Uniwersytet Wrocławski
Wydział Matematyki i Informatyki

Spłaszczone hierarchiczne przedziały ograniczające jako nowa struktura dla metody śledzenia promieni

Piotr Leszczyński

Praca magisterska na kierunku:
Informatyka
Promotor:
dr Andrzej Łukaszewski

Wrocław 2009

Streszczenie

W poniższej pracy opisane są struktury danych wykorzystywane w algorytmach śledzenia promieni.

We wprowadzeniu znajdują się ogólne informacje o metodzie śledzenia promieni. Można w nim przeczytać, gdzie i dlaczego się ją stosuje, są tam zilustrowane i opisane przykładowe efekty jakie można dzięki tej metodzie uzyskać. Jest tam również podana motywacja, która towarzyszyła mi przy pisaniu tej pracy.

Pierwszy rozdział opisuje teorię. Jest to krótkie wprowadzenie do aktualnego stanu w dziedzinie generowania obrazów metodą śledzenia promieni. Są w nim opisane podstawowe algorytmy i struktury, które były do tej pory wykorzystywane. Zawarte są w nim również techniki budowy i przeglądania oraz kilka szczegółów implementacyjnych.

Kolejny rozdział zawiera opis nowych struktur. Zostały one przeze mnie zaprojektowane i zaimplementowane. Są w nim dokładnie przedstawione algorytmy ich budowy i przeglądania.

W trzecim rozdziale opisane są szczegółowe wyniki testów przeprowadzonych na opisanych w rozdziałach pierwszym i drugim strukturach. Wyniki te przedstawiają rozmaite porównania działania tych struktur dla kilku wybranych scen.

Ostatni, czyli czwarty rozdział zawiera podsumowanie, są to moje osobiste wnioski dotyczące działania i możliwości rozwoju opisanych w rozdziale drugim struktur.

Do pracy dołączony jest także program, a jego dokładniejszy opis znajduje się w dodatku na końcu pracy.

Słowa kluczowe

ray tracing, śledzenie promieni, SBVH, BVH, BIH, kdTree

Spis treści

Wprowadzenie

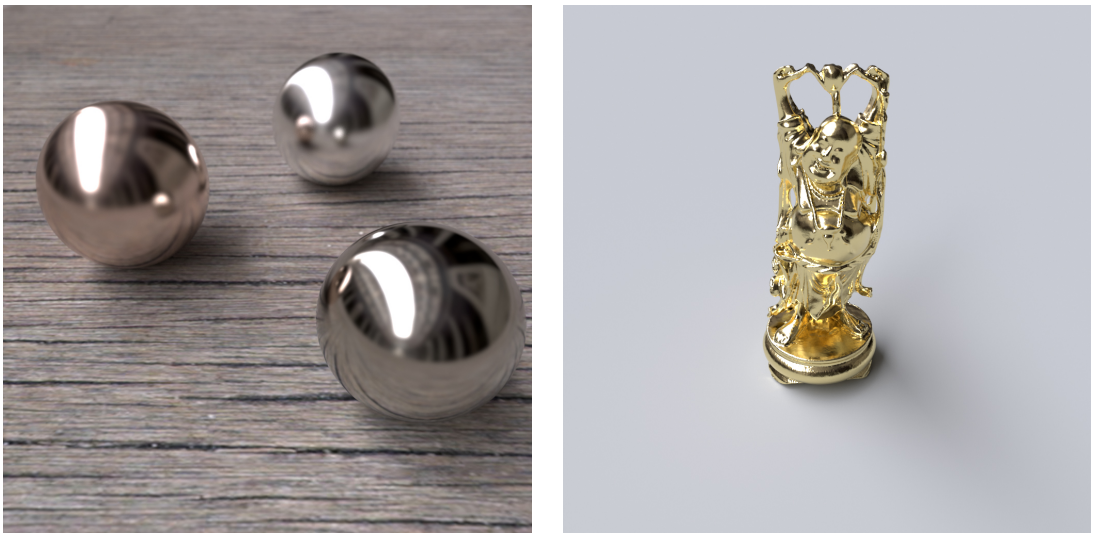
Technika śledzenia promieni jest używana do generowania realistycznych obrazów. Obecnie zdobywa ona coraz większe uznanie w świecie grafiki komputerowej. Dzieje się to głównie dzięki efektom jakie możemy osiągnąć przy jej wykorzystaniu. Przykłady takich efektów przedstawiam na zamieszczonych poniżej rysunkach (rys.1, rys.2, rys.3 i rys.4). Wzrastające zainteresowanie doprowadziło do dużego rozwoju algorytmów w tej dziedzinie. Implementacje tych algorytmów stały się bardzo efektywne, wykorzystują bowiem maksymalnie sprzęt komputerowy. Należy również wspomnieć, że rozwój tej techniki wiele zawdzięcza rozwojowi przemysłu komputerowego, miniaturyzacji i ciągłemu wzrostowi mocy obliczeniowej. Jeszcze kilka lat temu nikt nawet nie myślał o stosowaniu jej w czasie rzeczywistym. Teraz dla mniejszych scen z prostszymi efektami można osiągać interaktywne czasy animacji już na dobrej klasy komputerach osobistych. Oczywiście o wiele więcej można uzyskać na klastrach komputerowych, które są wykorzystywane w przemyśle filmowym. Zastosowanie klastrów w tej technice ma uzasadnienie w tym, że algorytm śledzenia promieni można w bardzo naturalny sposób urównoleglić. To znaczy, że każdy promień może być liczony osobno i przetwarzany przez inną jednostkę.

(a) (b)

Rysunek 1. Obrazy przedstawiające realizm efektów możliwych do osiągnięcia przy zastosowaniu techniki śledzenia promieni. Niektóre widoczne efekty: załamanie obrazu przy przechodzeniu przez przezroczyste powierzchnie zgodne z prawami fizyki, częściowe odbicia sceny od szklanych powierzchni, „miękkie” cienie i kaustyki.

Źródła: Rysunek 1(a) pochodzi ze strony: http://www.fizyka.umk.pl/~milosz/, rysunek 1(b) pochodzi z artykułu [SWS05].

Realizm widoczny na załączonych rysunkach można osiągnąć dzięki temu, że promienie generowane w programie symulują w pewien sposób rozchodzenie się światła w przestrzeni. W algorytmie nie dzieje się to dokładnie w taki sam sposób jak w rzeczywistości, ale jakby od drugiej strony. Jest to tak zwane wsteczne śledzenie promieni. Promienie generowane są z kamery przez każdy piksel obrazu i śledzona jest ich ścieżka do światła. Ścieżka ta powinna zachowywać jak najwięcej znanych własności fizycznych. Odwzorowanie idealne rzeczywistego światła jest praktycznie niemożliwe. W rzeczywistości promienie świetlne mają postać fal elektromagnetycznych, z których tylko część trafia do ludzkiego oka. Fale te mają różne długości, odbijają się od niektórych przedmiotów, a przez niektóre przechodzą zmieniając jedynie kierunek. W programie komputerowym ograniczamy się do pewnego podzbioru tych fal i każdą z nich reprezentujemy jako promień. W celu przyspieszenia obliczeń i uzyskania jak najlepszych efektów, najlepiej jest wybrać te promienie, które mają największy wkład w powstanie obrazu. Z tego powodu śledzi się je, zaczynając właśnie od kamery. Sekret ogromnej skalowalności algorytmu względem liczby procesorów kryje się w tym, że każdy z promieni ma pewien, zupełnie niezależny od innych wkład w powstanie obrazu, dlatego każdy z nich można obliczać równolegle na osobnych jednostkach.



(a) metalowe kule (b) złoty Budda

Rysunek 2. Wyrenderowane obiekty z zastosowaniem fizycznych właściwości odbijających metali.

Źródło: Rysunki pochodzą ze strony internetowej: http://www.pbrt.org.

Mimo powyżej opisanych ograniczeń technika ta wciąż jest zbyt wolna do interaktywnych zastosowań. Nie jest też wspierana sprzętowo, tak jak standardowa

grafika rastrowa przez karty graficzne. Powstały już jednak prototypowe urządzenia mogące sprzętowo przyspieszyć śledzenie promieni [SWS05]. W przyszłości można się spodziewać jeszcze lepszych rozwiązań.

Realizm powstałych obrazów jest na tyle dobry, iż stosuje się technikę śledzenia promieni mimo problemów związanych z wydajnością. Robi się to jednak głównie przy tworzeniu filmów, animacji, reklam, czy wizualizacji, gdzie nie wymaga się żadnej interaktywności. Często tworzy się obrazy poświęcając wydajność dla relistycznego wyglądu. Właśnie w tym celu korzysta się z ogromnych mocy obliczeniowych oferowanych przez klastry komputerowe.

Za ciekawostkę pokazującą relację efektywności do otrzymywanych wyników może posłużyć tutaj fakt, że w 1995 roku średni czas potrzebny na wyrenderowanie jednej klatki filmu „Toy Story" wynosił 2 godziny. W 2005 roku średni czas potrzebny na wygenerowanie jednej klatki filmu „Cars", z którego kadry znajdują się na rysunku 3, wynosił aż 15 godzin! Stało się to, mimo iż w tym czasie wydajność komputerów wzrosła o jakieś 300 razy. Zalety jakie daje artystom ta technologia sprawiają, że czas przestaje grać zasadniczą rolę i poświęca się go dla uzyskania lepszych efektów.

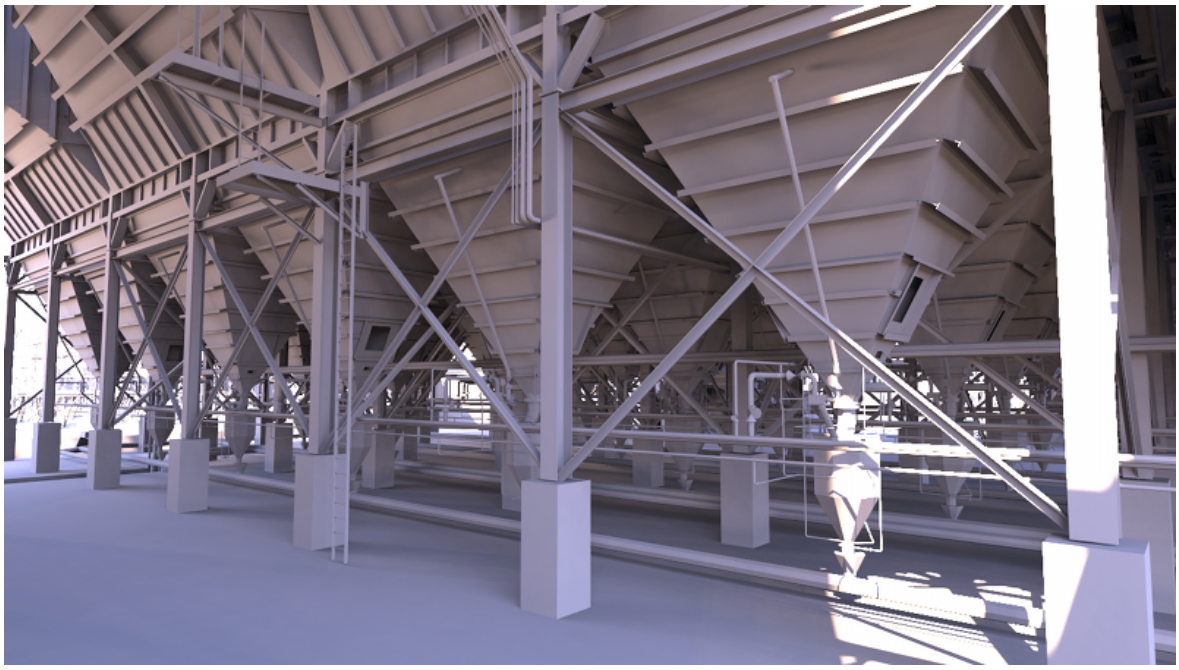
(a) Mater i Lightning McQueen (b) Doc Hudson

Rysunek 3. Obrazy wygenerowane podczas tworzenia filmu „Cars".

Źródła: Rysunek 3(a) pochodzi ze strony: http://www.meeko.org, rysunek 3(b) pochodzi z artykułu [CFLB06].

Pomimo tego, że technika śledzenia promieni została użyta po raz pierwszy około 40 lat temu, dalej znajdowane są w niej nowe ścieżki rozwoju. Proponowane są rozwiązania polepszające zarówno wydajność, jak i jakość generowanych obrazów. W tym miejscu chcę napisać o motywacji, która mi towarzyszyła podczas pisania tej pracy. Zastanawiałem się dość długo, co jeszcze można zrobić, by proces generowania obrazów bardziej przyspieszyć. Dzięki temu mogłoby powstać więcej filmów, czy animacji z pięknymi efektami graficznymi w znacznie krót-

szym czasie. Duże korporacje mogłyby zaoszczędzić sporo czasu i tworzyć jeszcze więcej. Na rynek realistycznych animacji mogłyby zacząć wchodzić nowe firmy niedysponujące tak ogromnym sprzętem. Nieco inną kwestią jest możliwość uzyskania interaktywnych czasów dla prostszych scen i animacji. Dobra grafika nawet w prostych grach, czy programach może zachwycać i być dla nich ogromnym plusem. Ingo Wald w swojej rozprawie doktorskiej [Wal04] opisał wiele możliwych sposobów na przyspieszenie całego algorytmu, rozwiązań zarówno algorytmicznych, jak i sprzętowych. Wyżej opisane cele można osiągnąć przez przyspieszenie działania głównej części całego algorytmu, czyli silnika, który jest odpowiedzialny za efektywne znajdowanie przecięcia danego promienia ze sceną. Najważniejszą częścią tego silnika jest struktura przechowująca scenę. Właśnie o takich strukturach będzie ta praca. Opiera się ona głównie na bardzo interesującym artykule, który ukazał się w roku 2008 [DHK08]. Struktura w nim opisana wykorzystuje większość zalet współczesnych procesorów. Została ona przeze mnie zaimplementowana. Wprowadziłem również kilka pomysłów na jej modyfikację. Pomysły te są dokładnie opisane w poniższej pracy, a więcej na temat implementacji można znaleźć w dodatku A.



