Politechnika Łódzka
Instytut Informatyki

Technologia OpenCL w symulacji dynamiki bryły sztywnej

Praca dyplomowa inżynierska

Wydziaż Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr inż. Dominik Szajerman
Dyplomant: Łukasz Kowalczyk
Nr albumu: 157889

Łódź, 2014

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.pl

Spis treści

Rozdział 1. Wprowadzenie . . . 3

1.1. Wstęp . . . 3
1.2. Cel . . . 4
1.3. Założenia . . . 4
1.4. Zakres pracy . . . 4

Rozdział 2. OpenCL . . . 5

2.1. Wprowadzenie . . . 5
2.2. Składniki środowiska OpenCL . . . 5
2.3. Architektura OpenCL . . . 6
2.4. Paralelizm danych . . . 6
2.5. Grupy robocze oraz zadania . . . 7
2.6. Zarządzanie pamięcią w OpenCL . . . 7
2.7. Urządzenia wykorzystane do testów . . . 8
2.7.1. NVIDIA GTX 760 . . . 8
2.7.2. Intel®Core™i5-3570K . . . 9

Rozdział 3. Wizualizacja symulacji z wykorzystaniem OpenGL . . . 10

3.1. Wstęp . . . 10
3.2. Wykorzystanie biblioteki OpenGL . . . 10

Rozdział 4. Seperating Axis Theorem . . . 13

4.1. Zasada działania . . . 13
4.2. Projekcja na płaszczyznę . . . 13
4.3. Algorytm SAT . . . 14
4.4. Iloczyn wektorowy . . . 15
4.5. Znalezienie MTV . . . 15
4.6. Problemy związane z używaniem Seperating Axis Theorem . . . 16

Rozdział 5. Wyznaczanie momentu kolizji oraz reakcji . . . 17

5.1. Sprawdzenie kolizji . . . 17

5.2. Kwaterniony 18

Rozdział 6. Wykorzystanie OpenCL w symulacji 20

6.1. Inicjalizacja OpenCL 20

6.1.1. Struktura obiektów 20

6.1.2. Stałe 20

6.1.3. Funkcje pomocnicze 20

6.1.4. Kernel 21

6.2. Uruchomienie kernela przez hosta 21

Rozdział 7. Implementacja aplikacji 23

7.1. Architektura 23

7.2. Struktury danych 24

7.3. Przesyłanie danych między CPU a GPU 25

Rozdział 8. Dynamika bryły sztywnej 26

8.1. Bryła sztywna 26

8.1.1. Ruch postępowy 26

8.1.2. Ruch obrotowy 26

8.2. Zasady dynamiki Newtona 27

8.2.1. I zasada dynamiki 27

8.2.2. II zasada dynamiki 27

Rozdział 9. Wyniki 29

Rozdział 10. Podsumowanie i wnioski 31

Bibliografia 32

Spis rysunków 33

Rozdział 1

Wprowadzenie

1.1. Wstęp

Początki obliczeń z wykorzystaniem akceleracji układów GPU związane są z przetwarzaniem potokowym oraz obliczeniami równoległymi. Jedne z pierwszych superkomputerów umożliwiały zdecydowanie zwiększoną wydajność dla zadań, które były podzielone na operacje powtarzalne. Pierwszy układ GPU został wprowadzony na rynek dzięki firmie Nvidia w 1999 roku [7]. Wtedy też rozpoczętko wykorzystywać te układy do obliczeń ogólnego przeznaczenia.
W roku 2001, wraz z wprowadzeniem programowalnego potoku renderingu oraz wsparcia dla obliczeń liczb zmiennoprzecinkowych, zaczęto wykorzystywać układy GPU do obliczeń niezwiązanych z grafiką. Pierwsze wersje shaderów były dość ograniczone (np. dla Vertex Shadera 1.1 możliwe było użycie jedynie 128 instrukcji). Kolejne wersje znacznie zwiększały swoje możliwości (512 instrukcji dla Shader Model 3.0 oraz do 64 000 instrukcji dla Shader Model w wersji 4.0) [8].
Zainteresowanie możliwościami obliczeniowymi kolejnych układów było duże, dlatego też w listopadzie 2006 roku Nvidia opracowała technologię CUDA. Pozwala ona na wykorzystanie mocy obliczeniowej procesorów GPU do rozwiązywania problemów numerycznych zdecydowanie wydajniej, niż wykonałby je procesor ogólnego zastosowania. Niestety tachnologia ta może być wykorzystana jedynie z kartami firmy Nvidia. W 2009 roku opracowany został przez firmę Apple Inc. otwarty framework OpenCL, który podobnie jak CUDA pozwala wykorzystać procesory graficzne do wykonania obliczeń, jednak w przeciwieństwie do technologii firmy Nvidia, OpenCL współpracuje z układami graficznymi wszystkich wiodących producentów a ponadto umożliwia uruchomienie zapisanych w nim kerneli na procesorach ogólnego przeznaczenia.
Wykorzystanie układów GPU znalazło zastosowanie w wielu dziedzinach, zaczynając od wsparcia naukowców przy tworzeniu nowych leków, poprzez obliczanie symulacji ruchu ciał niebieskich, przyspieszenia obliczeń związanych z kryptografią, aż po wykorzystanie w

grach komputerowych do obliczań zwiazanych z fizyką (symulacje zachowań cieczy, efekty cząsteczkowe czy też wsparcie w obliczaniu kolizji).

1.2. Cel

Celem pracy jest stworzenie symulacji bryły sztywnej wykorzystując otwarty framework OpenCL do akceleracji obliczeń, a zatem do przemieszczania obiektów, znajdowania kolizji z innymi obiektami oraz reagowania na zderzenia. Praca zakłada również implementację komunikacji części bazowej programu z układem GPU oraz implementację uproszczonego wyświetlenia symulowanych brył przy użyciu technologii OpenGL.

1.3. Założenia

Praca zakłada wykorzystanie technologii OpenCL w wersji 1.2 do wykonania obliczeń fizycznych. Użycie technologii OpenGL w wersji 3.1 do wyświetlania efektów działania aplikacji zostanie wsparte dodatkową biblioteką glfw w wersji 3 [5]. Część bazowa programu umożliwiająca wykorzystanie OpenCL oraz OpenGL została napisana w języku C++. Do tworzenia samego projektu posłużyło środowisko IDE QTCreator 1.8.4 a do zarządzenia procesem kompilacji narzędzie Cmake w wersji 2.8.
Symulacja brył odbywa się w przestrzeni posiadającej grawitację ziemską przybliżoną do wartości 9.81 $\frac{m}{s_2}$ oraz podłoże posiadające wagę zbliżoną do nieskończoności (m = ∞), na które nie działa siła grawitacji.

1.4. Zakres pracy

Pierwszy rozdział pracy dotoczy technologii OpenCL. Zostanie w nim omówiona architektura oraz ogólna zasada działania. Przedstawiony zostanie również sprzęt użyty do tworzonego oprogramowania.
Kolejny rozdział dotyczy technologii OpenGL. Przedstawione zostaną wstępne informacje o tej technologii, przedstawiony rozwój wraz z nowymi jej wersjami oraz omówione wykorzystanie technologii w projekcie.
Następny rozdział zawiera omówienie algorytmu sprawdzającego kolizję. Omówiony zostanie algorytm sprawdzania zderzeń między obiektami symulacji, zastosowane rozszerzenie podstawowej wersji algorytmu oraz problemy, które towarzyszą wykorzystaniu tej metody testowania kolizji.
Ostatni rozdział przedstawia konkretne zastosowanie technologii OpenCL przy obliczeniach związanych z symulacją.

