PRACA DYPLOMOWA - INŻYNIERSKA

Konrad Rodzik
nr albumu: 215039

NVIDIA CUDA - Technologia przyśpieszająca metodę śledzenia promieni

Kierownik pracy:
Prof. nazw. dr hab. inż. Dariusz Sawicki

.....
ocena pracy

.....
data i podpis Przewodniczącego
Komisji Egzaminacyjnej Dyplomowego

Warszawa, 2011

Życiorys

Urodziłem się 2 lutego 1988 r. w Rawie Mazowieckiej. W roku 2004 ukończyłem gimnazjum im. Haliny Konopackiej w Rawie Mazowieckiej i rozpocząłem naukę w Liceum Ogólnokształcącym im. Marii Skłodowskiej Curie w klasie o profilu matematyczno-informatycznym. Po zdaniu matury w 2007 roku zostałem studentem Politechniki Warszawskiej na wydziale Elektrycznym na kierunku Informatyka. Aktualnie kształce się w specjalizacji "Inżynieria Oprogramowania".

W roku 2010 zdobyłem I miejsce na "Ogólnopolskiej Konferencji Inżynierii Gier Komputerowych (IGK)." W roku 2009 na tej samej konferencji udało mi się wywalczyć II miejsce na podium.

Moje zainteresowania to: gry komputerowe, nowe technologie, film oraz motocykle

Streszczenie

Tytuł: NVIDIA CUDA - Technologia przyśpieszająca metodę śledzenia promieni”
W niniejszej pracy inżynierskiej przedstawiony został problem i przeprowadzone zostały badania nad metodą śledzenia promieni przy wykorzystaniu technologi NVIDIA CUDA. Do pracy została napisana aplikacja testowa, badająca wsteczną metodę śledzenia promieni na procesorach CPU oraz kartach graficznych GPU. Zrealizowana praca badawcza opiera się na badaniach testowych wydajności raytracingu na różnych konfiguracjach sprzętowych. W pracy został opisany projek, implementacja oraz przeprowadzone testy.

Abstract

Title: NVIDIA CUDA - Technology for speeding up raytracing
This engineering paper was presented problem and research about raytracing using NVIDIA CUDA technology. For the work was written test aplication whitch studies backward raytracing based on computer processors CPU and also on GPU graphics cards. Realized research work is based on raytracing test performance on different hardware configurations. In this paper was described design, implementation and conducted tests.

Spis treści

Podziękowania

Chciałbym bardzo podziękować moim rodzicom oraz mojej dziewczynie. Bez ich wsparcia i bodźców motywacyjnych powstanie tej pracy nie było by możliwe.

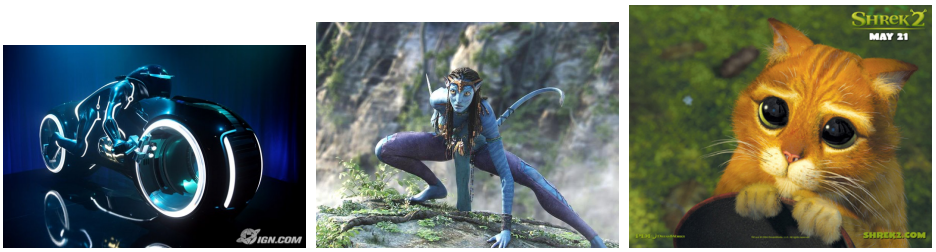
Na końcu lecz nie mniej ważne podziekowania dla mojego sprzętu, który dużo wycierpiał podczas powstawania niniejszej pracy. Karta graficzna wymieniana byla trzy razy...

Rozdział 1

Wstęp

1.1 Wprowadzenie

Raytracing jest techniką służącą do generowania foto realistycznych obrazów scen 3D. Na przestrzeni lat technika ta ciągle się rozwijała. Doczekała się wielu modyfikacji, które usprawniają proces generowania realistycznej grafiki. Takimi technikami mogą być między innymi PathTracing, Photon-Mapping, Radiostity i wiele innych. Z dnia na dzień wykorzystywanie raytracingu ciągle rośnie. W dzisiejszych czasach w grafice komputerowej oraz w kinematografii do uzyskania realistycznych efektów używana jest metoda śledzenia promieni. Dzięki takim zabiegom jesteśmy w stanie dosłownie zasymulować sceny oraz zjawiska, które nie muszą istnieć w rzeczywistym świecie. Czas generowania pojedynczej klatki/ujęcia takiej sceny niekiedy potrafi być liczony nawet w godzinach. Dlatego technika ta nie doczekała się jeszcze swojej wielkiej chwili w przemyśle rozrywkowym jakim są np. gry komputerowe oraz inne aplikacje generujące grafikę 3D w czasie rzeczywistym.



a) Tron Legacy b) Avatar c) Shrek

Rysunek 1.1: Ujęcia filmowe stworzone za pomocą technik komputerowych

1.2 Motywacja

Głównym bodźcem motywacyjnym do napisania tej pracy była chęć poszerzenia dotychczasowej wiedzy na temat przyśpieszania obliczeń przy pomocy nowej technologii NVIDIA CUDA. Dodatkowymi czynnikami motywacyjnymi było zamiłowanie do grafiki komputerowej oraz do tworzenia aplikacji obliczeń czasu rzeczywistego. Na co dzień zajmuję się programowaniem gier/aplikacji na komputery klasy PC oraz urządzenia mobilne. Uważam, że w przyszłości przedstawiona przeze mnie w tej pracy metoda śledzenia promieni będzie miała zastosowanie w grach oraz aplikacjach wykorzystujących grafikę czasu rzeczywistego.

1.3 Terminologia wykorzystywana w pracy

- Developer - Twórca/programista aplikacji
- CUDA - Technologia stworzona przez firmę NVIDIA w 2007 roku. Umożliwia równoległe obliczenia na mikroprocesorach karty graficznej
- Raytracing - metoda generowania obrazu za pomocą śledzenia promieni.
- Benchmark - Aplikacji testowa, która profiluje wydajność i zbiera informacje.
- Warp - blok wątków przydzielony na multiprocesor
- DirectX - technologia graficzna firmy Microsoft. Umożliwia wyświetlanie wysokiej jakości grafiki 2D/3D.
- Aliasing - Zdeformowany, o złej jakości obraz który powstaje podczas rastaryzacji, powodowany przez zbyt mała częstotliwość próbkowania na pojedyńczy pixel obrazu. Przeciwdziała się temu efektowi poprzez antyaliasing oraz w raytracingu poprzez super-sampling
- Super-sampling - sposób na zwiększenie jakości generowanych scen. Polega na śledzeniu wielu promieni świetlnych na pojedyńczy pixel generowanego obrazu.

1.4 Dostępne technologie, pozwalające zrównoleglić obliczenia na kartach graficznych

Poniżej zaprezętowane zostały trzy wybrane technologię wspomagające równoległe obliczenia na kartach graficznych. Niemniej jednak badania przepro-

wadzone i opisane w dalszej części pracy będą skupiały się na wykorzystaniu jednej z tych metod, a mianowicie technologii NVIDIA CUDA.

1.4.1 Open Computing Language (OpenCL)

Technologia tak zainicjowana została przez firmę Apple. Do inicjatywy i rozwijania tej technologii włączyły się w późniejszym czasie inne firmy takie jak: AMD, IBM, Intel, NVIDIA. W roku 2008 sformowana została grupa Khronos skupiająca powyższe firmy oraz wiele innych należących do branży IT. Grupa ta czuwa nad rozwojem technologii OpenCL. Technologia tak pozwala na pisanie kodu który jest przenośny między wieloma platformami: komputery, urządzenia przenośne, klastry obliczeniowe. OpenCL pozwala rozpraszać obliczenia na jednostki procesorowe CPU oraz na architektury graficzne GPU. Bardzo ważną zaletą OpenCL jest to, że pisanie z użyciem tej technologii nie jest zależne od sprzętu na jakim będzie ona uruchamiana.

1.4.2 ATI Stream Computing

Technologia ta została stworzona przez firmę AMD. Za pomocą tej platformy jesteśmy w stanie przeprowadzać złożone obliczenia na sprzęcie produkowanym przez AMD. W skład całego pakietu ATI Stream Computing wchodzi autorski język ATI Brook+ i kompilator tegoż języka. Dodatkowo ATI wspiera developerów własną biblioteka matematyczna (AMD Core Math Library) oraz narzędziami do profilowania wydajności kodu (Strem Kernel Analyzer). Technologia ATI konkuruje od dawna z technologią NVIDIA CUDA.

1.4.3 NVIDIA CUDA

CUDA (Compute Unified Device Architecture) jest technologia opracowaną przez firmę NVIDIA. Swoje początki CUDA miała w 2007 roku i do dziś jest wiodącą technologią strumieniowego przetwarzania danych z wykorzystaniem układów graficznych GPU. Dalszemu opisu niniejszej technologii poświęcony zostanie osobny rozdział.

a) Logo NVIDIA CUDA

b) Logo ATI Stream Computing

Rysunek 1.2: Loga dwóch konkurencyjnych ze sobą technologii przetwarzania równoległego na kartach graficznych

Rozdział 2

Cele pracy

2.1 Opracowanie techniki zrównoleglenia i przyspieszenia metody śledzenia promieni przy użyciu NVIDIA CUDA

Celem niniejszej pracy jest przeniesienie a zarazem zrównoleglenie algorytmu śledzenia promieni na procesory graficzne (GPU) firmy NVIDIA. Celem także jest przyśpieszenie obliczeń standardowego wstecznego Raytracingu w celu jak najszybszego generowania obrazów scen 3D.

2.2 Projekt uniwersalnej aplikacji - benchmark

W ramach projektu napisany został uniwersalny system Raytracingu działający na wielo rdzeniowych procesorach komputerowych (CPU), a także na kartach graficznych (GPU) firmy NVIDIA które obsługują technologie NVIDIA CUDA. Aplikacja testowa jest benchmarkiem, który jest w stanie przetestować zadane sceny 3D na wielu różnych konfiguracjach sprzętowych. Aplikacja ma za zadanie po uruchomieniu na komputerze użytkownika, testować wszelkie sceny z odpowiedniego katalogu. Dodatkowo zbierać potrzebne informacje o sprzęcie użytkownika oraz czasy generowania obrazów z każdej ze scen. Po przeprowadzeniu wszelkich testów aplikacja jest w stanie wyslać na adres e-mail developera (w tym przypadku autora pracy) wszelkie zgromadzone dane.