Rysunek 4. Rozchodzenie się światła w zacienionych miejscach pokazane na przykładzie modelu elektrowni.

Źródło: Rysunek pochodzi z artykułu [DHK08]

ROZDZIAŁ 1

Technika śledzenia promieni

Po raz pierwszy technika śledzenia promieni na komputerach została użyta w roku 1968. Zastosował ją Arthur Appel do rozwiązywania problemu niewidocznych powierzchni [App68]. Do dnia dzisiejszego bardzo wiele wysiłku zostało włożonego w polepszenie tej techniki. Polepszyła się zarówno jakość generowanych obrazów, jak i wydajność działania algorytmów. Wiele osób wprowadziło w tej dziedzinie mnóstwo rewelacyjnych pomysłów. Implementacje algorytmów, które na początku pojawiały się tylko w teorii, stały się bardzo efektywne. W tym rozdziale opiszę krótko dotychczasową pracę dotyczącą generowania obrazów metodą śledzenia promieni, skupiając się głównie na strukturach drzewiastych wykorzystywanych do przyspieszania znajdowania przecięć promieni ze sceną. Bardzo aktualny stan całej opisywanej dziedziny można znaleźć dość dobrze przedstawiony w pracy [WMG+07].

1.1. Podstawy ogólne

Wyróżnia się trzy główne metody śledzenia promieni:

- rzutowanie promieni (ang. ray casting) – w tej technice śledzi się jeden promień na każdy piksel obrazu, szukając jedynie jego pierwszego przecięcia ze sceną; w łatwy sposób można dodać cienie, robi się to przez śledzenie promienia z obliczonego punktu przecięcia do każdego źródła światła znajdującego się w scenie; jakość uzyskanego obrazu podobna jest do obrazów generowanych przez zwykłą kartę graficzną z buforem głębokości;

- rekurencyjne śledzenie promieni (ang. Whitted ray tracing) – technika zaproponowana przez Turnera Whitteda, opisana w artykule [Whi80], w tej technice również jest śledzony pojedyńczy promień na każdy piksel obrazu, jednak dodatkowo występuje tutaj rekursja, tj. promień może się odbijać lub załamywać, dzięki temu można uzyskać bardziej realistyczne efekty, takie jak odbicia lustrzane, czy załamanie obrazu przy przechodzeniu przez przezroczyste obiekty;

- rozproszone śledzenie promieni (ang. distributed ray tracing) – jest to najbardziej zaawansowana metoda, jak również najbardziej kosztowna; śledzonych może być wiele promieni na każdy piksel obrazu, dodatkowo w każdym

punkcie przecięcia jednego promienia ze sceną może zostać wygenerowanych bardzo wiele nowych promieni, które dalej są rekursywnie śledzone, w związku z tym złożoność obliczeniowa tej metody jest ogromna, pozwala ona jednak uzyskać najbardziej realistyczne obrazy, a w nich takie efekty jak: miękkie cienie, globalne oświetlenie sceny, mieszanie kolorów przy odbiciach, głębię ostrości, czy kaustyki;

Schematy działania wyżej opisanych metod przedstawione są na rysunku 1.1, a przykładowe obrazy pokazujące różnice w uzyskanych obrazach widoczne są na rysunku 1.2.

1.1.1. Algorytm rzutowania promieni

Podstawowy algorytm rzutowania promieni jest bardzo prosty. Składa się z kilku niżej opisanych kroków.
Dla każdego piksela p z płaszczyzny obrazu ip wykonaj kolejno:

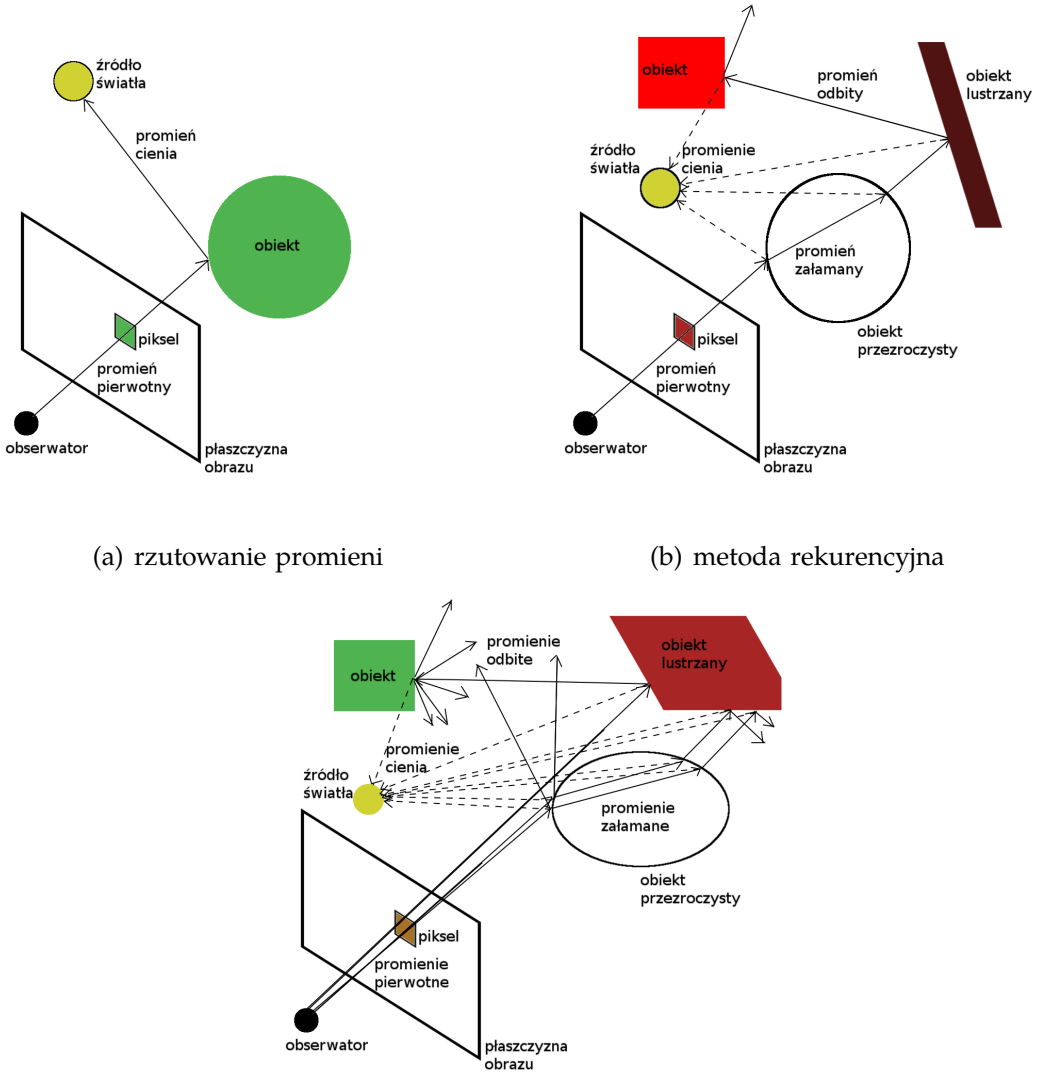
1. wygeneruj promień r z kamery c przechodzący przez piksel p
2. znajdź najbliższe przecięcie i promienia r ze sceną s
3. dla obliczonego przecięcia i oblicz kolor col (generując promień cienia i śledząc jego ścieżkę)
4. zapisz kolor col w pikselu p

Algorytmy rekurencyjnego, czy rozproszonego śledzenia promieni nie różnią się zbytnio od wyżej opisanego. Najważniejszą różnicą jest sposób obliczania natężenia oświetlenia w punkcie przecięcia oraz liczba śledzonych promieni i długości ścieżek mających wpływ na kolor danego piksela. Właśnie od liczby promieni i długości ścieżek zależy liczba wywołań funkcji szukającej najbliższego przecięcia promienia ze sceną. Wywołania tej funkcji pochłaniają najwięcej czasu tego algorytmu. Więcej o złożoności znajduje się w kolejnym punkcie.

1.1.2. Złożoność

Złożoność metody śledzenia promieni można oszacować w prosty sposób rozbijając algorytm na mniejsze składowe:

- generowanie promienia przechodzącego przez dany piksel obrazu, jest zwykłą operacją różnicy dwóch punktów w przestrzeni 3D (czyli jest to kilka mnożeń i dodawań);
- obliczanie koloru piksla także odbywa się w czasie stałym, jednak czasem, gdy mamy do czynienia z teksturami, potrzebnych jest nieco więcej operacji;



(a) rzutowanie promieni

(b) metoda rekurencyjna

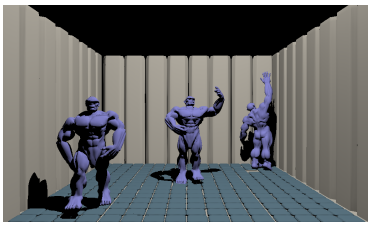
(c) metoda rozproszona

Rysunek 1.1. Porównanie schematów działania metod śledzenia promieni.

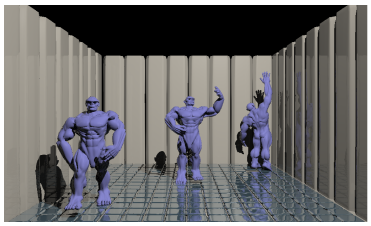
Źródło: Opracowanie własne.

- operacja znajdowania najbliższego przecięcia promienia ze sceną jest najbardziej kosztowna; złożoność tej operacji zależy w głównej mierze od wielkości sceny; przeważnie sceny składają się z trójkątów, dlatego na nich zamierzam się skupić; trywialnym sposobem znalezienia najlepszego przecięcia jest sprawdzenie wszystkich możliwości;

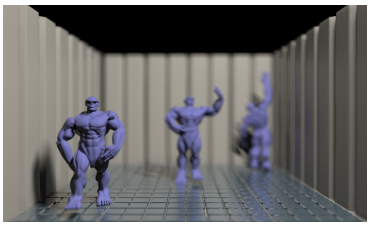
Z tej krótkiej analizy wynika, że złożoność najprostszego algorytmu śledzenia promieni dla obrazu o wysokości w, szerokości h i sceny zawierającej n trójkątów



(a) rzutowanie promieni



(b) metoda rekurencyjna



(c) metoda rozproszona

Rysunek 1.2. Porównanie efektów uzyskiwanych przy różnych metodach śledzenia promieni.

Źródło: Rysunek pochodzi z artykułu [BEL+07]

przedstawia się następująco:

$$\mathrm{O}(wh) \cdot \mathrm{T}(n)$$

Gdzie T(n) oznacza złożoność algorytmu szukania najbliższego przecięcia promienia ze sceną o n trójkątach. Najprostsza metoda sprawdzająca wszystkie możliwości ma liniową złożoność. O tym jak znajdować takie przecięcie szybciej, opisałem w rozdziale 1.2.2.

Złożoność algorytmu rekurencyjnego śledzenia promieni w teorii jest taka sama, gdyż dla każdego promienia liczymy rekurencyjnie pojedyńczą ścieżkę, której długość nie przkracza 5–8 poziomów. Wynika stąd, że algorytm ten jest do kilku razy wolniejszy.

Algorytm rozproszonego śledzenia promieni, w teorii również ma identyczną złożoność. Tutaj również liczba wywołań funkcji szukania najbliższego przecięcia może być ograniczona, jednak stała ograniczająca musiałaby być ogromna. W niektórych technikach śledzi się nawet od kilkuset do kilku tysięcy promieni na każdy piksel obrazu. Każdy z tych promieni podczas odbicia lub załamania może generować nowe promienie. Właśnie tej metody używa się przy produkcji najbardziej realistycznych obrazów.

1.2. Silnik

W tym punkcie opiszę zasadę działania silnika całego algorytmu. Czyli to, jak całość funkcjonuje, jak są znajdowane przecięcia i jak można to działanie przyspieszyć. Bardzo wiele w tej dziedzinie zostało już zrobione, a dokładniejsze informacje na temat dotychczasowych prac można znaleźć w artykule podsumowującym [WMG+07].

1.2.1. Obliczanie przecięć

W celu wygenerowania obrazu należy znaleźć najbliższe przecięcie promienia z trójkątem. W punkcie 1.1.2 jest napisane, że najprostszym sposobem na to jest przejrzenie wszystkich możliwości, czyli obliczenie przecięcia danego promienia z każdym trójkątem ze sceny.

Ta podstawowa operacja algorytmu śledzenia promieni, została już dokładnie przeanalizowana pod względem efektywności. Jednym z najlepszych rozwiązań jest algorytm zaproponowany przez Möllera i Trumborea. Szczegóły można znaleźć w artykule [MT97]. Przebadali oni kilka różnych algorytmów pod względem optymalności i właśnie ich okazał się jednym z najlepszych. Dokładna implementacja tego algorytmu w języku C znajduje się na stronie internetowej[1]. Podczas implementacji przetestowałem jeszcze algorytm Plückera. Korzysta on z iloczynów wektorowych i skalarnych. Dokładny opis można znaleźć na stronie internetowej[2].