Rozdział 2

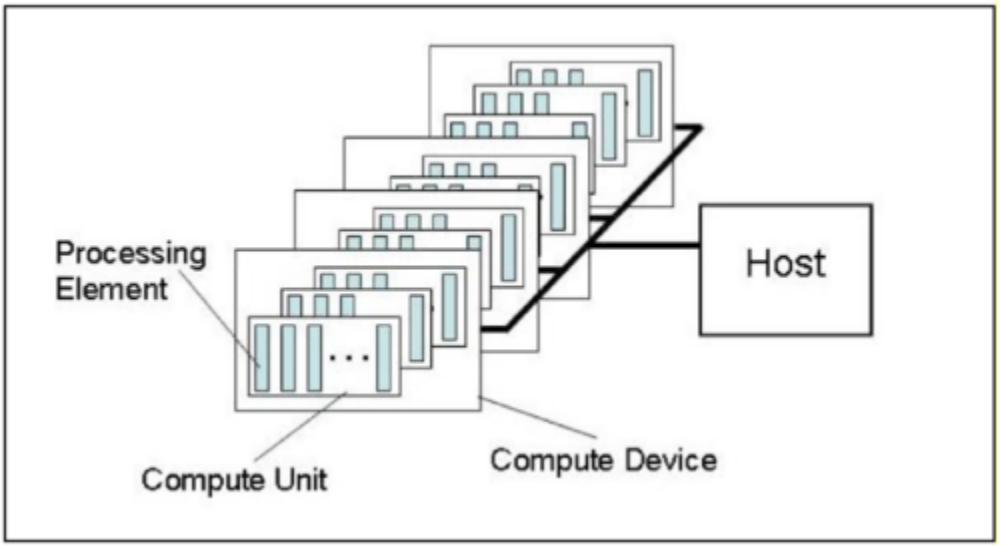
OpenCL

2.1. Wprowadzenie

OpenCL (Open Computing Language) jest standardem tworzenia oprogramowania stworzonym przez Khronos Group, który powstał w celu wsparcia w tworzeniu oprogramowania z wykorzystaniem mocy zarówno współczesnych kart graficznych (GPU) jak i procesorów (CPU). Jedną z głównych zalet OpenCL jest otwartość standardu, co umożliwia wykorzystanie technologii opierając się o sprzęt wielu producentów (np. Nvidia, AMD, Intel).

2.2. Składniki środowiska OpenCL

Idea funkcjonowania platformy OpenCL została przedstawiona na rysunku 2.1.
Środowisko składa się z hosta – „gospodarza", nakazujacego wykonanie konkretnego ciagu



Rysunek 2.1. Model platformy OpenCL.[1]

[1] Źródło: http://pclab.pl/zdjecia/artykuly/miekrzy/catalyst8_12/host.png

instrukcji urządzeniom obliczeniowym (Compute Device). Każde urządzenie posiada własny zestaw jednostek przetwarzających, na których istrukcje są wykonywane.

2.3. Architektura OpenCL

Architektura OpenCL składa się z trzech zasadniczych części: specyfikacji języka, API platformy i API czasu wykonania. Specyfikacja języka OpenCL definiuje składnię programów i kerneli[2], które oparte są na zmodyfikowanym standardzie ISO C99, zawierającym część dodanych, zmienionych oraz usuniętych słów kluczowych a takze rozszerzeniu standardu np. poprzez dodanie wsparcie dla przetwarzania równoległego.
API platformy umożliwia hostowi uruchamianie stworzonych kerneli na urządzeniach obliczeniowych. Realizuje ono również koncepcję kontekstu. Kontekst jest kontenerem grupującym urządzenie z przeznaczoną dla niego zawartością pamięci oraz kolejkami zadań. Przy pomocy API platformy realizowane jest przekazywanie danych między hostem a urządzeniem obliczeniowym.
API czasu wykonania wykorzystuje dostarczone przez platformę konteksty do kontrolowania kompatybilnych urządzeń. Przy jego pomocy odbywa się zarządzanie kolejkami zadań, obiektami pamięci i kernelami. Za pośrednictwem API czasu wykonania odbywa się również kolejkowanie kerneli na konkretnych urządzeniach.

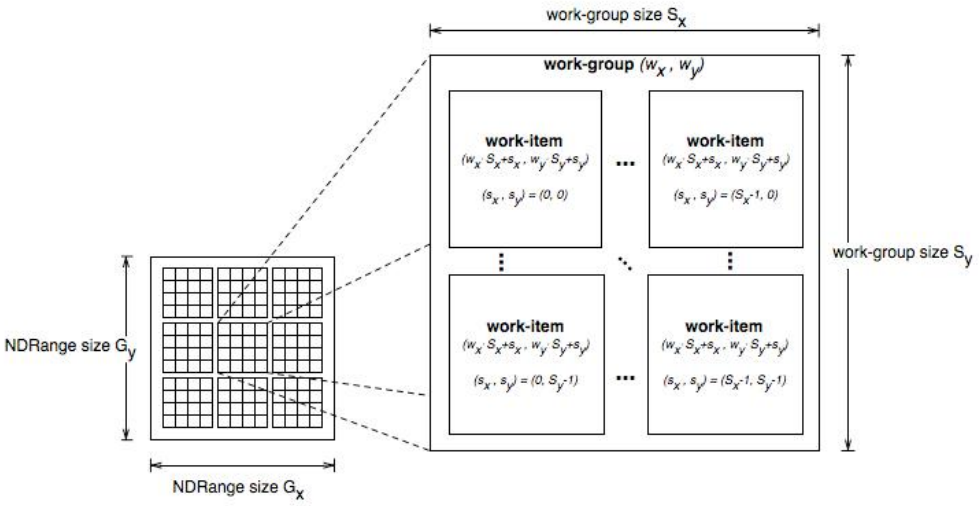
2.4. Paralelizm danych

OpenCL, podobnie jak karty graficzne, które są główną grupą urządzeń wykorzystujących tę technologię, działa na zasadzie paralelizmu danych. W przeciwieństwie do paralelizmu zadań, który zakłada wykonanie różnych ciągów instrukcji w tym samym czasie, OpenCL zakłada wielokrotne wykonanie identycznego zadania wykorzystując jednak inne zestawy danych.
Paralelizm danych jest tutaj zrealizowany za pośrednictwem programów, które składają się z jednego lub więcej kerneli. Program poza samym kernelem może też zawierać dodatkowo dane stałe oraz funkcje, z których korzystają kernele. Podczas wykonania kernela przedstawione w nim zadanie wykonywane jest równolegle na elementach przetwarzających urządzenia obliczeniowego, tzw. work-itemach.

[2] funkcja wykonywana na urządzeniu, korzystajaca korzystajaca z jego zasobów

2.5. Grupy robocze oraz zadania

Jak przedstawiono na rysunku 2.2, ciągi instrukcji powiązane są ze sobą w grupy robocze. Obszar NDRange jest metodą służącą do organizacji pamięci i zaplanowania zadań



Rysunek 2.2. OpenCL NDRange [3].

w architekturze OpenCL. NDRange to jedno-, dwu- lub trójwymiarowa przestrzeń. Jest ona podzielona na grupy robocze. Każda grupa posiada swój indeks, po którym można określić jej pozycję w przestrzeni. Każdy ciąg instrukcji, stanowiący odrębne zadanie kernela, dysponuje unikalnym globalnym numerem identyfikacyjnym a także unikalnym lokalnym numerem identyfikacyjnym wewnątrz każdej z grup.
Tak podzielona przestrzeń wynika z architektury OpenCL. Dzięki identyfikatorom można skorzystać z bariery lokalnej, czyli wstrzymać wykonywanie kolejnych operacji dla zadań w ramach danej grupy roboczej do momentu zsynchronizowania operacji na wszytkich zadaniach grupy.

2.6. Zarządzanie pamięcią w OpenCL

W OpenCL używane są wymienione niżej cztery przestrzenie pamięci.

1. Pamięć globalna - wszystkie wątki w grupach roboczych posiadają pełen dostęp (zapis i odczyt) do tej pamięci. Jest największym dostępnym obszarem pamięci jednak oferuje najwolniejszy dostęp
2. Pamięć prywatna - dostęp do pamięci jedynie dla zadania, które zaalokowało pamięć podczas wykonywania danej instancji kernela.

[3] Źródło: http://www-igm.univ-mlv.fr/ dr/XPOSE2008/CUDA_GPGPU/opencl_threads.jpg

3. Pamięć lokalna - wszystkie wątki w danej grupie roboczej posiadają pełen dostęp do pamięci, jednak jest ona nieosiągalna dla wątków z innych grup. Możliwe jest zmapowanie obszaru pamięci globalnej na pamięć lokalną w celu uzyskania przyspieszonego czasu dosępu, jednak nie jest to wspierane przez wszystkie urządzenia.
4. Pamięć stała. W tej kategorii pamięci przechowywane są stałe wartości podczas wykonania całej operacji kernela. Jest ona inicjalizowana przez hosta.

2.7. Urządzenia wykorzystane do testów

2.7.1. NVIDIA GTX 760

Do testów została wykorzystana karta NVIDIA GeForce GTX 760, przedstawiona na rysunku 2.3.