Rozdział 3

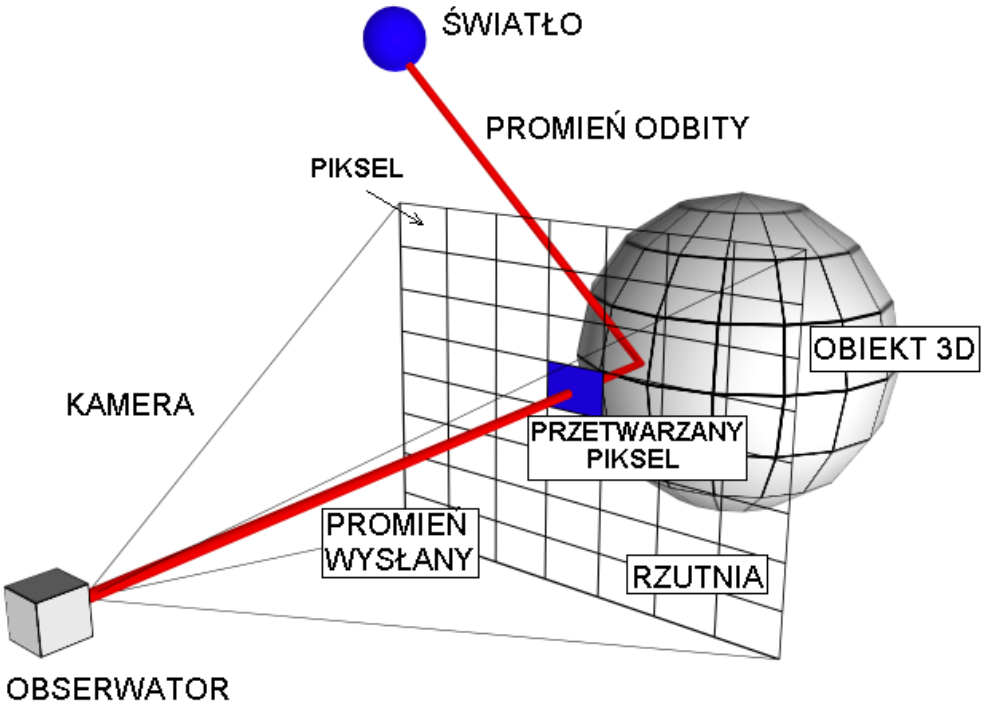
Wprowadzenie do Raytracingu

3.1 Wstępny opis

W rzeczywistym świecie promienie świetlne rozchodzą się od źródła światła do obiektów znajdujących się w świecie. Każde źródło światła wysyła nieskończoną liczbę swoich promieni świetlnych. Następnie te promienie odbijając się od obiektów i trafiają do oczu obserwatora powodując że widzi on określony kolor danego obiektu. Gdyby zaadaptować tą metodę do generowania realistycznej grafiki komputerowej, otrzymalibyśmy nieskończenie dokładny i realistyczny obraz. Z racji jednak na to, że sprzęt komputerowy ma ograniczone możliwości, a metoda ta jest bardzo nie efektywną metodą pod względem obliczeniowym. Najszerzej stosowaną metodą śledzenia promieni jest wsteczne śledzenie promieni (backward raytracing). W odróżnieniu od postępowego algorytmu śledzenia promieni (forward raytrcing), które opiera się na generowaniu jak największej liczby promieni dla każdego źródła światła. Algorytm wstecznego śledzenia promieni zakłada, że promienie śledzone są od obserwatora, poprzez scenę do obiektów z którymi kolidują. Poniżej przedstawiony jest poglądowy rysunek śledzenia pojedynczego promienia od obserwatora poprzez określony pixel na ekranie: 3.1

3.2 Rekursywna metoda śledzenia promieni

Przy omawianiu wstecznej metody śledzenia promieni warto wspomnieć o raytracingu rekursywnym. W zagadnieniu tym bada się rekurencyjnie promienie odbite zwierciadlane oraz załamane, które powstały z kolizji promieni pierwotnych z obiektami na scenie. Tak więc żywotność promienia pierwotnego wcale nie kończy się w momencie kolizji z obiektem sceny. To czy z danego promienia pierwotnego wygenerowane zostaną kolejne promienie w

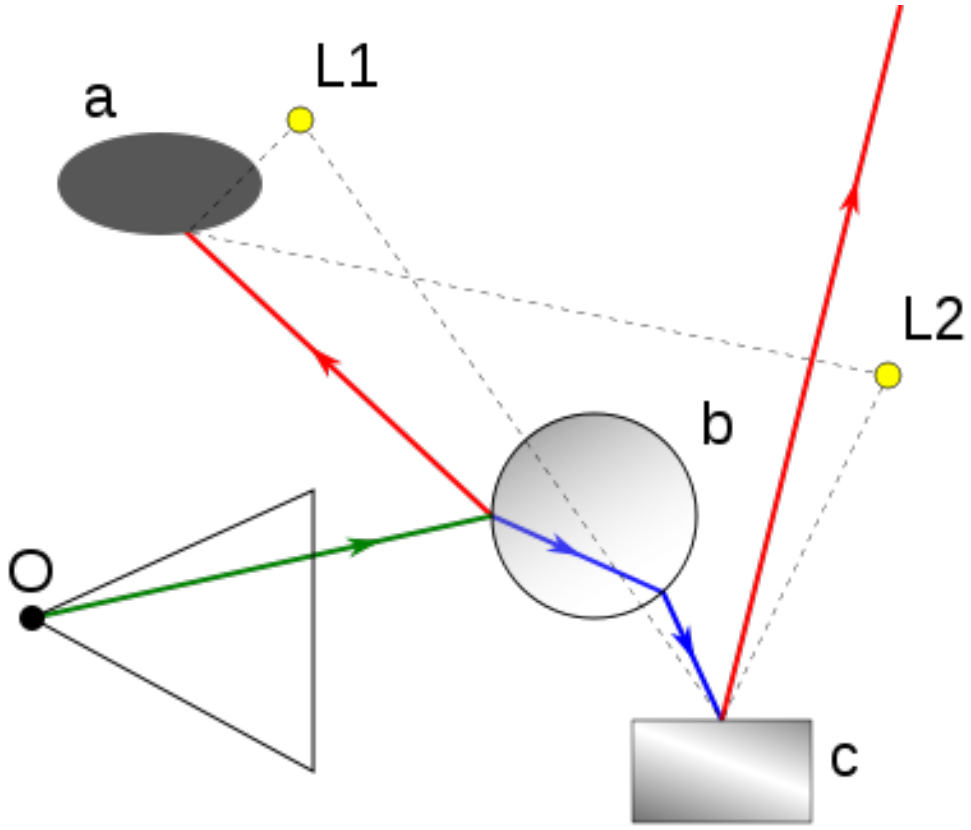


Rysunek 3.1: Sposób określania barwy piksela w raytracigu

bardzo dużej mierze zależy od materiału jakim pokryty jest dany obiekt sceny. Z pomocą tego rekursywnej metody śledzenia promieni jesteśmy w stanie zasymulować obiekty lustrzane oraz obiekty półprzezroczyste. Rekurencja w tej metodzie trwa do osiągnięcia maksymalnego stopnia zagłębienia. Kolor wynikowy danego pojedynczego Pixela powstaje z sumy kolorów, obiektu w jaki trafił promień pierwotny oraz kolorów obiektów w jakie trafiły promienie pierwotne. Poniżej przedstawiony jest poglądowy schemat zasady działania rekursywnej metody śledzenia promieni: 3.2

3.3 Przedstawienie algorytmu śledzenia promieni

Śledzenie promieni przez scenę rozpoczyna się od obserwatora określanego często jako kamery występującej na scenie. Przez każdy pixel ekranu śledzone są promienie które poruszają się po scenie. Gdy któryś ze śledzonych promieni napotka obiekt i zacznie z nim kolidować.



Rysunek 3.2: Zasada działania rekursywnego algorytmu ray tracingu

3.4 Przykład szósty

Oto przykładowy wydruk:

```
1 /* ta funkcja oblicza a+b */
2 int sum(int a, int b)
3 {
4   int suma=0;
5
6   suma=a+b;
7
8   return suma;
9 }
```

Rozdział 4

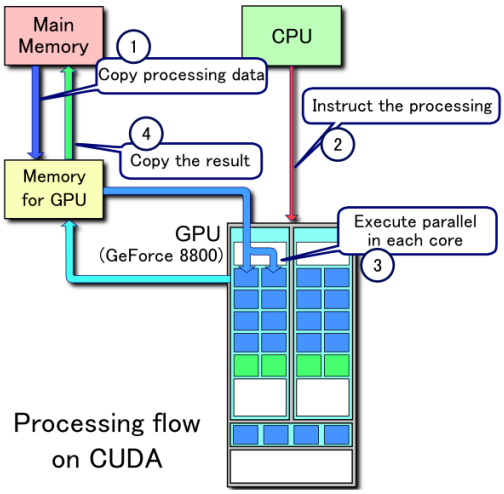
NVIDIA CUDA jako znakomita platforma do zrównoleglenia obliczeń

4.1 Wstępny opis

CUDA(Compute Unified Device Architecture) jest dość nową technologią wprowadzoną na rynek przez firmę NVIDIA. Technologia ta swój początek miała w 2007 roku. Od samego początku stała się ona wiodącą technologią przetwarzania strumieniowego z wykorzystaniem GPU. CUDA jako, że jest technologią stworzoną przez firmę NVIDIA, wspierana jest przez układy graficzne właśnie tej firmy. Wsparcie dla tej technologii rozpoczęło się od układów graficznych serii GeForce 8, Quadro oraz Tesla. Seria układów graficzny Quadro oraz Tesla są wyspecjalizowanymi układami obliczeniowymi do zastosowań naukowych. Natomiast serie GeForce można spotkać na co dzień w komputerach stacjonarnych oraz laptopach. Z pomocą technologii CUDA jesteśmy wstanie uzyskać wielokrotne przyśpieszenie w obliczeniach w stosunku do obliczeń na zwykłym procesorze CPU. na ryskunku 4.1 przedstawiony został przykładowy schemat przepływu obliczeń w CUDA.

4.2 Wspierane karty oraz zdolność obliczeniowa

We wstępnym opisie powiedziane było, że technologia CUDA zapoczątkowana była w układach graficznych serii GeForce, Tesla oraz Quadro. W tabeli 4.1 przedstawione zostało oficjalne wsparcie określonej wersji CUDA w poszczególnych układach graficznych.



Rysunek 4.1: Przykład przepływu przetwarzania w technologii CUDA.

Kolejną ważną rzeczą wyróżniająca karty graficzne jest ich zdolność obliczeniowa (ang. compute capability). Identyfikuje ona możliwości obliczeniowe danej karty graficznej w odniesieniu do technologii NVIDIA CUDA. W tabeli 4.2 przedstawione zostały możliwości kart graficznych w zależności od profilu CUDA.

4.3 Architektura

Karty graficzne GPU znacznie różnią się wydajnością od zwykłych procesorów CPU. Różnica w wydajności wynika głównie z faktu, iż procesory graficzne specjalizują się w równoległych, wysoce intensywnych obliczeniach. Karty graficzne składają się z większej liczby tranzystorów które są odpowiedzialne za obliczenia na danych. Nie posiadają natomiast takiej kontroli przepływu instrukcji oraz jednostek odpowiedzialnych za buforowanie danych jak procesory komputerowe CPU. Układy graficzne wspierające technologię CUDA zbudowane z multiprocesorów strumieniowych (ang. stream multiprocessor). Różne modele kart graficznych firmy NVIDIA posiadają różną liczne multiprocesorów, co przekłada się także na wydajność i zdolnosć obliczeniową danej architektury. Na rysunku 4.2 Przedstawiona jest przykładowa budowa takiego właśnie multiprocesora.

Każdy z multiprocesorów składa się z: (napisac z kad to wziete!!!!!)

