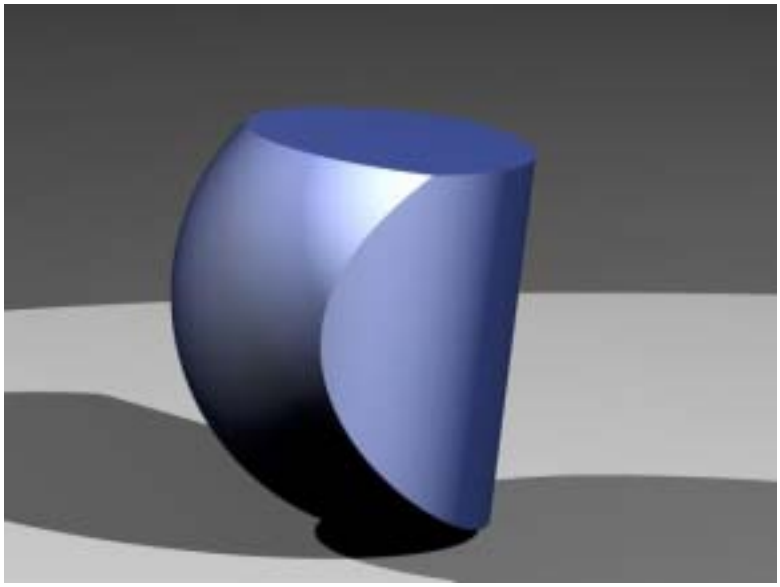
Dla przecięcia Boolowskiego, podobnie jak dla różnicy, stosujemy te same założenia co dla sumy. Tak jak różnica, przecięcie jest ideowo prostsze od sumy.



Rys. 4.14 – Przecięcie Boolowskie dwóch kul w trzech i dwóch wymiarach

W algebrze zbiorów przecięcie jest zbiorem w którym znajdują się wszystkie elementy zawierające się w obu zbiorach źródłowych. To samo dotyczy przecięcia Boolowskiego – tylko te fragmenty odcinków z pod-obiektu A które pokrywają się odcinkami z pod-obiektu B mogą znaleźć się w wynikowym zbiorze odcinków. Dokładnie na tym opiera się algorytm przedstawiony poniższym pseudokodem.

```
// Przeglądamy sekwencję A
for ( ; a < offa; a += 2 ) {

     // Pominięcie odcinków B które nie pokrywają żadnych A
     while ( *(b+1) <= *a )
          b += 2;
          jeśli nie ma więcej B – skopiuj resztę A i zakończ
     }

     // Odcinek z A ważny tylko gdy przecina się z B
     if ( *(a+1) > *b ) {

          // Zapisanie przecięcia
          kopiuj większy z (a[0], b[0]) do dest
          kopiuj mniejszy z (a[1], b[1]) do dest

          // Wzięcie pod uwagę pozostałych odcinków z B
          while ( *(b+1) <= *(a+1) ) {
               pomiń poprzedni odcinek z B
               if ( *b < *(a+1) ) {
                    kopiuj b[0] do dest
                    kopiuj minejszy z a[1], b[1] do dest
               }
          }
     }
}
```



Rys. 4.15 – Wyrenderowane przecięcie cylindra z kulą

4.3 Znajdowanie przecięć z obiektami zbudowanymi z trójkątów

4.3.1 Omówienie podejścia opartego na trójkątach

Idea CSG jest bardzo interesująca i pozwala na robienie rozmaitych ciekawych obiektów. Niestety przy pomocy CSG nie można symulować bardziej skomplikowanych kształtów. Tu z pomocą przychodzi geometria oparta na trójkątach.

Obecnie wszystkie najpopularniejsze programy do modelowania obiektów 3D generują obiekty zbudowane z trójkątów. Edytory te pozwalają na dowolne kształtowanie powierzchni z ograniczoną, lecz określoną dokładnością. Użytkownik może tworzyć zarówno powierzchnie B-sklejane jak i modyfikować położenia wierzchołków ręcznie.

Obiekty zbudowane z trójkątów, zwane z angielska meshami, są w istocie siatkami trójkątów. Każdy obiekt składa się z zestawu punktów-wierzchołków, na których rozpięte są trójkąty. Ponadto każdy wierzchołek ma przyporządkowany wektor normalny. Wektory normalne są interpolowane na trójkątach, doskonale symulując własności wypukłej lub wklęsłej powierzchni. W przypadku obiektów o rzadkiej siatce trójkątów, zazwyczaj można poznać że dany obiekt zbudowany jest z trójkątów jedynie obserwując jego krawędzie. Jeśli jednak siatka trójkątów jest gęsta, nie sposób odróżnić takiego obiektu od jego modelowego odpowiednika.

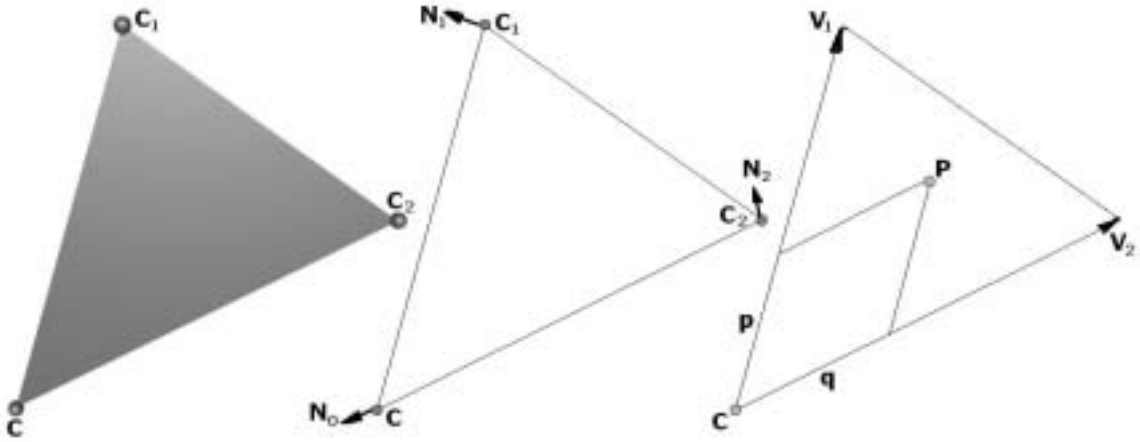
Jedyną wadą takiego podejścia, w stosunku do CSG, jest jego nadzwyczaj duża złożoność. Podczas gdy np. kulę można w CSG zasymulować dokładnie jednym obiektem, kula oparta na trójkątach może zawierać od 300 do 900 trójkątów, a nawet więcej.

Na szczęście obecne techniki, opisane w Rozdziale 5, znacznie redukują ilość obliczeń nawet dla tak dużych ilości obiektów. Zastosowanie jednorodnego opisu sceny, to jest przy użyciu tylko jednego typu obiektów – trójkątów, ma również dodatkowe zalety. Upraszcza się w ten sposób kod programu, a jednocześnie można stosować nowe efekty, nie możliwe przy bardziej skomplikowanych i różnorodnych obiektach.

4.3.2 Przecięcie promienia z trójkątem

Aby znaleźć przecięcie promienia z trójkątem, przyda się nam wiedza z Punktu 4.2.3 dotycząca przecięć z płaszczyzną. Trójkąt jest bowiem obiektem leżącym na płaszczyźnie. Tak więc najpierw musimy trafić w jego płaszczyznę, a następnie ograniczyć potencjalny obszar do samego trójkąta.

Trójkąt standardowo opisany jest przy pomocy trzech wierzchołków – C, C_1 i C_2. Opis ten możemy przekształcić, zapisując trójkąt jako jeden punkt C, będący „głównym" wierzchołkiem, oraz dwa wektory $V_1=C_1-C$ i $V_2=C_2-C$. V_1 i V_2 są wektorami krawędziowymi trójkąta i razem z punktem C wytyczają proste na których leżą te krawędzie. Wektor normalny płaszczyzny na której leży trójkąt jest iloczynem wektorowym tych dwóch wektorów $V=Norm(V_1xV_2)$.



Rys. 4.16 – Trójkąt w trzech i dwóch wymiarach

Przecięcia z płaszczyzną trójkąta dokonujemy zgodnie z Równaniem 4.12. Uzyskujemy w ten sposób skalar t. Następnie musimy sprawdzić, czy rzeczywiście trafiliśmy w trójkąt.

Zgodnie z powyższym opisem trójkąta, wektory V_1 i V_2 wytyczają bazę trójkąta na płaszczyźnie. A zatem punkt P, który już znamy, możemy opisać następującym równaniem:

$$P = C + V_1 p + V_2 q \qquad \text{(Równ. 4.25)}$$

Jeśli rozpiszemy powyższe równanie, otrzymamy równanie macierzowe:

$$V_1 p + V_2 q - Dt = O - C = X$$
$$\begin{bmatrix} V_1.x & V_2.x & -D.x \\ V_1.y & V_2.y & -D.y \\ V_1.z & V_2.z & -D.z \end{bmatrix} \bullet \begin{bmatrix} p \\ q \\ t \end{bmatrix} = \begin{bmatrix} X.x \\ X.y \\ X.z \end{bmatrix} \qquad \text{(Równ. 4.26)}$$

Znalezienie wartości p i q dla Równania 4.25 sprowadza się do policzenia odwrotności macierzy z Równania 4.26. Jako że t już znamy z przecięcia z płaszczyzną trójkąta, możemy uprościć Równanie 4.26.

$$\begin{bmatrix} V_1.x\, V_2.x \\ V_1.y\, V_2.y \\ V_1.z\, V_2.z \end{bmatrix} \bullet \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} P.x \\ P.y \\ P.z \end{bmatrix}$$

Ponieważ powyższy układ równań ma więcej równań niż niewiadomych, możemy je uprościć usuwając ostatnie linie:

$$\begin{bmatrix} V_1.x & V_2.x \\ V_1.y & V_2.y \end{bmatrix} \bullet \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} P.x \\ P.y \end{bmatrix} \qquad \text{(Równ. 4.27)}$$

Ostatecznie skalary p i q otrzymujemy rozwiązując Równanie 4.27. Odwrotność macierzy z tegoż równania możemy znaleźć w etapie prekalkulacji, gdyż jest ona stała dla danego trójkąta.

Skoro V_1 i V_2 są bazą trójkąta, nietrudno wykazać, że punkt P opisany Równaniem 4.25 leży wewnątrz trójkąta, jeśli spełnione są następujące trzy warunki:

$$\begin{cases} 0 \le p \le 1 \\ 0 \le q \le 1 \\ 0 \le p + q \le 1 \end{cases} \qquad \text{(Równ. 4.28)}$$

Jeśli wszystkie z powyższych warunków trzymają, oznacza to że trafiony przez nas punkt leży wewnątrz trójkąta.

Po trafieniu trójkąta, musimy jeszcze wyliczyć jego wektor normalny w trafionym punkcie. Wektor normalny jest interpolowany z wektorów krawędziowych. W wierzchołku C wektor normalny jest równy N_0, w C_1 jest równy N_1 a w C_2 - N_2. Można te zależności opisać następującymi równaniami:

$$\begin{cases} N = N_0 & , p = 0 \wedge q = 0 \\ N = N_1 & , p = 1 \wedge q = 0 \\ N = N_2 & , p = 0 \wedge q = 1 \end{cases}$$

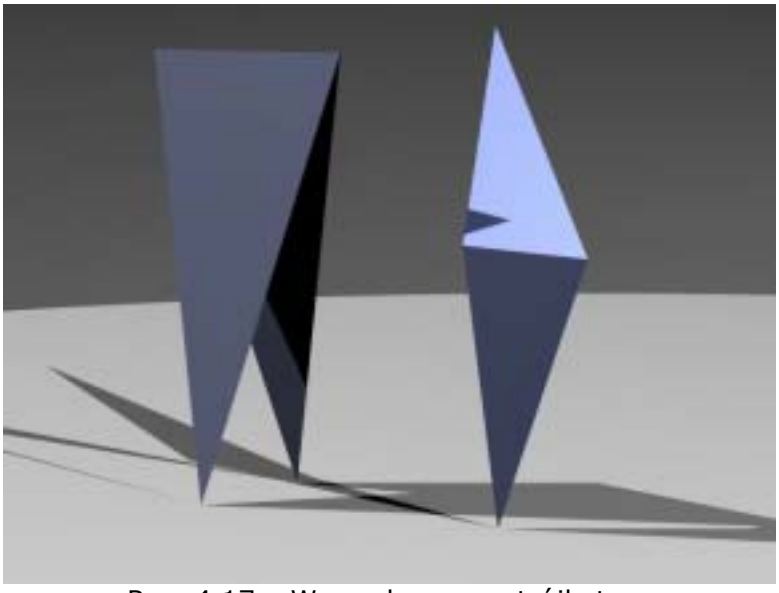
Ponadto na krawędziach trójkąta wektor normalny przyjmuje następujące wartości interpolowane:

$$\begin{cases} N = \mathrm{Norm}\left(\frac{N_0 + N_1}{2}\right) & , p = \frac{1}{2} \wedge q = 0 \\ N = \mathrm{Norm}\left(\frac{N_0 + N_2}{2}\right) & , p = 0 \wedge q = \frac{1}{2} \\ N = \mathrm{Norm}\left(\frac{N_1 + N_2}{2}\right) & , p = \frac{1}{2} \wedge q = \frac{1}{2} \end{cases}$$

Okazuje się, że najprostszym sposobem na zaspokojenie wszystkich sześciu wymagań jest następujące równanie interpolowanego wektora normalnego trójkąta:

$$N = \mathrm{Norm}\left((1 - p - q)N_0 + pN_1 + qN_2\right) \qquad \text{(Równ. 4.29)}$$

Powyższy sposób interpolacji również doskonale sprawdza się w przypadku interpolacji współrzędnych tekstury omówionych w Punkcie 6.2.4.



Rys. 4.17 – Wyrenderowane trójkąty

4.3.3 Obiekty Mesh

Trójkąty rzadko występują samotnie. Zazwyczaj budują większe obiekty, tzw. Meshe, będące siatkami trójkątów. Typowy obiekt typu Mesh może zawierać od kilkudziesięciu do kilku tysięcy trójkątów.

Obiekty Mesh nie są bezpośrednio renderowane. Zamiast trzymać trójkąty w hierarchii, przyporządkowane do Mesha, dodajemy je do sceny jako samodzielne obiekty. W ten sposób wykorzystujemy algorytmy optymalizacji liczby przecięć opisane w Rozdziale 5.

Warto dodać, że ponieważ w scenie mogą występować miliony trójkątów, a nawet ilości większego rzędu, musimy w jakiś sposób zredukować pamięć potrzebną na pamiętanie wszystkich danych. Ponieważ i tak dane wejściowe mają ograniczoną dokładność, wszystkie liczby dotyczące trójkątów (punkty wierzchołków, wektory normalne, itp.) możemy pamiętać jako liczby zmiennoprzecinkowe pojedynczej precyzji.

5 Optymalizacja ilości przecięć pojedynczego promienia

5.1 Cel optymalizacji ilości przecięć

Typowy renderowany obraz może zawierać setki obiektów, z których każdy może składać się z setek tysięcy pod-obiektów. Podczas renderowania generowane są setki milionów promieni, zatem przecięcie każdego promienia z każdym obiektem w celu wyznaczenia najbliższego przecinanego obiektu dla promienia jest czasowo nieopłacalne. Dlatego potrzebne są sposoby zmniejszania ilości przecięć dla pojedynczego promienia.

Spośród różnych algorytmów redukcji ilości przecięć promienia z obiektami [8], [12], [14], na potrzeby niniejszej pracy wybrano metodę drzew BSP. Przy wyborze kierowano się prostotą algorytmu. Jeśli chodzi o efektywność, to zgodnie z [9] trudno jest stwierdzić czy inne wydajne metody (takie jak metoda przestrzeni wokseli) są bardziej czy mniej efektywne. Zatem zdecydowano się na BSP, ponieważ jest to relatywnie prosta, intuicyjna metoda.

5.2 Metoda drzew BSP – podstawy budowy struktury

BSP oznacza Binarne Partycjonowanie Przestrzeni (*ang. Binary Space Partitioning*). Drzewo BSP jest drzewiastą strukturą hierarchiczną, której każdy węzeł zawiera typowo dwa węzły-dzieci.

Idea drzew BSP opiera się na strukturze zwanej drzewem binarnym, pozwalającej na szybkie przeszukiwanie posortowanego zbioru elementów. W przypadku drzew BSP używanych w raytracingu, sortowanie dotyczy grupowania elementów w przestrzeni pod względem zawierania się w objętości węzłów drzewa.

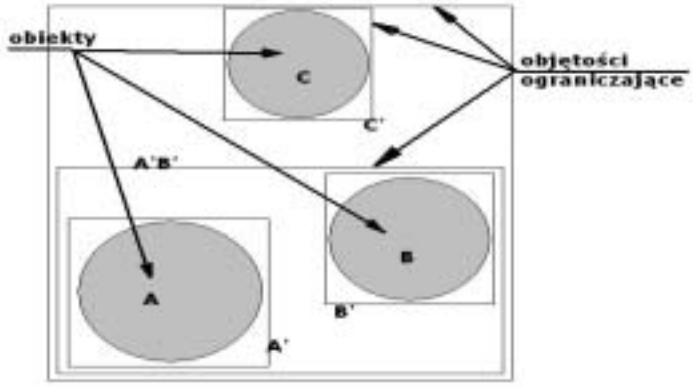
Własnością drzewa binarnego jest to, że aby znaleźć szukany element w zbiorze N elementowym wystarczy $\log_2 N$ porównań. W raytracingu liczba ta musi zostać pomnożona przez liczbę węzłów-dzieci, ponieważ dla każdego węzła musimy znaleźć to dziecko z którym przecina się promień. Tak więc na przykład dla sceny zawierającej milion obiektów mamy jedynie około 40 przecięć. Liczba ta może być w rzeczywistości kilkakrotnie wyższa ze względu na potrzeby przeszukiwania wielu liści drzewa, natomiast i tak jest nieporównywalnie mniejsza od samej liczby obiektów w scenie.

Budowanie drzew BSP sprowadza się do problemu dodawania pojedynczego obiektu do istniejącego drzewa. Dodawanie obiektów do drzewa jest ściśle związane ze sposobem jego przeszukiwania, który z kolei zależy od wybranej strategii przeszukiwania. Jest to o tyle istotne, że dobór strategii ma znaczny wpływ na późniejszą ilość przecięć promieni z obiektami. Wybrana strategia została opisana w punkcie 5.2.3.