Karta została wyposażona w 1152 rdzeni CUDA (OpenCL uznaje je jako elementy

Rysunek 2.3. NVIDIA GeForce GTX 760 [4].

przetwarzające) zebranych w 32 jednostkach obliczeniowych. Każdy z rdzeni jest taktowany zegarem o częstotliwości 1085 MHz. Maksymalna wydajność karty podczas obliczeń podwójnej precyzji wynosi 94 GFLOPsów (tysięcy operacji zmiennoprzecinkowych na sekundę), zaś podczas obliczeń w pojedynczej precyzji 2258 GFLOPSów. Karta została wyposażona w 2GB pamięci RAM typu GDDR5, taktowanej efektywną prędkością 1502 MHz. Wspiera standard OpenCL w wersji 1.2 oraz 2.0. [4]

[4] Źródło: http://www.guru3d.com/articles-pages/geforce-gtx-760-gigabyte-windforce-review,1.html

2.7.2. Intel®Core™i5-3570K

Do testów został wykorzystany również procesor Intel®Core™i5 model 3570K przedstawiony na rysunku 2.4.

Rysunek 2.4. Intel®Core™i5-3570K [5].

Posiada cztery rdzenie, które mogą na raz przetwarzać maksymalnie do czterech wątków. Procesor pracuje w zakresie częstotliwości 3.4 - 3.8 GHz. Jest to urządzenie zrealizowane w technologii 22nm o archtekturze noszącej nazwę Ivy Bridge. Posiada on 128 KB pamięci L1 Data Cache oraz 128 KB L1 Instruction Cache. Urządzenie to wspiera standard OpenCL w wersji 1.2[6].

[5] Źródło: http://www.chip.pl/images/testy/podzespoly-pc/procesory/intel-core-i5-3570k/56068.jpg/image_previe

[6] Źródło: http://ark.intel.com/pl/products/65520

Rozdział 3

Wizualizacja symulacji z wykorzystaniem OpenGL

3.1. Wstęp

OpenGL jest biblioteką graficzną przeznaczoną do tworzenia trójwymiarowej grafiki. Pierwsza wersja została opracowana w 1992 przez firmę Silicon Graphics Inc. na potrzeby stacji graficznych IRIS. Technologia zystała jednak popularnośc dopiero po zaimplementowaniu jej do współpracy z systemem z rodziny Microsoft Windows. Następnie jej rozwojem zajęła się organizacja ARB, w której skład wchodzili przedstawiciele firm 3DLabs, Apple, ATI, Dell, IBM, Intel, NVIDIA, SGI oraz Sun [6].
Sukces technologii polega na jej implementacji dla wszystkich wiodących systemów operacyjnych, niezależności od platformy sprzętowej (wraz z wsparciem sterowników dla większości kart graficznych) oraz ogólnie dostępnej specyfikacji. Aktualna wersja OpenGL to 4.4 (wydana 22 lipca 2013 roku). Technologia jest aktywnie wspierana przez Khronos Group.

3.2. Wykorzystanie biblioteki OpenGL

Jest wiele sposobów zrealizowania wizualizacji obrazów trójwymiarowych wykorzystując bibliotekę OpenGL. Najprostsze rozwiazania generują cały obraz w jednej funkcji oraz wykorzystują metody, które powstały jeszcze w wersji 1.2 biblioteki. Na potrzeby pracy zostało jednak wykorzytane rozwiązanie wprowadzone w openGL w wersji 3. Zamiast rysować każdy trójkątosobno w sekwencji glBegin-glEnd, trzecia odsłona OpenGL umożliwia przyspieszenie wykonania renderingu dzięki rysowaniu całej tablicy wierzchołków.
Pierwszą czynnością jest stworzenie bufora danych:

```
glGenBuffers(1, &vertexbuffer);
```

a następnie należy dowiązać bufor jako bieżący:

```
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
```

Możliwe są cztery niżej opisane typy dowiązania:

1. `GL_ARRAY_BUFFER` - buforowy obiekt tablic wierzchołków. Dane przechowywane w obiektach buforowych tablic wierzchołków są zgodne z formatem używanym w tablicach wierzchołków. Umożliwia to bezpośrednie korzystanie z tablic wierzchołków przy użyciu funkcji: glArrayElement, glDrawArrays, glMultiDrawArrays, glDrawElements, glDrawRangeElements i glMultiDrawElements. Jedyna różnica polega na sposobie dostarczenia danych tablic wierzchołków do funkcji: glVertexPointer, glNormalPointer, glColorPointer, glSecondaryColorPointer, glIndexPointer, glEdgeFlagPointer, glFogCoordPointer i glTexCoordPointer. Zamiast wskaźnika na dane tablicy (parametr pointer powyższych funkcji) należy podać położenie początku danych w bieżącym obiekcie buforowym tablicy wierzchołków. W praktyce jeden obiekt buforowy tablic wierzchołków może być zatem strukturą służącą do przechowywania danych wielu tablic wierzchołków.
2. `GL_ELEMENT_ARRAY_BUFFER` - buforowy obiekt indeksowych tablic wierzchołków. Obiekty buforowe indeksowych tablic wierzchołków są bezpośrednio przystosowane do współpracy z funkcjami: glDrawElements, glDrawRangeElements i glMultiDrawElements obsługującymi indeksowe tablice wierzchołków. Jedyna różnica polega na sposobie dostarczenia danych tablicy indeksów. Zamiast wskaźnika na dane tablicy z indeksami (parametr indices dwóch pierwszych funkcji) podaje się położenie początku danych w bieżącym obiekcie buforowym indeksowej tablicy wierzchołków. W przypadku funkcji glMultiDrawElements parametr indices zawiera adresy położenia danych kolejnych tablic indeksów w bieżącym obiekcie buforowym.
3. `GL_PIXEL_UNPACK_BUFFER` - buforowy obiekt odczytu danych pikseli. Obiekty buforowe odczytu danych pikseli można wykorzystać jako źródło danych pikseli dla następujących funkcji biblioteki OpenGL: glBitmap, glColorSubTable, glColorTable, glCompressedTexImage1D, glCompressedTexImage2D, glCompressedTexImage3D, glCompressedTexSubImage1D, glCompressedTexSubImage2D, glCompressedTexSubImage3D, glConvolutionFilter1D, glConvolutionFilter2D, glDrawPixels, glPixelMapfv, glPixelMapuiv, glPixelMapusv, glPolygonStipple, glSeparableFilter2D, glTexImage1D, glTexImage2D, glTexImage3D, glTexSubImage1D, glTexSubImage2D i glTexSubImage3D. W parametrach powyższych funkcji wskazujących źródło danych pikseli zamiast adresu tablicy z danymi wskazujemy położenie danych w obiekcie buforowym.
4. `GL_PIXEL_PACK_BUFFER` - buforowy obiekt zapisu danych pikseli. Obiekty buforowe zapisu danych pikseli można wykorzystać jako bufor danych pikseli dla następujących funkcji biblioteki OpenGL: glGetCompressedTexImage, glGetConvolutionFilter, glGetHistogram, glGetMinmax, glGetPixelMapfv, glGetPixelMapuiv, glGetPixelMapusv, glGetPolygonStipple, glGetSeparableFilter, glGetTexImage i glReadPixels. W parametrach powyższych funkcji wskazujących tablicę na dane pikseli zamiast adresu tablicy na dane wskazujemy położenie danych w obiekcie buforowym.

Następnie należy uzupełnić bufor:
`glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL`
Ostatni parametr informuje OpenGL, że dane z buforu będą pobrane raz lub będą pobierane sporadycznie, jednak będą służyły wielokrotnie do zapisu do obiektu OpenGL. Parametr ten jest jedynie sugestią pomagajacą w optymalizacji. Pozostałymi możliwymi wartościami parametru są:

— `GL_STREAM_DRAW` - jednokrotne lub sporadyczne pobieranie danych i przeważnie wykorzystanie ich do zapisu do obiektu OpenGL,
— `GL_STREAM_READ` - jednokrotne lub sporadyczne pobieranie danych i przeważnie wykorzystanie ich do odczytu z obiektu OpenGL,
— `GL_STREAM_COPY` - jednokrotne lub sporadyczne pobieranie danych i przeważnie wykorzystanie ich zarówno do odczytu z obiektu OpenGL jak i do zapisu
— `GL_STATIC_READ` - jednokrotne lub sporadyczne pobieranie danych i wielokrotne ich wykorzystanie do odczytu z obiektu OpenGL,
— `GL_STATIC_COPY` - jednokrotne lub sporadyczne pobieranie danych i wielokrotne wykorzystanie ich zarówno do odczytu z obiektu OpenGL jak i do zapisu,
— `GL_DYNAMIC_DRAW` - wielokrotne pobieranie danych i wielokrotne ich wykorzystanie do zapisu do obiektu OpenGL,
— `GL_DYNAMIC_READ` - wielokrotne pobieranie danych i wielokrotne ich wykorzystanie do odczytu z obiektu OpenGL,
— `GL_DYNAMIC_COPY` - wielokrotne pobieranie danych i wielokrotne wykorzystanie ich zarówno do odczytu jak i zapisu obiektu OpenGL.