- I-Cache - bufor instrukcji
- MT Issue - jednostka która rozdziela zadania dla SP i SFU

Zdolność obliczeniowa (wersja)	GPUs	Cards
1.0	G80	GeForce 8800GTX/Ultra/GTS, Tesla C/D/S870, FX4/5600, 360M
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b	GeForce 8400GS/GT, 8600GT/GTS, 8800GT, 9600GT/GSO, 9800GT/GTX/GX2, GTS 250, GT 120/30, FX 4/570, 3/580, 17/18/3700, 4700x2, 1xxM, 32/370M, 3/5/770M, 16/17/27/28/36/37/3800M, NVS420/50
1.2	GT218, GT216, GT215	GeForce 210, GT 220/40, FX380 LP, 1800M, 370/380M, NVS 2/3100M
1.3	GT200, GT200b	GTX 260/75/80/85, 295, Tesla C/M1060, S1070, CX, FX 3/4/5800
2.0	GF100, GF110	GTX 465, 470/80, Tesla C2050/70, S/M2050/70, Quadro 600,4/5/6000, Plex7000, 500M, GTX570, GTX580
2.1	GF108, GF106, GF104	GT 420/30/40, GTS 450, GTX 460

Tabela 4.1: Zestawienie kart graficznych oficjalnie wspierających technologię CUDA.

- C-Cache -bufor stałych (ang. constant memory) o wielkości 8KB, który przyspiesza odczyt z obszaru pamięci stałej
- 8 x SP - 8 jednostek obliczeniowych tzw stream processors, które wykonują większość obliczeń pojedynczej precyzji (każdy zawiera własne 32-bitowe rejestry)
- 2 x SFU - jednostki specjalne (ang. special function units). Zadaniem ich jest obliczanie funkcji przestępnych, np. trygonometrycznych, wykładniczych i logarytmicznych, czy interpolacja parametrów.
- DP -procesor, który wykonuje obliczenia podwójnej precyzji

Zdolność obliczeniowa	1.0	1.1	1.2	1.3
Funkcje atomowe w pamięci globalnej	-	✓	✓	✓
Funkcje atomowe w pamięci współdzielonej	-	-	✓	✓
Ilość rejestrów na multiprocesor	8192	8192	16384	16384
Maksymalna liczba warpów na multiprocesor	24	24	32	32
Maksymalna liczba aktywnych wątków na multiprocesor	768	768	1024	1024
Podwójna precyzja	-	-	-	✓

Tabela 4.2: Porównanie zdolności obliczeniowych kart graficznych wspierających NVIDIA CUDA.

- SM - pamięć współdzielona (ang. shared memory) o wielkości 16KB.

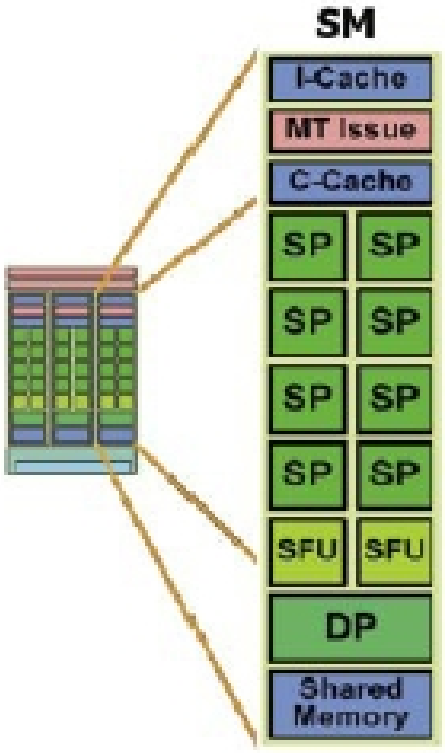
4.4 Rodzaje pamięci w architekturze CUDA

- Pamięć globalna (ang. global memory) - Ta pamięć jest dostępna dla wszystkich wątków. Nie jest pamięcią buforowaną. Dostęp do niej trwa od około 400 do 600 cykli. Pamięć ta służy przede wszystkim do zapisuj wyników działań programu obliczeniowego.

- Pamięć lokalna (ang. local memory) - Ma taki sam czas dostępu jak pamięć globalna (400-600 cykli). Nie jest także pamięcią buforowaną. Jest ona zdefiniowana dla danego wątku. Każdy wątek CUDA posiada własną pamięć lokalną. Zajmuje się ona przechowywaniem bardzo dużych struktur danych. Pamięć ta jest najczęściej używana gdy obliczenia danego wątku nie mogą być w całości wykonane na dostępnych rejestrach procesora graficznego.

- Pamięć współdzielona (ang. shared memory) - Jest to bardzo szybki rodzaj pamięci, dorównujący szybkości rejestrów procesora graficznego. Przy pomocy tej pamięci, wątki przydzielone do jednego bloku są wstanie się ze sobą komunikować. Należy jednak obchodzić się ostrożnie z tym rodzajem pamięci, gdyż mogą powstać Momoty w których wątki w jednym bloku będą chciały jednocześnie zapisywać i odczytywać z tej pamięci. Występowanie takich konfliktów w odczycie i zapisie powoduje duże opóźnienia.

- Pamięć stała (ang. const memory) - Ta pamięć w odróżnieniu do powyższych rodzajów pamięci, jest buforowaną pamięcią tylko do od-



Rysunek 4.2: Przykładowy schemat multiprocesora strumieniowego.

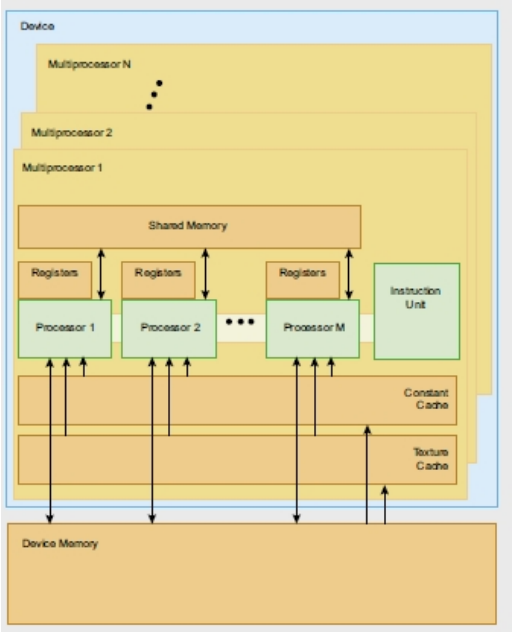
czytu. Gdy potrzebne dane znajdują się aktualnie w buforze dostęp do nich jest bardzo szybki. Czas dostępu rośnie gdy danych nie ma w buforze i muszą być doczytane z pamięci karty.

- Pamięć Tekstur (ang. texture memory) - Jest pamięcią podobną do pamięci stałej gdyż udostępnia tylko odczyt danych. Jest także pamięcią buforowaną. W pamięci tej bufor danych został zoptymalizowany pod kątek odczytu danych z bliskich sobie adresów. Najkorzystniejszą sytuacją jest gdy wątki dla danego warpa (grupa 32 wątków zarządzanych przez pojedynczy multiprocesor) odczytują adresy, które znajdują się blisko siebie. CUDA w swojej implementacji udostępnia możliwość posługiwania się teksturami 1D,2D,3D.

- Rejestry - Jest to najszybszy rodzaj pamięci. Dostęp do niego nie powoduje żadnych dodatkowych opóźnień, chyba że próbujemy odczytać z rejestru do którego dopiero co zostało coś zapisane. Każdy multiprocesor w urządzeniu CUDA posiada 8192 lub 16384 rejestrów 32-bitowych. Zależy to od wersji(zdolności obliczeniowej) danego urządzenia. W celu uniknięcia powyższych konfliktów ilość wątków na po-

jedynczy multiprocesor ustawia się jako wielokrotność liczby 64. NAPISAC Z KAD TO WIEM!!!!!!!!!!!!1

Na obrazku 4.3 poniżej przedstawiony został poglądowy schemat pamięci w architekturze CUDA.



Rysunek 4.3: Schemat pamięci.

4.5 Przykładowy program pod architekturę CUDA

Poniżej przedstawiony został przykład programu napisanego w języku C dla architektury CUDA. Program ten uruchamiany jest na wielu wątkach karty graficznej, każdy z tych wątków niezależnie wpisuję do tablicy swoje ID. Ważną informacją przy pisaniu kodu dla architektury CUDA jest to, że funkcje uruchamiane przez wątki mają specjalne oznaczenia:

- global - funkcje taką wywołać można tylko z CPU, a wykonuje się ona na GPU
- host - funkcjawykonuje się i może być wywołana tylko z kodu wykonywanego na CPU
- device - funkcja wykonuje się i może być wywołana tylko z kodu wykonywanego na GPU

Należy także pamiętać że funkcje dla wątków CUDA muszą zawsze zwracac wartosc void.

```
1  #include <stdlib.h>
2  #include <cuda_runtime.h>
3  #include <cutil.h>

5  // definicja funkcji która będzie uruchamiana
6  // równolegle na wątkach CUDA
7  __global__ void testFunction(int *data)
8  {
9    // obliczamy index tablicy a zarazem wątku
10   int id = blockIdx.x * blockDim.x + threadIdx.x;
11   // Zaposujemy do tablicy ID wątku
12   data[id] = id;
13 }

15 // W funkcji main wywołujemy powyższą funkcje
16 // dla wątków CUDA
17 int main()
18 {
19   // Na poczatku należey zainicjować urządzenie CUDA
20   cudaSetDevice(0);

22   // alokujemy pamieć na karcie graficznej
23   int *tablica;
24   cudaMalloc((void**)&tablica, sizeof(int) * ARRAY_SIZE);

26   // Ustalamy wielkosc bloku i karty
27   dim3 dimBlock(BLOCK_SIZE, 1);
28   dim3 dimGrid(ARRAY_SIZE / dimBlock.x, 1);

30   // wywołujemy naszą funkcję obliczeniową
31   testFunction<<<dimGrid, dimBlock>>>(tablica);

33   // Tworzymy tablice w pamieci ram i kopujemy
34   // dane z karty graficznej do pamieci ram.
35   int *tablica2 = (int*)malloc(sizeof(int) * ARRAY_SIZE);
36   cudaMemcpy(tablica, tablica2, sizeof(int) * ARRAY_SIZE,
37   cudaMemcpyDeviceToHost);

39   return 0;
40 }
```

Jak widzimy na powyższym listingu kodu gdy wywołujemy funkcję CUDA określamy na ilu wątkach ma się ona uruchomić i w jakie grupy mają być one pogrupowane. Na rysunku JAKIS RYSSSSSSSSSSSSSSSss przedstawiony został schemat pokazujący jak mogą wyglądać używane wątki w całej kgracie, pogrupowane w odpowiednie bloki. Podczas programowania na karty graficzne CUDA należy pamiętać o różnych dostępnych rodzajach

pamięci i wybrać tą najłaściwszą. Jeśli nie przemyślimy dobrze problemu jaki sobie założyliśmy rozwiązać przy pomocy technologii CUDA, może się zdażyć, że nasze rozwiązanie będzie działało gorej niż na procesorze CPU. Należy także poinformować o tym, że brak jest narzędzi, które wspomagały by śledzenie przepływu wykonywania programu tzw debugowanie. Z tym problemem borykaja się wszystkie technologie zwiazane z GPGPU(obliczenia przeprowadzane na kartach graficznych).