5.3 Przecięcia z sześcianami ograniczającymi

Zanim przejdziemy do właściwych algorytmów budowy i przeglądania drzew BSP, musimy jeszcze ustalić jaką formę przyjmą węzły drzewa. Węzły muszą być przestrzennym uogólnieniem wszystkich obiektów które będą „obejmować". Dobór typów obiektów

rezydujących w węzłach musi być podyktowany szybkością obliczania przecięć z nimi, łatwością znajdowania minimalnych wymiarów okalających różne typy obiektów, a także ilością nadmiarowej przestrzeni dodawanej do obiektów.



Rys. 5.1 – Przykład hierarchii objętości ograniczających

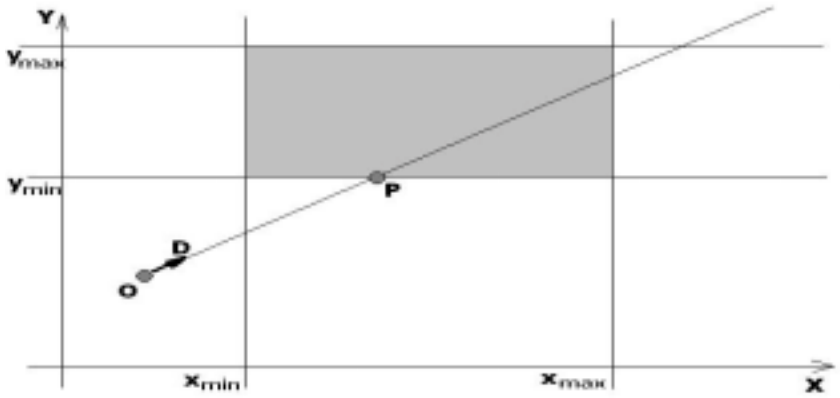
Najpopularniejszymi objętościami ograniczającymi (*ang. bounding volumes*), czyli typami obiektów węzłowych, są sfery i sześciany (prostopadłościany). Sześciany mogą być dodatkowo wyrównane do osi (tzn. mieć wszystkie krawędzie równoległe do odpowiednich osi układu współrzędnych). Wybrano wyrównane do osi sześciany (*ang. axis-aligned bounding boxes*), ze względu na ich lepsze możliwości optymalizacji pod względem szybkości przecięć, a także prostotę budowy i składowania danych (dokładne kryteria przedstawione w Tabeli 5.1).

Tab. 5.1 – Ocena różnych typów objętości ograniczających

typ objętości	wielkość danych	ilość instrukcji algorytmu przecięcia			
		dodaw.	mnoż.	dziel.	pierw.
sfera	4	13	11	1	1
sześcian	18	36	33	3	0
sześcian wyrównany	6	6	6	0	0

W powyższej tabeli nie przedstawiono ilości instrukcji porównań. Dla wykonuje się jedno porównanie w celu stwierdzenia, czy została ona trafiona przez promień. Jeśli nie, to nie wykonuje się ani dzielenia ani pierwiastka. Nadal jednak pozostaje duża liczba wykonanych dodawań i mnożeń. Dla zwykłego sześcianu wykonuje się w trakcie obliczeń sześć porównań. Ponieważ jednak na dzisiejszych procesorach porównania są nieefektywne (długi potok rozkazu, jednostka przewidywań skoków nie może wnioskować na podstawie danych), rozwiązanie to nie jest optymalne. Najlepszym rozwiązaniem jest sześcian wyrównany do osi – wymaga najmniejszej liczby elementarnych operacji, a dzielenie przez składową wektora promienia D można zastąpić mnożeniem przez odwrotność wyliczoną na etapie prekalkulacji. Rozwiązanie to nie wymaga w ogóle porównań, jedynie operacji min i max (np. warunkowe wykonywanie rozkazów - predykacja).

Sześcian ograniczający składowany jest w postaci sześciu liczb, z których każda określa minimalną i maksymalną pozycję na odpowiedniej osi. Pozycje te są współrzędnymi płaszczyzn na których leżą boki sześcianu. Budowanie sześcianów ograniczających jest bardzo proste. Np. dla obiektów zbudowanych z trójkątów wystarczy znaleźć minimalne i maksymalne wartości dla każdej osi spośród wszystkich wierzchołków danego obiektu.



Rys. 5.2 – Wyrównany sześcian ograniczający w dwóch wymiarach

Przecięcie sześcianu ograniczającego jest bardzo zbliżone do przecięcia ze zwykłym sześcianem (Punkt 4.2.4). Jako że sześcian jest częścią wspólną trzech „płyt" ograniczonych płaszczyznami, musimy przeciąć poszczególne płaszczyzny a następnie wyznaczyć minimum i maksimum na promieniu. Przecięcie zostało opisane Równaniem 5.1, przy założeniu że sześcian składa się z sześciu płaszczyzn o współrzędnych xmin, xmax, ymin, ymax, zmin, zmax.

$$\begin{cases} t_{xmin} = (xmin - O.x) / D.x \\ t_{xmax} = (xmax - O.x) / D.x \\ t_{ymin} = (ymin - O.y) / D.y \\ t_{ymax} = (ymax - O.y) / D.y \\ t_{zmin} = (zmin - O.z) / D.z \\ t_{zmax} = (zmax - O.z) / D.z \end{cases} \qquad \text{(Równ. 5.1)}$$

$$\begin{cases} t_{min} = \max(t_{xmin}, t_{ymin}, t_{zmin}) \\ t_{max} = \min(t_{xmax}, t_{ymax}, t_{zmax}) \end{cases}$$

Wyliczone wartości t_{min} i t_{max} stanowią minimalny i minimalny punkt przecięcia (zgodnie ze wzorem 4.1). Jeśli $t_{min}t \geq t_{max}$ to sześcian nie został trafiony.

Powyższy wzór jest analogiczny do wzoru z [9], jednak na potrzeby niniejszej pracy został uproszczony operacjami *min* i *max*, które pozwalają na wykorzystanie szczególnych właściwości współczesnych procesorów, takich jak rozkazy *min* i *max*, czy też wykonanie warunkowe lub predykacja.

5.4 Algorytm budowy i przeglądania drzew BSP oparty na heurystyce

Dobór kryterium porównawczego użytego przy budowie drzew BSP ma znaczący wpływ na ilość porównań wykonanych podczas przeglądania drzewa, czyli poszukiwania najbliższego przecięcia. Zdecydowano się na heurystykę bazującą na prawdopodobieństwie trafienia obiektu.

Jako że każdy węzeł jest wyrównanym do osi sześcianem (prostopadłościanem), pokrywa pewną objętość. Kiedy dodajemy nowy obiekt do węzła, musimy wybrać, do którego dziecka go dodamy. Wybieramy dziecko, które po powiększeniu tak aby okalało

dodany obiekt, będzie miało mniejsze prawdopodobieństwo trafienia przez dowolny promień. Zakładamy, że prawdopodobieństwo trafienia danego obiektu jest wprost proporcjonalne do powierzchni obiektu.

Następujący pseudokod pokazuje w jaki sposób dodajemy obiekt do drzewa BSP.

```
DodajObiekt( węzeł, obiekt )
{
     dopóki węzeł nie jest liściem {
          dla wszystkich dzieci węzła oblicz nową powierzchnię
          węzeł = dziecko o min powierzchni po dodaniu obiektu
     }
     węzeł = nowy węzeł zawierający ( węzeł.obiekt, obiekt )
}
```

Powyższa funkcja zawiera algorytm iteracyjny dodawania obiektu do drzewa, którego korzeń jest przekazany do niej jako *węzeł*. Funkcja schodzi w dół drzewa aż do liścia, przy czym przy wyborze kierunku schodzenia w danym węźle kieruje się tzw. minimalnym kosztem dodania obiektu do węzła – w naszym przypadku wybiera węzeł-dziecko o mniejszej powierzchni po dodaniu obiektu.

Po zbudowaniu drzewa BSP możemy z niego korzystać szukając przecięć z obiektami. Następujący pseudokod pokazuje algorytm przeglądania drzewa.

```
ZnajdźPrecięcie( promień, korzeń )
{
     kolejka priorytetowa ( priorytet = odległość )
     odległość = przecięcie( promień, korzeń )
     dodaj do kolejki ( odległość, korzeń )
     najlepsza_odl = nieskończoność
     najbliższy_obj = 0
     dopóki kolejka niepusta {
          węzeł = pobierz z kolejki węzeł o min odległości
          jeśli najlepsza_odl < odległość węzła
               przerwij pętlę
          jeśli węzeł jest liściem {
               odległość = przecięcie( promień, węzeł.obiekt )
               jeśli odległość < najlepsza_odl {
                    najlepsza_odl = odległość
                    najbliższy_obj = węzeł.obiekt
               }
          } w przeciwnym razie {
               dla każdego dziecka węzła {
                    odległość = przecięcie( promień, dziecko )
                    jeśli odległość < najlepsza_odl
                         dodaj do kolejki( odległość, dziecko )
               }
          }
     }
     zwróć najbliższy_obj
}
```

Powyższa funkcja używa kolejki priorytetowej do składowania węzłów, które promień przecina i które są bliższe niż najbliższy dotychczas przecięty obiekt. Z każdym przeciętym obiektem dozwolona odległość przecięcia zmniejsza się, więc węzły i obiekty które leżą dalej od ostatniego przecięcia są pomijane. Pozwala to na zadanie maksymalnej dopuszczalnej odległości na wejściu do funkcji – można to wykorzystać w sytuacjach gdy wiemy, że promień nie przecina pewnych obszarów sceny, albo dla promieni cieni (Punkt 7.5).

6 Wyznaczanie koloru powierzchni

6.1 Omówienie idei tekstur

Podczas tworzenia syntetycznych obrazów istotną rolę odgrywa kolor. W raytracingu ważne jest aby móc określić jaki kolor ma powierzchnia trafionego przez promień obiektu. Najprostszym sposobem jest przypisanie koloru do całego obiektu na stałe. Jednak aby osiągnąć podstawowy stopień realizmu trzeba uwzględnić różnorodność kolorystyczną powierzchni pojedynczego obiektu.

Rozwiązaniem problemu różnorodności kolorystycznej powierzchni obiektów są tekstury. Tekstura w grafice komputerowej jest bitmapą symulującą powierzchnię. Jako że bitmapa jest powierzchnią dwuwymiarową, dla każdego punktu dowolnego teksturowanego obiektu musi istnieć przekształcenie z trójwymiarowej przestrzeni obiektu do dwuwymiarowej przestrzeni bitmapy. Przekształcenie to nosi miano mapowania tekstury.

W kolejnych punktach niniejszego rozdziału zajmiemy się mapowaniem współrzędnych tekstur aby móc oglądać kolorowe obiekty na renderowanych obrazach. Istnieje kilka różnych metod mapowania współrzędnych, niezależnych od typów obiektów dla których są używane. Jedynie obiekty zbudowane z trójkątów, ze względu na swą specyfikę, mają własną metodę mapowania współrzędnych.

Dla uproszczenia algorytmu zakładamy, że tekstury są obrazami kwadratowymi o długości boku będącej potęgą dwójki.

Współrzędne tekstury w danym punkcie P zapisuje się jako [u,v].

Opisane algorytmy stworzono na potrzeby niniejszej pracy analogicznie do algorytmów teksturowania używanych w bitmapowych algorytmach syntezy obrazu (jak np. w OpenGL).

6.2 Wyznaczanie współrzędnych tekstury

6.2.1 Współrzędne sferyczne

Mapowanie współrzędnych z powierzchni kuli na prostokąt lub kwadrat bitmapy opiera się na założeniu, że współrzędnymi [u,v] stają się kąty pomiędzy wektorem promienia kuli oraz wektorami bazowymi kuli. Można to określić jako mapowanie kątowe. W związku z tym, prawa i lewa krawędź bitmapy, która jest teksturą, pokrywają się na obiekcie lub przylegają do siebie, natomiast górna i dolna krawędź zwijają się do pojedynczego piksela.

Z obiektem, którego powierzchnię teksturujemy, związujemy tzw. obiekt mapujący, który potrafi przekształcić współrzędne punktu P z przestrzeni sceny [x,y,z] do przestrzeni bitmapy [u,v]. Dla współrzędnych sferycznych obiekt ten ma centrum C oraz wektory Bk wskazujący na tył obiektu i Up wskazujący na górę obiektu.

Współrzędną v, która na bitmapie jest współrzędną y, wyliczamy jako kąt pomiędzy wektorem Up a promieniem (P-C), gdzie P jest trafionym punktem na powierzchni obiektu do którego jest przypisana tekstura i obiekt mapujący.

$$v = \frac{1}{\pi}\operatorname{acos}\left(\frac{Up \circ (P - C)}{|P - C|}\right) \qquad \text{(Równ. 6.1)}$$

Jak widać, w powyższym równaniu wyliczamy współrzędną v, która przyjmuje wartości z zakresu [0,1]. Wystarczy pomnożyć v przez wysokość bitmapy, aby uzyskać współrzędną y piksela na tej bitmapie.

Operację arcus cosinus można stablicować ze skończoną dokładnością, ponieważ bitmapa sama ma skończoną dokładność.

Współrzędną u (na bitmapie x) wyliczamy jako kąt pomiędzy rzutem wektora (P-C) na płaszczyznę w której leży wektor Bk (wektorem normalnym tej płaszczyzny jest Up) a wektorem Bk.

$$\begin{aligned} P' &= (P - C) - Up(Up \circ (P - C)) \\ u &= \frac{1}{\pi}\operatorname{acos}\left(\frac{Bk \circ P'}{|P'|}\right) \end{aligned} \qquad \text{(Równ. 6.2)}$$

Wyliczone u dotyczy tylko połowy bitmapy. Musimy jeszcze stwierdzić po której stronie kuli u leży – możemy to zrobić sprawdzając znak iloczynu skalarnego pomiędzy wektorem P' a wektorem Rt, który wskazuje prawą stronę kuli.

6.2.2 Współrzędne planarne

Mapowanie współrzędnych na płaszczyznę opiera się na rzucie wektora P-C na tą płaszczyznę. Płaszczyzna, która jest obiektem mapującym, określona jest przez punkt zaczepienia C, wektor normalny V oraz dwa wektory V_X i V_Y definiujące osie X i Y tekstury.

$$\begin{aligned} P' &= (P - C) - V(V \circ (P - C)) \\ &\begin{cases} u = \frac{1}{w} P' \circ V_X \\ v = \frac{1}{h} P' \circ V_Y \end{cases} \end{aligned} \qquad \text{(Równ. 6.3)}$$

W powyższym równaniu w i h są współczynnikami skalowania tekstury odpowiednio w osiach X i Y.

Po wyliczeniu u i v musimy jeszcze uwzględnić zawijanie tekstury. Jeśli płaszczyzna jest większa niż mapowana tekstura, przy obliczaniu współrzędnych bitmapy musimy podzielić je modulo długość odpowiedniego boku tekstury – jeśli długości boków są potęgami dwójki jest to prosta operacja logiczna *and*.

6.2.3 Współrzędne cylindryczne

Mapowanie współrzędnej u na cylinder jest takie jak dla kuli. Natomiast współrzędna v jest rzutem wektora (P-C) na oś cylindra. Cylinder określony jest przez punkt zaczepienia C i wektor osi V.

$$v = \frac{1}{h} V \circ (P - C) \qquad \text{(Równ. 6.4)}$$

Tak jak w przypadku współrzędnych planarnych, h jest współczynnikiem skalowania w osi Y tekstury. Musimy również zadbać o zawijanie v, to jest podzielić je modulo dwa przez długość boku tesktury.

6.2.4 Współrzędne dla trójkątów

W przypadku obiektów zbudowanych z trójkątów współrzędne tekstur wyliczamy w inny sposób. Każdy wierzchołek w takim obiekcie zawiera bowiem własne współrzędne tekstury, które musimy interpolować po trójkątach. Interpolacji dokonujemy w identyczny sposób jak interpolowaliśmy wektor normalny trójkąta w Punkcie 4.3.2.

$$T = T_0(1 - p - q) + T_1 p + T_2 q \qquad \text{(Równ. 6.5)}$$

Wektory T w powyższym wzorze oznaczają współrzędne tekstury T=[u,v]. T_0, T_1 i T_2 to współrzędne tekstury w odpowiednich wierzchołkach trójkąta.

7 Materiały powierzchni i oświetlenie

7.1 Wprowadzenie pojęcia materiału

Opisane w Rozdziale 6 tekstury służą generowaniu koloru powierzchni obiektów, zwanego też pigmentem. Kolor jest podstawową własnością powierzchni która jednak ulega modyfikacji pod wpływem światła pochodzącego z otoczenia obiektów.

Materiał jest drugą cechą podstawową obiektów, obok tekstury. Określa on sposób interakcji powierzchni obiektów ze światłem, w jaki sposób światło modyfikuje kolor powierzchni, jak się od niej odbija i jak przez nią przenika.

W niniejszym rozdziale opiszemy symulowane efekty świetlne zachodzące na powierzchni obiektów oraz rodzaje obiektów generujących światło.

7.2 Model cieniowania Phonga

Cieniowanie jest sposobem odbijania światła przez powierzchnię pod różnymi kątami. Ilość odbitego światła pochodzącego od określonego źródła w określonym kierunku zależy bowiem od kilku czynników.

Phong [2] zaproponował podstawowy, uproszczony model cieniowania. W modelu tym, z powierzchnią obiektu oddziaływają trzy składowe oświetlenia: rozproszona (*ambient*), dyfuzyjna (*diffuse*) i bezpośrednia (*specular*). Składowa *ambient* pochodzi od światła równomiernie rozproszonego w całej scenie, składowe *diffuse* i *specular* pochodzą od konkretnego źródła światła, przy czym *diffuse* to światło rozproszone na powierzchni obiektu z powodu jej fraktury, a *specular* to światło bezpośrednio odbite, które zaznacza na powierzchni charakterystyczne podświetlenie (*highlight spots*).

Obiekt lub sam materiał może zawierać osobne kolory dla osobnych składowych modelu Phonga. Każda składowa może mieć trzy osobne wartości dla osobnych składowych koloru czerwonej, zielonej i niebieskiej.