Po uzupełnieniu buforu możliwe jest przesłanie informacji o sposobie rysowania samych elementów, korzystając z metody `glDrawArrays(GLenum mode, GLint first, GLsizei count)`. Pierwszy atrybut informuje o rodzaju rysowanych prymitywów. Przyjmuje on takie same wartości jak funkcja glBegin, czyli `GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS oraz GL_PO`
Kolejny parametr określa pierszy element z tablicy wierzchołków, który ma być wykorzystany do renderingu. Ostatni atrybut przekazuje informację o początkowym indeksie wierzchołka a trzeci o liczbie renderowanych prymitywów. W przypadku stworzonej aplikacji dla typu `Box` użyty został typ `GL_QUADS` a w trzecim argumencie funkcji wpisana została wartość 24, ponieważ dla każdej z ścian sześcianu musimy podać cztery wierzchołki, aby poprawnie wyrenderować jedną ścianę bryły.

Rozdział 4

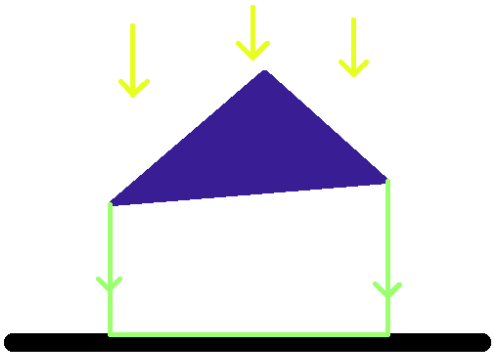
Seperating Axis Theorem

4.1. Zasada działania

Separating Axis Theorem, w skrócie SAT, jest metodą pozwalającą wykryć, czy dane dwa wielokąty wypukłe przecinają się [9]. Odpowiednie rozwinięcie algorytmu umożliwia również znalazienie najmniejszego wektora przeniknięcia obiektów. SAT jest algorytmem genetycznym, który pozwala szybko zweryfikować, czy nastąpiła kolizja między obiektami. Niweluje to konieczność stosowania złożonych obliczeniowo jak i czasowo algorytmów, umożliwiając sprawdzenie kolizji dla duzej liczby obiektów bez spadku wydajności aplikacji. Ogólna zasada działania Separating Axis Theorem polega na sprawdzeniu, czy istnieje oś dzieląca sprawdzane obiekty bez ich przecinania. Jeśli nie jest możliwe znalezienie takiej osi, oznacza to, iż zaistniała kolizja tych obiektów.

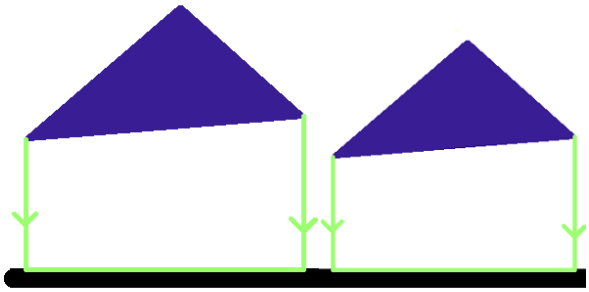
4.2. Projekcja na płaszczyznę

Ważnym zagadnieniem podczas omawiania działania algorytmu SAT są projekcje. Ideę projekcji można przedstawić jako cień rzucany przez obiekt na płaszczyznę, gdy światło skierowane jest prostopadle do płaszczyzny a obiekt znajduje się pomiędzy źródłem światła a płaszczyzną.



Rysunek 4.1. Projekcja na płaszczyznę (opracowanie własne).

Omówiona w wstępie rozdziału idea algorytmu w praktyce wykorzystuje projekcje i przedstawia się następująco: jeśli dwa wielokąty wypukłe nie kolidują ze sobą to istnieje conajmniej jedna oś, dla której projekcje obiektów nie zachodzą na siebie.



Rysunek 4.2. Projekcja na płaszczyznę (opracowanie własne).

4.3. Algorytm SAT

Algorytm wykonuje sprawdzenie nałożenia projekcji obiektów na osie dopóki nie trafi na pierwszą oś, dla której projekcje nie zachodzą na siebie. Jeśli nie znajdzie takiej osi, zwraca informację o przecinaniu się obiektów. W przeciwnym wypadku dostajemy informacje, że obiekty nie kolidują.

Aby stwierdzić czy projekcje na danej osi się nakładają, należy sprawdzić czy początkowy punkt pierwszego obiektu znajduje się blizej początku osi niż początkowy punkt drugiego sprawdzanego obiektu. Jeśli tak, to następnie wykonujemy sprawdzenie, czy najdalej wysunięty w prawo punkt pierwszego obiektu jest położony dalej niż poczatkowy punkt drugiego obiektu. Jeśli tak jest, oznacza to, iż dla tej osi zachodzi kolizja obiektów. Gdy

początek pierwszego obiektu jest położony na osi dalej niż drugi, wykonujemy analogiczne operacje, porównujac najdalej wysunięty punkt drugiej bryły z początkowym punktem projekcji pierwszego.
Teoretycznie istnieje nieskończenie wiele osi, które możemy sprawdzić. W praktyce wystarczy sprawdzić jedynie osie wzdłuż normalnych płaszczyzn, z których złożone są sprawdzane bryły oraz wzdłuż osi stworzonych z iloczynów wektorowych brzegów obiektów.

4.4. Iloczyn wektorowy

Wynikiem iloczynu wektorowego nowy wektor o kierunku prostopadłym do płaszczyzny utworzonej przez mnożone wektory o długości równej iloczynowi długości wektorów oraz sinusa kąta między nimi [10]. Zwrot utworzonego wektora wyznaczany jest regułą prawej ręki lub regułą śruby prawoskrętnej.Kolejność mnożenia odgrywa tu niemałe znaczenie,ponieważ wynik iloczynu wektorowego AxB nie równa się BxA. Wartość iloczynu wektorowego zdefiniowana jest następujacym wzorem:

$$\|\vec{a} \times \vec{b}\| = \|\vec{a}\| \bullet |\vec{b}\| \bullet \sin\gamma \tag{4.1}$$

Produkt iloczynu skalarnego wektorów a i b ma postać:

$$a \times b = \vec{a} \times \vec{b} == (a\_xb\_z - a\_zb\_y)\widehat{x} + (a\_zb\_x - a\_xb\_z)\widehat{y} + (a\_xb\_y - a\_yb\_y)\widehat{z} \tag{4.2}$$

co równoważne jest z zapisem:

$$\vec{a} \times \vec{b} = [a\_xb\_z - a\_zb\_y, a\_zb\_x - a\_xb\_z, a\_xb\_y - a\_yb\_y] \tag{4.3}$$

4.5. Znalezienie MTV

MTV (Minimum Translation Vector) jest to wektor opisujący minimalną część wspólną nachodzących obiektów, który jest używany do odepchnięcia kolidujących obiektów do stanu, w którym nie będą już się przecinać [9]. W pierwotnej wersji algorytmu SAT otrzymujemy jedynie informację o tym czy kolizja między bryłami zachodzi czy też nie. Jednakże wystarczy zapisywać wielkość nałożenia obiektów podczas sprawdzania ich projekcji i wybrać najmniejsze wartości dla każdej osi, aby otrzymać szukany MTV.