Ostatecznie zdecydowałem się na użycie rozwiązania zaproponowanego przez Möllera i Trumborea, gdyż daje ono bardzo dobre rezultaty. Algorytm ten można również łatwo przebudować tak, by w jednym przebiegu obliczał przecięcie z czterema różnymi trójkątami. Takie rozwiązanie wykorzystuje instrukcje SIMD (ang. Single Instruction Multiple Data) procesora, dzięki czemu przyspiesza cały algorytm średnio o około 3 razy. Wiecej o tej technice w punkcie 1.2.4 na stronie 20.

Należy tutaj również wspomnieć o tym, że istnieją bardziej efektywne algorytmy, korzystają one jednak z wcześniej obliczonych wartości dla każdego trójkąta. Takie trójkąty zajmują więcej miejsca w pamięci komputera, przez co algorytmy te wykorzystuje się tylko w niektórych przypadkach.

1.2.2. Struktury

Małe sceny w grafice komputerowej składają się z kilku tysięcy trójkątów, w tych największych trójkąty liczy się w milionach. Oczywistym jest, że dla takich scen nie można sprawdzać przecięć danego promienia z każdym trójkątem, gdyż nie

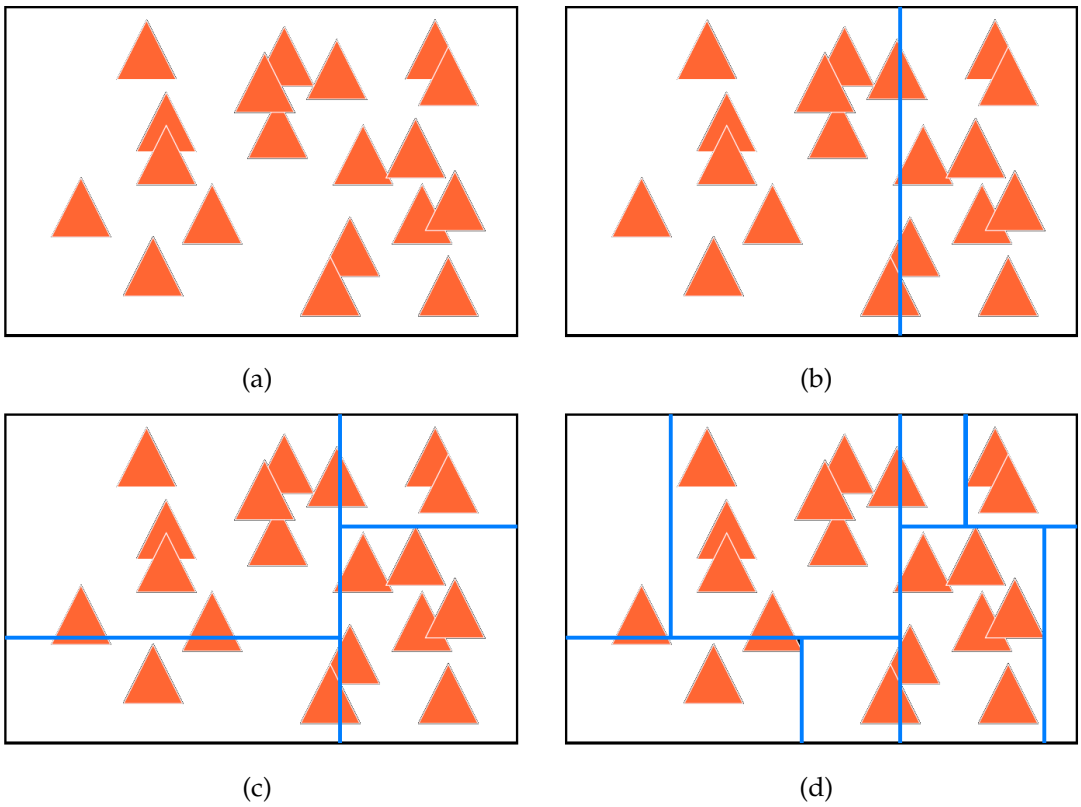
[1] http://jgt.akpeters.com/papers/Chirkov05/

[2] http://www.flipcode.org/archives/Introduction_To_Plcker_Coordinates.shtml

doczekamy się wyniku w rozsądnym czasie. W tym celu powstały różne struktury grupujące trójkąty w sposób umożliwiający bardziej efektywne znajdowanie najbliższego przecięcia poprzez eliminowanie sprawdzania przecięć z jak największą liczbą trójkątów. Struktury te zostały zaprojektowane dla najróżniejszych rodzajów scen. W tej pracy skupię się wyłącznie na strukturach drzewiastych, które potrafią radzić sobie z grupowaniem trójkątów w najczęściej spotykanych scenach. Istnieją jednak sceny z trójkątami rozmieszczonymi w sposób powodujący nieefektywne działanie struktur.

KD–drzewo

KD–drzewo uważane jest za najlepszą strukturę do przyspieszania śledzenia promieni. Jest to drzewo binarne, które dzieli przestrzeń na części za pomocą płaszczyzn. Przykład w 2D pokazuje rysunek 1.3.



Rysunek 1.3. Schemat powstawania kd–drzewa. W kolejnych krokach przestrzenie dzielone są płaszczyzną na dwie podprzestrzenie.

Źródło: Prezentacja Gordona Stolla z konferencji SIGGRAPH 2005.

Każdy węzeł wewnętrzny drzewa posiada informację o osi i płaszczyznie podziału oraz wskaźnik na „dziecko”. Z kolei liść drzewa posiada liczbę znajdujących

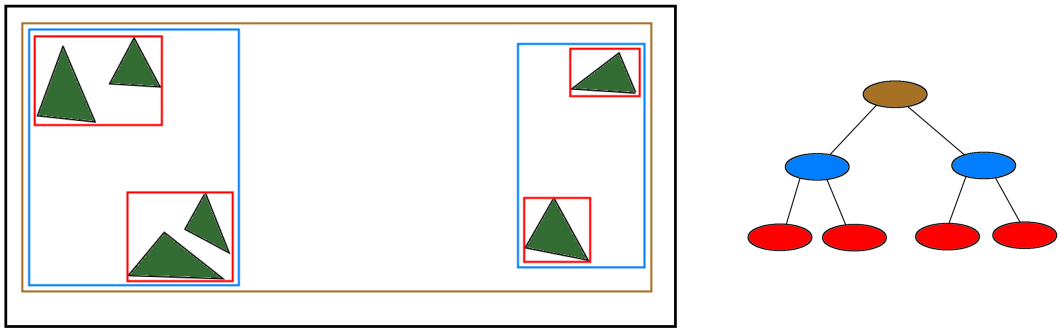
się w nim trójkątów oraz wskaźnik na pierwszy z nich. Wszystkie te informacje można zmieścić w zaledwie 8 bajtach, dzięki czemu struktura ta jest bardzo kompaktowa. Ta własność pomaga zmieścić bardzo dużo węzłów, czyli sporą część drzewa w bardzo szybkiej pamięci cache procesora nawet dla ogromnych scen.

W artykule [WH06] jest dokładnie opisana bardzo efektywna budowa kd–drzewa. Podczas tej budowy wykorzystuje się kilka technik umożliwiających szybką i bardziej optymalną budowę drzewa, m.in. heurystykę pól powierzchni (SAH, ang. Surface Area Heuristics), która jest opisana w punkcie 1.2.3 w dalszej części pracy. W tym miejscu opiszę tylko kilka charakterystycznych własności kd–drzewa. Największą zaletą jest czas szukania przecięcia promienia ze sceną, który jest nieco lepszy niż w strukturach opartych na hierarchiach ograniczających o których będzie mowa w kolejnych punktach tego rozdziału. Do słabych stron kd–drzewa należy skomplikowana implementacja. Utrudnia ją prosty fakt, że podczas budowy przestrzeń dzielona jest płaszczyznami na podprzestrzenie. Trójkąty znajdujące się na granicy tych podprzestrzeni muszą znaleźć się w obu poddrzewach wychodzących z danego węzła. Fakt ten utrudnia również oszacowanie pamięci, którą może zająć gotowa struktura (występują podwójne odwołania do trójkątów). Kolejnym negatywnym wnioskiem wynikającym z tego faktu jest utrudniona przebudowa już gotowego drzewa, która mogłaby być bardzo pomocna przy renderowaniu kolejnych klatek animacji, w których nie doszło do zbyt dużych zmian w scenie.

Hierarchiczne bryły ograniczające

Struktura hierarchicznych brył ograniczających (BVH, ang. Bounding Volume Hierarchies) jest również drzewem binarnym. Bryła ograniczająca (BV, ang. Bounding Volume) jest najczęściej prostopadłościanem o ścianach biegnących wzdłuż osi układu współrzędnych (AABB, ang. Axis Aligned Bounding Box), która ogranicza pewną ważną dla nas przestrzeń. Jak nazwa wskazuje, drzewo jest hierarchią tych brył ograniczających, co znaczy, że każdy węzeł zawiera bryłę, która ogranicza jego „dzieci". W liściach drzewa znajdują się trójkąty. Dla każdego z tych trójkątów w prosty sposób znajduje się bryłę ograniczającą AABB. Przykład takiej struktruktury można zobaczyć na rysunku 1.4.

Najważniejszą różnicą w porównaniu do kd–drzewa jest tu fakt, że podczas budowy drzewa w jednym kroku zamiast rozdzielania przestrzeni dzielimy zbiór trójkątów na dwa rozłączne podzbiory. Dzięki temu rozwiązaniu eliminujemy możliwość podwójnych wystąpień trójkątów w drzewie. Umożliwia to także oszacowanie pamięci potrzebnej do skonstruowania drzewa i zarezerwowanie jej przed budową. Podczas rozdzielania trójkątów na dwa podzbiory teoretycznie dla n trójkątów istnieje 2^n możliwości podziału, jednak dość oczywiste jest, że nie trzeba ich wszystkich sprawdzać. Patrząc na położenie trójkątów w przestrzeni można



Rysunek 1.4. Przykład struktury BVH.

Źródło: Opracowanie własne.

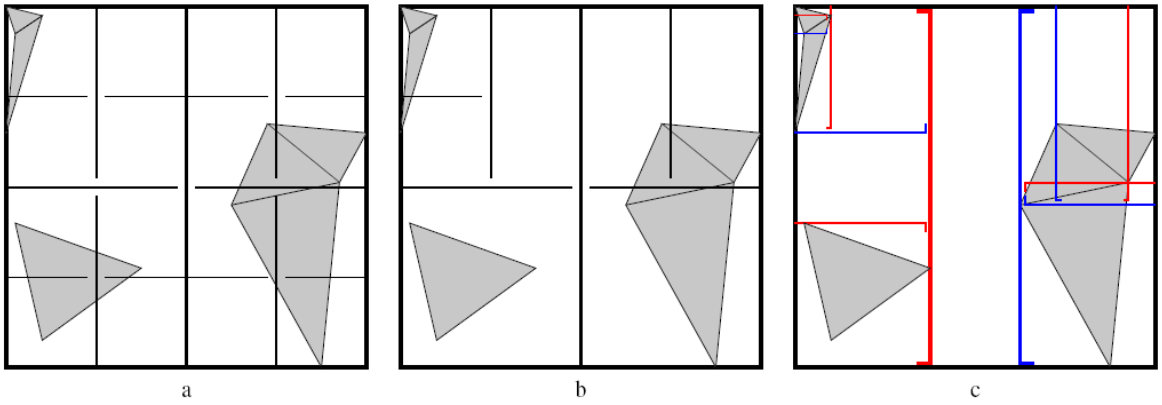
zejść do liniowej liczby możliwości ($3n$ możliwości przy przeglądaniu wszystkich trzech osi). W dalszej części pracy zostanie omówiona technika podziału na koszyki, dzięki której nie trzeba tak dokładnie sprawdzać położenia wszystkich trójkątów. Do zalet struktury BVH należy również błyskawiczna możliwość aktualizacji, która bardzo pomaga w interaktywnych animacjach. Wszystkie te własności można wykorzystać podczas budowy, która jest zdecydowanie bardziej efektywna od budowy kd–drzewa. Do negatywnych cech należy z pewnością rozmiar pojedyńczego węzła i co jest z tym związane całej struktury. Węzeł drzewa BVH zajmuje zdecydowanie więcej miejsca niż węzeł kd–drzewa. Samo trzymanie pojedyńczej bryły AABB wymaga 24 bajtów. Efektywne implementacje korzystają z węzłów 32 bajtowych. Przez taki rozmiar zdecydowanie mniej węzłów może zmieścić się w pamięci cache procesora i przeglądanie BVH dla większych scen może być wolniejsze niż przeglądanie kd–drzewa. Negatywną własnością jest również fakt, że dwie bryły ograniczające mogą na siebie nachodzić, przez co podczas trawersowania drzewa może zajść potrzeba częstszego przeglądania obojga dzieci. Może być to konieczne nawet wówczas, gdy w bliższej bryle otaczającej znajdziemy trójkąt przecinany przez dany promień.

Powstały już bardzo efektywne implementacje BVH, które wykorzystują zaczerpniętą z kd–drzew i nieco zmodyfikowaną heurystykę SAH o której więcej znajduje się w punkcie 1.2.3 na stronie 15. Najwięcej ciekawych i bardzo przydatnych informacji na temat budowy struktury hierarchicznych brył ograniczających można znaleźć w artykule [Wal07].