Składową *ambient*, która dotyczy wszystkich świateł w scenie, mnożymy przez kolor *ambient* obiektu i dodajemy do koloru wynikowego.

$$ambient = K_A \bullet A \qquad \text{(Równ. 7.1)}$$

gdzie K_A jest kolorem *ambient* powierzchni obiektu (może być równa K_D z Równania 7.2) a A jest współczynnikiem pochodzącym od wszystkich źródeł światła w scenie. A może być skalarem lub składać się z trzech liczb dla oddzielnych składowych koloru.

Składowa *diffuse* jest liczona osobno dla każdego światła i dodawana do koloru wynikowego. Dla pojedynczego światła zależy ona wyłącznie od kąta padania światła na punkt na powierzchni obiektu, ponieważ po odbiciu światło dyfuzyjne jest rozproszone równomiernie we wszystkich kierunkach. Jeśli światło pada pod kątem 0 stopni, składowa dyfuzyjna jest równa 0. Jeśli natomiast światło pada prostopadle do powierzchni, składowa ta przyjmuje wartość maksymalną. Dlatego jest ona wprost proporcjonalna do kosinusa

kąta pomiędzy wektorem padania światła –L a wektorem normalnym powierzchni N w trafionym przez promień punkcie P obiektu (ponieważ |L|=|N|=1 to jest to iloczyn skalarny tych dwóch wektorów).

$$diffuse = \sum_i K_D \bullet D_i(-L_i \circ N) \qquad \text{(Równ. 7.2)}$$

W powyższym równaniu sumujemy składowe pochodzące od kolejnych źródeł światła i. K_D jest kolorem dyfuzyjnym powierzchni obiektu, D_i jest współczynnikiem dyfuzyjnym i-tego źródła światła dla kąta padania 90 stopni, L_i jest znormalizowanym wektorem od źródła światła do powierzchni obiektu w trafionym punkcie P, czyli wektorem padania promieni światła, oraz N jest wektorem normalnym powierzchni w punkcie P. Składowa D_i, tak jak składowa A w Równaniu 7.1 może być skalarem lub składać się z trzech liczb dla oddzielnych składowych koloru.

Składowa *specular*, tak jak składowa *diffuse* jest liczona osobno dla każdego światła a potem sumowana. Ilość światła *specular* jest wprost proporcjonalna do kąta pomiędzy wektorem patrzenia na powierzchnię, czyli wektorem promienia D, oraz odbitym wektorem światła R. Dla osiągnięcia połysku powierzchni wyliczony współczynnik podnosi się do potęgi – im większa potęga tym większe wrażenie połysku.

Jeśli wektor światła jest równy –L, to wektor odbity liczy się ze wzoru:

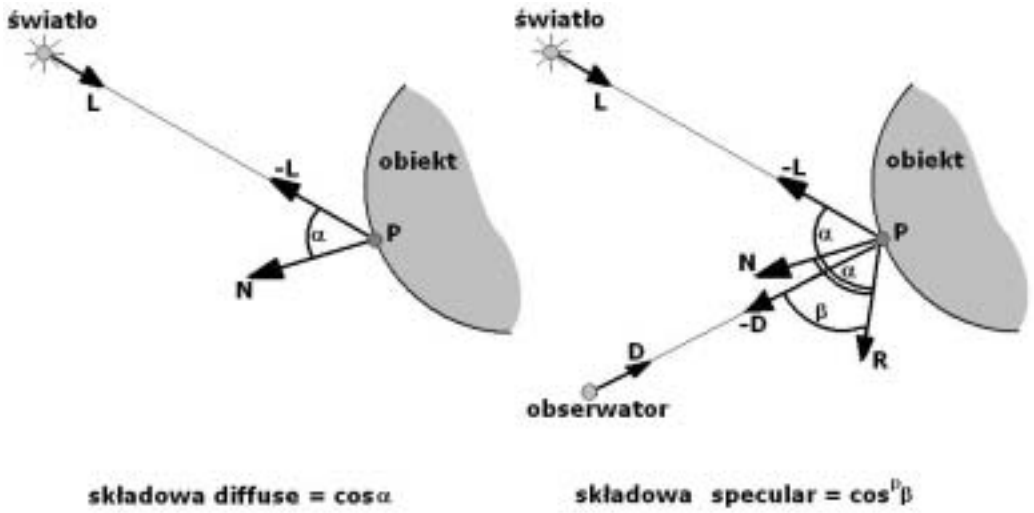
$$R = L - 2N(L \circ N) \qquad \text{(Równ. 7.3)}$$

gdzie N jest wektorem normalnym powierzchni a L jest wektorem promieni światła padającego na powierzchnię.

Zgodnie z powyższym, równanie składowej *specular* jest następujące:

$$specular = \sum_i K_S \bullet S_i(-D \circ R_i)^p \qquad \text{(Równ. 7.4)}$$

gdzie K_S jest kolorem *specular* powierzchni (często biały lub taki jak K_D), S_i jest współczynnikiem *specular* dla i-tego światła (jedna lub trzy składowe tak jak A i D_i), D jest wektorem kierunkowym promienia, R_i jest odbitym wektorem światła a p jest współczynnikiem połysku materiału, tzw. potęgą *specular*.

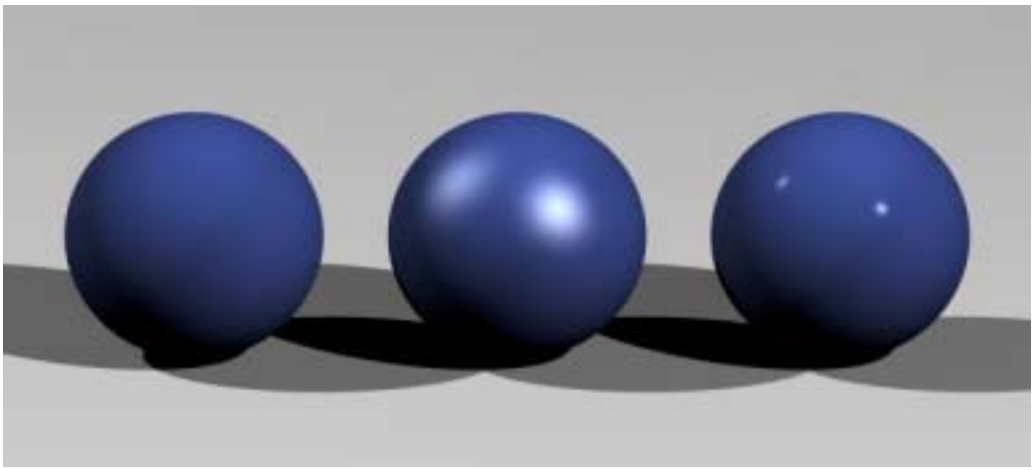


Rys. 7.1 – Składowe dyfuzyjna i bezpośrednia

Ostatecznie kolor promienia (wynikowego piksela obrazu) jest równy:

$$K = ambient + diffuse + specular \qquad \text{(Równ. 7.5)}$$

gdzie *ambient*, *diffuse* i *specular* pochodzą odpowiednio z Równań 7.1, 7.2 i 7.4. Kolory powierzchni K_A, K_D i K_S mogą pochodzić z tekstury, mogą też być stałe dla materiału.



Rys. 7.2 – Wyrenderowane kule z materiałami o różnym wykładniku Phonga: po lewej bez składowej *specular*, pośrodku p=10, po prawej p=300

7.3 Modele oświetlenia

7.3.1 Światło punktowe

Światła są nierenderowalnymi elementami sceny wpływającymi bezpośrednio na wygląd powierzchni obiektów o określonych kolorach. Sposób interakcji światła z powierzchniami jest określony modelami cieniowania, natomiast samo światło generowane jest przez obiekty tzw. „światła".

Sposób wytwarzania światła jest zależny od poszczególnych obiektów-świateł. Natężenie światła może zmieniać się w funkcji odległości lub być stałe w każdym punkcie przestrzeni. Światło zmienne w funkcji f(x) symuluje bliskie źródła (np. lampy), natomiast światło stałe symuluje źródła odległe (np. słońce).

Model światła punktowego (*ang*. *Point Light* lub *Omni Light*) opiera się na założeniu że obiekt generuje światło we wszystkich kierunkach. Innymi słowy, światło pochodzi z jednego punktu w przestrzeni i jest wypromieniowywane w kulę.

W wybranym punkcie przestrzeni P, wektor L światła pochodzącego od źródła znajdującego się w punkcie C jest zatem równy:

$$L = \mathrm{Norm}(P - C) \qquad \text{(Równ. 7.7)}$$

Jeśli natężenie podstawowe światła wynosi a_0, natężenie a_P w punkcie P jest równe:

$$a_P = a_0 \, \mathrm{f}(|P - C|) \qquad \text{(Równ. 7.8)}$$

gdzie f(x) jest funkcją natężenia od odległości.

7.3.2 Światło stożkowe

Światło stożkowe (*ang. Spot Light*) jest podobne do światła punktowego ponieważ też pochodzi z jednego punktu, natomiast jest ograniczone przez stożek. Promienie świetlne wzdłuż osi stożka mają największe natężenie. Promienie świetlne leżące na płaszczyźnie stożka mają natężenie równe 0. Definiuje się wewnętrzny stożek, w którym wszystkie promienie świetlne mają takie samo, maksymalne natężenie.

Światło stożkowe definiuje się przy pomocy punktu generującego światło C, wektora osi stożka L oraz kątów zewnętrznego α_{max} i wewnętrznego α_{min} stożka ($\alpha_{min} < \alpha_{max}$).

Dla wybranego punktu przestrzeni P, światło stożkowe wpływa na niego wtedy i tylko wtedy gdy kąt pomiędzy wektorem światła wyliczanym z Równania 7.7 a osią stożka:

$$\text{Norm}(P - C) \circ L > \cos \alpha_{max} \qquad \text{(Równ. 7.9)}$$

Natężenie światła w punkcie P dla promieni świetlnych leżących w wewnętrznym stożku (zgodnie ze wzorem 7.9, przy czym zamiast α_{max} jest α_{min}) oblicza się z Równania 7.8, natomiast natężenie światła dla promieni świetlnych leżących pomiędzy stożkiem wewnętrznym a zewnętrznym zmienia się nieliniowo i jest wyliczane ze stosunku cosinusów:

$$a_P = a_0 \, \text{f}(|P - C|) \frac{(\text{Norm}(P - C) \circ L) - \cos \alpha_{max}}{\cos \alpha_{min} - \cos \alpha_{max}} \qquad \text{(Równ. 7.10)}$$

7.3.3 Światło cylindryczne

Światło cylindryczne (*ang. Directional Light*) symuluje odległe źródła światła, gdyż jego promienie świetlne są wzajemnie równoległe. Typowe światło cylindryczne jest ograniczone cylindrem o promieniu r_{max}. W wewnętrznym cylindrze o promieniu r_{min} wszystkie promienie świetlne mają takie samo natężenie. Natężenie pomiędzy cylindrem wewnętrznym i zewnętrznym maleje nieliniowo jak dla stożka.

Światło cylindryczne definiuje się przy pomocy punktu generującego światło C, wektora osi cylindra L oraz promieni wewnętrznego r_{min} i zewnętrznego r_{max} cylindra ($r_{min} < r_{max}$).

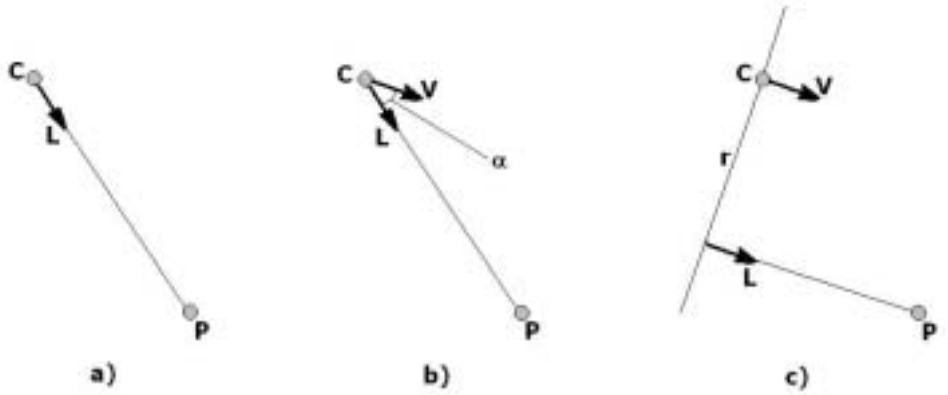
Dla wybranego punktu przestrzeni P, światło cylindryczne wpływa na niego jeśli długość wektora P'-C jest mniejsza od promienia zewnętrznego cylindra r_{max}, przy czym P' jest rzutem punktu P na płaszczyznę bazową światła cylindrycznego określoną punktem zaczepienia C i wektorem normalnym L.

$$\begin{aligned} P' &= P - L((P - C) \circ L) \\ |P' - C|^2 &< {r_{max}}^2 \end{aligned} \qquad \text{(Równ. 7.11)}$$

W powyższym równaniu testujemy kwadrat odległości, dzięki czemu nie musimy liczyć kosztownego pierwiastka.

Dla punktów P leżących pomiędzy cylindrami wyliczamy natężenie światła analogicznie jak dla światła stożkowego, przy czym w tym przypadku natężenie zmienia się kwadratowo:

$$\begin{aligned} d &= (P - C) \circ L \\ a_P &= a_0 \, \text{f}(d) \frac{{r_{max}}^2 - |P' - C|^2}{{r_{max}}^2 - {r_{min}}^2} \end{aligned} \qquad \text{(Równ. 7.12)}$$



Rys. 7.3 Rodzaje świateł – a) punktowe, b) stożkowe, c) cylindryczne

7.3.4 Funkcja natężenia światła od odległości

Wraz z oddalaniem się punktu od źródła światła, maleje jasność, czyli natężenie światła. W terminologii angielskiej określa się to mianem *attenuation* – gaśnięcia. Z fizyki wiadomo, że natężenie maleje w funkcji odwrotnie kwadratowej w zależności od odległości.

W raytracingu nie jesteśmy jednak ograniczeni fizyką i możemy przyjąć dowolną funkcję. Na przykład dla źródeł odległych, np. dla słońca, możemy założyć, że natężenie światła jest stałe i nie zależy od odległości.

Poniżej przedstawiono kilka popularnych funkcji natężenia od odległości.

$$\begin{aligned} \mathrm{f}(x) &= 1 \\ \mathrm{f}(x) &= \frac{1}{x} \\ \mathrm{f}(x) &= \frac{1}{x^2} \\ \mathrm{f}(x) &= a^{-x} \end{aligned} \qquad \text{(Równ. 7.13)}$$

Funkcje te mogą zostać użyte do określenia natężenia światła w zależności od odległości x od źródła. Na ich wartości można nałożyć ograniczenia – zwłaszcza ograniczenie górne w celu redukcji nadmiernego „prześwietlenia" bliskich obiektów. Oczywiście w każdym z równań świateł (7.8, 7.10, 7.12) mamy również stały współczynnik jasności a_0, który pomaga przy dostrojeniu źródeł światła. Dodatkowo funkcję a^{-x} możemy stablicować w celu przyspieszenia obliczeń.

7.4 Znajdowanie cieni

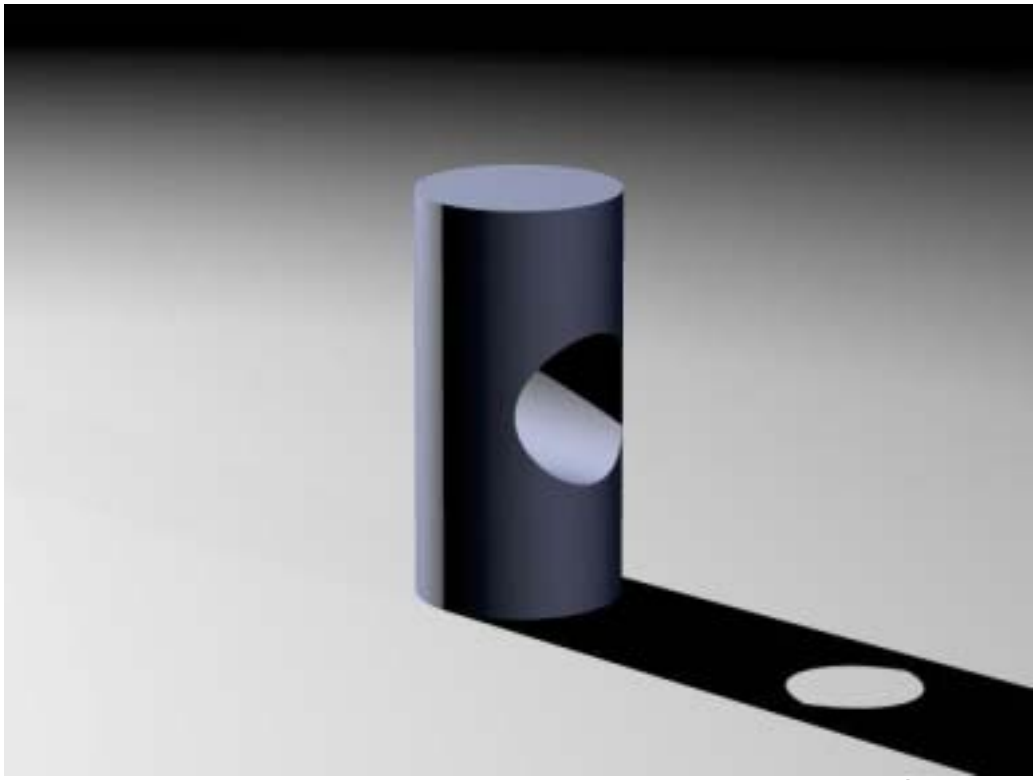
Cienie w raytracingu uzyskuje się poprzez sprawdzanie, czy dany punkt jest oświetlony przez światła. Wartości natężenia światła dyfuzyjnego (*diffuse*) i bezpośredniego (*specular*) nie uwzględnia się dla świateł zasłoniętych.

Aby sprawdzić, czy konkretne światło jest zasłonięte i nie wnosi udziału do oświetlenia punktu, generuje się tzw. promień cienia (*shadow ray*). Promień ten wychodzi z obiektu i jest skierowany w kierunku światła:

$$\begin{cases} O_S = P \\ D_S = -L \end{cases} \qquad \text{(Równ. 7.14)}$$

Ponadto przyjmuje się maksymalną odległość – zasięg promienia. Zasięg ten jest nie większy niż odległość trafionego punktu P od źródła światła.