4.6. Problemy związane z używaniem Seperating Axis Theorem

Istnieje kilka zauważalnych wad zastosowania SAT.
Pierwszą jest możliwość wykorzystania jedynie brył wypukłych. Jedną metodą na zniwelowanie tej wady jest podzielenie złożonego obiektu na kilka brył tak, aby każda z brył składających się na obiekt była wypukła.
Drugą wadą jest sytuacja, w której jeden obiekt jest zawarty w drugim. W takim wypadku obliczony wektor MTV może zawierać błędne wartości oraz zwrot. Rozwiązaniem będzie dodanie do sprawdzania każdej osi testu na zawieranie.
Dodatkowo należy pamiętać, że testowane płaszczyzny mogą się powtarzać, np. dla sześcianu wystarczy przestestować trzy płaszczyzny zamiast sześciu, ponieważ dla każdej z równoległych płaszczyzn potrzeba tylko jednego sprawdzenia. Przyspiesza to czas sprawdzania kolizji i znacząco zmniejsza liczbę potrzebnych iteracji.

Rozdział 5

Wyznaczanie momentu kolizji oraz reakcji

5.1. Sprawdzenie kolizji

Sam proces symulacji dla każdej ramy czasowej jest podzielony na dwie części. Pierwsza część wykorzystuje zaktualizowane z poprzedniego cyklu dane dla obiektów, tzn. aktualnie działające na obiekt wektory sił i aktualizuje położenie oraz orientację brył. W stworzonej symulacji funkcja aktualizująca przekazuje oprócz danych o bryłach również czas, jaki upłynął od ostatniej aktualizacji (delta time). Dzięki temu atrybutowi możliwe jest prawidłowe obliczenie przemieszczenia obiektu od odstatniej aktualizacji niezależnie od mocy obliczeniowej urządzenia, na którym symulacja jest uruchamiana. Kolejnym etapem jest sprawdzenie obiektów pod kątem kolizji oraz zaktualizowanie wartości wektorów brył. Krok ten jest znacznie bardziej wymagający obliczeniowo, ponieważ każda z brył biorąca udział w symulacji testowana jest przeciw pozostałym bryłom.
Aby przyspieszyć obliczenia, przestrzeń, w której uruchamiana jest symulacja, można podzielić na mniejsze obszary metodą drzewa ósemkowego. Dzięki zastosowaniu tego rozwiązania sprawdzane między ze sobą będą jedynie bryły znajdujące się w tej samej podprzestrzeni. Jeśli obiekt znajdzie się na granicy dwóch lub wiecej podprzestrzeni, zostanie dodany do listy sprawdzanych obiektów w każdym obszarze.
Do wyznaczenia momentu kolizji wykorzystany został algorytm SAT opisany w poprzednim rozdziale. Funkcja sprawdzająca wykonuje przetestowania każdego aktywnego obiektu z pozostałymi obiektami symulacji. Należy zaznaczyć, iż obiekty dla których wartości wektorów są zbliżone do zera, zostają usówane z puli obiektów aktywnych. Dla takich obiektów nie są aktualizowane wartości sił, biorą jedynie udział w sprawdzaniu kolizji z aktywnymi obiektami. W momencie wykrycia kolizji z aktywnym obiektem, czyli poprzez nadanie wektorom sił nowych wartości, obiekt znów staje się aktywny.
Poza binarną odpowiedzią, czy kolizja między bryłami wystąpiła, funkcja wykonuje dodatkowe obliczenia. Do podstawowej wersji algorytmu dodane jest wyznaczenie wektora określającego najmniejszą część wspólną kolidujących ze sobą brył (MTV). Korzystając z wektora MTV możliwe jest wyznaczenie wektora rotacji po kolizji brył. Aby to zrobić,

należy wyznaczyć wektor prostopadły do danego. Wyznaczony wektor może zostać uzyty do nadania bryle nowego kierunku oraz prędkości dla ruchu obrotowego. Sama rotacja bryły jest przedstawiona w postaci kwaternionu.

5.2. Kwaterniony

Kwaternion jest rozszerzeniem liczb zespolonych na cztery wymiary i posiada trzy pierwiastki urojone (x,y,z) oraz część rzeczywistą w. Kwaternion możemy zapisać w postaci:

$$q = w + xi + yj + zk \tag{5.1}$$

gdzie $w, x, y, z \in \mathbb{R}$

W praktyce kwaternion jest strukturą czterech liczb. Aby przedstawić rotacje bryły w tej postaci, należy użyć odpowiedniego algorytmu. Należy podać kąt oraz oś wokół której odbywa się obrót. Procedura wygląda następujaco [1]:

$$Angle := Angle * 0.5\text{f}$$
$$S := sin(Angle)$$
$$Out[x] = Axis[x] * S$$
$$Out[y] = Axis[y] * S$$
$$Out[z] = Axis[z] * S$$
$$Out[w] := cos(Angle)$$

W podanym powyżej przykładzie, `Angle` określa kąt a `Axis` oś obrotu a wynik zapisywany jest w `Out`.

Kwaterniony można mnożyć, dlatego też możliwe jest złożenie obrotów wokół wszystkich osi w jednym kwaternionie. Przykład przemnożenia dwóch kwaternionów:

$$Out[x] := q1[w] * q2[x] + q1[x] * q2[w] + q1[y] * q2[z] - q1[z] * q2[y]$$
$$Out[y] := q1[w] * q2[y] + q1[y] * q2[w] + q1[z] * q2[x] - q1[x] * q2[z]$$
$$Out[z] := q1[w] * q2[z] + q1[z] * q2[w] + q1[x] * q2[y] - q1[y] * q2[x]$$
$$Out[w] := q1[w] * q2[w] - q1[x] * q2[x] - q1[y] * q2[y] - q1[z] * q2[z]$$

Sama procedura wykonania rotacji wymaga przekształcenia kwaternionu na macież 3x3, który przedstawia przykład:

$$xx := q[x] * q[x], yy = q[y] * q[y]$$
$$zz := q[z] * q[z]$$
$$xy := q[x] * q[y], xz = q.x * q[z]$$
$$yz := q[y] * q[z], wx = q.w * q[x]$$

[1] Źródło: http://www.flipcode.com/documents/matrfaq.html#Q58

$$wy := q[w] * q[y], wz = q.w * q[z]$$
$$Out[1,1] := 1.0f - 2.0f * (yy + zz)$$
$$Out[1,2] := 2.0f * (xy + wz)$$
$$Out[1,3] := 2.0f * (xz - wy)$$
$$Out[2,1] := 2.0f * (xy - wz)$$
$$Out[2,2] := 1.0f - 2.0f * (xx + zz)$$
$$Out[2,3] := 2.0f * (yz + wx)$$
$$Out[3,1] := 2.0f * (xz + wy)$$
$$Out[3,2] := 2.0f * (yz - wx)$$
$$Out[3,3] := 1.0f - 2.0f * (xx + yy)$$

Możliwe jest uproszczenie powyższej funkcji podając punkt jaki mamy przekształcić, dzięki czemu od razu otrzymamy przekształcony w przestrzeni punkt. Uproszczenie wygląda następująco:

$$xx := q[x] * q[x], yy = q[y] * q[y]$$
$$zz := q[z] * q[z]$$
$$xy := q[x] * q[y], xz = q[x] * q[z]$$
$$yz := q[y] * q[z], wx = q[w] * q[x]$$
$$wy := q[w] * q[y], wz = q[w] * q[z]$$
$$Out[x] := (1.0f - 2.0f * (yy + zz)) * point.x$$
$$+(2.0f * (xy - wz)) * point.y$$
$$+(2.0f * (xz + wy)) * point.z$$
$$Out[y] := (2.0f * (xy + wz)) * point.x$$
$$+(1.0f - 2.0f * (xx + zz)) * point.y$$
$$+(2.0f * (yz - wx)) * point.z$$
$$Out[z] := (2.0f * (xz - wy)) * point.x$$
$$+(2.0f * (yz + wx)) * point.y$$
$$+(1.0f - 2.0f * (xx + yy)) * point.z$$

Rozdział 6

Wykorzystanie OpenCL w symulacji

6.1. Inicjalizacja OpenCL

W części inicjalizującej odczytany zostaje plik kernela. Został on podzielony na cztery głowne sekcje - strukturę obietków, stałe, dodatkowe funkcje oraz samą funkcję kernela.

6.1.1. Struktura obiektów

W tej części zdefiniowana została struktura odzwierciedlająca dane, jakie posiadają obiekty użyte w symulacji. Aby dane przekazane z pamięci RAM do pamięci GPU mogły zostać poprawnie odczytane, kolejność definiowanych składowych obiektu musi być identyczna z kolejnością zdefiniowaną w części programu hosta.