Hierarchiczne przedziały ograniczające

Struktura hierarchicznych przedziałów ograniczających (BIH, ang. Bounding Interval Hierarchies) została zaprojektowana specjalnie do renderowania animacji.

Charakteryzuje się krótkim czasem budowy, przez co idealnie pasuje do renderowania scen, dla których czas budowy struktury przyspieszającej ma wyraźny wpływ na sumaryczny czas przetwarzania jednej klatki. Jak już nazwa wskazuje jest ona bardzo podobna do struktury BVH. Różnicą są jedynie obszary ograniczające, które tutaj maja postać przedziałów, a nie prostopadłościennych brył, jak to było w przypadku BVH. Przedziały te tworzone są przez dwie płaszczyzny równoległe do jednej z osi układu współrzędnych. Dla każdego węzła ta oś może być inna. Przykład tej struktury w dwóch wymiarach można znaleźć na rysunku 1.5.



Rysunek 1.5. Przykład budowy struktury BIH.

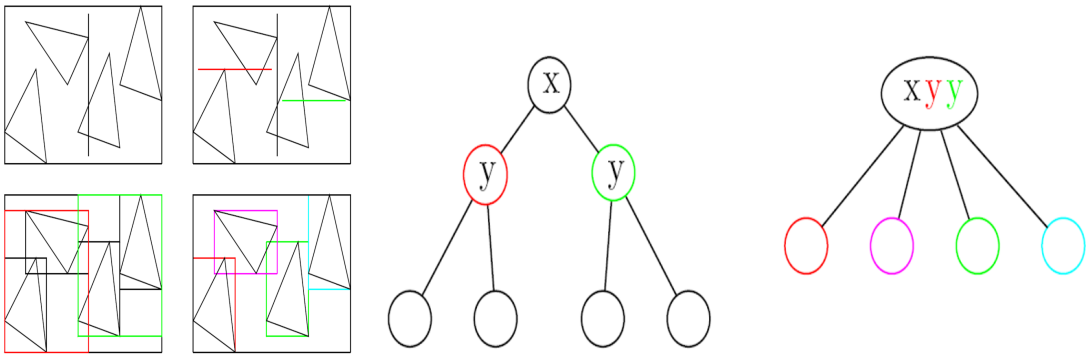
Źródło: Rysunek zapożyczony z artykułu [WK06]

Budowa jest oparta na bryłach AABB trójkątów, podobnie jak było to w drzewie BVH. Przebiega ona jednak zdecydowanie szybciej. Płaszczyzny podziału podczas budowy wybierane są z wcześniej wyliczonych propozycji, dzięki tej operacji oszczędza się mnóstwo czasu. Węzły drzewa BIH w dobrej implementacji zajmują zaledwie 12 bajtów pamięci, co jak już wspominałem jest dużym plusem, gdyż może przyspieszyć przeglądanie drzewa. Zyskując na czasie budowy i zmniejszając pamięć potrzebną na jeden węzeł przez ucięcie czterech ścian bryły AABB, drzewo straciło na jakości. Strata ta widoczna jest przy szukaniu najbliższego przecięcia promienia ze sceną. Więcej informacji o całej strukturze, jej budowie, przeglądaniu i porównaniu z innymi można znaleźć w pracy [WK06].

Spłaszczone hierarchiczne bryły ograniczające

Struktury opisane wyżej należą do najczęściej stosowanych struktur przyspieszających w technice śledzenia promieni. W roku 2008 ludzie z Uniwersytetu w Ulm wpadli na pomysł przebudowania drzew binarnych na drzewa czwórkowe [DHK08], zwane spłaszczonymi hierarchiami brył ograniczających (SBVH, ang.

Shallow Bounding Volume Hierarchies). Gotowa konstrukcja drzewa SBVH różni się od konstrukcji drzewa BVH jedynie liczbą dzieci danego węzła. Sama budowa polega na skonstruowaniu binarnego drzewa BVH, a następnie przekształceniu go w drzewo czwórkowe SBVH. Bardzo dobrze pokazuje to rysunek 1.6.



Rysunek 1.6. Przykład przebudowy struktury BVH w strukturę SBVH.

Źródło: Rysunek pochodzi z artykułu [DHK08]

Za takim rozwiązaniem budowy drzewa przemawia architektura dzisiejszych procesorów. Procesory są w stanie wykonywać operacje na słowach 128 bitowych. Instrukcje takie zwane są SIMD. Przy ich użyciu, operacje mnożenia, dodawania, odejmowania, czy dzielenia 4 liczb zmiennoprzecinkowych zajmują prawie 4 razy mniej czasu niż przy wykorzystaniu zwykłych instrukcji. We wcześniej opisanych strukturach instrucje te były wykorzystywane do efektywniejszego śledzenia pakietów promieni. Rozwiązanie to zwiększało efektywność głównie w metodzie rzutowania promieni. W przypadku pozostałych metod zysk nie był już tak znaczący. Dzięki nowej strukturze spłaszczonych hierarchicznych brył ograniczających instrukcje te zyskują większe znaczenie właśnie w metodach rekurencyjnego i rozproszonego śledzenia promieni, gdzie mogą być wykorzystane na całej długości śledzonej ścieżki. Więcej o instrukcjach SIMD znajduje się w punkcie 1.2.4 na stronie 20.

Węzły powstałe po spłaszczeniu struktury zwiększyły rozmiar. W dobrej implementacji jeden węzeł drzewa SBVH zajmuje 128 bajtów. Rozmiar ten jest dość duży, ale mniejsza liczba węzłów powoduje zmniejszenie rozmiaru całej struktury.

W artykule opisującym efektywną budowę drzewa BVH [Wal07] porównane są przedstawione do tej pory struktury, za wyjątkiem drzew SBVH. Czas budowy liczony na jednym wątku jest zdecydowanie najsłabszy dla kd–drzew. Na wybranym przez autorów artykułu komputerze, drzewo to jest budowane ze średnią prędkością około 150 do 300 tysięcy trójkątów na sekundę. Zdecydowanie lepsze

czasy budowy można osiągnąć dla drzew BVH, które powstają z prędkością od 1,35 do 2,3 miliona trójkątów na sekundę. Wydajność przeglądania tych struktur przy podanych czasach budowy jest zbliżona. Najlepsze czasy budowy uzyskano dla drzew BIH, wynoszą one od 2 do 3 milionów trójkątów na sekundę. Niestety stało się to kosztem wydajności przeglądania, która według autorów wynosi od 60% do 80% wydajności osiąganej dla optymalnych drzew BVH.

1.2.3. Techniki budowy i przeglądania

Podczas opisywania struktur przyspieszających wspomniałem o różnych technikach ich budowy. W kolejnych puntach techniki te zostaną dokładniej omówione. Dzięki nim można w krótszym czasie uzyskać dobrze zbalansowane drzewa, które umożliwiają szybkie znalezienie przecięcia danego promienia ze sceną.

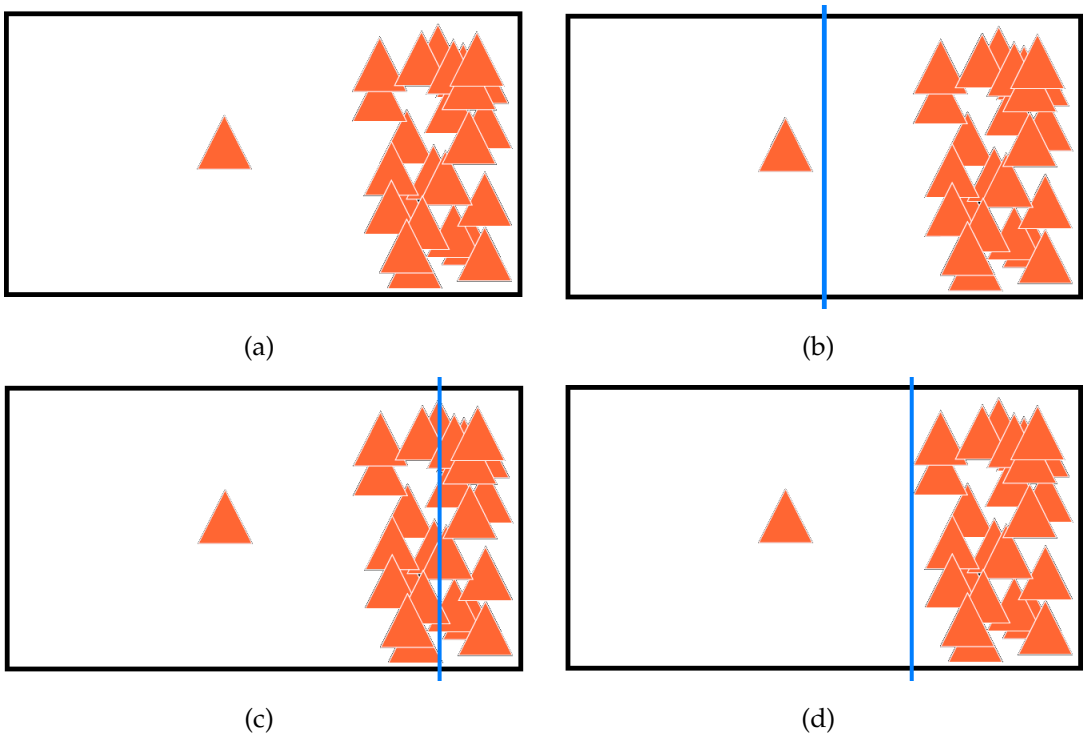
Heurystyka pól powierzchni

Heurystyka pól powierzchni (SAH, ang. Surface Area Heuristics) została stworzona dla kd–drzew, a następnie zaadoptowana do struktur opartych na bryłach ograniczających. Obecnie jest to najlepsza znana metoda pomagająca zbudować optymalne drzewo. Dokładny opis jej zastosowania w budowie obu rodzajów drzew można znaleźć w artykułach [WH06] i [Wal07]. Heurystyka pól powierzchni wykorzystuje zasadę prawdopodobieństwa. Najlepsze miejsce podziału oblicza się minimalizując koszt z wzoru 1.1.

$$Koszt(V \rightarrow \{L, R\}) = K_T + K_I(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R) \tag{1.1}$$

Wzór ten oblicza koszt trawersowania drzewa powstałego przy podziale bryły V na dwie bryły L i R, które zawierają odpowiednio N_L i N_R trójkątów. K_T określa koszt przejścia do niższego poziomu drzewa, czyli jest to koszt metody przecięcia promienia z bryłą AABB. K_I natomiast określa koszt przecięcia promienia z jednym trójkątem. Funkcja SA oblicza pole powierzchni danej bryły. Jak łatwo zauważyć z wzoru 1.1 wynika, że jej wynik jest mniejszy, gdy większa liczba trójkątów jest ograniczona przez mniejszą bryłę. Praktycznym przykładem może być sytuacja, w której promień trawersując strukturę z większym prawdopodobieństwem trafi w większą bryłę, dlatego lepiej, gdy bryła ta ma jak najmniej trójkątów. W dobrze zbudowanym drzewie puste obszary są duże, dzięki czemu mogą zostać szybko przeglądnięte. Przykładem może być rysunek 1.7.

Najlepszym miejscem podziału jest więc minimum funkcji $Koszt$. Znajduje się je przez obliczenie wyniku tej funkcji dla różnych podziałów (dyskretna aproksymacja). Takie obliczenia zajmują sporo czasu, dlatego często bada się tylko wybrane podziały. O tym jak zrobić to efektywnie będzie opisane w następnym punkcie.



Rysunek 1.7. Przykład kroku podziału trójkątów na dwie części. Rysunek 1.7(a) przedstawia scenę z trójkątami. Rysunek 1.7(b) przedstawia podział przestrzeni na dwie równe części, ale nie bierze pod uwagę liczby trójkątów występującej w obu tych częściach. Rysunek 1.7(c) przedstawia podział trójkątów w medianie, tu z kolei nie jest brany pod uwagę podział przestrzeni. Na rysunku 1.7(d) jest przedstawiony podział korzystający z heurystyki SAH, zawiera on duży prawie pusty obszar, przez co maleje prawdopodobieństwo wystąpienia wielu wywołań funkcji obliczającej przecięcie promienia z trójkątami.

Źródło: Rysunek zapożyczony z prezentacji Gordona Stalla z konferencji SIGGRAPH 2005

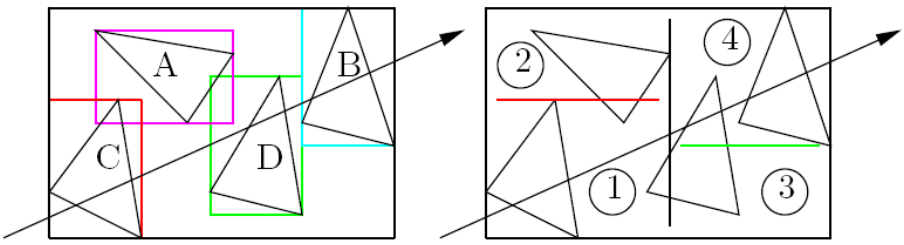
Istnieje jeszcze dość często wykorzystywana przy prostych implementacjach struktur heurystyka mediany. Wyróżnia się heurystykę mediany przestrzeni, pokazaną na rys.1.7(b) i mediany trójkątów, pokazaną na rys.1.7(c). Budowa przy pomocy tych heurystyk jest błyskawiczna, ponieważ ogranicza się liczbę operacji wykonywanych na każdym poziomie budowy drzewa. Metoda ta ma jednak jedną ogromną wadę, którą jest dużo wolniejsze trawersowanie. Trawersowanie nieco lepiej działa w przypadku użycia heurystyki mediany przestrzennej, w której lepiej rozdzielane są puste przestrzenie. W heurystyce mediany trójkątów otrzymujemy dobrze zbalansowane drzewa, ze względu na liczbę trójkątów.