Rozdział 5

Omówienie aplikacji testowej

5.1 Założenia

Na potrzeby niniejszej pracy zostało opracowane autorskie rozwiązanie uniwersalnego wstecznego raytracera działającego zarówno na procesorze CPU jak i również na ko-procesorach graficznych GPU firmy NVIDIA. Aplikacja testowa jest wstanie generować wynikowe obrazy scen 3D składających się z kul, prostopadłościanów oraz płaszczyzn. Na każdy z elementów sceny jest możliwość nałożenia dowolnej tekstury oraz doboru odpowiednich parametrów materiału. Dodatkowo na scenie możliwe jest umieszczanie świateł punktowych. Aplikacja sama w sobie jest benchmarkiem, który potrafi przetestować zadaną liczbę scen 3D na komputerze użytkownika. Zebrane wyniki z obliczeń jest wstanie przesłać na wybrany adres e-mail (w tym przypadku za zgodą użytkownika do developera). Aplikacja przy generowaniu obrazu sceny 3D bierze pod uwagi różne właściwości materiału danego obiektu. Docelowo generowane są takie efekty jak: oświetlenie, odblask, cienie, wielokrotne odbicia i załamania, tekstury. Przy użyciu materiałów o różnych parametrach jesteśmy wstanie uzyskać bardzo ciekawie wyglądające obiekty np: lustro, szkło, metale i wiele innych.

5.2 Implementacja

Aplikacja testowa została napisana w języku C++, wykorzystując biblioteki standardowe pochodzące z języka C. Wersja śledzenia promieni przy użyciu technologii CUDA została napisana w tzw. "C for CUDA". Dodatkowo do wyświetlania wynikowych obrazów użyta została biblioteka Microsoft DirectX 9.0. Program przeznaczony jest do uruchamiania na systemach z ro-

dziny Windows. Aplikację testową można nazwać swoistym benchmarkiem. Działanie jej składa się z 5 ważnych punktów:

- wczytywanie scen do testow
- testowanie zadanych scen na procesorze CPU.
- testowanie zadanych scen na karcie graficznej GPU.
- zapisywanie wynikowych obrazów scen na dysk użytkownika
- zebranie informacji o testowanych scenach i wysłanie ich na mail developera.

Przebieg działania:
Aplikacja na samym początku wczytuje plik benchamarku z rozszerzeniem *.rtb. Plik ten zawiera w sobie spis scen (pliki *.rtm) które mają być przetestowane przez raytracer. Następnie rozpoczyna się testowanie zadanych scen na procesorze CPU. Gdy wszystkie sceny zostaną przetestowane na procesorze, rozpoczyta się raytracing na karcie graficznej z użyciem CUDA. Na koniec gdy już wszystkie sceny zostały wygenerowane przy użyciu CPU oraz GPU, wynikowe obrazy generowane przez raytracer zapisywane są na dysku użytkownika. Zebrane informacje z profilowania każdej ze scen zostają zapisane do pliku oraz wysłane na adres e-mail developera.

Statystyki związane z kodem aplikacji testowej:

- 61 plików kodu
- 9386 linii kodu
- 272697 bajtów kodu

5.3 Zestaw testów

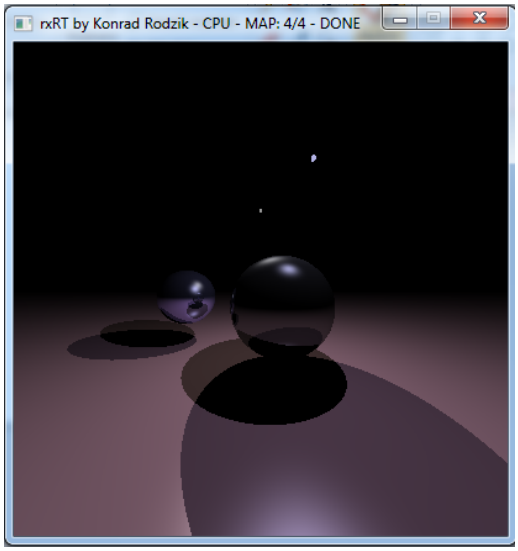
By wykazać przyśpieszenie pomiędzy śledzeniem promieni na procesorze CPU a kartą graficzną GPU przygotowany został zestaw 10 scen testowych. Testowana jest wydajność generowania scen o różnej budowie i występujących na niej prymitywach. W scenach tych testowanych jest wiele parametrów takich jak:

- Odbicia promieni od obiektów na scenie
- Załamania promieni w obiektach na scenie

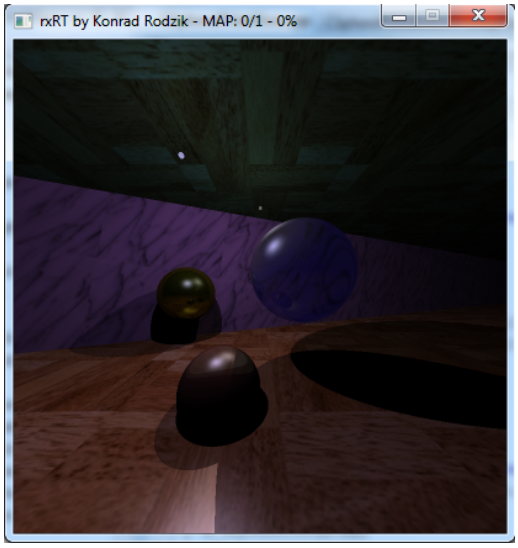
- Teksturowanie obiektów sceny
- Rodzaj oraz ilość prymitywów wyświetlanych na scenie
- Jakość generowanego obraz (super sampling)
- Rozdzielczość generowanego obrazu

5.4 Przykłady wygenerowanych obrazów

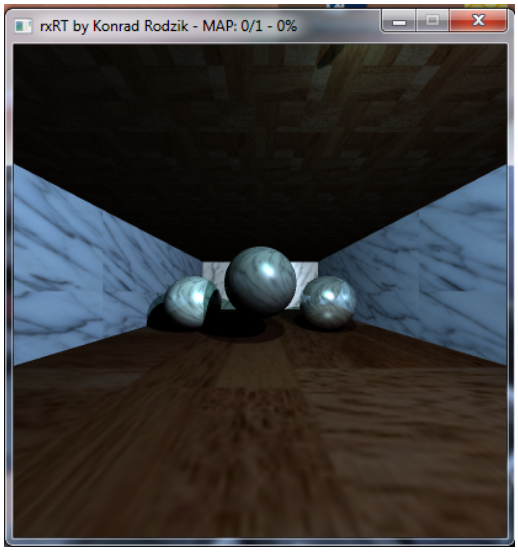
W rozdziale tym przedstawione zostały wyniki generowania scen przez aplikacje testową. Każda z tych scen była generowana na procesorze CPU oraz na karcie graficznej GPU.



Rysunek 5.1: Przykładowa wygenerowana scena 1.



Rysunek 5.2: Przykładowa wygenerowana scena 2.



Rysunek 5.3: Przykładowa wygenerowana scena 3.

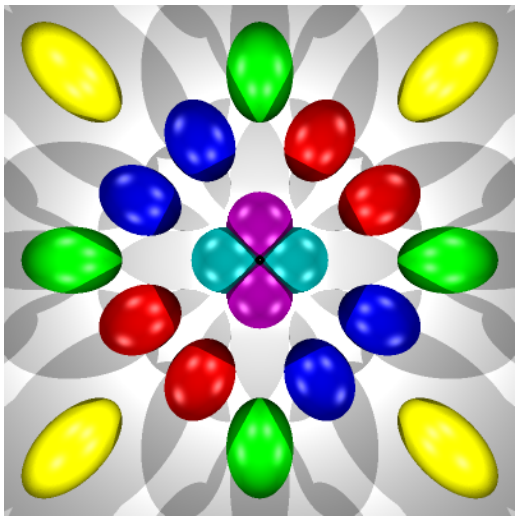
Rozdział 6

Porównanie wydajności Raytracera działającego na CPU oraz na GPU

Tutaj przedstawię wszelkie wyniki generowania obrazów ze scen oraz wszelkie wartości czasowe im odpowiadające. Porównanie wyników dla raytracingu CPU vs GPU. Z każdych wynikowych danych badań przeprowadzonych na różnych konfiguracjach sprzętowych wygenerowany zostanie wykres przedstawiający różnice czasowe w generowaniu scen a tym samym przyspieszenie wstecznego raytracingu jakie udało się uzyskać. Zakładam wstępnie testy 10 scen. Na każdej z nich różna konfiguracja obiektów oraz materiałów im nadanych. Chce pokazać jakie właściwości materiałów i jaka ilość prymitywów oraz ich rodzajów wpływa na zmniejszanie/zwiększanie wydajności raytracingu.

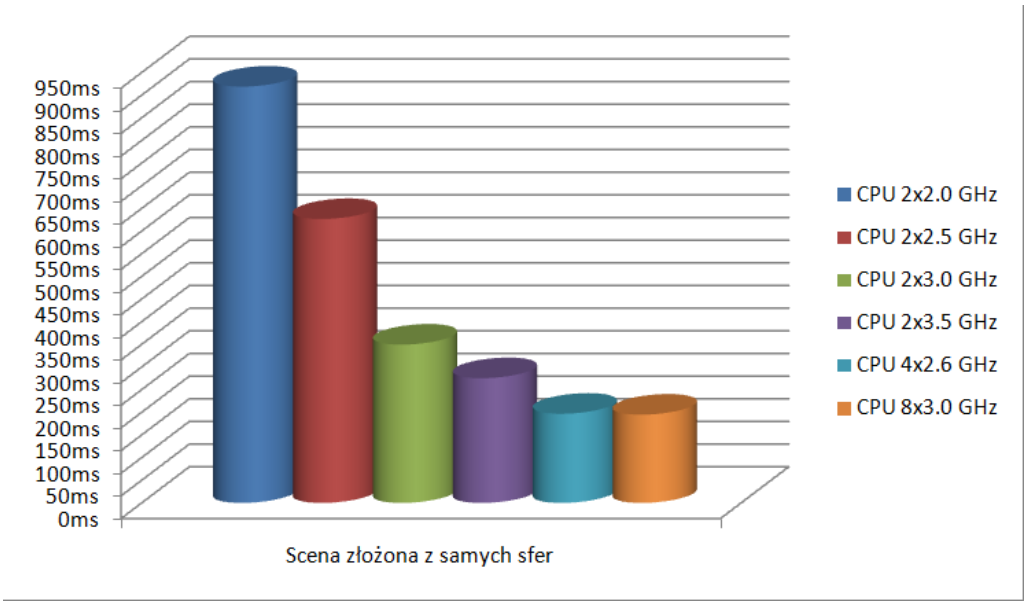
6.1 Test1 - Scena złożona z samych sfer

Testowi poddana została scena zawierająca jako prymitywy same sfery. Na scenie jest dokładnie 20 sfer, które mają nałożone różne materiały. Na scenie tej znajdują się także 4 światła punktowe.