Detekcja zasłonięcia realizowana jest poprzez sprawdzenie, czy promień cienia trafia w jakikolwiek obiekt pomiędzy trafionym punktem a światłem. Obliczenia związane z promieniami cieni są prostsze w stosunku do obliczeń promieni przecięć, ponieważ nie musimy znać najbliższego przecinanego obiektu, lecz jakikolwiek obiekt na danym dystansie. Procedury przecięć z Rozdziału 5 są zatem nieco uproszczone.



Rys. 7.4 – Wyrenderowany cylinder z dziurką rzucający cień

8 Dodatkowe algorytmy stosowane podczas syntezy obrazów

8.1 Symulacja efektów fizycznych poprzez generowanie promieni wtórnych

Podstawowymi efektami wprowadzającymi realizm do generowanych scen są tekstury i cienie. Przez zastosowanie promieni wtórnych można dalej zwiększyć realizm, symulując kolejne efekty.

Promienie wtórne są promieniami przecięć generowanymi na powierzchni obiektów. Promienie te zawsze zaczepione są w trafionym punkcie: O' = P. Umożliwiają one symulowanie odbić oraz przezroczystości powierzchni.

Promienie wtórne liniowo zwiększają ilość obliczeń. Istotne jest założenie ograniczeń na ich ilość – po trafieniu kolejnego punktu promieniem wtórnym generowane są bowiem kolejne. Podstawowym ograniczeniem jest maksymalna liczba iteracji, tzw. głębokość. Innym ograniczeniem może być ilość światła wnoszonego przez promień. Każdy promień wtórny posiada współczynnik wkładu s<1, który maleje wraz z kolejnymi promieniami wtórnymi i jest ustalany dla konkretnych materiałów. Po osiągnięciu minimalnej założonej wartości tego współczynnika przestaje się generować kolejne promienie.

8.1.1 Obliczanie promieni odbitych – efekt lustra

Promienie odbite wylicza się z następującego wzoru:

$$\begin{cases} O' = P \\ D' = D - 2N(N \circ D) \end{cases} \qquad \text{(Równ. 8.1)}$$

gdzie P jest trafionym punktem, N jest wektorem normalnym w tym punkcie (już po zaaplikowaniu mapy chropowatości jeśli obiekt taką posiada) a D jest poprzednim wektorem kierunku promienia który trafił w punkt P. O' i D' są odpowiednio punktem zaczepienia i wektorem kierunku nowego, odbitego promienia.

Współczynnik wkładu s dla promienia odbitego jest przyjmowany jako współczynnik odbicia r. Jest on stały dla typowych materiałów, natomiast dla bardziej skomplikowanych modeli może być zależny od danych początkowych (np. kąt patrzenia czy kąt padania światła).

8.1.2 Kolejność obliczania promieni wtórnych

Podczas tworzenia obrazu, dla każdego piksela wyliczamy promienie pierwotne. Do kolorów wyliczonych dla promieni pierwotnych dodajemy kolory pochodzące od promieni wtórnych. Tym samym nasuwa się rekurencyjny algorytm:

```
UtwórzObraz()
{
     dla każdego piksela
          piksel[ x, y ] = PrzetwarzajPromień( x, y )
}

kolor PrzetwarzajPromień( O, D )
{
     kolor = oblicz kolor promienia
     dla każdego promienia wtórnego {
          s = oblicz wkład promienia wtórnego
          kolor += s * PrzetwarzajPromień( wtórny )
     }
     zwróć kolor
}
```

Powyższy sposób nie jest jednak jedyny. Alternatywą jest algorytm iteracyjny, który oblicza kolejno wszystkie promienie z poziomów.

```
UtwórzBlokObrazu( szer, wys )
{
     kolejka FIFO promieni
     dodaj wszystkie promienie pierwotne do kolejki
     zeruj wszystkie piksele
     dopóki kolejka niepusta {
          promień = pobierz promień z kolejki
          kolor = oblicz kolor promienia
          piksel[ x, y ] += kolor
          dla każdego promienia wtórnego {
               dodaj promień wtórny do kolejki [x,y]
          }
     }
}
```

Algorytm ten jest prostszy niż algorytm rekurencyjny, ponieważ nie wymaga ciągłego wywoływania funkcji. Ponadto jeśli liczymy wszystkie promienie z jednego poziomu jednocześnie (np. wszystkie promienie pierwotne jeden po drugim), korzystamy z większej koherencji tych promieni, ponieważ dwa promienie z jednego poziomu mają dużo większą szansę przeciąć się z tymi samymi obiektami niż dwa promienie z różnych poziomów.

8.2 Składanie promieni w obraz wynikowy

W ostatniej fazie mamy do czynienia ze zbiorem kolorów wyliczonych dla promieni pierwotnych. Najprostszym sposobem zakończenia całego cyklu jest przypisanie pojedynczym pikselom obrazu kolorów odpowiadających im promieni oraz składowanie obrazu. Realizuje to poniższy pseudokod.

```
obraz o szerokości w i wysokości h
dla każdego piksela o współrzędnych 0<x<w i 0<y<h {
     promień = kamera -> wygeneruj promień( x, y )
     kolor = znajdź kolor promienia
     obraz[ x, y ] = kolor
}
zapisz obraz do pliku
```

Podejście w którym ilość promieni jest równa ilości pikseli generowanego obrazu sprzyja powstawaniu tzw. aliasingu. Aliasing polega na „kanciastości” czy „pikselowatości” krawędzi obiektów. Najprostszym sposobem uniknięcia aliasingu jest wygenerowanie obrazu o N razy większej rozdzielczości (typowo 2-4 razy większej) i uśrednieniu pikseli. Innymi

słowy, dla pojedynczego piksela generuje się N^2 promieni, których kolory się uśrednia. Takie podejście liniowo zwiększa ilość generowanych promieni.

8.3 Przetwarzanie rozproszone

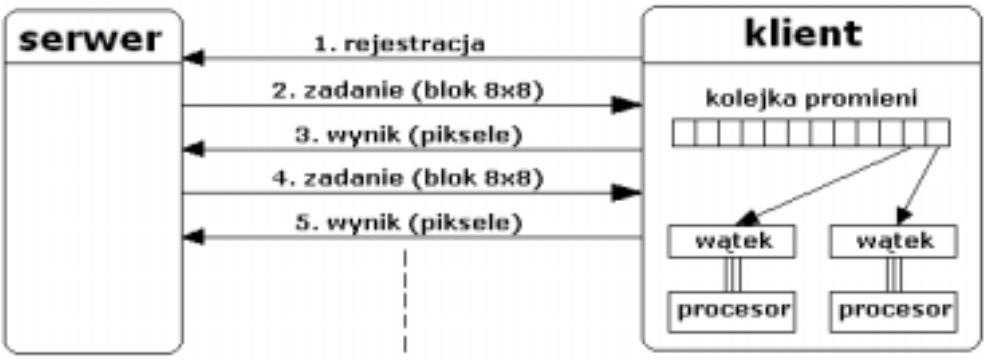
Raytracing w sposób naturalny można podzielić na szereg mniejszych, niezależnych zadań. Podział ten może zostać wykonany na poziomie pojedynczych promieni (potok obróbki promienia – Rozdział 2), albo też między promieniami (pojedyncze promienie są przetwarzane przez różne jednostki) lub między pikselami (pojedyncze piksele lub zbiory pikseli są przetwarzane przez różne jednostki).

Ze względu na obecny brak możliwości zrównoleglenia obliczeń na poziomie potoku obróbki promienia podział ten nie został uwzględniony.

Oprogramowanie powstałe dla celów niniejszej pracy może w dwojaki sposób wykorzystywać przetwarzanie równoległe: w obrębie jednego komputera i rozproszone na wielu komputerach.

W [16], [17] przedstawiono metodę podziału obrazu na bloki. W niniejszej pracy zaproponowano modyfikację tego podziału. Obraz dzieli się na bloki o wymiarach 8x8 pikseli. Oprogramowanie działa w architekturze klient-serwer. Zdalne kopie programu rejestrują się w serwerze. Serwer przesyła im dokładną kopię danych – scenę. Następnie dzieli generowany obraz i do każdego klienta wysyła po dwa bloki 8x8 do przetworzenia. Klienci przetwarzają bloki. Ponieważ każdy klient otrzymał dwa bloki, po przeliczeniu pierwszego odsyła wynik (kolory pikseli) do serwera i przechodzi do przetworzenia drugiego bloku. W tym czasie serwer odbiera wynik i wysyła kolejny blok do przetworzenia do klienta. Dlatego też unika się przestojów i oczekiwania na dane – każdy z klientów cały czas jest zajęty i wykorzystany w (prawie) 100%.

Podział obrazu na bloki 8x8 empirycznie jest bardziej optymalny niż np. podział na poziome linie, ponieważ w kwadratowym bloku zachowana jest większa koherencja wiązki promieni, a co za tym idzie lepsza lokalność przestrzenna danych.



Rys. 8.1 – Architektura klient-serwer

Program wykorzystuje również maszyny wieloprocesorowe. Tworzy tyle wątków renderujących ile w systemie jest procesorów. Każdy wątek jest przyporządkowany do odpowiedniego procesora. Wątek renderujący z puli wygenerowanych promieni pobiera kolejne promienie i je przetwarza. Wszystkie wątki korzystają ze wspólnej puli promieni. Nie ma przy tym konfliktu RAW (*ang. read-after-write*), ponieważ dane są przeważnie odczytywane – zapisywane są tylko dane dotyczące pojedynczego promienia (w obrębie jednego wątku) oraz dane wynikowe (kolory pikseli).

9 Projekt i struktura programu

9.1 Założenia projektowe

Po dobraniu różnych algorytmów uczestniczących w syntezie obrazu, przed przystąpieniem do realizacji projektu będącego częścią niniejszej pracy poczyniono szereg założeń, mających na celu określenie ram działania programu.

Program służy do generowania obrazów w oparciu o dobrane algorytmy. Jego danymi wejściowymi jest opis sceny, a danymi wyjściowymi obraz. Dane wejściowe mogą być w dowolnym formacie, jeśli tylko nie przekraczają dopuszczalnego zestawu obiektów akceptowanych przez program, oraz jeśli istnieje odpowiednia wtyczka (*ang. plug-in*) importująca.

Program jest rozszerzalny pod kątem danych wejściowych i wyjściowych. Dane wejściowe są importowane poprzez wtyczki – uniwersalne moduły dodające obiekty do sceny istniejącej w programie. Program może zostać rozszerzony o różne typy obiektów geometrycznych, świateł, kamer, o różne rodzaje algorytmów przeglądania sceny, dodatkowe efekty wizualne, oraz sposoby zrównoleglenia obliczeń. Możliwości te są udostępniane dzięki zdefiniowaniu jednorodnych interfejsów odpowiednich modułów programu.

Dodatkową zaletą zaprojektowanej architektury jest przenaszalność. Program może zostać przystosowany na innych systemach niż obecna implementacja (Windows), dzięki odseparowaniu elementów specyficznych dla systemu operacyjnego od zasadniczej części programu.

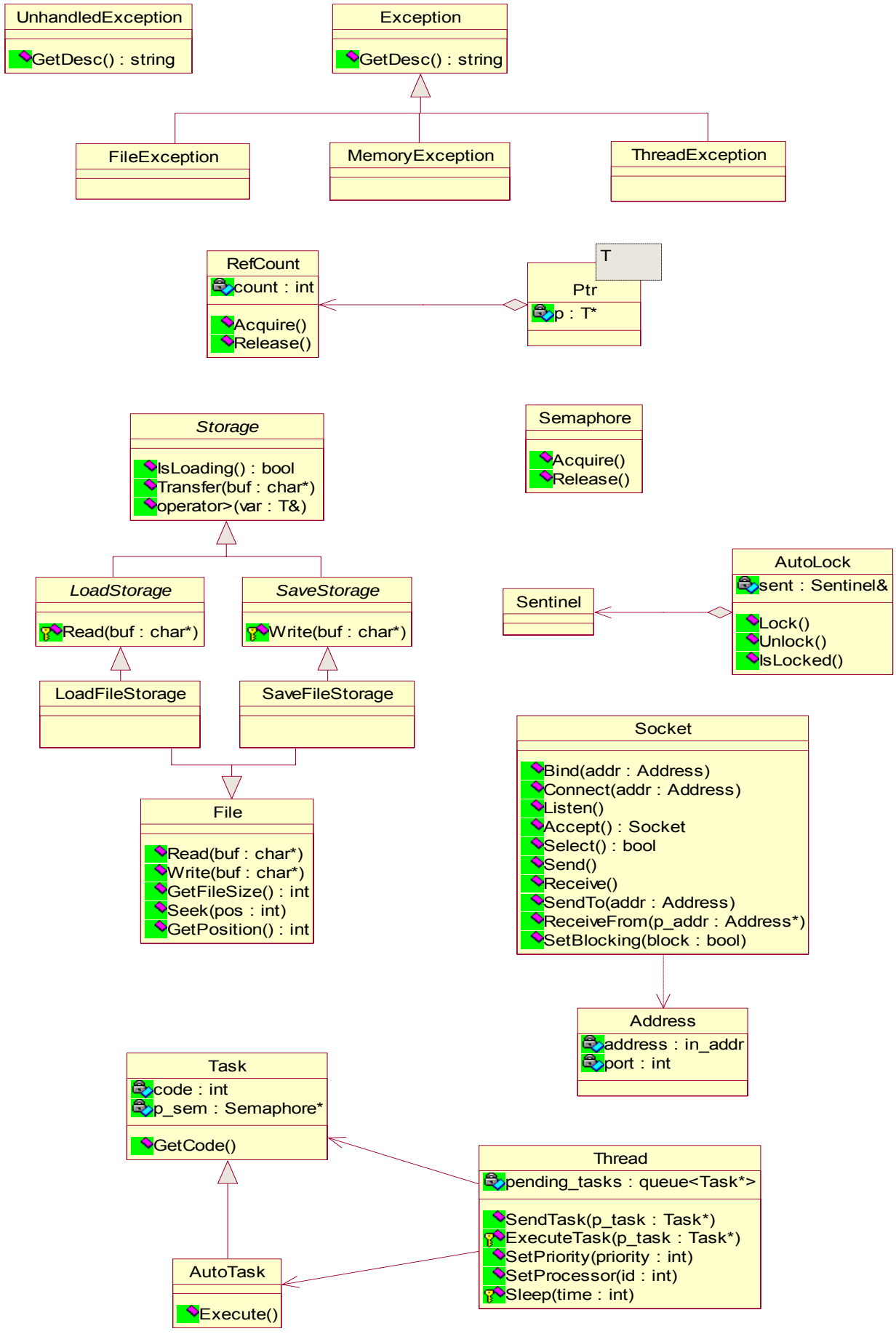
Do implementacji wybrano język C++, ponieważ daje on dużą swobodę działania. Język ten jest obecnie ustandaryzowany i cieszy się dużą popularnością. Umożliwia on programowanie obiektowe, co pozwala na projektowanie na wyższym poziomie abstrakcji i jednocześnie bezpośrednie odzwierciedlenie projektu w implementacji. Mechanizmy wbudowane w język C++ umożliwiają zastosowanie idiomów (takich jak np. bezpieczny wskaźnik), które zmniejszają liczbę błędów w programie, a tym samym przyspieszają czas tworzenia i ułatwiają późniejsze modyfikacje i rozszerzanie.

9.2 Konstrukcja programu

W następujących punktach przedstawiono i opisano schematy klas – strukturę, sposoby tworzenia obiektów i komunikacji pomiędzy nimi.

9.2.1 Podstawowe klasy systemowe

W celu uniezależnienia się od systemu operacyjnego wyodrębniono szereg klas implementujących różne specyficzne aspekty. W momencie przenoszenia programu na inny system operacyjny należy przepisać te klasy tak, aby udostępniały identyczny interfejs.



Rys. 9.1 – Podstawowe klasy systemowe

Prosta struktura wyjątków umożliwia łatwą obsługę sytuacji krytycznych. Do sygnalizacji poważnych błędów służy klasa *UnhandledException*. Klasa *Exception* jest bazową dla *FileException*, *MemoryException* i *ThreadException*, oraz ewentualnie innych wyjątków, które sygnalizowane są w momencie wystąpienia błędów możliwych do usunięcia lub ominięcia (np. brak pamięci, czy błąd dostępu do pliku).

Bezpieczny wskaźnik (*ang. safe pointer*) *Ptr<T>* jest szablonem. Umożliwia automatyczne zwalnianie pamięci (obiektu) w momencie gdy zginie ostatnia referencja do niego. Wskaźnika tego używa się tak samo jak zwykłego wskaźnika, z tą różnicą że nie wywołuje się operatora *delete*. Bezpieczny wskaźnik działa w oparciu o zliczanie referencji. Klasa *T* (parametr szablonu) musi posiadać funkcje *Acquire()* (zwiększenie licznika) i *Release()* (zmniejszenie licznika i zwolnienie pamięci gdy licznik osiągnie 0). Klasa *RefCount* spełnia te warunki. Jest ona klasą bazową dla wszystkich innych klas obsługiwanych przez bezpieczny wskaźnik.

Klasa *Semaphore* implementuje semafor. Zawiera funkcje *Acquire()* i *Release()*. Klasa *Sentinel* implementuje mutex (*ang. mutualy exclusive*). Nie posiada ona funkcji, a jest obsługiwana przez statyczne obiekty klasy *AutoLock*, automatycznie zwalniające mutex po wyjściu z zasięgu (*}*).

Klasa *Socket* (wraz z *Address*) jest nadbudówką nad gniazda BSD. Niektóre funkcje są bowiem inaczej realizowane przez różne systemy operacyjne (np. przełączanie stanu blokowania).

Klasa *File* to podstawowe funkcje dostępu do plików. Klasa *Storage* umożliwia łatwą serializację obiektów przy użyciu tylko jednej funkcji (np. *LoadSave(Storage&)*).. Funkcja ta zapisuje obiekt jeśli argumentem jest obiekt *SaveStorage* a odczytuje jeśli *LoadStorage*. Do odczytywania i zapisywania obiektów z/do plików służą klasy *LoadFileStorage* i *SaveFileStorage*. Klasę *Storage* wykorzystano również do łatwego przesyłania sceny przez sieć.