Koszyki na trójkąty

Dobrą techniką przyspieszającą budowę drzewa jest podział trójkątów na koszyki. Przy podziale trójkątów danego węzła na dwie części, aby policzyć minimum funkcji kosztu dla heurystyki SAH należałoby obliczać wzór 1.1 dla każdego możliwego podziału. Zajmuje to bardzo dużo czasu. Badania wykazały, że dokładne znajdowanie minimum nie jest konieczne. Wystarczy znaleźć w miarę dobry punkt podziału. Czas działania można więc zmniejszyć dzieląc na początku trójkąty na równo oddalone w przestrzeni koszyki. Wszystkie kolejne obliczenia, takie jak liczenie minimum funkcji kosztu, wykonuje się na tych koszykach. W aktykule [Wal07] zostało napisane, że już drzewo zbudowane przy pomocy 8 koszyków daje zbliżone rezultaty do drzewa, w którym minimum szukane jest dla wszystkich możliwych podziałów. Czasy budowy takich drzew różnią się jednak znacznie, oczywiście na korzyść tego z zastosowaniem koszyków.

Kolejność przeglądania drzewa

Poza budową ważne jest też przeglądanie drzewa. Warto to robić optymalnie, dlatego należy zwrócić uwagę na kolejność przeglądania dzieci danego węzła. Oczywiście lepiej trawersować jako pierwsze dziecko to, którego bryła otaczająca znajduje się bliżej początku promienia. W przypadku znalezienia przecięcia z trójkątem należącym do tej bryły, trawersowanie kolejnego dziecka może okazać się zbędne, dzięki czemu można zaoszczędzić sporo czasu. Przykład prawidłowego wyboru kolejności trawersowania dzieci w drzewie SBVH znajduje się na rysunku 1.8.



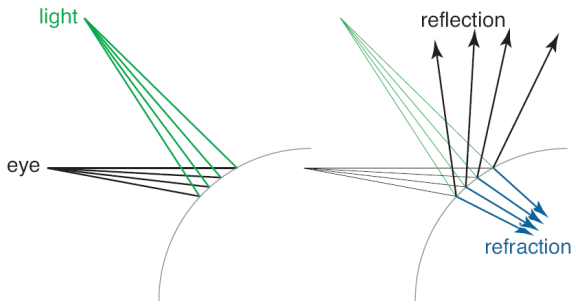
Rysunek 1.8. Przykład prawidłowego wyboru kolejności trawersowania dzieci dla danego promienia w drzewie SBVH.

Źródło: Rysunek pochodzi z artykułu [DHK08]

Trawersowanie pakietów promieni

Interesującą techniką przyspieszania przeglądania drzewa jest śledzenie pakietów promieni. Pakiet taki składa się przeważnie z 4, a czasem z 16 promieni. Wykorzystywana jest tu zbieżność promieni pierwotnych i instrukcje SIMD, o których więcej znajduje się w punkcie 1.2.4. Dzięki instrukcjom SIMD można szybciej znajdować przecięcia promieni z kolejnymi węzłami struktury. Zbieżność promieni pozwala na ograniczenie liczby trawersowań struktury. Możliwe jest to w związku z tym, że promienie z sąsiadujących pikseli w wielu przypadkach trafiają w ten sam obiekt, a czasem nawet w ten sam trójkąt w scenie. Taki obiekt znajduje się przeważnie w jednym bądź dwóch sąsiednich liściach, przez co nie trzeba dla kolejnych promieni trawersować początkowej ścieżki danego drzewa.

Problem pojawia się dla promieni odbitych bądź załamanych. Pakiet takich promieni może stać się rozbieżny, przez co traci się dla tych promieni możliwość zmniejszenia liczby trawersowań struktury. Zdecydowanie bardziej opłaca się w takim wypadku trawersować osobno każdy z promieni. Przykład takiej sytuacji znajduje się na rysunku 1.9.



Rysunek 1.9. Przykład rozbieżności pakietu promieni po odbiciu od kulistej powierzchni.

Źródło: Rysunek pochodzi z artykułu [BEL+07]

1.2.4. Szczegóły implementacji

Dobra struktura, przemyślana budowa i dobra technika trawersowania może bardzo poprawić wydajność algorytmu śledzenia promieni. Jednak detale implementacji w tym przypadku róznież mają bardzo duże znaczenie. Napisanie efektywnego kodu nie jest zadaniem łatwym. W tym rozdziale opiszę kilka szczegółów implementacyjnych, na które warto zwrócić uwagę, gdyż stosowanie lub nie, opisanych poniżej rozwiązań może dość znacznie wpłynąć na szybkość działania całego silnika.

Usunięcie rekursji

Algorytm śledzenia promieni nadaje się znakomicie do zaimplementowania za pomocą rekursji. Nie chodzi tu tylko o metodę rekuryncyjnego śledzenia promieni, ale samą budowę drzewa, czy sposób jego trawersowania.
Rekurencyjny algorytm budowy drzewa dla danego zbioru trójkątów S:

1. jeśli zbiór S jest dość mały, utwórz liść i zakończ algorytm budowy
2. podziel zbiór trójkątów S na dwa podzbiory S_1 oraz S_2 i utwórz węzeł wewnętrzny N
3. wywołaj procedurę budowy drzewa dla zbioru S_1, gotowe drzewo podepnij do węzła N jako lewe dziecko
4. wywołaj procedurę budowy drzewa dla zbioru S_2, gotowe drzewo podepnij do węzła N jako prawe dziecko

Rekurencyjny algorytm przeglądania drzewa dla danego promienia r:

1. jeśli aktualny węzeł jest liściem przetnij promień r ze wszystkimi trójkątami, które się w nim znajdują i zakończ procedurę
2. sprawdź przecięcie promienia r z lewym dzieckiem, jeśli przecina, to wywołaj procedurę przeglądania dla lewego dziecka
3. sprawdź przecięcie promienia r z prawym dzieckiem, jeśli przecina, to wywołaj procedurę przeglądania dla prawego dziecka

Wywołania rekurencyjne zajmują sporo czasu, jedną z przyczyn jest potrzeba rezerwowania pamięci na stosie dla takich wywołań. Wyjściem w tej sytuacji jest przerobienie obu wyżej opisanych rozwiązań na algorytmy, które w pętli, korzystając z wcześniej przygotowanego stosu sprawnie poradzą sobie zarówno z budową drzewa jak i z jego przeglądaniem. W rozwiązaniu tym zamiast wywołania rekuryncyjnego funkcji wystarczy wrzucić odpowiednie dane na stos i przejść do początku pętli, gdzie dane te są ze stosu zdejmowane i wykonuje się na nich wszystkie żądane operacje.

Podział trójkątów „w miejscu”

W każdym kroku budowy drzewa potrzebne jest podzielenie trójkątów na dwa podzbiory według pewnej zasady. Przeważnie polega to na wyborze jednej płaszczyzny. Następnie trójkąty, których środki ciężkości znajdują się po jednej jej stronie trafiają do jednego zbioru, pozostałe do drugiego. Jak już wcześniej wspomniałem,

trójkątów może być bardzo dużo, dlatego ich sortowanie mija się z celem szybkiego algorytmu. W takim wypadku najlepszym wyjściem jest przyjęcie rozwiązania z algorytmu *quicksort*. Scena trzymana jest w pojedyńczej tablicy jako wskaźniki na trójkąty. W momencie podziału, na jej początku i końcu umieszcza się wskaźniki, które pomogają przy podziale. Wskaźniki te iteruje się zatrzymując na niepasujących do danej części trójkątach. Trójkąty te są ze sobą zamieniane miejscami. Rozwiązanie to poza optymalną liniową złożonością ma zaletę w dużej efektywności, gdyż w pamięci zamieniane miejscami są tylko wskaźniki.

Instrukcje SIMD

W punkcie 1.2.2 wspomniałem o instrukcjach, które umożliwiają szybsze wykonywanie wielu operacji arytmetycznych. Instrukcje te noszą nazwę SIMD (ang. Single Instruction Multiple Data). Istnieją wyspecjalizowane procesory, stworzone właśnie dla architektury SIMD. Wykorzystywane są one w komputerach wektorowych, do obliczeń naukowo–technicznych, gdzie przetwarzanych jest wiele strumieni danych w oparciu o jeden strumień rozkazów. Dzisiejsze procesory skonstruowane w architekturze x86 posiadają listę standardowych rozkazów poszerzoną o zestawy instrukcji takie jak MMX, SSE, SSE2 itp. Właśnie te instrukcje wykorzystują architekturę SIMD i umożliwiają wykonywanie podstawowych operacji arytmetycznych na czterech 4-bajtowych liczbach w pojedyńczej instrukcji.

W teorii rozkazy takie powinny przyspieszyć cały program o 4 razy. W praktyce jest to mniej więcej 3-krotne przyspieszenie. W algorytmie, gdzie liczy się każdy szczegół przy poprawie efektywności jest to ogromny postęp. Instrukcje te mogą być wykorzystywane podczas trawersowania struktur drzewiastych, zarówno przy pomocy pakietów promieni dla drzew binarnych, jak i dla jednego promienia. W przypadku drzew czwórkowych, co jest zdecydowanie lepszym rozwiązniem, dla rekurencyjnej i rozproszonej metody śledzenia promieni. Przy trawersowaniu liścia można liczyć przecięcie promienia z czterema trójkątami na raz. Warto wykorzystać zestaw instrukcji SIMD również podczas budowy. Miejsce do ich wykorzystania można znaleźć bez większego problemu.

Alokowanie i wyrównywanie pamięci

Ważną zasadą podczas implementacji szybkich algorytmów jest wyeliminowanie z programu funkcji rezerwujących pamięć. Wywołania takie zajmują sporo czasu. Dlatego w dobrym rozwiązaniu warto zadbać o wstępną rezerwację pamięci na wszystkie potrzebne w czasie działania obiekty. Jest to trochę bardziej utrudnione w przypadku kd–drzew, gdzie jak pisałem w punkcie 1.2.2, nie zawsze wiadomo ile dokładnie miejsca może zająć gotowa struktura zbudowana dla sceny złożonej z n trójkątów. Ostatnim szczegółem implementacyjnym o którym chciałem wspo-

mnieć jest wyrównywanie wstawianych do pamięci obiektów do odpowiedniej wartości, przeważnie do liczby podzielnej przez 32. Takie rozmieszczenie umożliwia procesorowi lepsze poukładanie obiektów w pamięci cache, dzięki czemu program może działać nieco szybciej.

ROZDZIAŁ 2

Spłaszczone hierarchiczne przedziały ograniczające

W tym rozdziale opiszę dokładnie strukturę, która powstała jako połączenie wybranych cech z już wcześniej istniejących i omówionych w poprzednim rozdziale drzew. Struktura ta ze względu na swój wygląd nosi nazwę spłaszczonych hierarchicznych przedziałów ograniczających (SBIH, ang. Shallow Bounding Interval Hierarchies).

2.1. Pomysł

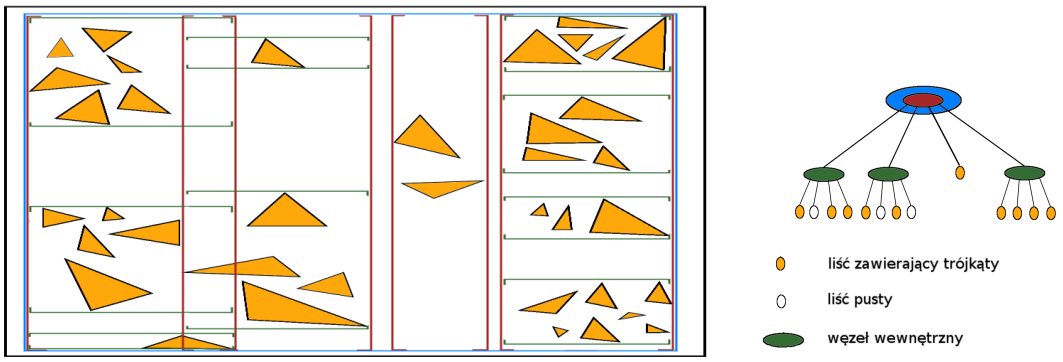
Czytając artykuł o strukturze SBVH [DHK08] i mając wiedzę na temat wcześniej powstałej struktury BIH [WK06], pomyślałem, że można spróbować połączyć oba rozwiązania. To znaczy zbudować strukturę, która będzie umożliwiała:

- efektywne trawersowanie przy pomocy instrukcji SIMD, nie tylko dla pakietów promieni pierwotnych;
- lepsze umieszczenie struktury w pamięci cache procesora przez zmiejszenie rozmiaru węzłów drzewa;
- szybką budowę, co może być wykorzystywane w przypadku interaktywnego renderowania animacji;

Struktura ta podobnie jak drzewo BIH miałaby w węzłach jedynie płaszczyzny ograniczające obszary z dwóch stron na jednej osi. Płaszczyzny te dzieliłyby jednak bryłę węzła wewnętrznego na 4 części tworząc drzewo czwórkowe, tak jak to jest w drzewie SBVH. Przykład tej struktury znajduje się na rysunku 2.1.