6.1.2. Stałe

Ta część zawiera dane stałe, wykorzystywane przez funkcje pomocnicze. Zdefiniowane tu zostały takie dane jak wartość grawitacji czy opór powietrza.

6.1.3. Funkcje pomocnicze

Dzięki wzorowaniu się na standardzie C99 OpenCL pozwala na wykorzystanie funkcji. Umożliwia to znaczne zwiększenie czytelności oprogramowania oraz szybsze wprowadzenie ewentualnych zmian. W celu przyspieszenia wykonywania kernela argumenty funkcji są przekazywane przez wskaźnik, co niweluje konieczność tworzenia obiektów tymczasowych, a co za tym idzie dodatkowego czasu na alokowanie oraz zwalnianie dużych obszarów pamięci.

6.1.4. Kernel

Tutaj zostają umieszczone funkcje wykonywane na danym urządzeniu obliczeniowym. W przypadku stworzonej aplikacji użyte zostały dwie funkcje - aktualizująca obiekty, oraz sprawdzająca kolizje.

6.2. Uruchomienie kernela przez hosta

Funkcja kernela na samym początku pobiera identyfikator:

```
unsigned int i = get_global_id(0);
```

Funkcja ta pobiera globalny identyfukator wątku. W przypadku stworzonej symulacji, operacje dla każdej bryły sztywnej biorącej udział w symulacji wykonywane są na oddzielnym wątku. Dzięki temu, identyfikator określa również daną bryłę, na której wykonywane są operacje. Odwoływanie się do elementów, na któych wykonywane są operacje, przypomina odwoływanie się do kolejnych elementów w tabeli.
Sam proces inicjalizacyjny rozpoczyna się od pobrania informacji o dostępnych platformach:

```
ret = clGetPlatformIDs (1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
                                &ret_num_devices);
```

W przypadku użytych urządzeń zostaną wykryte dwie plaftormy: karta NVIDIA GeForce GTX 760 oraz procesor Intel®Core™i5-3570K.
Następnie utworzony zostaje kontekst obliczeniowy oraz tworzona jest kolejka zadań:

```
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

Kolejka służy jako interfejs do wysyłania do urządzenia zadań (tj. kopiowanie danych między hostem a urządzeniem, wysłanie kernela).
Kolejnym krokiem jest utworzenie obiektu programu powiązanego z wcześniej stworzonym plikiem:

```
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
                        (const size_t *)&source_size, &ret);
```

Plik z programem jest tekstowy, zatem należy uprzednio wczytać go do pamięci jako łańcuch znakowy (char*).
Po wykonaniu tych operacji należy zbudować program, używając stworzonego obiektu programu:

```
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

Program budowany jest dla określonego urządzenia lub jeśli jako argument została podana wartość NULL, program jest budowany na wszystkich znalezionych urządzeniach.
Po zbudowaniu można stworzyć obiekt kernela z programu:

```
kernel = clCreateKernel(program, "updatePoints", &ret);
```

Jako argument podaje się nazwę funkcji z użytego pliku kernela. Stworzony obiekt kernela będzie można wysłać, po ustaleniu parametrów, za pomocą kolejki zadań do wykonania.
Następnie należy utworzyć zmienne typu cl_mem, które będą służyły jako bufory wejścia/wyjścia w pamięci urządzenia np.:

```
cl_a = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(myBody)*numBoxes,
                                    NULL, &ret);
```

Każda zmienna tego typu może zostać oznaczona jako tylko do odczytu (CL_MEM_READ_ONLY), tylko do zapisu (CL_MEM_WRITE_ONLY) oraz z pełnym dostępem (CL_MEM_READ_WRITE).
Po utworzeniu danych należy przetransferować dane z pamięci hosta do pamięci urządzenia, gdyż kernel nie posiada bezpośredniego dostępu do danych umieszczonych w pamięci hosta.

```
ret = clEnqueueWriteBuffer(command_queue, cl_a, CL_TRUE, 0,
      sizeof(myBody)*numBoxes, bodies, 0, NULL, NULL);
```

Jako, że kernel zostanie wysłany do kolejki zadań, należy odpowiednio ustawić parametry będące argumentami dla przesyłanej funkcji kernela, np.:

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_a);
```

Gdy wszystkie etapy zostały wykonane, można przesłać obiekt kernela do kolejki zadań:

```
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_item_size,
                    &global_item_size, 0, NULL, NULL);
```

Następnie należy zaczekać na wykonanie wszystkich obliczeń przez urządzenie:

```
ret = clFinish(command_queue);
```

Jeśli ten krok zostanie pominięty a urządzenie nie skończy wykonywać kodu kernela, dane wyjściowe będą błędne.
Gdy kernel zakończy prawidłowo pracę, należy przekopiować wynik z pamięci urządzenia do pamięci hosta:

```
ret = clEnqueueReadBuffer(command_queue, cl_a, CL_TRUE, 0,
      sizeof(myBody)*numBoxes, bodies, 0, NULL, NULL);
```

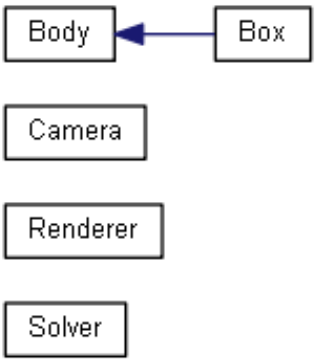
Gdy dane zostaną przekopiowane, należy usunąć zainicjowane wcześniej obiekty OpenCL.

Rozdział 7

Implementacja aplikacji

7.1. Architektura

Aplikacja została stworzona w języku C++ oraz wykorzystuje elementy obiektowe tego języka programowania. Klasy projektu przedstawione są na diagramie 7.1.



Rysunek 7.1. Podział klas (opracowanie własne).

Plik nagłówkowy `stdafx.h` zawiera informacje o wszystkich dołączonych do projektu plikach źródłowych, bibliotek oraz zdefiniowane stałe. Dzięki wykorzystaniu takiego rozwiązania możliwe jest łatwe zarządzanie projektem oraz zmniejszone jest ryzyko błędu wielokrotnego dołączania tej samej biblioteki lub pliku źródłowego.
Plik `main.cpp` zawiera inicjalizację instancji klas odpowiedzialnych za renderowanie oraz komunikację z układem GPU. Tworzone są tu również obiekty symulacji. Znajduje się tutaj również główna pętla aplikacji.
Klasa `Camera` zawiera implementację uproszczonej kamery, umożliwiającej poruszanie się po renderowanej scenie. `Renderer` odpowiada za stworzenie okna aplikacji, ciągłe generowanie obrazu symulowanych brył, przemieszczanie kamery oraz przełączanie trybu pomiędzy wyswietlaniem brył oraz samych ich siatek. Klasa `Solver` obejmuje komunikację z GPU oraz posiada implementację obliczeń fizycznych wykorzystujących CPU. Klasa `Renderer`, `Solver` oraz `Camera` zostały stworzone wykorzystując wzorzec projektowy

Singleton[1]. Celem zastosowania wzorca jest ograniczenie liczby stworzonych instancji danej klasy do jednej oraz zapewnienie globalnego dostępu do stworzonego obiektu. W przypadku każdej klasy korzystającej z tego wzorca, implementacja sprowadza się do stworzenia statycznej metody zwracającej referencję do instancji obiektu. Wywoływana metoda dodatkowo tworzy obiekt klasy przy pierwszym jej uzyciu. Należy pamiętać, aby zarówno kontruktor jak i instancja klasy były składowymi prywatnymi, aby uniemożliwić stworzenie większej ilości obiektów danej klasy.
`Body` jest klasą bazową, zawierającą strukturę obiektu, metody ustawiające, pobierające oraz obliczające odpowiednie elementy struktury. `Box` jest klasą dziedziczącą po klasie `Body`. Ustawia ona typ tworzonej bryły oraz nadpisuje część metod obliczających z klasy bazowej. Dodatkowo plik `Phys.cl` zawierający logikę obliczanych sił oraz przemieszczeń obiektów na urządzeniu GPU.

7.2. Struktury danych

Ważnym elementem aplikacji jest struktura `myBody` zawarta w klasie `Body`. Zawiera ona następujace dane:

— center - centralny punkt obiektu w postaci trzech wartości typu zmiennoprzecinowego, odzwierciedlających położenie w globalnej przestrzeni trójwymiarowe,
— lengths - długości odcinków od środka obiektu do jego krawędzi prowadzonych wzdłuż lokalnych osi układu współrzędnych,
— velocity - aktualna wartość wektora sumy prędkości, jakie działają na obiekt,
— angularVelocity - aktualna wartość wektoru prędkości katowej obiektu wyrażona w postaci trzech liczb skalarnych,
— weight - masa bryły,
— orientation - aktualny obrót obiektu wyrażony w postaci kwaternionu,
— isActive - zmienna binarna określająca stan obiektu,
— type - zmienna wskazująca na typ bryły,
— points - obliczone współrzędne wierzchołków, służące do poprawnego wyrenderowania bryły.

Identyczna struktura znajduje się w pliku `Phys.cl`. Położenie kolejnych elementów struktury zarówno w pliku kernela jak i pliku klasy musi być identyczne, gdyż w innym przypadku, np. zamieniając w strukturze kernela miejscami pola center oraz lengths, wartości używane od obliczeń byłyby zmienione i powodowałyby błędne wyniki.

[1] Singleton - kreacyjny wzorzec projektowy. Gwarantuje, że klasa będzie miała tylko jeden egzemplarz i zapewnia globalny dostęp do niego. [2]

7.3. Przesyłanie danych między CPU a GPU

Idea przesyłania danych pomiędzy pamięcią RAM a pamięcią urządzenia GPU została opisana w rozdziale piątym. Po utworzeniu kontekstu oraz kolejki zadań, stworzony jest bufor o wielkości równej pamięci potrzebnej na przechowanie struktury dla jednej bryły pomnożonej przez liczbę brył biorących udział w symulacji. Dostęp do bufora jest typu CL_MEM_READ_WRITE, co oznacza, iż możliwe jest zarówno odczytywanie jak i zapisywanie do niego danych. Następnie uzupełniane są argumenty przesyłane do funkcji kernela:

— wkaźnik na bufor obiektów,
— liczba obiektów biorących udział w symulacji,
— czas, jaki upłynął pomiędzy kolejnymi iteracjami pętli.

Po wykonaniu operacji należy przekopiować z powrotem dane z bufora do listy obiektów. Umożliwienie zarówno odczytu jak i zapisu do jednego bufora pozwoliło zmniejszyć zapotrzebowanie na pamięć układu GPU. Aby jednak było to możliwe, operacje wykonywane na danych znajdujących się w buforze musiały zostać zaimplementowane w taki sposób, by nie ingerować w elementy struktury wykorzystywane do sprawdzania kolizji z innymi bryłami.

Rozdział 8

Dynamika bryły sztywnej

8.1. Bryła sztywna

Bryła sztywna inaczej ciało sztywne jest to obiekt, w którym poszczególne punkty (np. cząsteczki) pozostają w stałych odległościach od siebie. Ciało sztywne nie zmienia swojego kształtu oraz posiada stałą gęstość.
Opisując dynamikę bryły sztywnej należy zwrócić uwagę na dwa rodzaje ruchu.

8.1.1. Ruch postępowy

Każdy odcinek łączący dwa dowolne punkty ciała bryły będzie równoległy do odcinka łączącego te same dwa punkty, ale w poprzednim położeniu. Wszystkie punkty bryły poruszają się w danym momencie z jednakową prędkością liniową a podczas ruchu jednostajnie zmiennego będą miały również takie samo przyspieszenie. Dlatego też ruch postępowy dla bryły sztywnej jest identyczny jak dla ruchu punktu materialnego

8.1.2. Ruch obrotowy

Wszystkie punkty bryły sztywnej, za wyjątkiem tych leżących na osi obrotu, poruszają się po okręgach o promieniu równym `r`, równych odległości punktów od osi obrotu. Prędkość kątowa bryły będzie wynosić

$$\omega = \frac{v_i}{r_i} \tag{8.1}$$

gdzie `v` to prędkość liniowa. Zatem energię kinetyczną ciała można obliczyć z zależności:

$$E_i = \frac{1}{2} m r^2 \omega^2 \tag{8.2}$$

Całkowita energia kinetyczna bryły w tym ruchu jest równa sumie energii kinetycznych jej poszczególnych punktów. Z definicji bryły sztywnej wynika, że w tym ruchu wszystkie punkty będą miały taka samą prędkość kątową . Tak więc całkowitą energię kinetyczną bryły można przedstawić w postaci wzoru:

$$E_c = \frac{1}{2} (\sum m_i r_i^2) \omega^2 \tag{8.3}$$

Iloczyny

$$m_i r_i \tag{8.4}$$

pochodzą od poszczególnych punktów ciała. Sumę tych iloczynów nazywa się momentem bezwładności `I` bryły sztywnej względem osi obrotu.

$$I = \sum_{i}^{n} m_i r_i^2 \tag{8.5}$$

Moment bezwładności bryły zależy nie tylko od osi obrotu, ale także od kształtu i rozkładu gęstości ciała, wyrażanej w $kg * m^3$ W zależności od rodzaju bryły, moment bezwładności występuje pod inną postacią. Dla sześcianu wynosi ona $I = \frac{ms^2}{6}$, gdzie `m` oznacza masę a `s` długość krawędzi. Dla kuli będzie to wartość $I = \frac{2mr^2}{5}$ gdzie `m` oznacza masę a `r` promień kuli.
Jeżeli bryła sztywna wykonuje ruch obrotowy z przyspieszeniem kątowym, to dzieje się tak pod wpływem przyłożonej siły. Jeżeli siła F działa na punkt materialny bryły sztywnej to jej działanie automatycznie przenosi się na całą bryłę. Mówi się nie o samej sile, ale o tzw. momencie siły. Można go obliczyć ze wzoru:

$$\tau = rF \sin \alpha \tag{8.6}$$