Klasa *Thread* jest wątkiem. Obiekty tej klasy mają prywatną funkcję *Execute()*, która jest wykonywana na wątku związanym z obiektem. Do wątku wysyła się zadania, które są przesyłane poprzez kolejkę strzeżoną semaforem, i wykonywane w funkcji *ExecuteTask()* wywoływanej przez *Execute()*. Jeśli zadanie nie jest wyprowadzone z klasy *Task*, tylko z *AutoTask*, zamiast *ExecuteTask()* wywoływana jest funkcja *Execute()* klasy *AutoTask* (lub pochodnej). Zakończenie wątku odbywa się podczas wywołania destruktora obiektu i jest realizowane poprzez wysłanie specjalnego zadania przerywającego obsługę kolejki w funkcji *Execute()*.

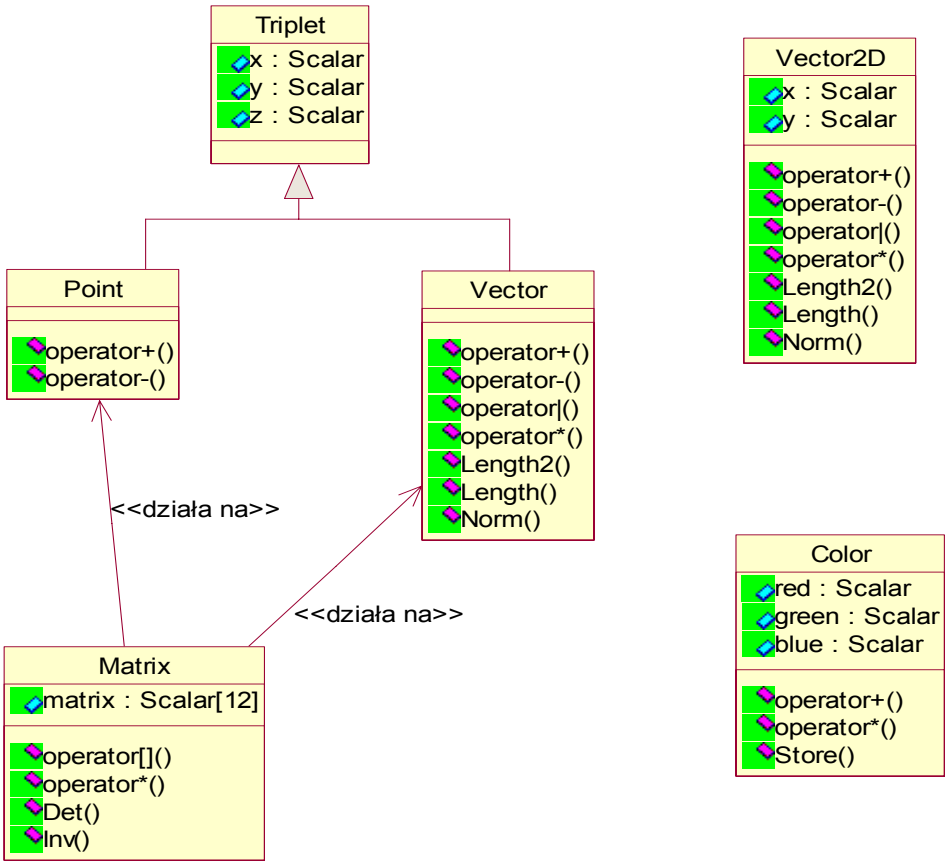
9.2.2 Klasy algebry wektorowej

Raytracing wykorzystuje elementy algebry wektorowej. Jedną z unikalnych własności języka C++ jest przeciążanie operatorów. Korzystając z tej dogodności zaimplementowano szereg klas służących do działania na wektorach liczb udostępniających semantykę zgodną z semantyką języka C++.

Podstawowym typem używanym w programie jest *Scalar*. Obecnie jest on zadeklarowany jako *double*, ale w każdej chwili może być przyjęty inny, np. *float*.

Klasa *Point* reprezentuje punkt a klasa *Vector* wektor w przestrzeni trójwymiarowej. Na punktach nie można wykonywać działań wektorowych, ale można do nich dodawać (i odejmować) wektory, a różnicą między dwoma punktami jest wektor. *Vector* z kolei udostępnia szereg operatorów dla różnych działań – dodawanie +,-, iloczyn skalarny |, iloczyn wektorowy *, mnożenie i dzielenie przez skalar *,/, oraz funkcje *Length2()* (kwadrat długości), *Length()* (długość) i *Norm()* (wektor jednostkowy). Ze względu na taką samą zawartość i podobne znaczenie, klasy *Point* i *Vector* są pochodnymi *Triplet*.

Klasa *Matrix* odpowiada macierzy 4x4 i może być używana do transformacji afinicznych. Punkt w takich transformacjach jest przyjęty jako $[x\ y\ z\ 1]^T$ a wektor jako $[x\ y\ z\ 0]^T$. Ostatni wiersz macierzy jest przyjęty zawsze jako [0 0 0 1].



Rys. 9.2 – Klasy algebry wektorowej

Klasa *Vector2D* służy do reprezentacji punktów lub wektorów w dwóch wymiarach. Jest używana do przechowywania i działania na współrzędnych tekstur.

Klasa *Color* udostępnia podstawowe działania na kolorach. Składowe koloru (*red*, *green* i *blue*) są typu *Scalar* i przyjmują wartości z zakresu [0, 1] ([0 0 0] – czarny, [1 1 1] – biały). Kolory można dodawać i mnożyć przez skalar, a także składować w postaci *unsigned char[3]* (wykonywana jest saturacja).

9.2.3 Budowa sceny

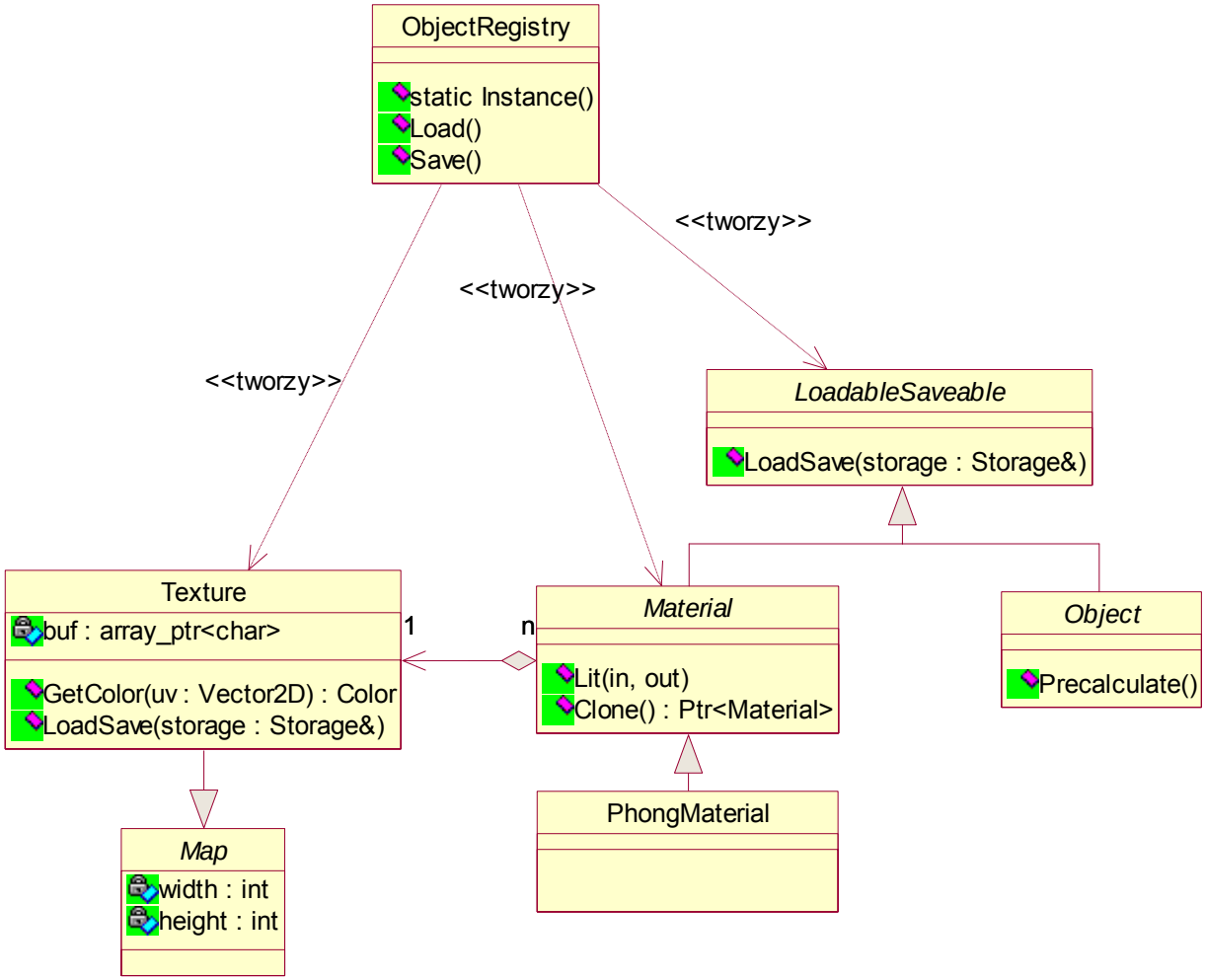
Scena zawiera obiekty będące podklasami klasy *Object*. *Object* reprezentuje element trójwymiarowej sceny. Są trzy rodzaje klas abstrakcyjnych wyprowadzonych z *Object*: *GeomObject* – renderowalny obiekt geometryczny, *Light* – źródło światła oraz *Camera* – obiekt kamery syntezujący promienie pierwotne.

Zaimplementowano trzy rodzaje świateł: *PointLight* (punktowe), *SpotLight* (stożkowe) i *DirLight* (cylindryczne). Zaimplementowano jeden rodzaj kamery – perspektywiczną – *CamPerspective*.

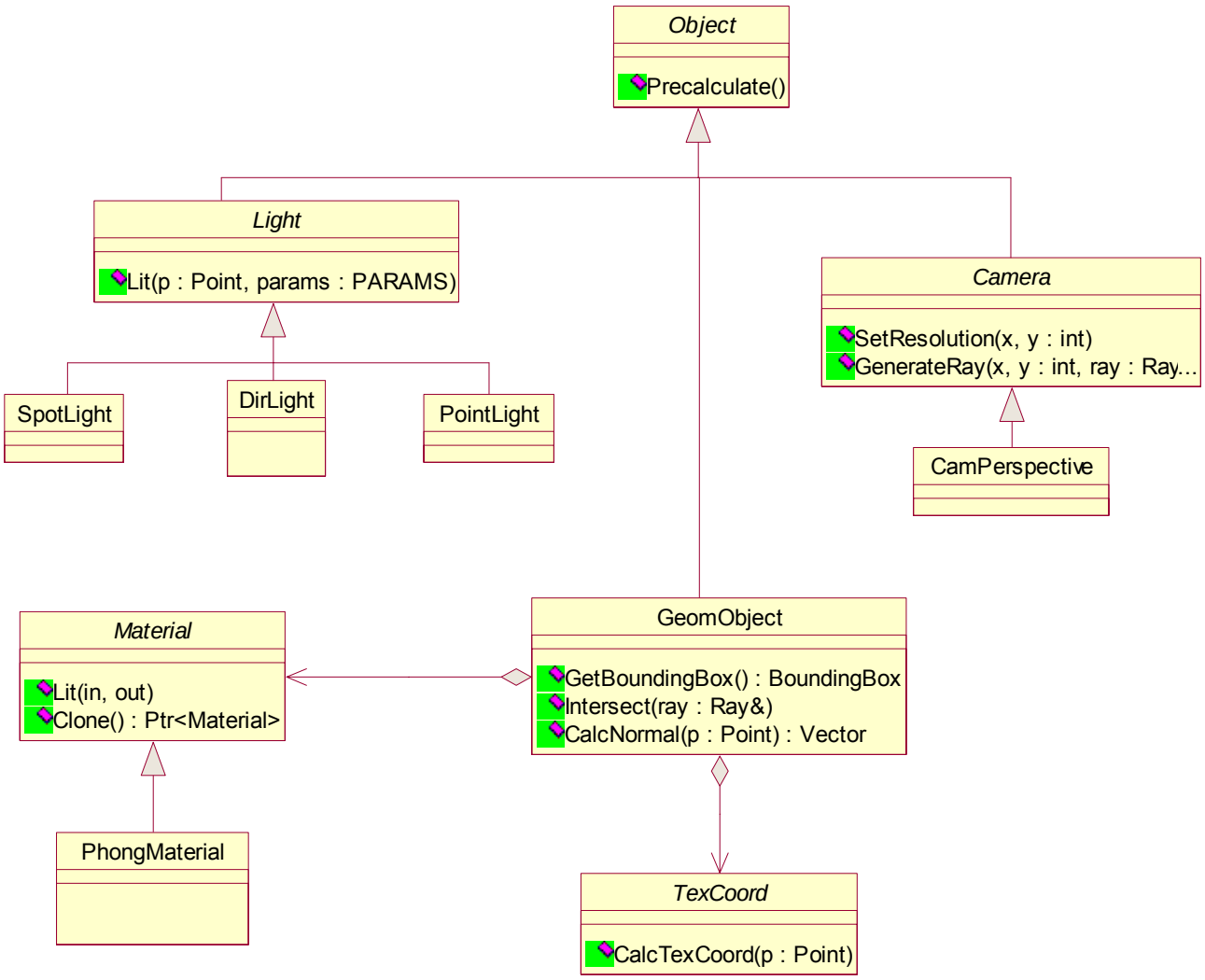
Obiekty *GeomObject* są renderowalne. Zaimplementowano pięć rodzajów opisanych obiektów prostych, oraz obiekt *Boolean* realizujący sumę, różnicę lub przecięcie (CSG). Klasa *Boolean* jest pochodną klasy *Compound*, zawierającej zbiór obiektów zamkniętych *GeomObject*. Klasa *GeomObject* ma dwie składowe: wskaźnik na materiał *Material* i wskaźnik na obiekt translacji współrzędnych *TexCoord*.

Klasa *Material* reprezentuje materiał (właściwości powierzchni obiektu) i potrafi wyliczyć kolor powierzchni po uwzględnieniu źródeł światła. Zaimplementowano jeden rodzaj materiału – opisany *PhongMaterial*. Materiał zawiera wskaźnik na teksturę *Texture*. Tekstura jest pochodną klasy *Map*. Jedna tekstura może być używana przez wiele materiałów, a jeden materiał może być używany przez wiele obiektów geometrycznych. Nie ma globalnego rejestru materiałów i tekstur, a zwalnianie ich odbywa się automatycznie dzięki właściwościom bezpiecznego wskaźnika.

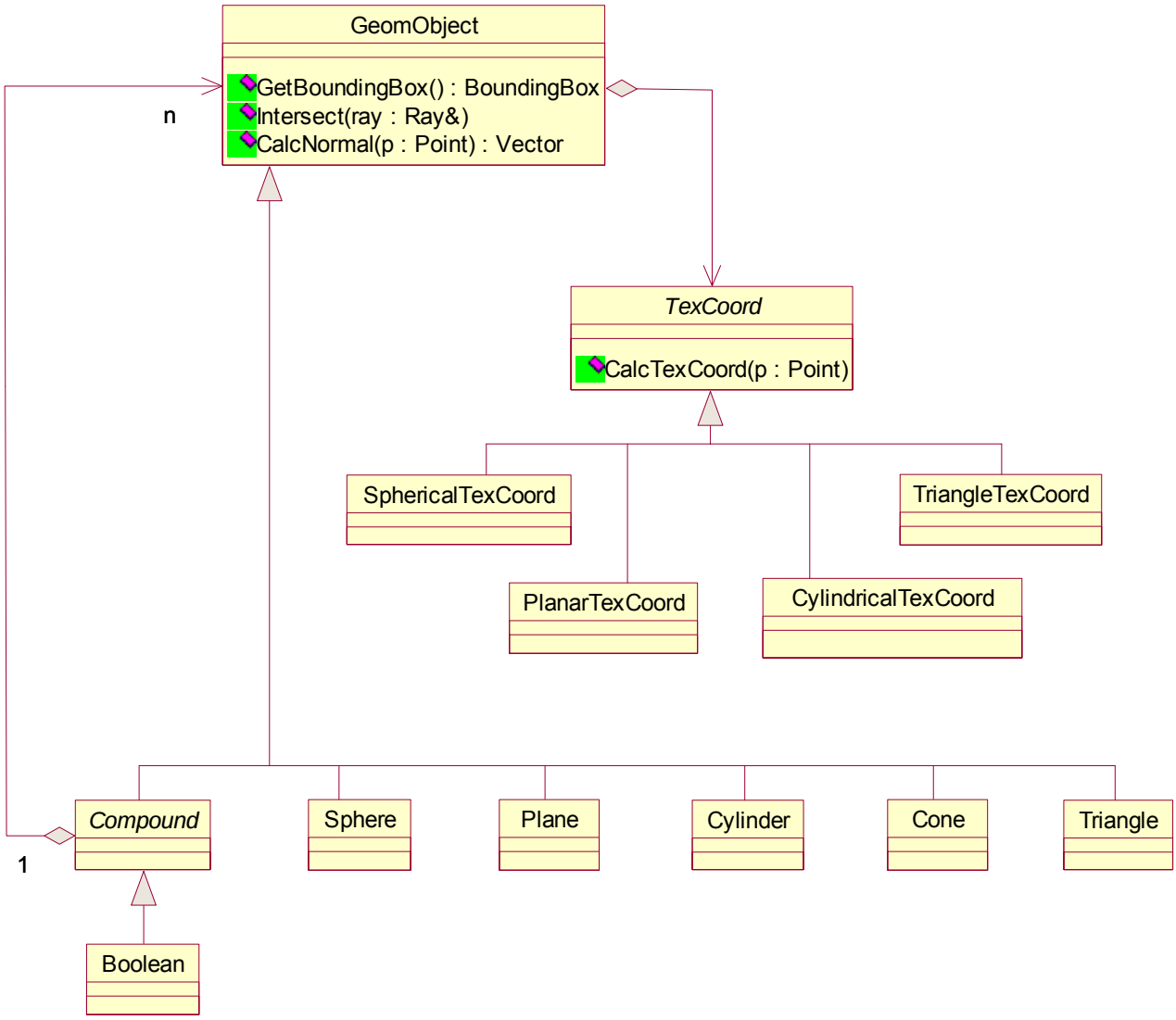
Podczas importowania sceny z zewnętrznego pliku obiekty tworzone są przez wtyczkę importera. Jednak scena może być zapisana do składu *Storage* i odtworzona z niego (składem może być plik lub sieć).