Zgodnie z powyższym opisem, węzeł wewnętrzny struktury SBIH musi zawierać conajmniej 6 płaszczyzn ograniczających. Dwie zewnętrzne można przekopiować z wyższego poziomu, a dla korzenia drzewa, z bryły ograniczającej całą scenę. Dodatkowo potrzebne jest oznaczenie osi, na której wykonywany jest podział i wskaźnik na pierwsze „dziecko” („dzieci” mogą być umieszczone w pamięci obok siebie, dlatego więcej wskaźników nie jest potrzebnych). Liście muszą zawierać liczbę trójkątów w nich zawartych oraz wskaźnik na pierwszy z nich.

Przy założeniu, że mamy dla całej sceny skonstruowaną bryłę ograniczającą, podczas trawersowania i wchodzenia do kolejnych węzłów bryła ta zostaje systematycznie zmniejszana. W wyniku tego, w każdym węźle mamy dostępną bryłę AABB. Sposób w jaki struktura ta jest skonstruowana, uniemożliwia jednak idealne ograniczenie danego poddrzewa, tak jak jest to możliwe w drzewie SBVH. W związku z tym można spodziewać się mniejszej wydajności podczas trawersowania.



Rysunek 2.1. Przykład sceny w 2D i stworzonej na niej struktury SBIH.

Źródło: Opracowanie własne.

Opisana w artykule [DHK08] konstrukcja drzewa SBVH przebiega w dwóch etapach. Na początku budowane jest drzewo BVH. Drzewo to następnie jest szybko przekształcane w drzewo SBVH. Autorzy, dzięki takiej budowie wykorzystali dokładnie wszystko to, co do tej pory istniało dla drzew BVH, między innymi heurystykę pól powierzchni opisaną w punkcie 1.2.3. Dzięki temu drzewo SBVH jest zbudowane optymalnie. Zastanawiałem się, czy można to rozwiązać w inny sposób, ponieważ w nowym drzewie SBIH nie będzie możliwe skonstruowanie go na przykład z drzewa BIH. Podziały w drzewie na dwóch kolejnych poziomach musiałyby być wykonywane na tej samej osi. Takie założenie można oczywiście przyjąć podczas budowy pierwszego drzewa, ale zdecydowanie ładniejsze byłoby rozwiązanie, w którym będzie od razu konstruowane drzewo czwórkowe.

2.2. Realizacja

Na początku zostało zrealizowane najprostsze rozwiązanie pomysłu opisanego w punkcie 2.1, w którym każdy liść trzymany był jako osobny węzeł. Podczas implementacji okazało się, że to rozwiązanie nie jest zbyt efektywne. Liście w drzewie można trzymać w sposób zaprezentowany w artykule [DHK08] opisującym drzewa SBVH. Każdy liść jest tam trzymany w wewnętrznym węźle, który jest dla

niego ojcem. Dzięki temu rozwiązaniu można ograniczyć całkowitą liczbę węzłów drzewa. Jednak wymaga ono również dwukrotnego zwiększenia pamięci potrzebnej na jeden węzeł. Obie struktury zostały dalej rozwijane i doprowadzone do końca. Ze względu na niewielkie różnice w budowie i w trawersowaniu są one w kolejnych punktach opisane równolegle.

2.2.1. Wygląd struktury

Pracę należy rozpocząć od projektu wyglądu węzła. Pierwszym pomysłem była minimalizacja jego rozmiaru. Jak już zostało opisane w punkcie 2.1, w węźle wewnętrznym trzeba trzymać conajmniej 6 liczb zmiennoprzecinkowych, wskaźnik na dziecko oraz oś, w której dokonano podziału. Jak łatwo policzyć taki węzeł zmieści się w 32 bajtach, co jest bardzo dobrym wynikiem. Należy przy tym rozwiązaniu wziąć pod uwagę fakt, że każdy liść musi być trzymany w osobnym węźle, co zwiększa dość znacznie rozmiar struktury.

Listing 2.1. Przykładowy kod źródłowy węzła drzewa SBIH.

```
struct
{
        float   plaszczyznaMin[3];
        float   plaszczyznaMax[3];
        int     wskaznik;
        int     dane;
};
```

Kolejny pomysł był zaczerpnięty z artykułu [DHK08], gdzie liście były przechowywane w węzłach, będących ich ojcami. Przy tym rozwiązaniu trzeba było zwiększyć rozmiar węzła do 64 bajtów. W takim węźle można już bez problemów trzymać 8 liczb zmiennoprzecinkowych określających płaszczyzny podziału, 4 wskaźniki, po jednym dla każdego dziecka, na kolejny węzeł lub na pierwszy trójkąt (w zależności, od tego, czy dziecko jest liściem, czy węzłem wewnętrznym) oraz liczbę trójkątów dla dzieci, które są liśćmi. Dodatkowo przeznacza się 2 bity na przetrzymywanie osi podziału.

Listing 2.2. Przykładowy kod źródłowy węzła drzewa SBIH2.

```
struct
{
        float   plaszczyznaMin[4];
        float   plaszczyznaMax[4];
        int     wskaznik[4];
        int     liczbaTrojkatow[4];
};
```

2.2.2. Budowa struktury

Budowa struktury SBIH jest bardzo podobna do budowy hierarchicznych struktur drzewiastych opisanych w punkcie 1.2.2. W tym punkcie budowa ta zostanie opisana dokładnie.
Opis zmiennych pomocniczych:

n_i liczba trójkątów w i-tym kubełku;

bb_i bryła AABB otaczająca wszystkie trójkąty znajdujące się w i-tym kubełku;

C_i środek ciężkości i-tego trójkąta;

BB_i bryła AABB i-tego trójkąta;

$N_{L,j}$ suma wszystkich n_i od 0 do j, $N_{L,j} = \sum_{i=0}^{i=j} n_i$;

$N_{R,j}$ suma wszystkich n_i od $j+1$ do n, gdzie n to liczba wszystkich kubełków, $N_{R,j} = \sum_{i=j+1}^{i=n} n_i$;

$A_{L,j}$ pole powierzchni bryły otaczającej trójkąty znajdujące się w kubełkach od 0 do j, $A_{L,j} = SA(\bigcup_{i=0}^{i=j} bb_i)$;

$A_{R,j}$ pole powierzchni bryły otaczającej trójkąty znajdujące się w kubełkach od $j+1$ do n, $A_{R,j} = SA(\bigcup_{i=j+1}^{i=n} bb_i)$;

Algorytm budowy:

1. oblicz bryły otaczające dla wszystkich trójkątów (można to zrobić efektywnie z wykorzystaniem instrukcji SIMD), w tym kroku oblicz również środki ciężkości wszystkich trójkątów oraz bryłę V otaczającą całą scenę
2. wrzuć na stos węzeł zawierający wszystkie trójkąty
3. wykonuj poniższą pętlę dopóki stos jest niepusty:
   (a) ściągnij z wierzchołka stosu węzeł z danymi do stworzenia drzewa;
   (b) jeśli w danym węźle jest mało trójkątów, to stwórz z niego liść i zakończ;

(c) dla kazdej z trzech osi:

i. podziel trójkąty na kubełki według wzoru znajdującego się w artykule [Wal07] (dla każdego kubełka obliczaj podczas tej operacji n_i oraz bb_i);

ii. za pomocą zmiennych n_i i bb_i wylicz $N_{L,j}$, $N_{R,j}$ oraz $A_{L,j}$ i $A_{R,j}$;

iii. korzystając ze zmiennych $N_{L,j}$, $N_{R,j}$, $A_{L,j}$ i $A_{R,j}$ oblicz z wykorzystaniem SAH najlepszy podział (niech to będzie miejsce l);

iv. uaktualnij zmienne $N_{L,j}$, $N_{R,j}$, $A_{L,j}$ i $A_{R,j}$ tak jakby kubełki $0, ..., l$ oraz $l + 1, ..., n$ tworzyły osobne węzły;

v. oblicz z wykorzystaniem nowych zmiennych dwa najlepsze podziały (jeden dla koszyków od 0 do l, drugi dla koszyków od $l + 1$ do n);

vi. zapamiętaj wszystkie 3 podziały, jeśli wynik SAH jest lepszy niż dla pozostałych osi;

(d) posortuj trójkąty w tablicy według otrzymanego podziału metodą podaną w punkcie 1.2.4;

(e) wrzuć na stos dane potrzebne do stworzenia poddrzew z 4 otrzymanych podzbiorów trójkątów.

Struktury SBIH i SBIH2 różnią się jedynie szczegółami implementacyjnymi w tworzeniu węzłów wewnętrznych i liści. Ogólny schemat algorytmu jest identyczny. Jak łatwo zauważyć, techniki budowy hierarchicznych drzew opisane w punkcie 2.2.2 zostały tu umiejętnie wykorzystane. Większa zmiana była potrzebna jedynie w przypadku implementacji heurystyki SAH, która w tym wypadku jest wywoływana trzykrotnie podczas tworzenia jednego węzła. Dzieje się tak, ponieważ każdy z tych trzech podziałów musi odbywać się wzdłuż jednej osi. Dokładna implementacja algorytmu budowy opisanych drzew znajduje się w pliku `traversetree.cpp`, który znajduje się w źródłach programu. Plik ten jest dołączony do pracy, więcej informacji można znaleźć w dodatku A.

2.2.3. Uaktualnianie struktury

Poza budową struktura ta nadaje się idealnie do szybkiego uaktualniania. Uaktualnianie takie jest bardzo przydatne w scenach animacji, gdy mało obiektów w kolejnych klatkach zmienia swoje położenie. Najlepsze efekty można osiągnąć, gdy położenie to zmieniają w podobny sposób trójkąty do siebie przylegające. Takie uaktualnienie drzewa można zrobić w liniowym czasie względem liczby trójkątów w drzewie. Możliwe i bardzo intuicyjne jest tu rozwiązanie rekurencyjne, jednak jak już wcześniej wspominałem, warto się go pozbyć.

Algorytm przedstawia się następująco:

1. wrzuć na stos korzeń drzewa z wyzerowaną bryłą otaczającą (bryła ta zostanie uaktualniona)
2. wykonaj poniższą pętlę dopóki stos jest niepusty:
   (a) zdejmij węzeł z wierzchołka stosu;
   (b) jeśli to liść, to oblicz dla niego nową bryłę otaczającą i uaktualnij bryłę otaczającą ojca danego węzła;
   (c) jeśli to węzeł wewnętrzny, to wrzuć na stos jego kolejne dziecko z wyzerowaną bryłą otaczającą, jeśli to ostatnie dziecko, to po uaktualnieniu go, uaktualnij również bryłę otaczającą jego rodzica (ewentualnie bryłę otaczającą całą scenę).

W pliku `traversetree.cpp` można znaleźć implementację tego rozwiązania. Jak już wspomniałem w poprzednim punkcie, plik ten znajduje się w źródłach programu opisanego w dodatku A.

2.2.4. Trawersowanie struktury

Ostatnim interesującym algorytmem jest trawersowanie drzewa przez dany promień. Optymalny algorytm bierze pod uwagę dwie kwestie:

- kolejność przeglądania poddrzew danego węzła — faktem jest, że przecięcie liczymy za jednym zamachem dla wszystkich 4 poddrzew, ale warto je przeglądać w prawidłowej kolejności, można dzięki takiemu rozwiązaniu zaoszczędzić sporo czasu
- właściwe ograniczanie sprawdzanego obszaru podczas zagłębiania się w drzewie — tu trzeba uważać na to, by nie liczyć przecięć z przedziałami ograniczającymi znajdującymi się poza bryłą otaczającą wynikającą z hierarchii drzewa, można to zrobić przez odpowiednie zarządzanie wartością określającą koniec promienia

Mój algorytm przedstawia się następująco:

1. wrzuć korzeń drzewa na stos;
2. dopóki stos jest niepusty wykonaj:
   (a) zdejmij węzeł ze stosu;
   (b) jeśli to liść, to przetnij promień ze wszystkimi trójkątami, które się w nim znajdują (jeśli przecięcie znalezione, to uaktualnij promień) i kontynuuj pętlę;

(c) jeśli to węzeł wewnętrzny, to:

i. ustaw kolejność trawersowania dzieci w zależności od kierunku promienia na osi, wzdłuż której istnieje podział w danym węźle;

ii. ustaw tymczasowe (na czas przecinania z płaszczyznami) maksimum promienia (w tym miejscu trzeba uważać, by maksimum promienia było ustawione na minimalną wartość z jego aktualnej długości i otaczającej ten fragment sceny bryły AABB, jeśli się to ustawienie zaniedba, to w praktyce odwiedzimy prawie każdy węzeł, ponieważ tylko promień równoległy do danej płaszczyzny jej nie przetnie);

iii. policz przecięcie z płaszczyznami;

iv. wrzuć dzieci, z którymi przecięcie dało wynik pozytywny na stos, na stos pomocniczy wrzuć bryłę AABB ograniczającą te dzieci.

Implementacja algorytmów trawersowania zaimplementowanych struktur znajduje się w pliku `tracer.cpp` w źródłach programu dołączonego w dodatku A.

ROZDZIAŁ 3

Wyniki

Testy zostały przeprowadzone na 4 zaimplementowanych strukturach:

- hierarchicznych bryłach ograniczających (BVH);
- spłaszczonych hierarchicznych bryłach ograniczających (SBVH);
- hierarchicznych przedziałach ograniczających (BIH);
- spłaszczonych hierarchicznych przedziałach ograniczających z węzłem o minimalnym rozmiarze (SBIH);
- spłaszczonych hierarchicznych przedziałach ograniczających, w których liście wewnętrzne zawarte są w „rodzicu" (SBIH2).