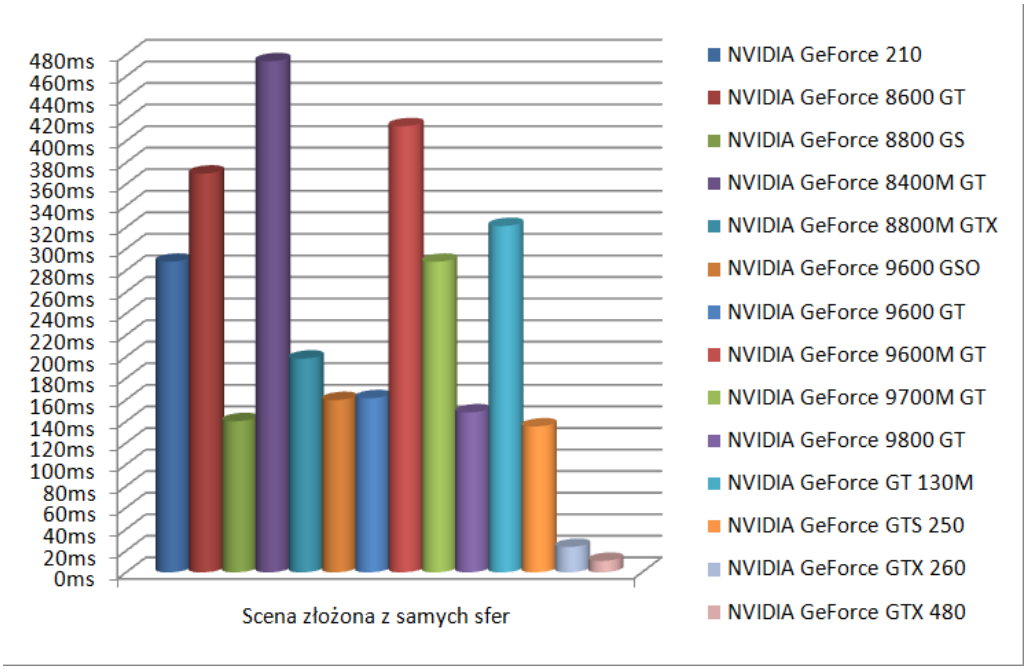


Rysunek 6.1: Scena złożona ze sfer

WYKRESY:



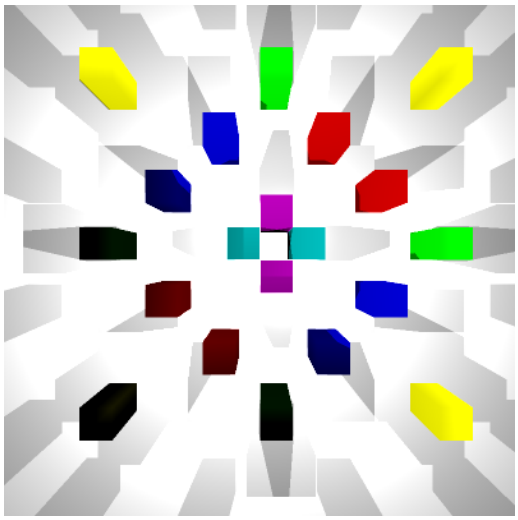
Rysunek 6.2: Wykresy CPU dla sceny złożonej z samych sfer



Rysunek 6.3: Wykresy GPU dla sceny złożonej z samych sfer

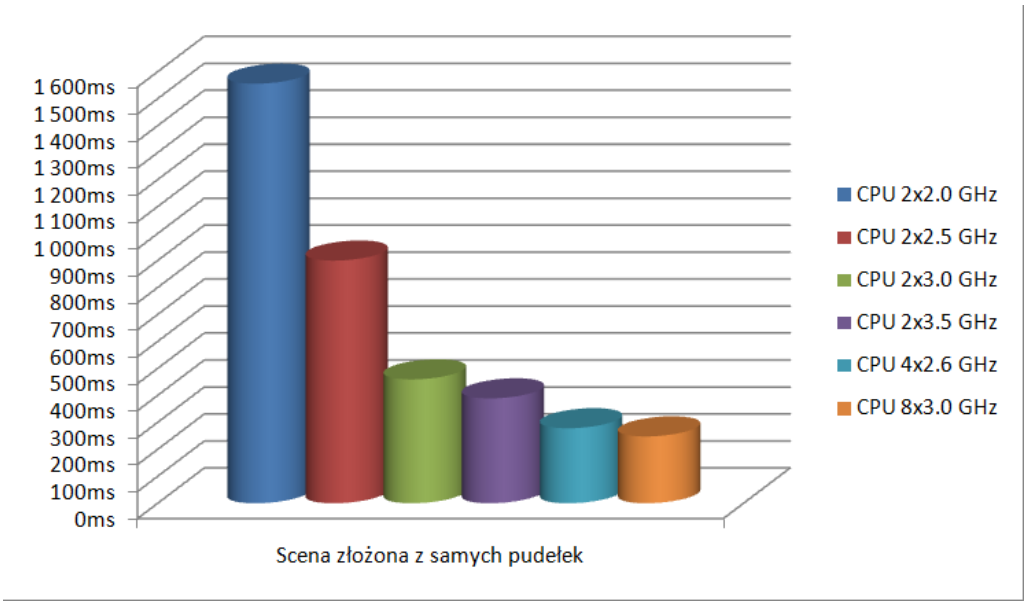
6.2 Test2 - Scena złożona z samych pudełek

W tym teście na scenie zostało umieszczonych 20 pudełek (boxów). Każde z tych pudełek posiada swój własny materiał. Scena również oświetlana jest przez 4 źródłą światła punktowego.

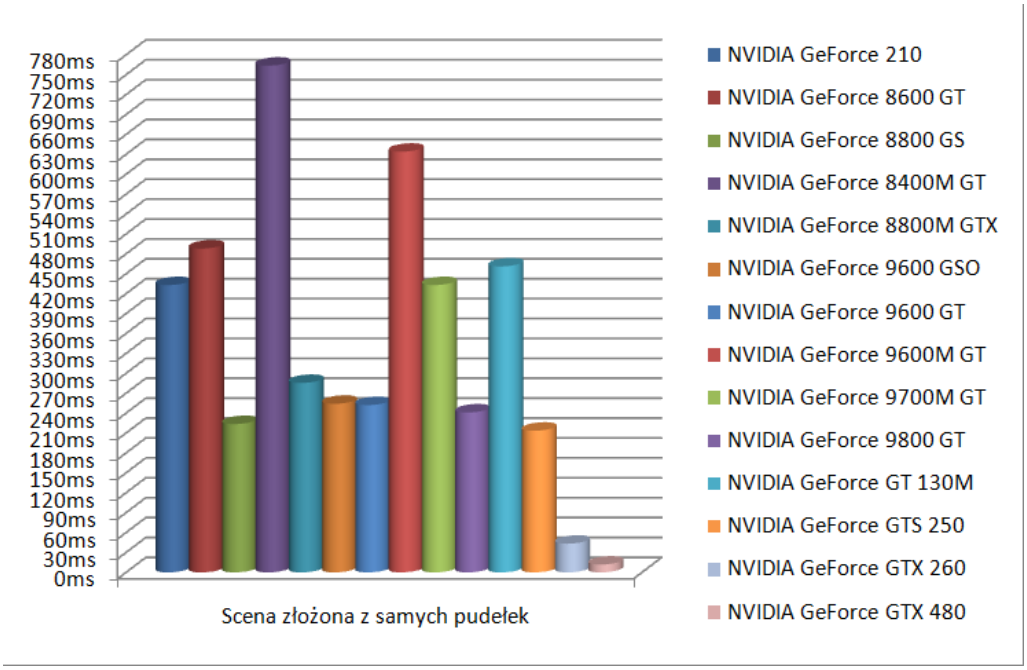


Rysunek 6.4: Scena złożona ze sfer

WYKRESY:



Rysunek 6.5: Wykresy CPU dla sceny złożonej z samych pudełek

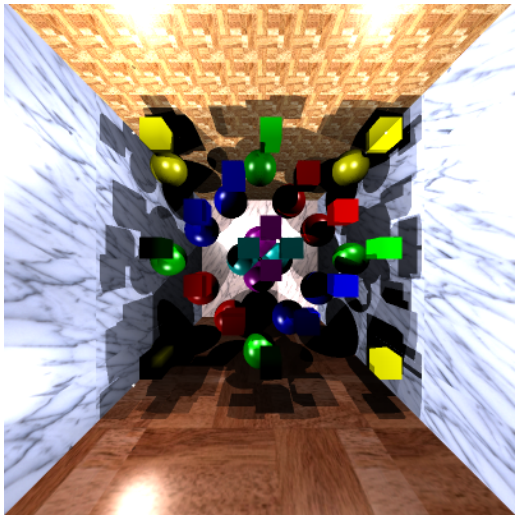


Rysunek 6.6: Wykresy GPU dla sceny złożonej z samych pudełek

6.3 Test3 - Scena z różnymi prymitywami

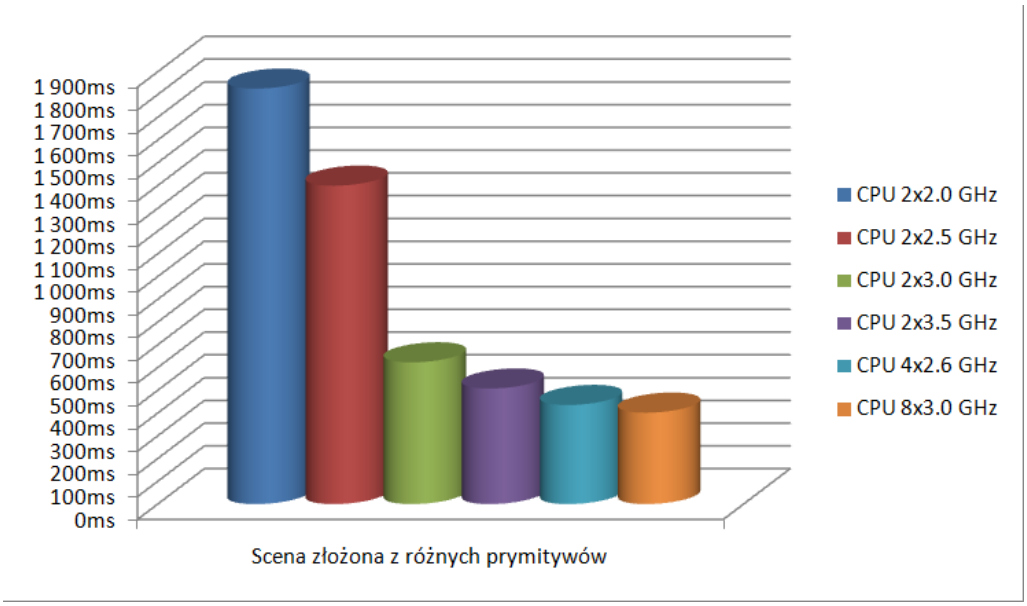
Testowi poddana została scena z różnego rodzaju prymitywami. Każdy z prymitywów posiada swój materiał, niektóre także teksture. Scena składa się dokładnie z:

- dwudziestu sfer
- dwudziestu pudełek
- sześciu płaszczyzn
- czterech źródeł światła punktowego.