Rys. 9.3 – Budowa sceny (część I)



Rys. 9.4 – Budowa sceny (część II)



Rys. 9.5 – Budowa sceny (część III)

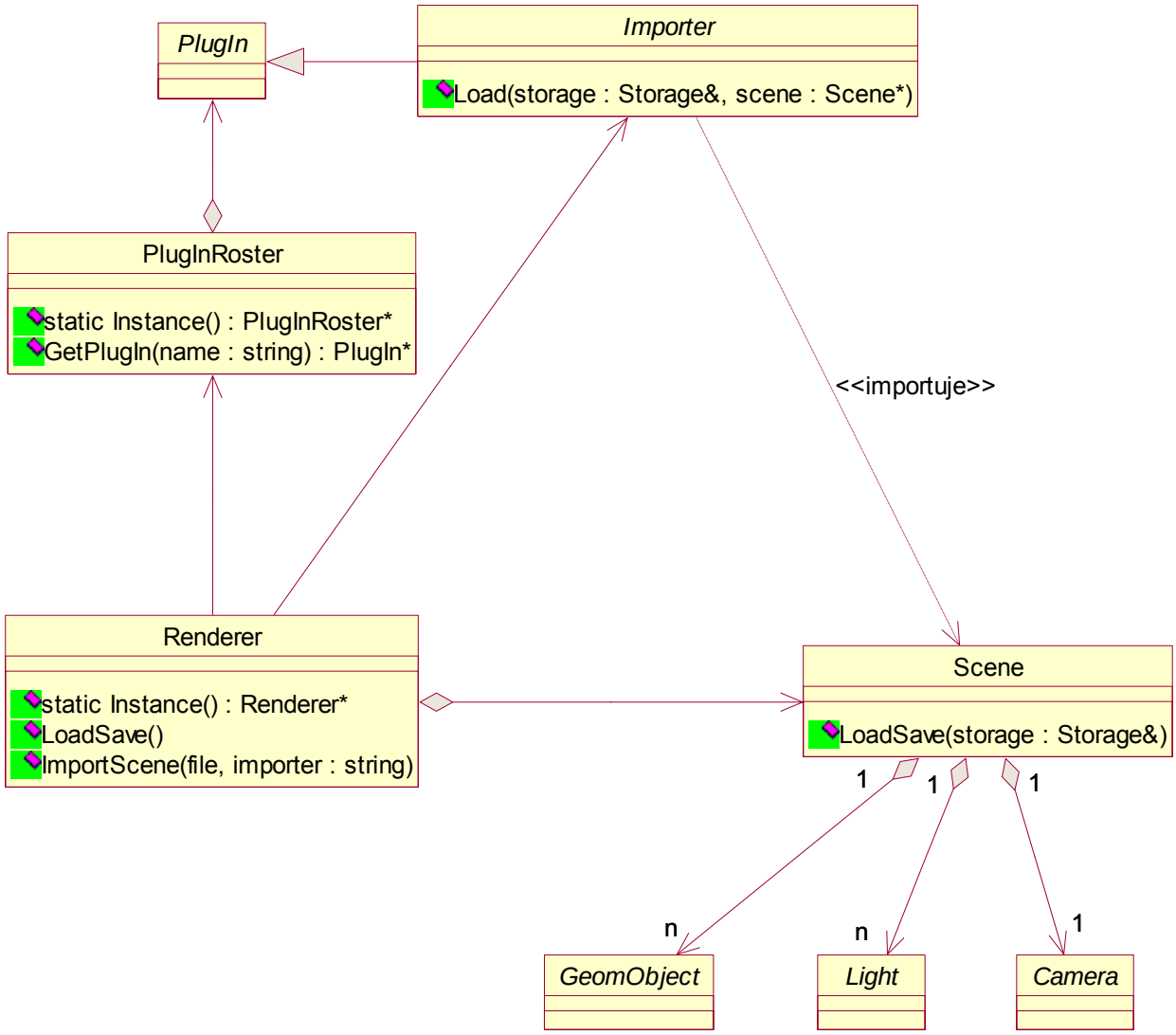
Klasy *Object*, *TexCoord* i *Material* są klasy *LoadableSaveable* i tym samym wspierają ich odczytywanie i zapisywanie. Klasa *Texture* posiada w tym celu funkcję *LoadSave()*. Globalny obiekt klasy *ObjectRegistry* służy do zapisywania i odczytywania obiektów i jest używany przez klasę *Scene* przedstawioną w 9.2.4. Obiekt klasy *ObjectRegistry* potrafi utworzyć odpowiednie obiekty podczas odczytywania ze *Storage*, podczas zapisywania bowiem zapisuje on również nazwę klasy obiektu. Obiekt *ObjectRegistry* oparty jest na wzorcu (*ang. Design Pattern*) zwanym *Abstract Factory*.

9.2.4 Struktura renderera

Renderer jest główną częścią programu odpowiedzialną za syntezę obrazów. Pojedynczy, globalny obiekt klasy *Renderer* umożliwia: importowanie sceny z zewnętrznych plików, zapisywanie odczytywanie sceny do/z składów *Storage*, rejestrowanie i wyrejestrowywanie procesorów bloków oraz renderowanie obrazu.

Podczas importowania sceny, renderer pobiera odpowiednią wtyczkę (pochodną klasy *Importer*). Wtyczka wczytuje poszczególne obiekty, światła i kamerę i dodaje je do sceny

(klasa *Scene*) zawartej w rendererze. Zarówno po zaimportowaniu sceny jak i po załadowaniu z pliku, scena jest przygotowywana do renderowania (dla wszystkich obiektów wywoływana jest funkcja *Precalculate()*).

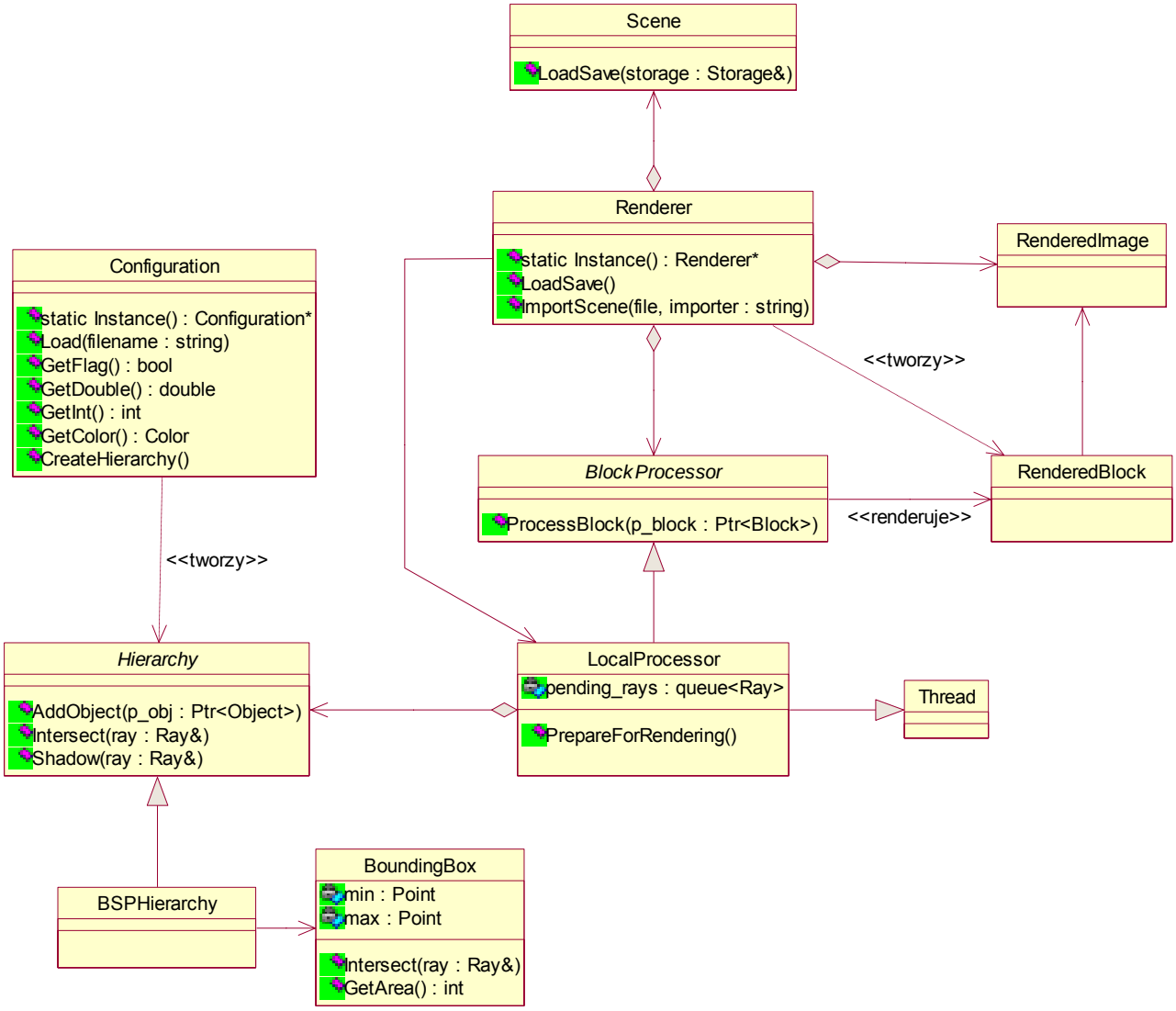


Rys. 9.6 – Schemat renderera (część I)

Wszelkie renderowane mapy bitowe są pochodnymi klasy *Block*. Blok jest to prostokątny zbiór pikseli, które mają zostać wypełnione przez renderer bądź jego części. Proces renderowania rozpoczyna się po przekazaniu do renderera bloku klasy *RenderedImage*. Po zakończeniu destruktor *RenderedImage* aplikuje antyaliasing i zapisuje obraz do pliku.

Proces renderowania przygotowany jest zarówno do wykonania na jednej jednostce obliczeniowej jak i w systemie rozproszonym. W rendererze rejestruje się procesory bloków (pochodne klasy *BlockProcessor*) które wykonują obliczenia. Renderer dokonuje podziału zadań. Zaimplementowany schemat rozdziału zadań polega na podzieleniu renderowanego obrazu na kwadratowe bloki (domyślnie 8x8, definiowalne przez użytkownika) i późnym

przydzielaniu ich procesorom bloków (*ang. lazy dispatch*). Na początku każdy procesor dostaje trzy bloki do przetworzenia. Jeśli dany procesor skończy pierwszy z otrzymanych bloków, otrzymuje następny. W ten sposób każdy procesor ma jeden blok który obecnie renderuje i dwa oczekujące. Zadania przekazywane są do procesorów w postaci bloków *RenderedBlock*.

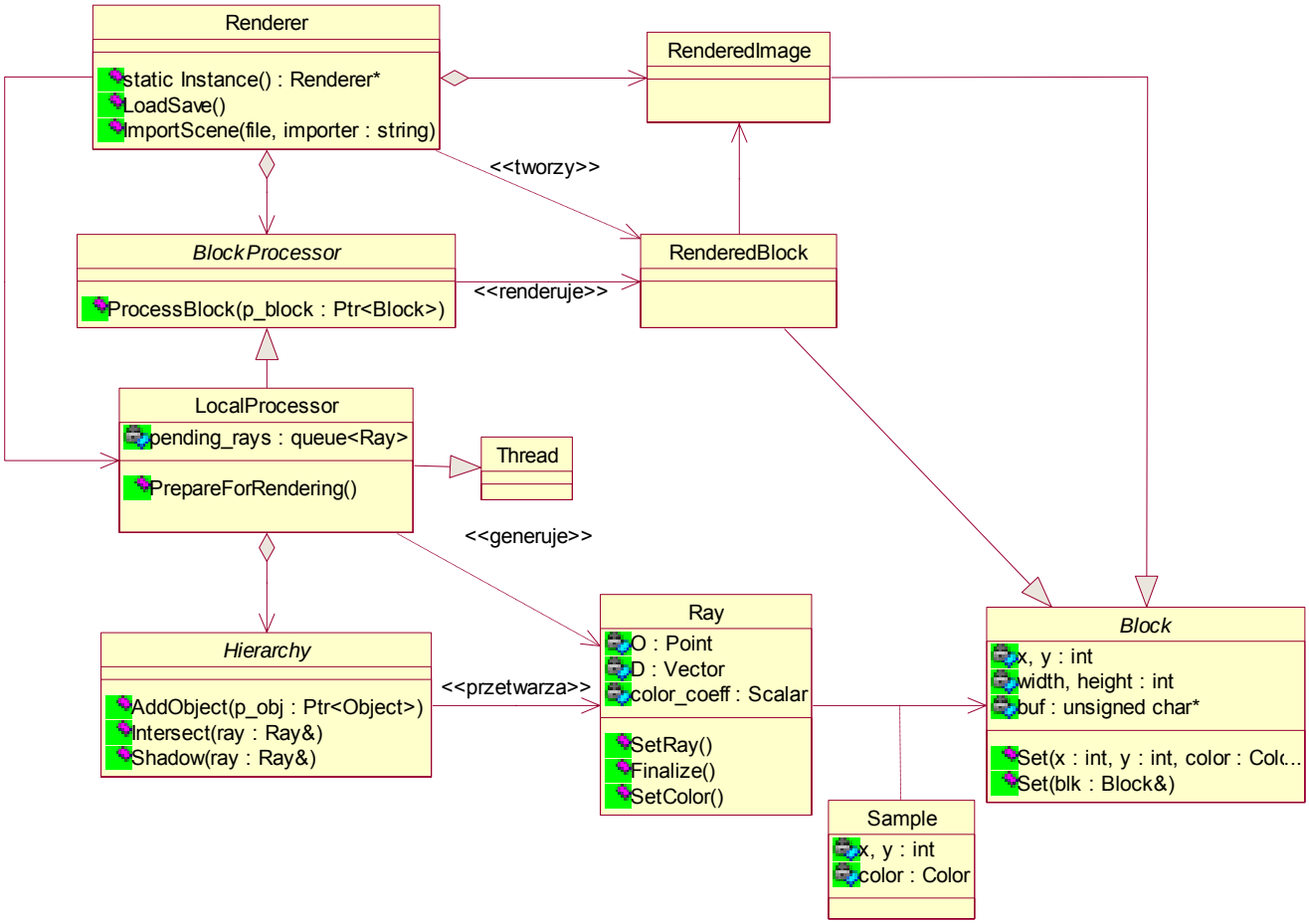


Rys. 9.7 – Schemat renderera (część II)

LocalProcessor to lokalny procesor bloków. Implementuje on właściwy algorytm raytracingu. Domyślnie renderer tworzy jeden lokalny procesor. *LocalProcessor* posiada jeden wątek renderujący do którego przekazywane są bloki – zadania. Może on zostać przystosowany do pracy wieloprocesorowej. Konfiguracja lokalnego procesora odbywa się poprzez globalny obiekt *Configuration*, odczytujący dane konfiguracyjne z pliku użytkownika.

Potok renderowania zrealizowany został następująco. Lokalny procesor pobiera kolejno otrzymane bloki – zadania. Dla każdego bloku generuje zestaw promieni pierwotnych i umieszcza je w kolejce. Lokalny procesor bowiem ma dostęp do kamery,

świateł oraz zbioru obiektów posiadanych przez renderer. Obiekty są trzymane w specjalnej strukturze klasy *Hierarchy* optymalizującej liczbę przecięć z obiektami. Zaimplementowano jeden algorytm optymalizacji przecięć metodą drzew BSP budowanych w oparciu o heurystykę powierzchni sześcianów okalających (Rozdział 5) – jest to klasa *BSPHierarchy*. Istnieje możliwość wymiany tego algorytmu na inny poprzez implementację innej klasy pochodnej od *Hierarchy*.



Rys. 9.8 – Schemat renderera (część III)

Lokalny procesor dla kolejnych promieni pobieranych ze swojej kolejki (algorytm LIFO lub FIFO – wybór użytkownika) używa hierarchii obiektów do znalezienia przecięcia z najbliższym obiektem. Po znalezieniu przecięcia procesor wybiera światła nie zasłonione przez inne obiekty (promienie cienia), oblicza parametry oświetlenia obiektu, oraz przez uwzględnienie materiału wylicza kolor trafienia. Kolor ten jest dodawany do koloru próbki (na schemacie funkcja *Ray::SetColor()* korzysta z *Sample* aby wstawić kolor w odpowiednim miejscu bloku). Na koniec procesor generuje promienie wtórne, które dodaje do kolejki promieni (ograniczenia generacji definiowane przez użytkownika to stopień zagłębienia oraz minimalny współczynnik koloru).

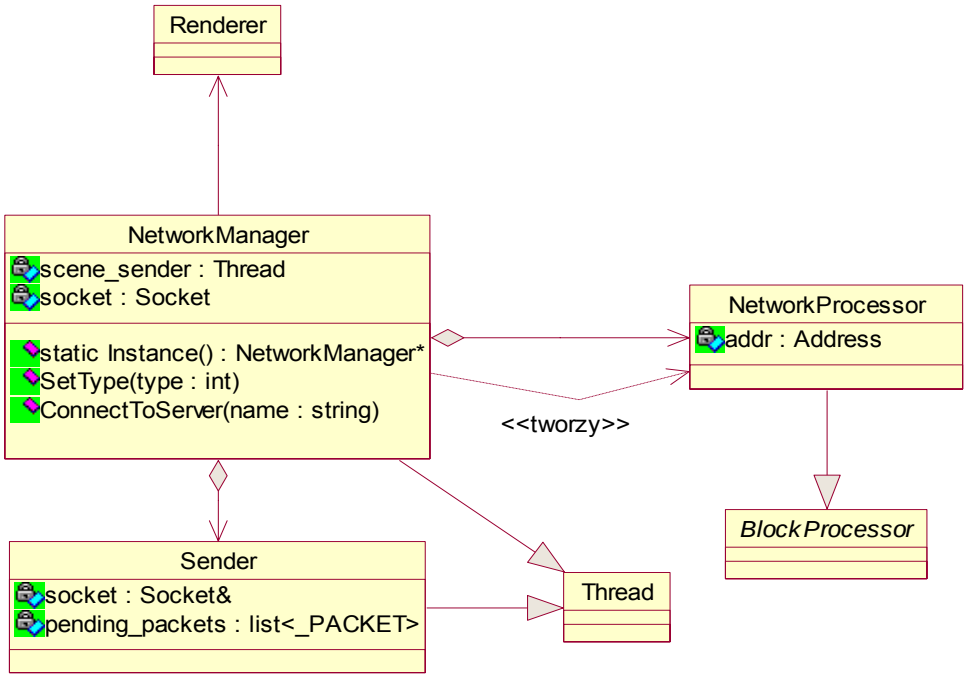
Renderowanie jednego bloku w procesorze lokalnym kończy się w momencie wyczerpania kolejki promieni.