Do testów zostały wykorzystane sceny pobrane z repozytorium animacji Uniwersytetu w Utah, które można znaleźć na stronie internetowej[1]. Przykładowe zrzuty ekranu wybranych klatek animacji znajdują się na rysunku 3.1. Dokładniejsze dane dotyczące scen, animacji można znaleźć w tabeli 3.1.

nazwa sceny	liczba trójkątów	liczba klatek
wooddoll	6658	29
marbles	8800	500
toasters	11141	246
hand	17135	44
ben	78029	30
fairy forest	174117	21
exploding dragon	252572	16

Tabela 3.1. Statystyka scen użytych w testach.

Źródło: Opracowanie własne.

Wybór scen nie jest przypadkowy. Zostały one stworzone w celach testowych. Przedstawiają najczęściej spotykane w animacjach rodzaje ruchu. W scenie *hand* występuje naturalny ruch palców ręki. Ruch biegnącej postaci możemy znaleźć

[1]http://www.sci.utah.edu/~wald/animrep/

w scenach *ben* i *wooddoll*. W animacjach zawsze występują pewne obiekty stałe, takie jak ściany, czy podłogi. Nie zmieniają one położenia w czasie. W scenach testowych znaleźli miejsce reprezentanci właśnie takich scen, są to *fairy forest* oraz *toasters*, gdzie pewne obiekty poruszają się po ograniczonym obszarze, który także jest renderowany. Inną często spotykaną sytuacją w animacjach są obiekty poruszające się niezależnie od siebie, zmieniające kierunki ruchu pod wpływem odbić. Można je zobaczyć w scenie *marbles*, w której piłeczki wpadają do niewidzialnego pojemnika i odbijając się od siebie układają się na jego dnie. Ostatnia scena *exploding dragon* przedstawia smoka rozpadającego się w drobny pył. Taka sekwencja jest bardzo trudna do utrzymania w efektywnej strukturze. Trójkąty, które na początku były blisko siebie, były połączone, w kolejnych klatkach mogą poruszać się zupełnie niezależnie w różnych kierunkach.

Większość testów została przeprowadzona na komputerze z procesorem AMD Sempron 2500+ obsługującym instrukcje SSE i SSE2 taktowanym z częstotliwością 1,4 GHz i wyposażonym w 1 GB pamięci RAM DDR. W celu sprawdzenia równoległości zastosowanych algorytmów został wykorzystany komputer z procesorem Intel Core 2 Quad Q9300 taktowany z częstotliwością 2,5 GHz, również wykorzystującym instrukcje SSE i SSE2, posiadającym 4 GB pamięci RAM DDR2.

3.1. Budowa

Pierwszym testem było porównanie czasów budowy całej struktury. Wyniki dla scen testowych znajdują się w tabeli 3.2. Budowa struktury SBVH jest minimalnie dłuższa niż budowa struktury BVH. Tak jak zostało opisane w punkcie 1.2.2, drzewo SBVH powstaje przez przebudowanie drzewa BVH. Kolejną ciekawą obserwacją jest czas budowy drzew, których budowa opiera się na spłaszczonym podziale przestrzeni płaszczyznami, czyli SBIH i SBIH2. Dla mniejszych scen czasy te są większe niż czasy budowy drzew opartych na bryłach ograniczających, ale wraz ze wzrostem wielkości scen, czasy te rosną dużo wolniej. Jest to spowodowane tym, że podczas budowy struktury hierarchicznych brył ograniczających, na każdym etapie dzielimy zbiór trójkątów na 2 podzbiory, a przy spłaszczonych hierarchicznych przedziałach ograniczających na 4. Dzięki temu początkowy duży zbiór trójkątów dość szybko rozdziela się na więcej mniejszych, na których operacje wykonywane są już znacznie szybciej. Różnicę tę również widać porównując strukturę BIH, w której podczas budowy również dochodzi do podziału na 2 części, z SBIH. Budowa struktury BIH jest dużo wolniejsza.

Kolejnym testem było sprawdzenie czasu aktualizacji struktury. Taka aktualizacja powinna być błyskawiczna, ponieważ odbywa się to przez zwykłe przeglądnięcie drzewa i aktualizację jego węzłów. Złożoność czasowa tej operacji jest li-

(a) ben (b) hand (c) toasters

(d) fairy forest (e) wooddoll (f) exploding dragon

(g) marbles

Rysunek 3.1. Przykładowe zrzuty ekranu z rendrowanych animacji.

Źródło: Sceny pochodzą z Utah Animation Repository.

niowa względem rozmiaru sceny (przejrzenie wszystkich trójkątów) i względem rozmiaru struktury (przejrzenie wszystkich węzłów). Z tabeli 3.3 porównując czasy aktualizacji do czasów budowy drzew (patrz tab. 3.2), można wywnioskować, że ta operacja rzeczywiście jest błyskawiczna i w przypadku wszystkich struktur czasy są zbliżone z lekkim wskazaniem na struktury spłaszczonych hierarchicznych przedziałów ograniczających. Dzieje się tak, ponieważ drzewa tych struktur mają zdecydowanie mniej węzłów wewnętrznych do przeglądnięcia, a ich mniej-

nazwa sceny	BVH	SBVH	BIH	SBIH	SBIH2
wooddoll	34,3	26,9	44,9	32,4	21,8
marbles	38,8	39,2	64,7	72,9	74,4
toasters	51,5	50,8	83,9	91,5	96,8
hand	77,8	79	126,6	110,6	80
ben	426,2	424,9	639,1	277,2	214,6
fairy forest	1029,4	1055,2	1474,2	903,8	741,0
exploding dragon1	1371,2	1372,1	2026,1	665,9	582,0
exploding dragon2	1406	1406,6	2076,2	782,4	677

Tabela 3.2. Porównanie średnich czasów budowy od podstaw dla różnych struktur (czasy podane w ms).

Źródło: Opracowanie własne.

szy rozmiar pozwala na umieszczenie całych struktur w szybkiej pamięci cache procesora.

nazwa sceny	BVH	SBVH	BIH	SBIH	SBIH2
wooddoll	1,6	2	1,6	1,3	1,6
marbles	2,3	2,8	2	2,1	2,6
toasters	2,8	3,5	2,4	2,6	3,3
hand	4,6	5,6	4,2	3,8	4,8
ben	20,6	25	18,2	14,3	19,5
fairy forest	51,4	61,8	43,3	38,2	51,2
exploding dragon1	77,8	91,4	67,4	50,6	69
exploding dragon2	75	91,6	66	51,6	69,8

Tabela 3.3. Średnie czasy uaktualniania struktury dla jednej klatki (czasy podane w ms).

Źródło: Opracowanie własne.

Ważnym aspektem, o którym wspomniałem kilka razy w drugim rozdziale jest rozmiar pamięci jaki zajmuje cała struktura. Rozmiar ten w testach został policzony dokładnie przez przemnożenie liczby zajętych węzłów przez rozmiar jednego węzła. Wynik tego działania został uśredniony dla wszystkich klatek animacji. W tym wypadku należy zwrócić uwagę na inne rozmiary drzew w zależności od parametru budowy, jakim jest maksymalna liczba trójkątów w jednym liściu. Im jest ich więcej, tym drzewo może zostać zbudowane szybciej i tym mniej pamięci

zajmuje. Rozmiary drzew przedstawionych w tabeli 3.4, odnoszą się do drzew zbudowancyh z identycznym parametrem, z maksymalną liczbą trójkątów w liściu wynoszącą 16. W tej tabeli warto zwrócić uwagę na bardzo mały rozmiar struktur SBIH i SBIH2 w pamięci. Węzły tych struktur są bardzo kompaktowe, a dzięki spłaszczeniu ich liczba utrzymana jest na bardzo niskim poziomie.

nazwa sceny	BVH	SBVH	BIH	SBIH	SBIH2
wooddoll	73,5	38	76	9,5	6,5
marbles	52	51,5	103	52	27,5
toasters	67	67	131	67,5	36
hand	99,5	100	198	36,5	26,5
ben	453	454	900	43	33
fairy forest	1023	1021	2024	251	172,5
exploding dragon1	1475	1475	2933	57	45
exploding dragon2	1492	1494	2933	91	68

Tabela 3.4. Porównanie pamięci zajmowanej przez struktury. Pamięć liczona w kilobajtach.

Źródło: Opracowanie własne.

3.2. Trawersowanie

Najważniejszą własnością, od której zależy efektywność działania, czyli jakość struktury jest czas renderowania jednej klatki. Ważnym aspektem oczywiście jest rozmiar renderowanej sceny oraz wielkość bryły zawierającej renderowaną scenę w stosunku do tego co obejmuje widok z kamery. Gdy obiekt zajmuje nieznaczną część widoku z kamery, wtedy spora liczba promieni pierwotnych może być szybko odrzucona przez test przecięcia z bryłą otaczającą scenę. Kamera we wszystkich scenach testowych została jednak ustawiona w taki sposób, by bryła otaczająca całą scenę obejmowała większą część kadru. Dla wszystkich struktur ustawienia kamery są identyczne. Duży wpływ na czas renderowania jednej klatki, przy korzystaniu z opisanych struktur ma również sama budowa sceny, czyli rozmieszczenie trójkątów. W tabeli 3.5 scena *exploding dragon* została podzielona na dwie części. Część pierwsza to 8 pierwszych klatek, część druga to 8 ostatnich. W drugiej części smok rozpada się na drobne kawałeczki, którymi są pojedyńcze trójkąty. Każdy taki trójkąt porusza się w innym kierunku, co znacznie utrudnia renderowanie. Wyniki pokazują, że czasy dla drugiej części tej animacji, szczególnie dla drzew

SBIH i SBIH2, są dużo większe niż te dla pierwszej jej części. Pokazuje to trudności jakie struktury te mają przy całkowicie losowo zmieniających się scenach.

nazwa sceny	BVH	SBVH	BIH	SBIH	SBIH2
wooddoll	255	218	751,6	1216	903
marbles	241	215	253	255,3	191,6
toasters	934,4	699,4	2499	2699	1243
hand	845	417	1988	3066	1158
ben	521,2	395,3	2344	6550	1666
fairy forest	1789	1227	7824	13315	4310
exploding dragon1	451,2	252,7	1196	4435	2215
exploding dragon2	1061	581	10054	27452	23462

Tabela 3.5. Porównanie czasów renderowania obrazów po całkowitej przebudowie struktury (czasy podane w ms).

Źródło: Opracowanie własne.

Gdy renderowana scena animacji nie zmienia się znacznie, warto zadbać, by struktura, w której się ona znajduje była dobrze aktaulizowana dla kolejnych klatek. Aktualizacja taka jest bardzo szybka, co pokazuje tabela 3.3. Efektywność przeglądania zaktualizowanej struktury możemy odczytać z tabeli 3.6. Własność ta została sprawdzona tylko dla niektórych scen. Szczególną uwagę należy zwrócić na sceny *toasters* i *fairy forest*, w których występują jednocześnie obiekty stałe i obiekty ruchome. Tak zbudowane są praktycznie wszystkie sceny w większości dzisiejszych animacji. W tabeli 3.6 można jeszcze zauważyć dość znaczne różnice dla czasów renderowania obrazów po aktualizacji z wykorzystaniem struktury SBVH w porównaniu do renerowania przy pomocy tej struktury po całkowitej przebudowie. Być może jest to spowodowane tym, iż bryły otaczające dla węzłów wewnętrznych muszą zawierać cztery bryły na niższym poziomie. Podczas aktualizacji przy małej zmianie każdej z tych czterech brył, bryła „rodzica" może dość znacznie się powiększyć, przez co dużo więcej promieni trafia właśnie w tę bryłę i jest liczone dużo więcej przecięć. W strukturach SBIH i SBIH2 mamy również drzewa czwórkowe, jednak tam budowa oparta jest na podziale płaszczyznami (w jednym węźle wzdłuż jednej osi), dzięki czemu małe zmiany w węzłach „dzieci" nie powodują wielkich zmian w węźle „rodzica".

Innym interesującym parametrem, opisującym w pewien sposób efektywność działania struktury, jest średnia liczba trójkątów przecinanych w czasie jednej sekundy. Wyniki obrazujące tę statystykę znajdują się w tabeli 3.7. Wartości zostały wyliczone przez podzielenie średniej liczby przecinanych trójkątów podczas ren-

nazwa sceny	BVH	SBVH	BIH	SBIH	SBIH2
wooddoll	281,3	843	818	1423	997
marbles	254,2	534,1	310,1	321,5	228,5
toasters	978	3731	2779	2920	1322
hand	916	4274	2284	3478	1258
ben	584	30643	2883	8514	2321
fairy forest	2261	89424	13480	24325	6375

Tabela 3.6. Porównanie czasów renderowania obrazów po aktualizacji struktury, pełna przebudowa co 4 klatki (czasy podane w ms).