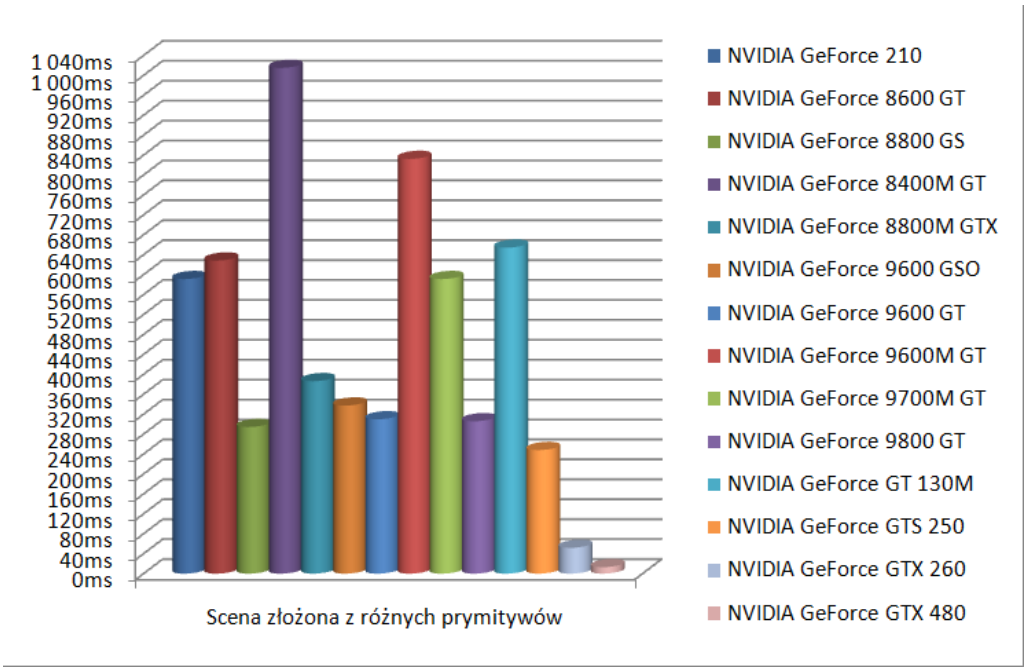


Rysunek 6.7: Scena złożona ze sfer

WYKRESY:



Rysunek 6.8: Wykresy CPU dla sceny złożonej z różnych prymitywów

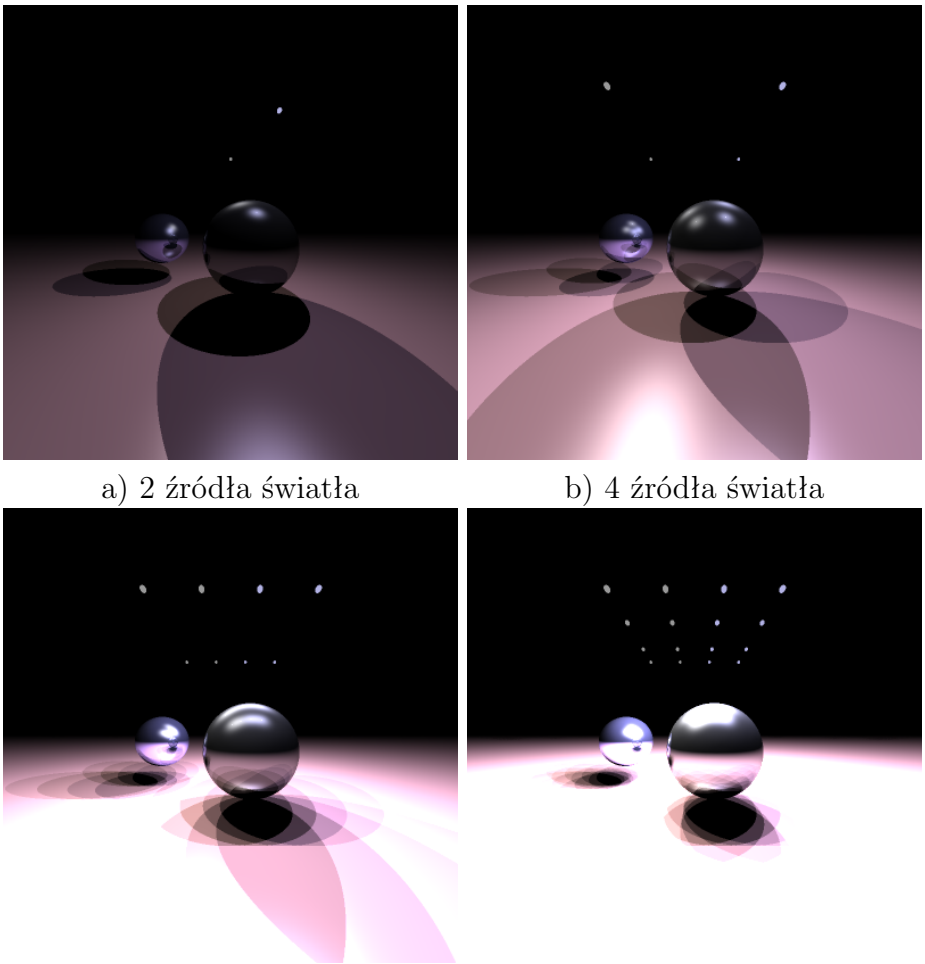


Rysunek 6.9: Wykresy GPU dla sceny złożonej z różnych prymitywów

6.4 Test4 - Różna liczba świateł punktowych na scenie

W teście sprawdzone zostało jaki wpływ ma liczba świateł punktowych na szybkość generowanej sceny. Testy zostały wykonane dla czterech różnych wariantów:

- dwa źródła światła punktowego.
- cztery źródła światła punktowego.
- osiem źródeł światła punktowego.
- szesnaście źródeł światła punktowego.

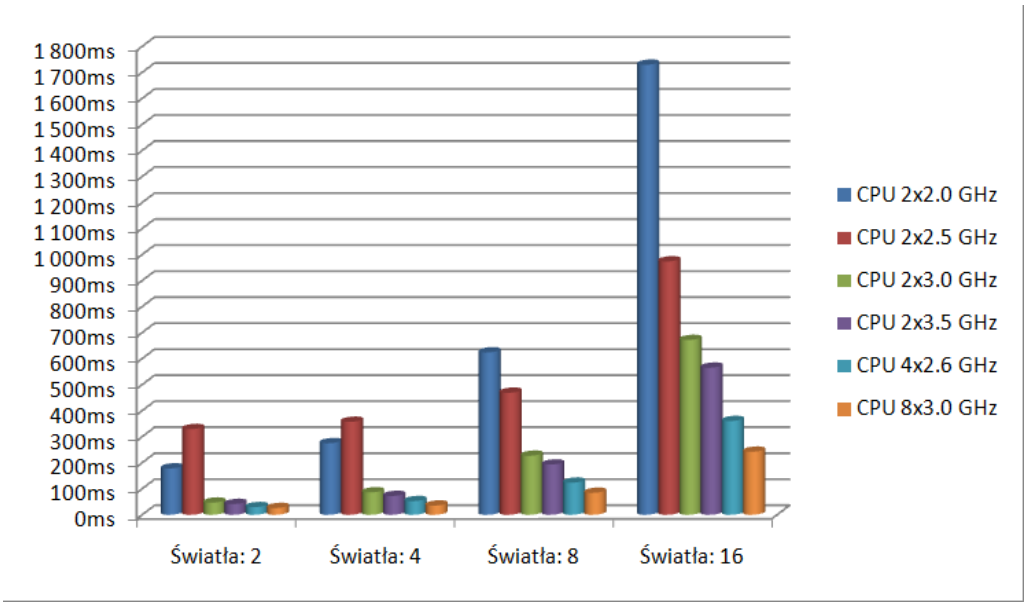


a) 2 źródła światła b) 4 źródła światła

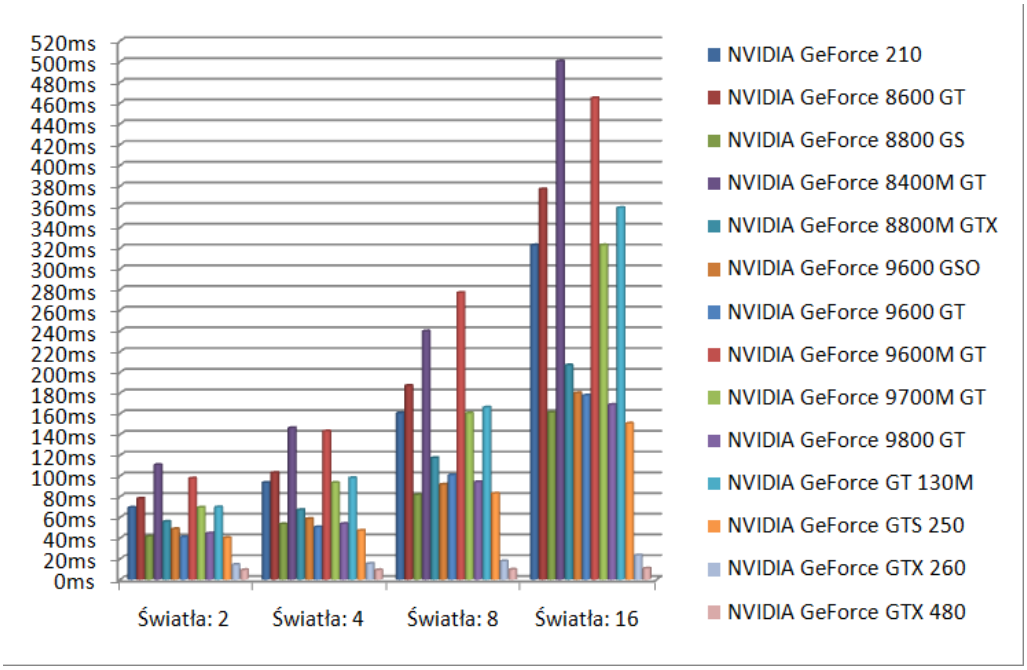
c) 8 źródeł światła d) 16 źródeł światła

Rysunek 6.10: Obrazy wygenerowane w teście o różnej liczbie świateł

WYKRESY:

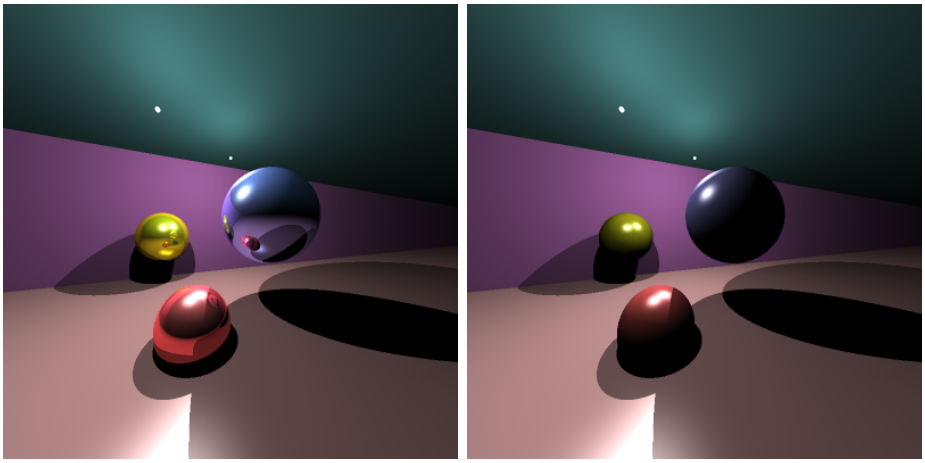


Rysunek 6.11: Wykresy CPU dla sceny złożonej z różnej ilości źródeł światła



Rysunek 6.12: Wykresy GPU dla sceny złożonej z różnej ilości źródeł światła

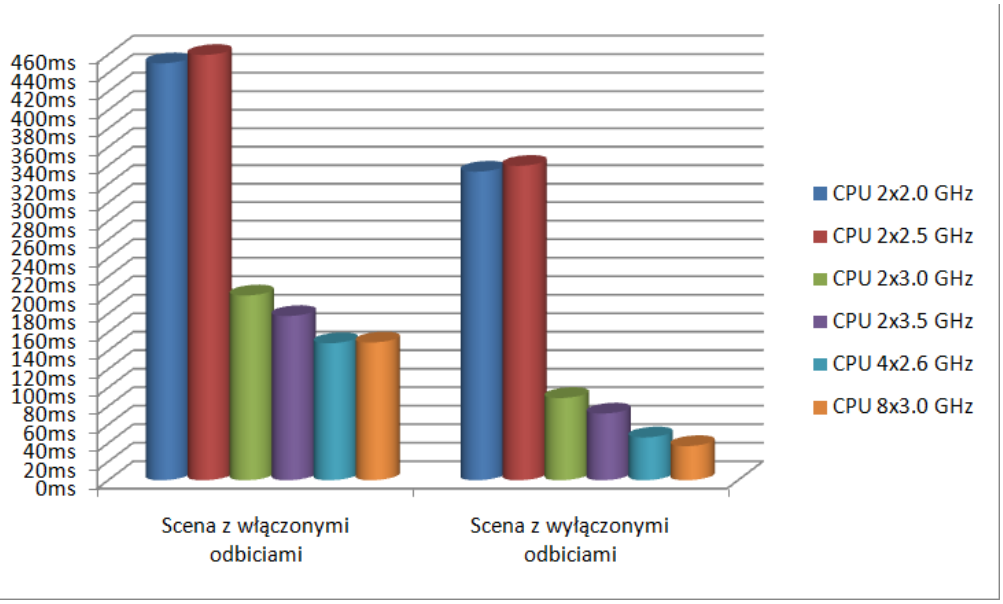
6.5 Test5 - Scena tesująca odbicia promieni od obiektów



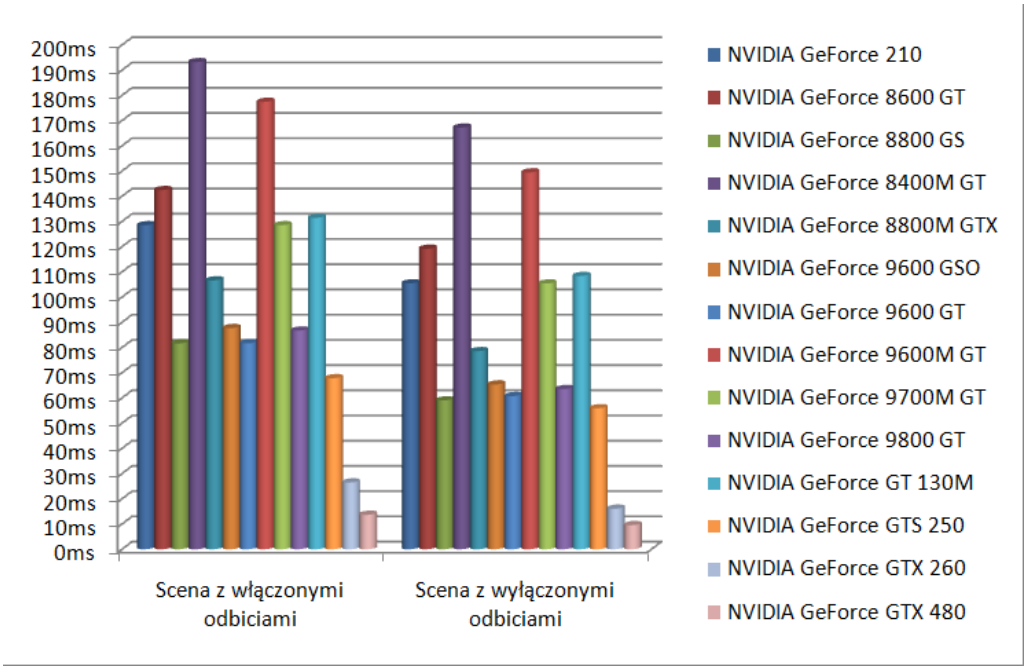
a) włączone odbicia promieni b) wyłączone odbicia promieni

Rysunek 6.13: Obrazy wygenerowane w teście odbić promieni świetlnych

WYKRESY:

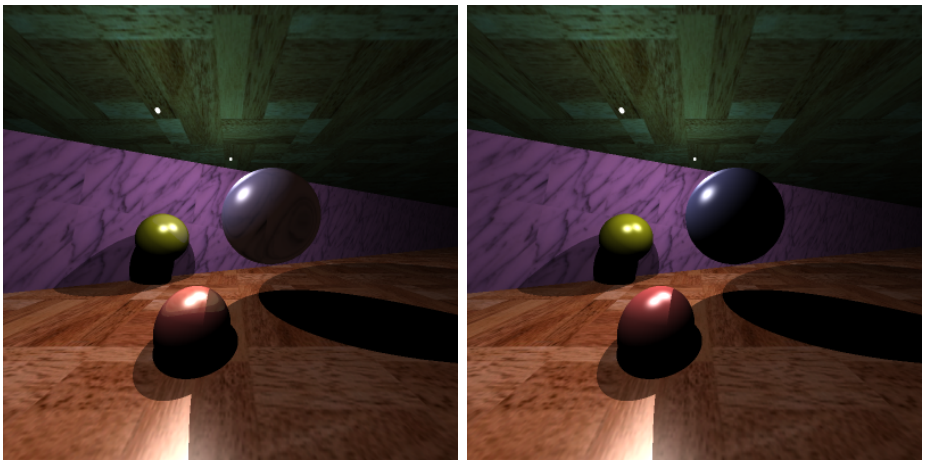


Rysunek 6.14: Wykresy CPU dla sceny z włączonymi odbiciami promieni od obiektów



Rysunek 6.15: Wykresy GPU dla sceny z włączonymi odbiciami promieni od obiektów

6.6 Test6 - Scena tesująca załamania promieni w obiektach

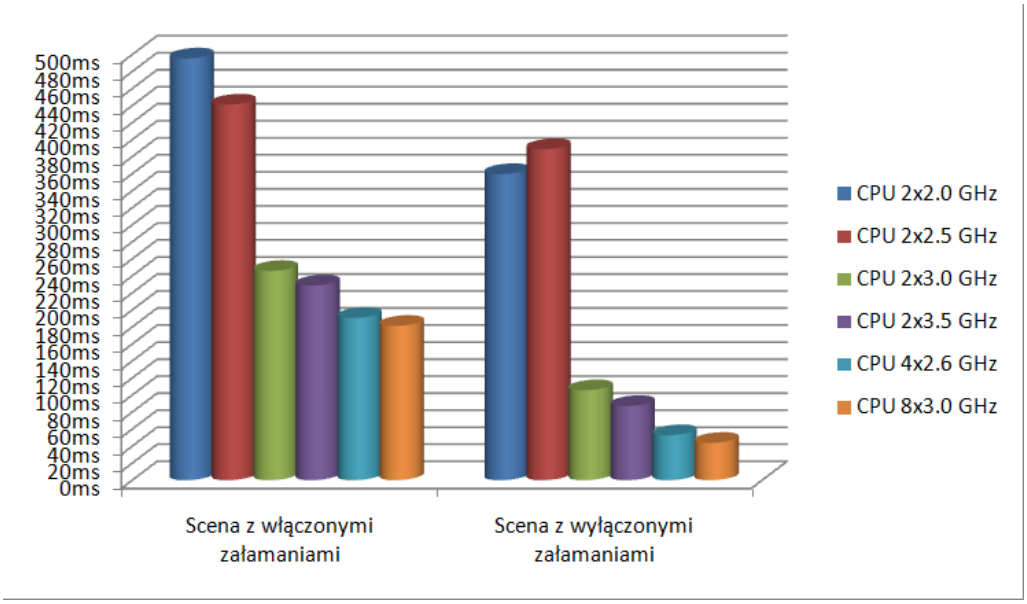


a) włączone załamania promieni

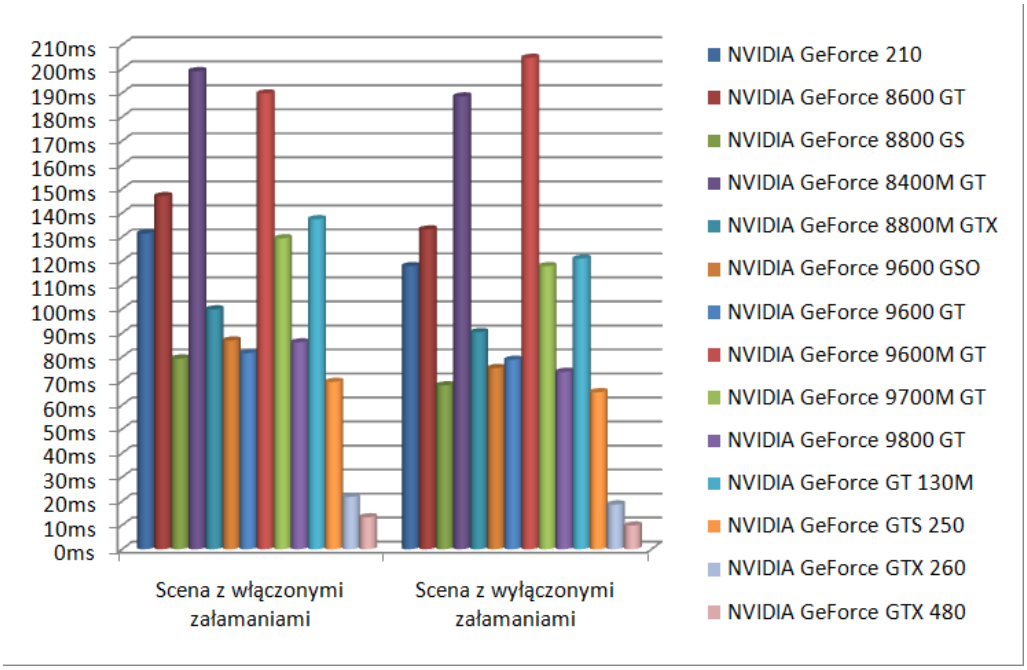
b) wyłączone załamania promieni

Rysunek 6.16: Obrazy wygenerowane w teście załamań promieni świetlnych

WYKRESY:

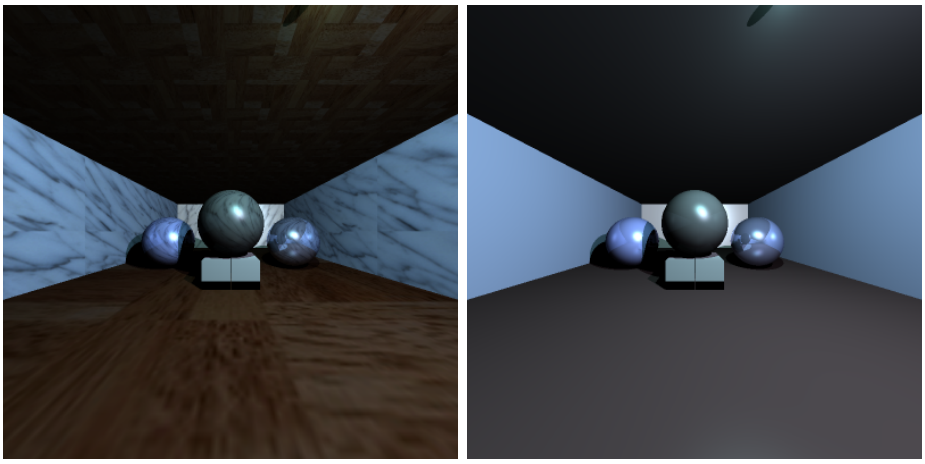


Rysunek 6.17: Wykresy CPU dla sceny z włączonymi załamaniami promieni w obiektach



Rysunek 6.18: Wykresy GPU dla sceny z włączonymi załamaniami promieni w obiektach

6.7 Test7 - Scena tesująca teksturowanie obiektów

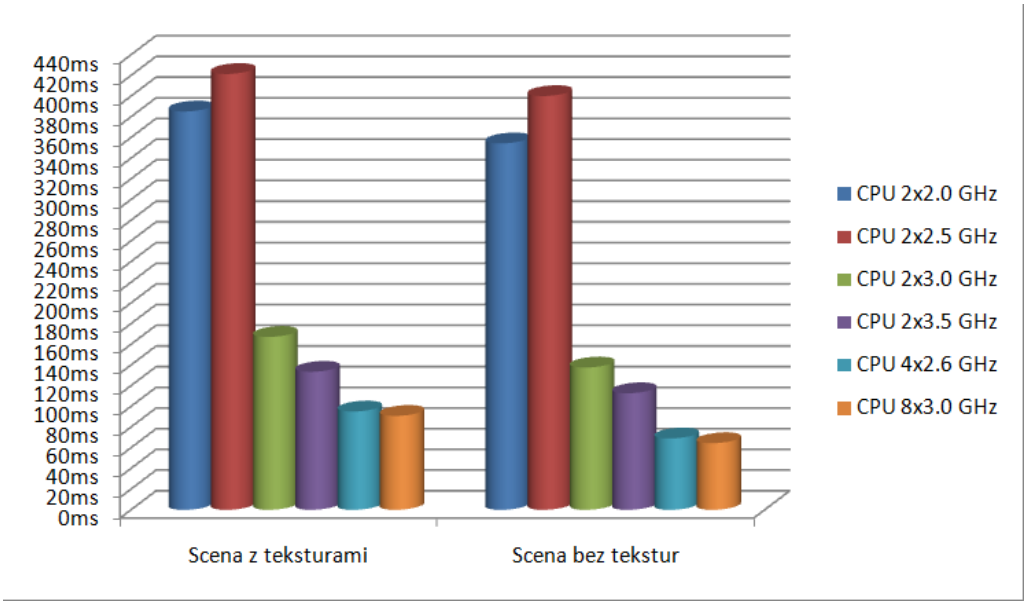


a) włączone teksturowanie obiektów

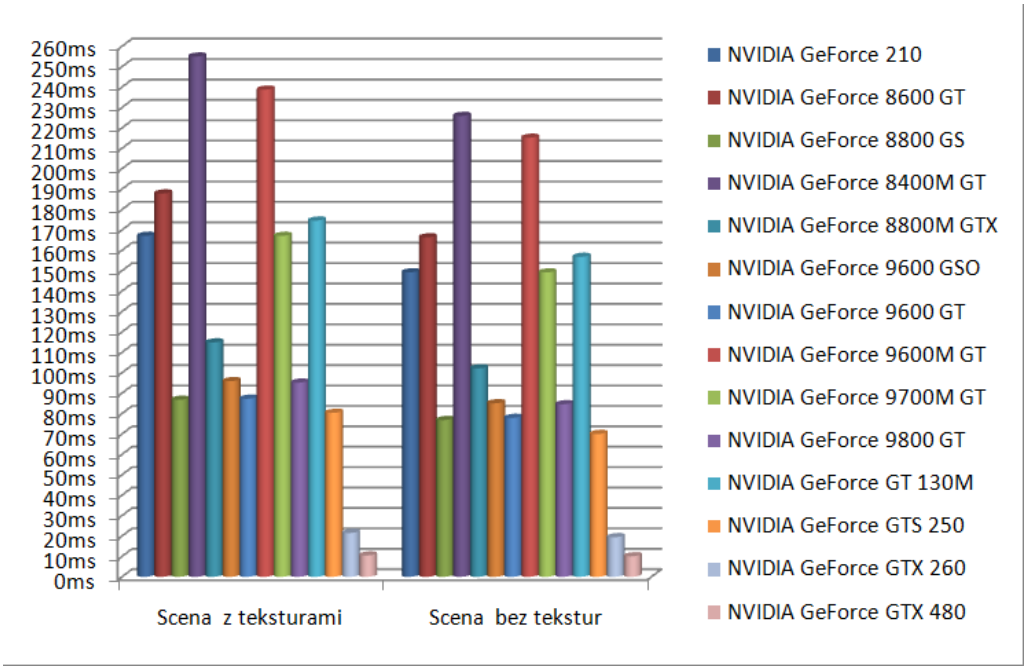
b) wyłączone teksturowanie obiektów

Rysunek 6.19: Obrazy wygenerowane w teście teksturowania obiektów

WYKRESY:



Rysunek 6.20: Wykresy CPU dla sceny z teksturami na obiektach



Rysunek 6.21: Wykresy GPU dla sceny bez tekstur na obiektach

6.8 Test8 - Różna rozdzielczość generowanych scen

W teście zbadana zostanie zależność szybkości generowania sceny w zależności od rozdzielczości wynikowego obrazu. Testowany jest tutaj spadek wydajności od liczby generowanych promieni. Przetestowane zostały dwa warianty scen:

- Rozdzielczość obrazu 400 na 400 pikseli.
- Rozdzielczość obrazu 800 na 600 pikseli.

OBRAZKI
WYKRESY

6.9 Test9 - Rożny stopień dokładności generowanych scen

W tym teście zbadana została szybkość wygenerowanego obrazu od jakości tego obrazu. Stosowany jest tu tak zwany super-sampling(DODAC DO UZYWANYCH SLOW). Jest to wzmożone próbkowanie promieni na pojedyńczy pixel ekranu. Przetestowane zostały dwa warianty scen:

- Scena ze śledzeniem pojedyńczego promienia na pixel.
- Scena ze śledzeniem czterech promieni na pojedyńczy pixel.

OBRAZKI
WYKRESY
Jak widać na załączonych wykresach, zauważalny jest spadek w szybkości generowanych scen wraz ze wzrostem śledzonych promieni na pojedyńczy pixel. Im mniejszy współczynnik super-samplingu tym szybciej generowana jest scena. Jednak jej jakość nie jest najlepsza. Pojawia się tu zjawisko tzw. aliasingu(DODAC I WYJASNIC NA GORZE w URZYWANYCH OKRESLENIACH).

6.10 Podsumowanie testów

Rozdział 7

Podsumowanie i wnioski

Podsumuje jasna dla wszystkich rzecz, że obliczenia na kartach graficznych przewyższają wydajnością obliczenia na zwykłych wielordzeniowych procesorach komputerowych. Dodatkowo opisze moje przemyślenia i wnioski które objawiły mi się podczas pisania uniwersalnej aplikacji wstecznego raytracingu. Na koniec opowiem trochę o tym co udało się zrobić, osiągnąć a czego nie i w jaki sposób chciałbym dalej rozwijać prace nad raytracingiem.

7.1 Napotkane problemy

Pracując nad niniejszą pracą, oraz mając do czynienia a znową technoogią NVIDIA CUDA napotkałem kilka problemów. Niemniej jednak 3 z nich były głównymi problemami. Dwa natury softwarowej oraz jeden natury hardwarowej.

- Technologia NVIDIA CUDA nie wspiera rekurencji. W tym wypadku implementacja odbić i załamań światła nie była możliwa w standardowy sposób. Poradziłem sobie tworząc sztuczną rekurencje w pętli używając własnorecznie stworzonego stosu.
- Testowanie(debugowanie) algorytmów pod technologię NVIDIA CUDA jest możliwe tylko i wyłącznie jeśli posiada się sprzęt z dwoma kartami graficznymi wspierającymi właśnie tą technologię. Nie posiadałem takiego sprzętu wieć moje sprawdzanie poprawniści algorytmów odbywało się metodą prób i błędów oraz porównywaniem wynikowych obrazów z wersją raytracera działającą na procesorze CPU.
- Ostanim ważnym problemem, któremu musiałem stawić czoło był problem natury sprzętowej. Sprzęt jaki miałem do dyspozycji posiadał

fabrycznie wadliwy układ graficzny (GeForce 8400M). Podczas implementacji raytracingu układ ten przepalił się około 4 razy i tyle samo razy był wymieniany w serwisie na nowy.

7.2 Perspektywy kontynuacji

Niniejszą pracę oraz badania nad metodą śledzenia promieni chciałbym dalej rozwijać w ramach kolejnej pracy dyplomowej. W planach do zrealizowania mam między innymi:

- Ulepszenie metody wstecznego raytracingu adaptując algorytmy photon-mappingu oraz path-tracingu
- Próba stworzenia własnego hybrydowego algorytmu śledzenia promieni w celu kolejnych przyśpieszeń generowania scen.

Bibliografia

[1] J. Bloch. Effective Java. Addison–Wesley, Stoughton, USA, second edition, 2008. http://www.thinkingblackberry.com/.

[2] E. Ciurana. Developing with Google App Engine. Apress, Berkeley, USA, 2008.

[3] Community. BlackBerry Developer Zone. http://na.blackberry.com/eng/developers/.

[4] S. Hartwig and M. Buchmann. Empty Seats Traveling. http://research.nokia.com/files/NRC-TR-2007-003.pdf.

[5] C. King. Advanced BlackBerry Development. Apress, Berkeley, USA, 2009.

[6] A. Rizk. Thinking BlackBerry. http://www.thinkingblackberry.com/.

[7] D. Sanderson. Programming Google App Engine. O'Reilly Media, Sebastopol, USA, 2009.

[8] Charles Severance. Using Google App Engine. O'Reilly Media, Sebastopol, USA, 2009.

[9] Wikipedia. Stosunek liczby samochodów do zaludnienia. http://pl.wikipedia.org/wiki/Stosunek˙liczby˙samochod%C3%B3w˙do˙zaludnienia.

Spis rysunków

Spis tabel