9.2.5 Budowa warstwy sieciowej

Obiekt globalny *NetworkManager* umożliwia rozproszoną syntezę obrazu przez sieć. Może on działać w jednym z dwóch trybów: jako serwer lub klient. Serwer jest właściwą instancją renderującą obraz. Klient udostępnia swoją moc obliczeniową i renderuje tylko bloki, które wyznaczy mu serwer. Do jednego serwera może być podłączonych wiele klientów.

Do komunikacji zaproponowano protokół oparty na wiadomościach działający na gniazdach bezpołączeniowych UDP. Można w nim wyróżnić trzy etapy: rejestrację klienta, renderowanie (wykonywanie zadań) i wyrejestrowanie klienta. W sieci lokalnej, serwer wysyła do wszystkich (*ang. broadcast*) wiadomość HITHERE. Odbierają ją instancje *NetworkManagera* nie połączone (stan nieustalony – ani serwer, ani klient). Dzięki temu użytkownik może wybrać, do którego serwera chce podłączyć klienta. W Internecie użytkownik musi podać dokładny adres IP, ponieważ nie jest tam możliwe wysyłanie wiadomości typu *broadcast*.

Aby się połączyć z serwerem, klient wysyła wiadomość JOIN. Następnie tworzy on gniazdo nasłuchowe strumieniowe TCP i wykonuje funkcję *Listen()* czekając na połączenie z serwerem. Serwer po otrzymaniu żądania połączenia JOIN również tworzy gniazdo TCP i łączy się z klientem. To połączenie TCP jest wykorzystywane do przesłania sceny. Scena jest przesyłana przy użyciu obiektów pochodnych klasy *Storage*. Serwer w celu przesłania sceny wykorzystuje osobny wątek, który nie zakłóca normalnej pracy. Po pomyślnym przesłaniu sceny obie strony zamykają połączenie i uznają rejestrację za zakończoną z sukcesem. Jeśli w trakcie transferu sceny wystąpią komplikacje, połączenie jest przerywane a rejestracja zaniechana.



Rys. 9.9 – Budowa warstwy sieciowej

Po zarejestrowaniu klienta, serwer tworzy obiekt *NetworkProcessor* będący procesorem bloków i odpowiadający klientowi. W momencie gdy Renderer przekazuje mu zadanie do wykonania, *NetworkManager* wysyła je do klienta przy pomocy komunikatu BLOCK. Klient odbiera komunikat, a dysponuje dokładną kopią sceny, więc wykonuje

zadanie na swoim lokalnym procesorze. Po zakończeniu zadania odsyła piksele zawarte w wyrenderowanym bloku komunikatem RESULT. Każdy komunikat ma 512 bajtów, więc wynik bloku 8x8 mieści się w jednym komunikacie, a wynik bloku 16x16 w czterech. Wynik po odebraniu przez serwer jest akceptowany i przekazywany przez *NetworkProcessor* do renderera, który generuje następne zadanie. Ponieważ renderer pilnuje, aby każdy procesor miał przekazane co najmniej trzy zadania, klient nie czeka na następne zadanie i od razu wykonuje kolejne zadanie które było przekazane wcześniej. Po odesłaniu wyniku, klient wysyła również statystyki (ilość przetworzonych promieni i pikseli) komunikatem STATS.

W momencie przerwania pracy klienta, klient wysyła komunikat GOODBYE. Serwer co pewien czas (domyślnie 10 sekund) wysyła do klientów komunikat CHECK aby sprawdzić czy jeszcze pracują. Komunikaty JOIN, BLOCK, RESULT i CHECK wymagają odesłania potwierdzenia o odebraniu komunikatem RECEIPT. *NetworkManager* posiada specjalny wątek *Sender* służący do wysyłania komunikatów. Dla wyżej wymienionych komunikatów wznawia on ich wysłanie jeśli nie otrzymuje RECEIPT. Jeśli po pewnym czasie (domyślnie 5 sekund) nie otrzyma odpowiedzi, stwierdza że adres jest „martwy". W przypadku serwera oznacza to wyrejestrowanie klienta i przekazanie jego zadań do puli zadań niewykonanych, które rozdzielane są w pierwszej kolejności.

9.2.6 Analiza wzrostu wydajności podczas obliczeń w systemie rozproszonym

W celu przetestowania modułu sieciowego wyrenderowano prostą scenę złożoną z 11 różnorakich obiektów prostych. Nie miała przy tym znaczenia złożoność sceny, ale ilość promieni – renderowany obraz był rozmiarów 1600x1200 (800x600 z antialiasingiem 2x2).

Do testu wykorzystano cztery komputery z procesorami Athlon z zegarami 1470MHz (XP1700+), 1600MHz (XP1900+), 1400MHz i 1000MHz (jako serwer pracował procesor 1470MHz). Dla porównania uruchomiono renderowanie najpierw na jednym z nich (1470), a potem na wszystkich jednocześnie. Biorąc pod uwagę zegary tych procesorów, ich sumaryczna wydajność jest 3.72 raza wyższa niż jednego z nich (1470). Wyniki okazały się następujące:

Tabela 7.1 – Czasy renderowania na jednym i czterech komputerach

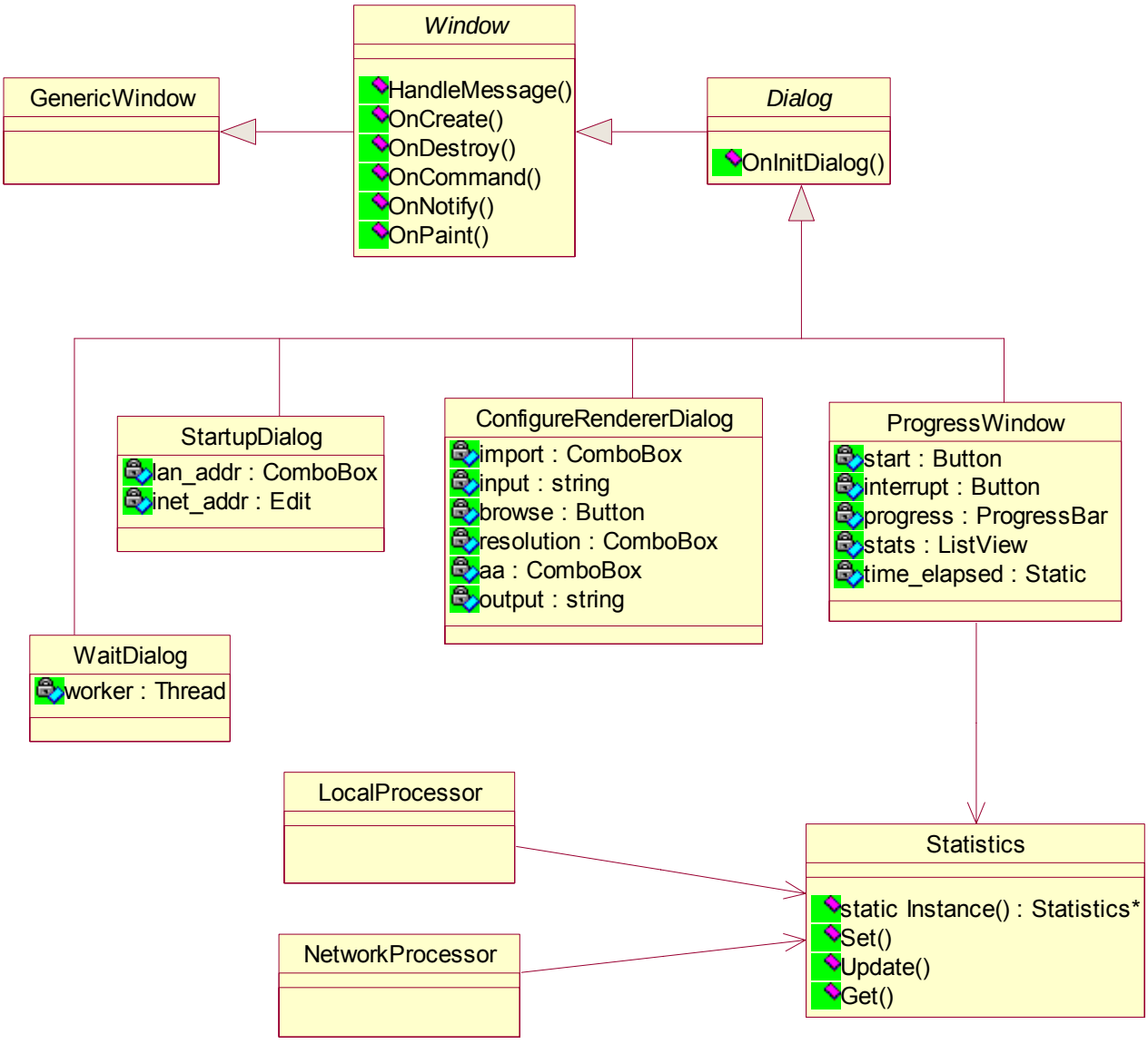
		razem			
	sam 1470	serwer 1470	1600	1400	1000
liczba promieni (w tysiącach)	3308	613	1063	873	760
czas obliczeń	23s	7s			

Dla pewności pomiary wykonano kilkakrotnie (zauważono wahania nie znaczące na poziomie 5 tysięcy promieni). W zespole czterech komputerów obraz był wygenerowany w 7 sekund, czyli 3.28 raza szybciej niż na jednym komputerze (1470). Jest to wynik na poziomie 88% mocy sumarycznej czterech użytych komputerów. Pozostała moc została wykorzystana na zarządzanie siecią; widać spadek mocy serwera, który ma wyższy zegar niż procesor 1400, a jednak przetworzył mniej promieni.

Na wyniki nieznaczny wpływ miała konfiguracja komputerów – pierwsze dwa to Athlony XP działające z pamięciami DDR266, a dwa pozostałe to starsze Athlony z pamięciami SDR133. Nie mogło mieć to dużego wpływu ze względu na niewielką scenę i małe tekstury.

9.2.7 Struktura interfejsu użytkownika

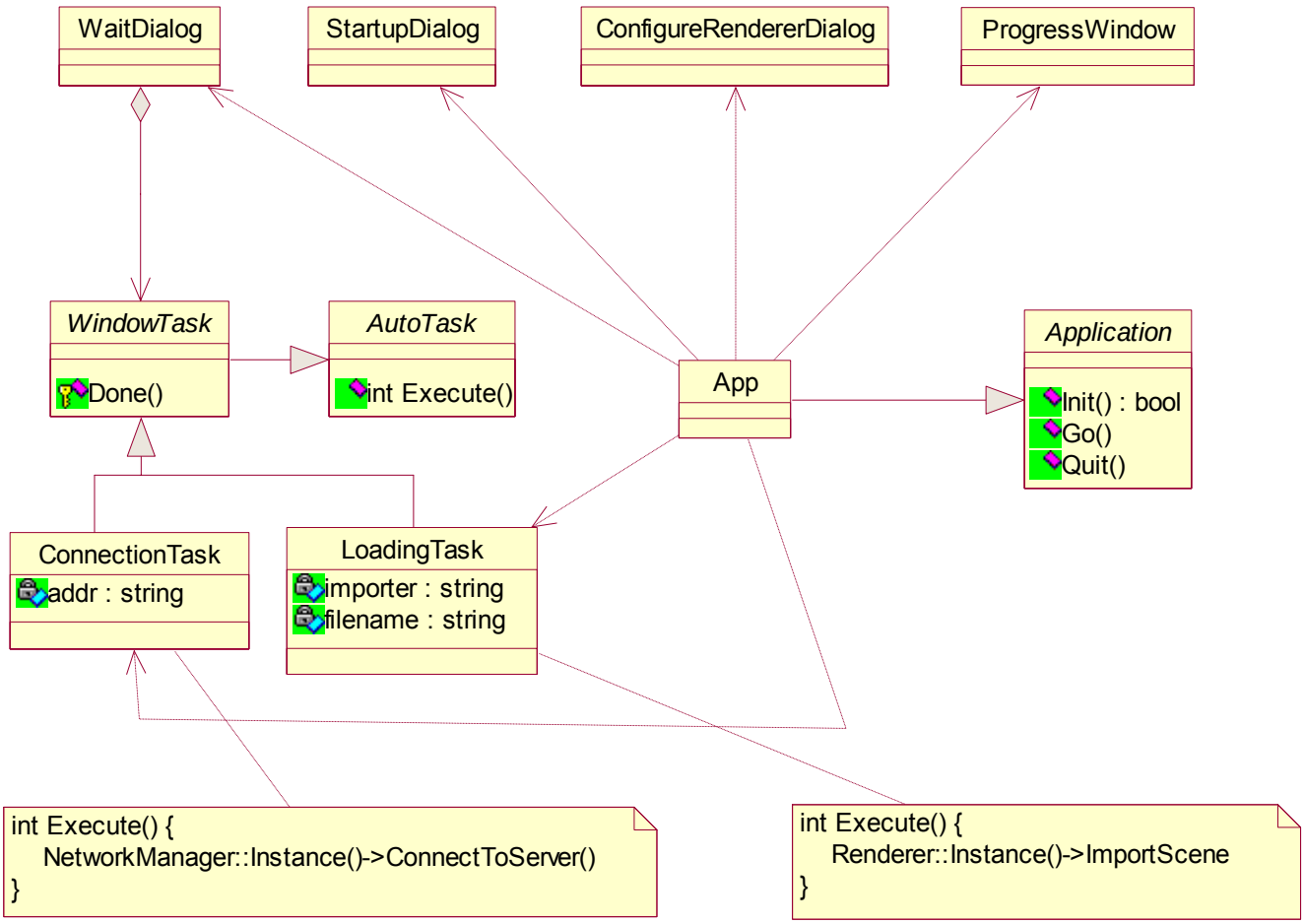
Interfejs użytkownika jest elementem zależnym od systemu operacyjnego. Interfejs dla implementacji programu na potrzeby niniejszej pracy, pod systemem Windows, został oparty na prostej bibliotece klas pokrywającej (*ang. wrapper*) funkcjonalność Windows API. Głównym obiektem jest obiekt klasy *App*. Inicjuje on program, wczytuje konfigurację i wyświetla kolejne konfiguracyjne okna dialogowe.



Rys. 9.10 – Struktura interfejsu użytkownika (część I)

Są cztery rodzaje okien dialogowych: *WaitDialog* służy do wykonywania długich operacji w tle (łączenie się z serwerem *ConnectionTask* i wczytywanie sceny *LoadingTask*), *StartupDialog* pozwalające użytkownikowi wybrać tryb pracy programu (serwer lub klient), *ConfigureRendererDialog* pozwalające skonfigurować serwer (wczytać scenę, ustawić rozdzielczość i nazwę pliku wynikowego) oraz *ProgressWindow*. Trzy pierwsze są oknami

modalnymi, a *ProgressWindow* oknem niemodalnym wyświetlającym postęp w renderowaniu oraz sumaryczny czas.



Rys. 9.11 – Struktura interfejsu użytkownika (część II)

9.3 Instrukcja obsługi

Program przetestowano pod systemem Windows 2000.

Po uruchomieniu program wczytuje plik konfiguracyjny *config.txt*. W pliku tym użytkownik może zdefiniować rozmiary bloku (zadania renderera), kolor tła, sposób działania kolejki lokalnego procesora bloków oraz ograniczenia dla promieni wtórnych (maksymalna głębokość rekurencji i minimalny współczynnik promienia).

Następnie użytkownik wybiera tryb pracy programu: serwer, klient w sieci LAN i klient w sieci Internet. W przypadku klienta sieci LAN na liście pojawiają się dostępne serwery (może to odbywać się z opóźnieniem do 10s) a w przypadku klienta w sieci Internet należy podać adres serwera.

Jeśli program zostanie uruchomiony jako klient, podczas połączenia pobierana jest z serwera scena, po czym otwiera się okno postępu. Wszystkie elementy tego okna są nieaktywne, oprócz przycisku umożliwiającego wyjście, a tym samym rozłączenie z serwerem.