Źródło: Opracowanie własne.

derowania animacji przez czas jej renderowania. Chciałbym dodać w tym miejscu, że test ten został przeprowadzony na strukturach, których wszystkie liście mają maksymalnie po 16 trójkątów. Dzięki takiemu założeniu wyniki można porównywać. Jak łatwo odczytać z tabeli 3.7 struktura BVH umożliwia przecinanie średnio od 0,5M do 1M trójkątów na sekundę. Struktura SBVH od około 0,7M do 1,3M trójkątów w ciągu jednej sekundy. Dużo lepiej radzą sobie struktury oparte na podziale płaszczyznami, BIH umożliwia przecinanie od około 2,5M do 3,5M, podczas gdy struktury SBIH i SBIH2 odpowiednio około 3M–5M i 2M–4M przecięć trójkątów na sekundę. Taki wynik może być spowodowany po pierwsze, mniejszą liczbą operacji potrzebnych podczas trawersowania drzewa (test przecięcia tylko z płaszczyznami równoległymi do jednej z osi układu współrzędnych), a po drugie, mniejszym rozmiarem węzłów, dzięki czemu, struktura może być zdecydowanie szybciej przeglądana, przez umieszczenie w pamieci cache, z której czytanie odbywa się błyskawicznie.

3.3. Parametry w strukturze SBIH2

Struktura SBIH2 została przetestowana dokładniej. W testach zostały sprawdzone m.in. różne parametry jej budowy. Na końcu zostały wykorzystane oczywiście parametry dające optymalne wyniki. Z tabeli 3.8 można odczytać prędkość budowy. Rośnie ona wraz ze zwiększaniem liczby koszyków i maleje wraz ze zwiększaniem liczby trójkątów w liściu. Przy dobieraniu odpowiednich parametrów należy zwrócić również uwagę na czasy trawersowania, które są przedstawione w tabeli 3.9. Widać, że dla pewnych parametrów budowa struktury się nie opłaca. Dla niektórych parametrów z kolei nie widać wyraźnej różnicy w czasach trawersowań. Gdy liczba koszyków dochodzi do 64, czasy renderowania niezancznie się zmniej-

nazwa sceny	BVH	SBVH	BIH	SBIH	SBIH2
wooddoll	0,45	0,99	2,95	4,1	3,83
marbles	1,1	1,33	2,5	2,5	2,16
toasters	0,92	1,29	2,85	2,79	2,27
hand	1,14	1,08	3,27	4,13	2,84
ben	0,71	1	3	3,96	3,53
fairy forest	0,61	0,9	3	3,8	2,95
exploding dragon1	1,03	0,70	3,3	5,07	4,87
exploding dragon2	0,51	0,37	3,44	5,1	5,06

Tabela 3.7. Porównanie wydajności działania struktur, w tabeli przedstawiona jest średnia liczba przecinanych trojkątów w ciągu jednej sekundy dla poszczególnych scen i struktur (liczba podana w milionach).

Źródło: Opracowanie własne.

szają. Powyżej tej liczby utrzymują się już na podobnym poziomie. Czasy budowy powyżej pewnej granicy rosną bardzo wyraźnie, dzieje się to wraz ze wzrostem liczby koszyków, która oznacza po prostu większą liczbę sortowań.

l.k. \ *m.l.t.*	4	8	16	32	64	128
16	177	177,8	181,4	181,1	178,9	165,4
32	193,5	186,9	187	187,1	187,2	173,2
64	216	215,4	214,5	220,3	211	189,5
128	270,6	270,7	268,4	267,1	251,2	224
256	390,2	378,7	383,5	371,8	345,9	298,5
512	604,5	598,3	616,2	601,9	540,1	425,5
1024	1062	1080	1060	1049,8	900,5	714,4

Tabela 3.8. Porównanie czasów budowy struktury SBIH2 w zależności od wybranych parametrów *l.k.* — liczba koszyków i *m.l.t.* — maksymalna liczba trójkątów w liściu (czasy podane w ms).

Źródło: Opracowanie własne.

l.k. \ *m.l.t.*	4	8	16	32	64	128
16	1681	1682	1686	1749	1762	2119
32	1582	1580	1584	1588	1649	2006
64	1616	1617	1615	1621	1672	2027
128	1604	1603	1605	1608	1656	2030
256	1584	1585	1584	1590	1636	2012
512	1581	1583	1580	1586	1639	2015
1024	1571	1572	1571	1576	1626	2007

Tabela 3.9. Porównanie czasów renderowania struktury SBIH2 w zależności od wybranych parametrów *l.k.* — liczba koszyków i *m.l.t.* — maksymalna liczba trójkątów w liściu (czasy podane w ms).

Źródło: Opracowanie własne.

3.4. Wielowątkowość algorytmów

W tabeli tab.3.10 znajdują się wyniki testu wielowątkowości renderowania dla wybranych scen. Wielowątkowość została zaimplementowana jedynie dla renderowania. Został przy tym wykorzystany fakt, że kolor każdego piksela obrazu może być obliczany niezależnie. Dzięki tej własności metody śledzenia promieni, nie widać praktycznie żadnych strat podczas urównoleglania programu dla żadnej z zaimplementowanych struktur. Wszystkie spisują się bardzo dobrze. W tym teście wykorzystano jedynie procesor 4-rdzeniowy, jednak nic nie stoi na przeszkodzie, by zastosować tę metodę na dużo większej liczbie rdzeni, czy procesorów.

	hand			*marbles*			*ben*		
struktura \ l. wątków	1	2	4	1	2	4	1	2	4
BVH	1946	981	499	519	260	144	1226	629	327
SBVH	541	267	144	274	140	81	524	274	143
SBIH	3679	1841	939	331	167	91	7301	3687	1854
SBIH2	1398	702	361	250	126	71	1971	1009	515

Tabela 3.10. Porównanie czasów renderowania obrazów dla różnej liczby wątków (czasy podane w ms).

Źródło: Opracowanie własne.

ROZDZIAŁ 4

Wnioski i plany rozwoju

Rozwiązanie opisane w tej pracy można jeszcze poprawić. Jedną z ważniejszych rzeczy może być urównoleglenie budowy. Przykładowe jest opisane w artykule [Wal07]. Z wyników osiągniętych przez jego autorów, można wywnioskować, że jest ono dobre.

Kolejnym usprawnieniem może być dodanie pakietów. Patrząc z jednej strony chcieliśmy to wyeliminować przez budowę struktury opartej na drzewie czwórkowym. Jednak w ten sposób przyspieszamy przeglądanie drzewa, czyli znajdowanie najbliższego przecięcia. dzięki pakietom promieni być może da się takie przecięcie znaleźć jeszcze szybciej wykorzystując techniki opisane w artykule [BEL+07]. Dodatkowo, między kolejnymi wywołaniami metod szukania najbliższego przecięcia wykonywanych jest mnóstwo innych operacji, takich jak odbijanie, załamywanie promienia, czy przeliczane kolorów. Operacje te mogą być wykonywane z wykorzystaniem instrukcji SIMD podczas trawersowania struktury z wykorzystaniem pakietów.

W wynikach zaprezentowanych w rozdziale 3 trawersowanie przy pomocy spłaszczonych hierarchicznych przedziałów ograniczających (SBIH oraz SBIH2) nie daje najlepszych rezultatów. Co pokazuje szczególnie tabela 3.5. Jednak w tabeli 3.7 widać, że wydajność struktury jest wysoka. Problem jest z jakością. Stąd wniosek, że być może da się poprawić heurystykę budowy opisaną w punkcie 2.2.2. Dobrym rozwiązaniem byłaby z pewnością heurystyka, dzięki której możnaby otrzymać podział trójkątów na 4 dobrze rozdzielone części.

Uważam również, że struktura zbudowana w stylu spłaszczonych hierarchicznych przedziałów ograniczających może być dużo łatwiej i efektywniej zrealizowana sprzętowo. Tu pod uwagę należy wziąć kilka czynników. Jednym jest na pewno pamięć, jaką struktury zajmują. W tym punkcie struktury BIH spisują się zdecywdowanie lepiej niż BVH, co widać w tabeli 3.4. Patrząc na pamięć można od razu wspomnieć kd–drzewa opisywane w punkcie 1.2.2, w których jeden trójkąt mógł trafić do kilku poddrzew. W tej strukturze jest to niemożliwe. Struktura SBIH może zostać dużo prościej zrealizowana niż struktura BVH, ponieważ przy przeglądaniu drzewa, na każdym poziomie potrzeba tylko przecięcia z płaszczyzną, a nie z całą bryłą ograniczającą, tak jak to jest w przypadku BVH. Instrukcje SIMD wykorzystane w pracy i implementacji dla drzew czwórkowych, mogą sprzętowo zostać zrealizowane bardzo podobnie, dzięki czemu struktury te powinny być bardzo szybkie w przeglądaniu. Ponadto wszystko może odbywać się równolegle.

Realizacja algorytmiczna takiego rozwiązania jest już dość prosta, a technologia również idzie dziś w kierunku zwiększania liczby rdzeni w procesorach, zamiast podwyższania mocy obliczeniowych, więc może już w niedalekiej przyszłości metoda śledzenia promieni zyska na zainteresowaniu dzięki realistycznym efektom i coraz bardziej interaktywnym czasom działania.

DODATEK A

Program *atracer*

Do powyższej pracy jest dołączony program *atracer*. Został on zaimplementowany specjalnie w celach testowych wszystkich wyżej opisanych struktur. Źródła programu oraz wszystkie potzebne pliki do jego uruchomienia, w tym przykładowe sceny opisane w rozdziale 3 znajdują się na płycie DVD.
W skład całego programu wchodzą:

- kod źródłowy i makefile;
- katalog ze scenami *scenes*, zawiera on sceny pobrane z repozytorium Uniwersytetu w Utah, niektóre ze scen zostały dopasowane do potrzeb programu;
- skrypt i dodatkowe pliki uruchomieniowe, w skrypcie znajdują się przykładowe wywołania programu;

Program został napisany w języku C++. Działa na systemach Unixowych. Bardzo istotne jest, by procesor obsługiwał instrukcje SSE i SSE2, gdyż program zawiera ich wywołania.

Opis działania:

1. zainicjowanie parametrów działania;
2. wczytanie scen wraz z teksturami do pamięci (wczytywanie scen w formacie *.obj*);
3. zarezerwowanie pamięci potrzebnej na budowane struktury;
4. dla kolejnych klatek animacji uruchamiane są metody budowy (lub aktualizacji) struktury, a następnie renderowania;

Najbardziej istotne klasy: *traversetree* i *tracer* znajdują się w plikach z odpowiadającymi im nazwami. W klasie *traversetree* znajduje się kod źródłowy wykorzystywany przy budowie i aktualizacji wszystkich opisanych struktur. Z kolei klasa *tracer* zawiera kod źródłowy służący do przeglądania tychże struktur.

Kompilacja programu odbywa się przez uruchomienie skryptu *make*. Wersję programu można wybrać wykomentowując odpowiednie wiersze w pliku *settings.h* (wiersze z deklaracją *#define*), który znajduje się w katalogu z kodem źródłowym. Przed wywołaniem polecenia *make* warto wykonać róznież *make clean*. Do kompilacji potrzebny jest kompilator g++ w wersji 4.2 lub wyższej.

Program potrzebuje na wejściu kilku parametrów. Mogą one być podane w dowolnej kolejności. Nie wszystkie są wymagane, dla niektórych przyjmowane są wartości domyślne.

Opis parametrów programu (każdy parametr podajemy poprzedzając go znakiem '-'):

- scn [nazwa sceny] — nazwa sceny zdefiniowanej w pliku *ustawienia.info*;
- bn [liczba] — określa liczbę koszyków, na jaką są dzielone trójkąty w każdym kroku budowy drzewa (wartość domyślna 8);
- mlt [liczba] — określa maksymalną liczbę trójkątów, jakie mogą znaleźć się w liściu drzewa (wartość domyślna 16);
- th [liczba] — określa liczbę wątków, jakie będą wykorzystywane podczas renderowania obrazów, maksymalnie 8 (domyślnie 1);
- res [liczba] [liczba] — określa rozdzielczość generowanych obrazów, najlepiej kwadrat o boku potęgi 2 (wartość domyślna 512 na 512);
- fpb [liczba] — określa co ile klatek drzewo ma zostać przebudowane w całości (wartość domyślna 1);
- f — jeśli podany, to wyświetla daną scenę w nieskończonej pętli (domyślnie wyłączone);
- save — zapisuje zrzuty ekranu z każdej wygenerowanej klatki (domyślnie wyłączone);
- statsFile [nazwa pliku] — zapisuje statystyki do podanego pliku (domyślnie wyłączone).

Bibliografia

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM. 5

[BEL+07] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based whitted and distribution ray tracing. In *GI '07: Proceedings of Graphics Interface 2007*, pages 177–184, New York, NY, USA, 2007. ACM. 8, 18, 41

[CFLB06] P.H. Christensen, J. Fong, D.M. Laur, and D. Batali. Ray tracing for the movie 'cars'. *Symposium on Interactive Ray Tracing*, 0:1–6, 2006. 3

[DHK08] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Computer Graphics Forum*, 27(4):1225–1233, June 2008. 4, 13, 14, 17, 23, 24, 25

[MT97] Möller and Trumbore. Fast, minimum storage ray-triangle intersection. *JGTOOLS: Journal of Graphics Tools*, 2, 1997. 9

[SWS05] Jörg Schmittler Sven Woop and Philipp Slusallek. Rpu: A programmable ray processing unit for realtime ray tracing. In *Proceedings of ACM SIGGRAPH 2005*, July 2005. 1, 3

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/. 4

[Wal07] Ingo Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007. 12, 14, 15, 17, 27, 41

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006. 11, 15

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. 5

[WK06] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering*, pages 139–150, June 2006. 13, 23

[WMG+07] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007. 5, 9