gdzie r jest odległością punktu przyłożenia siły od osi obrotu, a kąt alfa zawarty jest między prostą łączącą punkt przyłożenia siły z osią obrotu i wektorem siły F.
Zarówno dla ruchu postępowego jak i ruchu obrotowego obowiązują zasady dynamiki Newtona.

8.2. Zasady dynamiki Newtona

8.2.1. I zasada dynamiki

„W inercjalnym układzie odniesienia, jeśli na ciało nie działa żadna siła lub siły działające równoważą się, to ciało pozostaje w spoczynku lub porusza się ruchem jednostajnym prostoliniowym."
W odniesieniu do ruchu obrotowego pierwsza zasada dotyczy sytuacji kiedy bryła sztywna nie porusza się lub ma stałą prędkość kątową. Wtedy to wypadkowy moment sił względem danej osi obrotu, działających na ciało jest równy zero.

8.2.2. II zasada dynamiki

„Jeśli siły działające na ciało nie równoważą się (czyli wypadkowa sił $\vec{F}_w$ jest różna od zera), to ciało porusza się z przyspieszeniem wprost proporcjonalnym do siły wypadkowej, a odwrotnie proporcjonalnym do masy ciała."

Powyższą zasadę można wyrazić wzorem

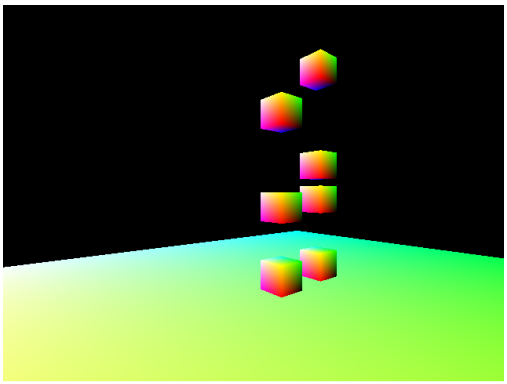
$$\vec{a} = \frac{\vec{F_w}}{m} \tag{8.7}$$

gdzie F_w oznacza wypadkową sił. Kierunek i zwrot przyspieszenia jest zgodny z kierunkiem i zwrotem siły. Stosując drugą zasadę dynamiki do ruchu obrotowego można wywnioskować, że jeśli bryła sztywna poddana jest działaniu stałego momentu sił to porusza się ona z przyspieszeniem kątowym wprost proporcjonalnym do tego momentu sił co do wartości i odwrotnie proporcjonalnym do momentu bezwładności.

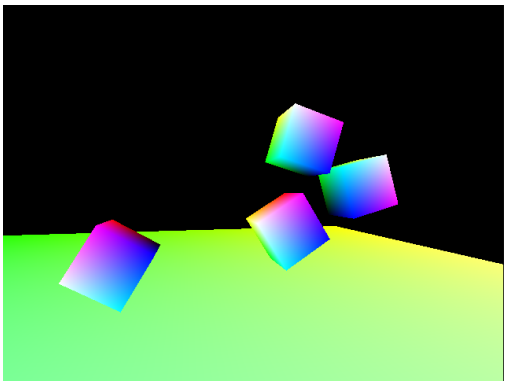
Rozdział 9

Wyniki

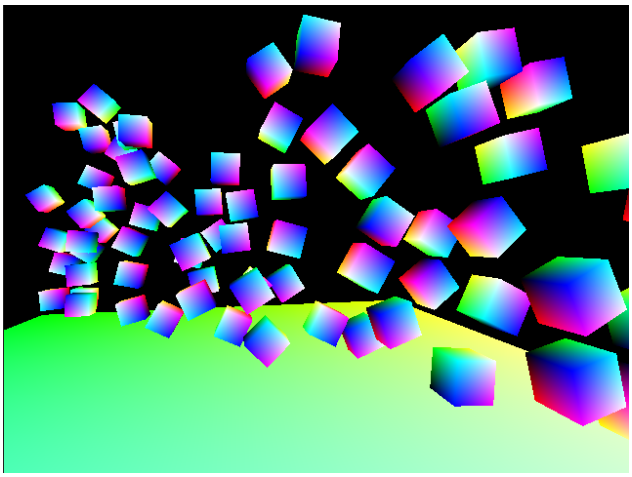
Po stworzeniu aplikacji przeprowadzone zostały testy mające na celu sprawdzić wydajność aplikacji pod kątem czasu wykonania oraz porównanie wyników, wykonując te same obliczenia korzystając jedynie z mocy CPU. Poniżej zamieszczone zostały wyniki tych pomiarów.



Rysunek 9.1. Przykładowy zrzut ekranu z aplikacji (opracowanie własne).



Rysunek 9.2. Zrzut ekranu z aplikacji po kolizji (opracowanie własne).



Rysunek 9.3. Zrzut ekranu z aplikacji po kolizji (opracowanie własne).

Liczba brył	Czas wykonania GPU (ms)	Czas wykonania CPU(ms)
1	46.3	16.7
2	46.3	16.7
5	46.3	16.7
10	46.3	16.7
100	52.4	167.8
1000	90.6	ponad 1000

Tabela przestawia uśrednioną wartość czasu potrzebną na wykonanie jednej iteracji zawierającej aktualizację pozycji obiektów oraz wyznaczanie kolizji i obliczanie wartości sił po zderzeniu. Każdy przedstawiony wynik to średnia arytmetyczna z 1000 pomiarów. Można zauważyć, iż czas potrzebny na wykonanie tych operacji w zakresie do 10 brył w symulacji jest niezmienny. Jednocześnie widać, że do tego momentu wykonywanie operacji na CPU przynosi większe korzyści, niż na GPU. Zdarzenie to spowodowane jest dodatkowym czasem potrzebnym na utworzenie buforów oraz przesyłanie ich do oraz z urządzenia GPU. Natomiast wraz ze wzrostem liczby symulowanych obiektów średni czas operacji zwięsza się zdecydowanie szybciej dla testu przeprowadzonego na CPU. Dzięki możliwości równoległego wykonywania operacji na wielu obiektach jednocześnie, czas potrzebny na wykonanie operacj rośnie zdecydowanie wolniej niż ma to miejsce przy wykorzystaniu CPU.

Rozdział 10

Podsumowanie i wnioski

W ninejszej pracy została pokazana symulacja bryły sztywnej z wykorzystaniem technologii OpenCL. Została również przedstawiona przenośność tego rozwiązania inne na plaftormy, takie jak procesory ogólnego zastosowania firmy Intel. Praca pokazuje również, że technologia OpenGL jest wystarczająca do wizualizacji przedstawionej symulacji. Dzięki wykorzystaniu mocy obliczeniowej karty Nvidia, wykorzystanie procesora CPU zostało odciążone. Dzięki możliwości równoległego przetwarzania instrukcji dla obiektów, wyniki obliczeń dla większej liczby obiektów były gotowe niemalże w tym samym czasie.
Niestety konieczność ciągłego przesyłania danych pomiędzy układem GPU a pamięcią RAM jest jednym z głównych powodów spowolnienia czasu wykonania całego cyklu obliczeń a co jest z tym związane, ograniczenie możliwości wykorzystania w pełni mocy układu. Nakład czasu potrzebny na przesłanie większej porcji danych został zredukowany dzięki ograniczeniu przesyłanych danych, pozwalając, by część z nich została obliczona już po stronie GPU oraz poprzez brak konieczności tworzenia dodatkowej tablicy danych na obiekty biorące udział w symulacji.
Wykorzystanie technologii OpenCL zapewnia przenośność oprogramowania, dzięki czemu umożliwia wykorzystanie podanego rozwiązania obliczeń fizycznych również na innych konfiguracjach wspierających odpowiednią wersję technologii. Dzięki temu możliwe jest przyspieszenie wykonania obliczeń korzystając nawet z budżetowych układów GPU dając zadawalające rezultaty.

Bibliografia

[1] David M Bourg, *Physics for Game Developers* , wydawnictwo O'Reilly Media, listopad 2001

[2] Erich Gamma, Richard Helm, *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, wydawnictwo Helion, wrzesień 2010

[3] gafferongames.com, *Rotation and Inertia Tensors*, `http://gafferongames.com/virtualgo/`, witryna internetowa, stan na 01 września 2014

[4] www.khronos.org, *OpenCL 1.2 Reference Pages*, `http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`, witryna internetowa, stan na 01 września 2014

[5] GLFW 3.0 , `http://www.glfw.org/docs/latest/` dokumentacja, stan na 1 września 2014

[6] Kurs OpenGL , `http://cpp0x.pl/kursy/Kurs-OpenGL-C++` dokumentacja, stan na 1 września 2014

[7] www.nvidia.pl , `http://www.nvidia.pl/object/visual-computing-pl/`, witryna internetowa, stan na 1 września 2014

[8] en.wikipedia.org , `http://en.wikipedia.org/wiki/High-level_shader_language/`, witryna internetowa, stan na 1 września 2014

[9] Marek Zając, *Algorytm SAT* `http://zajacmarek.com/wp-content/uploads/2012/12/Algorytm-SAT.pdf`, stan na 1 września 2014

[10] *Iloczyn wektorowy* `http://www.math.edu.pl/iloczyn-wektorowy`, witryna internetowa, stan na 1 września 2014

[11] pl.wikipedia.org ,`http://pl.wikipedia.org/wiki/Lista_moment%C3%B3w_bezw%C5%82adno%C5%9Bci`, witryna internetowa, stan na 1 września 2014

Spis rysunków

2.1. Model platformy OpenCL.[1] 5
2.2. OpenCL NDRange [2]. 7
2.3. NVIDIA GeForce GTX 760 [3]. 8
2.4. Intel®Core^{TM}i5-3570K [4]. 9

4.1. Projekcja na płaszczyznę (opracowanie własne). 14
4.2. Projekcja na płaszczyznę (opracowanie własne). 14

7.1. Podział klas (opracowanie własne). 23

9.1. Przykładowy zrzut ekranu z aplikacji (opracowanie własne). 29
9.2. Zrzut ekranu z aplikacji po kolizji (opracowanie własne). 29
9.3. Zrzut ekranu z aplikacji po kolizji (opracowanie własne). 30

...……. , dnia …….……..roku

……..
(IMIĘ I NAZWISKO STUDENTA)

……..
(NR ALBUMU)

……..
(KIERUNEK STUDIÓW)

……..
(RODZAJ I FORMA STUDIÓW)

OŚWIADCZENIE

Świadomy/a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że przedkładana praca magisterska / inżynierska / licencjacka (*) na temat:

……..

……..

…….....

została napisana samodzielnie.

Jednocześnie oświadczam, że ww. praca:

- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz. U. z 2000 r. Nr 80, poz. 904, z późniejszymi zmianami) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am w sposób niedozwolony,
- nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów wyższej uczelni lub tytułów zawodowych.

Jestem także świadomy/a, że praca zawiera rezultaty stanowiące własność intelektualną Politechniki Łódzkiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Uczelni.

……..

(PODPIS STUDENTA)

(*) - niepotrzebne skreślić