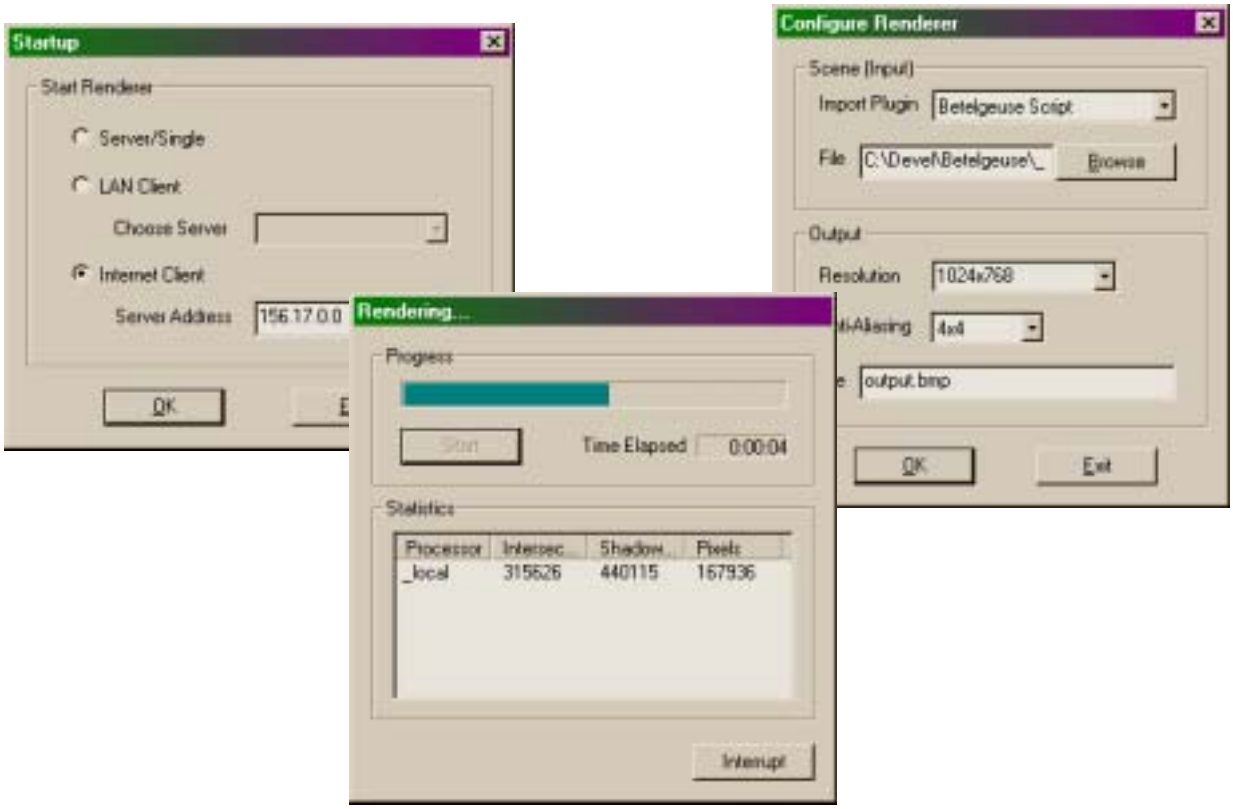
W przypadku uruchomienia w trybie serwera, otwiera się okno konfiguracji renderera. W oknie tym wybiera się typ importera i plik źródłowy zawierający scenę w formacie wybranego importera (domyślnie jest to *Betelgeuse Script*), rozdzielczość generowanego obrazu, tryb antialiasingu (1x1 – normalny, 2x2 – renderowany obraz 4 razy większy, 4x4 – obraz 16 razy większy), oraz nazwę pliku wynikowego. Program po zakończeniu zapisuje wygenerowany obraz w formacie Windows BMP 24bit.

Po skonfigurowaniu serwera otwiera się okno postępu. Aby rozpocząć generowanie obrazu należy wcisnąć przycisk „Start". Nie dzieje się to od razu, gdyż użytkownik może chcieć poczekać na zarejestrowanie się klientów. Rejestrowanie klientów jest robione zawsze, zarówno przed jak i po rozpoczęciu renderowania.

W czasie renderowania wyświetla się pasek postępu, a także całkowity czas renderowania. Na dole wyświetla się lista z zarejestrowanymi procesorami bloków (lokalny widoczny jest jako „_local"). Dla każdego z nich podana jest liczba promieni jaką przetworzył.

Renderowanie można w każdej chwili przerwać przyciskiem „Interrupt". Po zakończeniu używa się go również do wyjścia z programu.

Obraz wynikowy jest zapisywany po wyjściu programu. Jeśli renderowanie zostało przerwane, zapisywane jest tyle ile zostało wyliczone.



Rys. 9.12 – Okna programu

Program zawiera importer scen w formacie *Betelgeuse Script*, przygotowanym na potrzeby niniejszej pracy. Jest to format tekstowy, który jest przetwarzany w jednym przebiegu. Można w nim dołączać tekstury o dowolnych wymiarach zapisane w formacie Windows BMP o dowolnej głębokości kolorów. Dokładny opis formatu został zamieszczony na CD-ROMie dołączonym do pracy.

10 Podsumowanie

W niniejszej pracy przedstawiono architekturę programu do syntezy cyfrowych obrazów metodą raytracingu. Dobrano szereg algorytmów gwarantujących podstawową jakość obrazu, wyższą niż w przypadku algorytmów rasteryzujących, a także poprawiających efektywność obliczeń. Zastosowano algorytm podziału zadań dla obliczeń rozproszonych.

Program zaimplementowano dla systemu Windows i przetestowano pod Windows 2000. Wybór nowoczesnego języka programowania, jakim jest C++, oraz nowoczesnego kompilatora zagwarantował nie tylko lepsze możliwości implementacyjne, ale również wyższą wydajność kodu.

Przetwarzanie współbieżne przetestowano w systemie rozproszonym złożonym z czterech komputerów. Odnotowano 88% wykorzystanie ich mocy obliczeniowej.

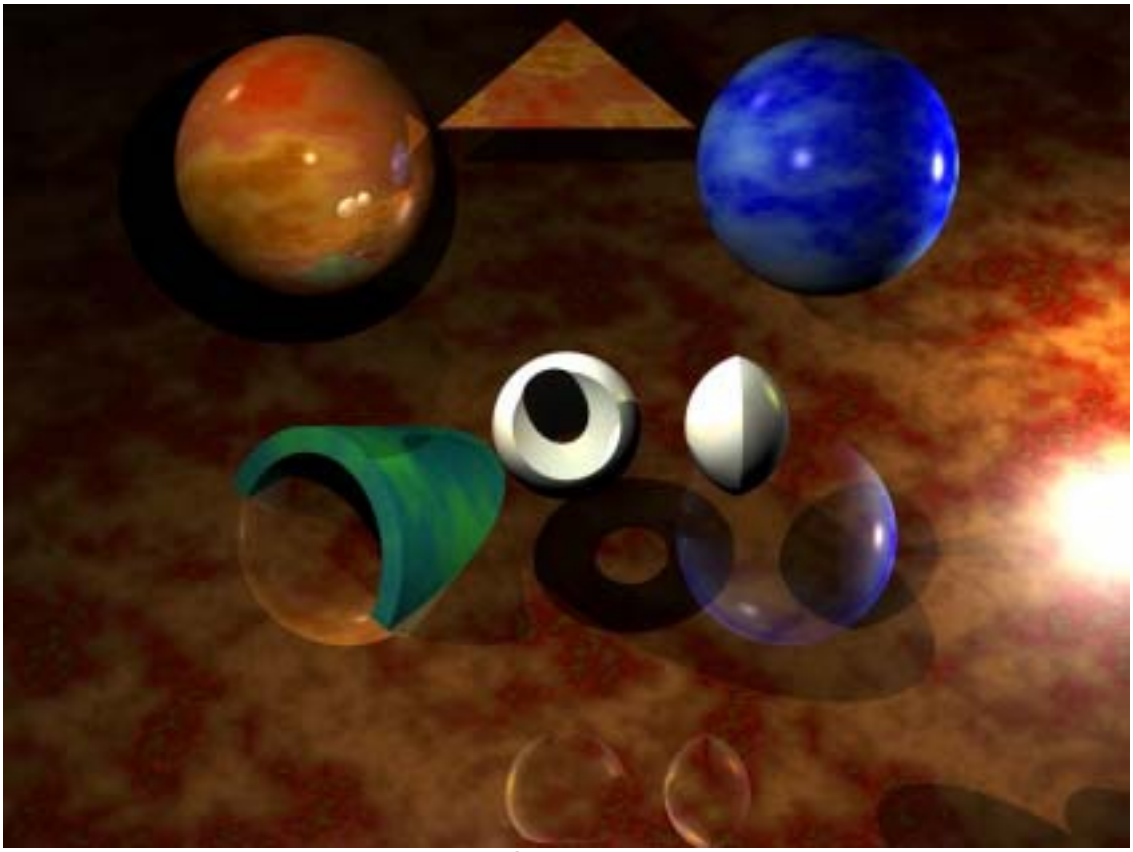
Zaimplementowany program, dzięki stosownej architekturze, może z powodzeniem zostać przeniesiony na inne platformy sprzętowe. Odpowiednie moduły programu mogą również zostać wymienione tak, aby móc skorzystać ze sprzętowego wspomagania niektórych obliczeń.

Program może zostać poszerzony o dodatkowe moduły, typy obiektów, efekty oraz rodzaje plików wejściowych. Interfejs użytkownika jest luźno powiązany z sercem programu, dlatego może zostać poszerzony lub zastąpiony innym.

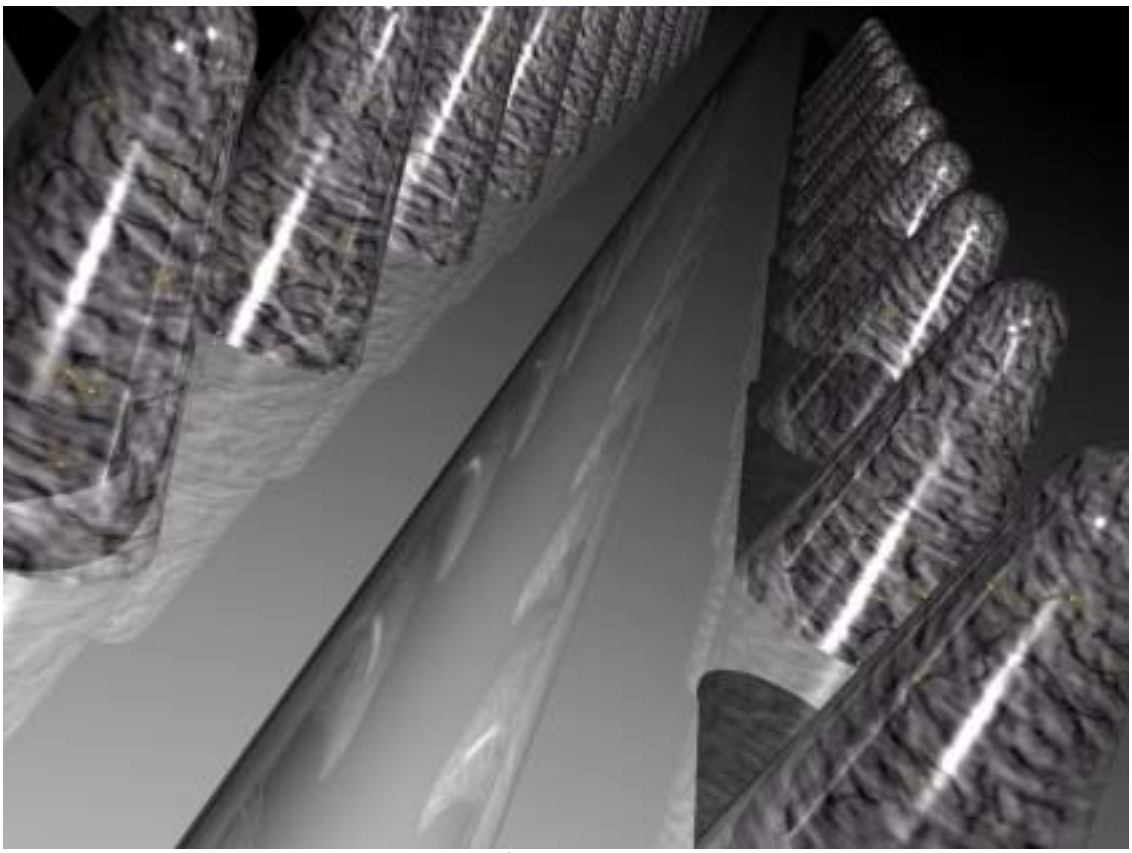
Poniżej zamieszczono obrazy kilku scen wyrenderowanych przy pomocy stworzonego programu. Następująca tabela prezentuje czasy renderowania tych scen na komputerze z procesorem Athlon 650MHz dla rozdzielczości docelowej 640x480 (antialiasing 1x1).

Tabela 10.1 – Czasy renderowania testowych scen

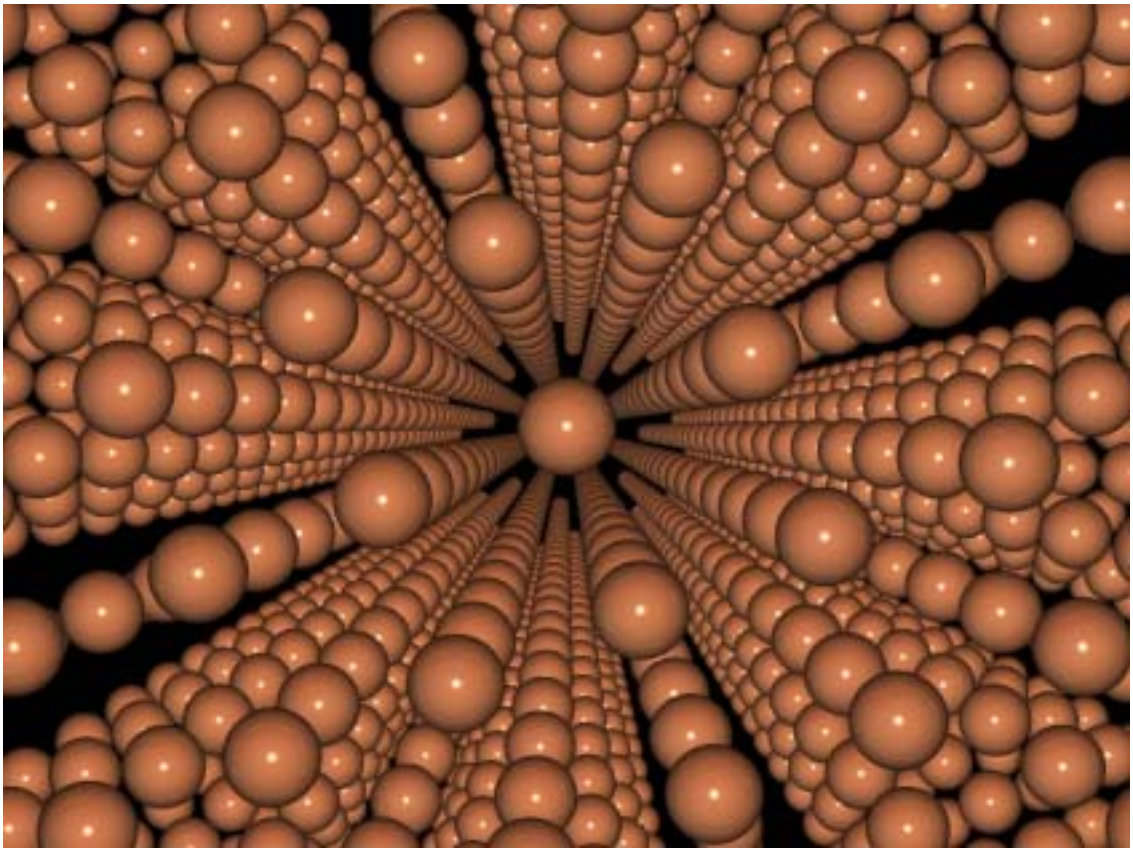
nr sceny	liczba obiektów	czas [s]
1	11	8
2	46	11
3	4 miliony	62



Rys. 10.1 – Scena 1, 11 obiektów prostych, czas renderowania – 8s



Rys. 10.2 – Scena 2, 46 obiektów prostych, czas renderowania – 11s



Rys. 10.3 – Scena 3, 4 miliony kul, czas renderowania – 62s

Bibliografia

[1] Turner Whitted, *An improved illumination model for shaded display*, Communications of the ACM, 23(6), str. 343-349, czerwiec 1980

[2] Bui-Tuong Phong, *Illumination for Computer Generated Images*, Communications of the ACM, 18(6), str. 311-317, czerwiec 1975

[3] A. Appel, *Some Techniques for Shading Machine Renderings of Solids*, AFIPS 1968 Spring Joint Computing Conference, str. 37-49, 1968

[4] James F. Blinn, *Models of Light Reflection for Computer Synthesized Pictures*, Computer Graphics, Proceedings, Annual Conference Series, 1977, ACM SIGGRAPH, str. 192-198

[5] James T. Kajiya, *Anisotropic reflectance models*, Computer Graphics, 19(4), Proceedings, Annual Conference, 1985, ACM SIGGRAPH, str. 15-21

[6] James T. Kajiya, *The Rendering Equation*, Computer Graphics, 20(4), Proceedings, Annual Conference Series, 1986, ACM SIGGRAPH, str. 143-150

[7] A.S. Glassner, *An Introduction to Ray Tracing*, Academic Press, 1989

[8] A.S. Glassner, *Space Subdivision for Fast Ray Tracing*, IEEE Computer Graphics and Applications, 4(10), str. 15-22, październik 1984

[9] Brian Smits, *Efficiency issues for ray tracing*, Journal of Graphics Tools, 3, 1998, str. 1-14

[10] Eric Haines, *Efficiency improvements for hierarchy traversal*, Graphics Gems II, Academic Press, San Diego, 1991, str. 267-273

[11] Eric Haines i D.P. Greenberg, *The Light Buffer: A Shadow-Testing Accelerator*, IEEE Computer Graphics and Applications, 6(9), str. 6-16, wrzesień 1986

[12] T. K. Kay i J. T. Kajiya, *Ray tracing complex scenes*, Computer Graphics, 20, Proceedings, Annual Conference Series, 1986, ACM SIGGRAPH, str. 269-278

[13] A. Woo, *Fast ray-box intersection*, Graphics Gems, Academic Press, San Diego, 1990, str. 395-396

[14] B. Arnaldi, T. Priol i K. Bouatouch, *A new space subdivision method for ray tracing CSG modelled scenes*, The Visual Computer 3, 2, sierpień 1987, str. 98-108

[15] H. Weghorst, G. Hooper i D.P. Greenberg, *Improved Computational Methods for Ray Tracing*, ACM Trans. on Graphics, 3(1), str. 52-69, styczeń 1984

[16] J.G. Cleary, B.Wyvill, G.M. Birtwistle i R. Vatti, *Multiprocessor Ray Tracing*, Research Report Nr. 83/128/7, Dept. of Computer Science University of Calgary, 1983

[17] E. Reinhard i F.W. Jansen, *Rendering large scenes using parallel ray tracing*, First Eurographics Workshop of Parallel Graphics and Visualization, wrzesień 1996, str. 67-80

[18] A. Fujimoto i K. Iwata, *Accelerated Ray Tracing*, Proc. CG Tokyo 1985, str. 41-65

[19] S.M. Rubin i T. Whitted, *A 3-Dimensional Representation for Fast Rendering of Complex Scenes*, Computer Graphics, 14(3), str. 110-116, lipiec 1980

[20] J. Amanatides i A. Woo, *A Fast Voxel Traversal Algorithm for Ray Tracing*, Eurographics '87, Proceedings of the European Computer Graphics Conference and Exhibition, sierpień 1987, str. 3-10

[21] Edytor E. Haines, *Ray Tracing News*
http://www.acm.org/tog/resources/RTNews/html/index.html

[22] BART – *Benchmark for Animated Ray Tracing*, http://www.ce.chalmers.se/BART