Uniwersytet Jagielloński
Wydział Matematyki i Informatyki
Zespół Katedr i Zakładów Informatyki Matematycznej

Wielowątkowa symulacja N ciał z implementacją w architekturze CUDA

Autor
Damian Stachura

Opiekun
dr Maciej Ślusarek

20 sierpnia 2018

Spis treści

1 Przedstawienie problemu symulacji N ciał

Symulacja N ciał jest zagadnieniem z mechaniki klasycznej, które polega na wyznaczeniu toru ruchów wszystkich ciał danego układu o danych masach, prędkościach i położeniach początkowych w oparciu o prawa ruchu i założenie, że ciała oddziałują ze sobą zgodnie z prawem grawitacji Newtona.

1.1 Szczególne przypadki

Problem wyznaczenia dokładnego ruchu dowolnej liczby ciał jest trudny, więc można znaleźć wiele prac skupiających się jedynie na ustalonej, małej liczbe ciał.

1.1.1 Problem dwóch ciał

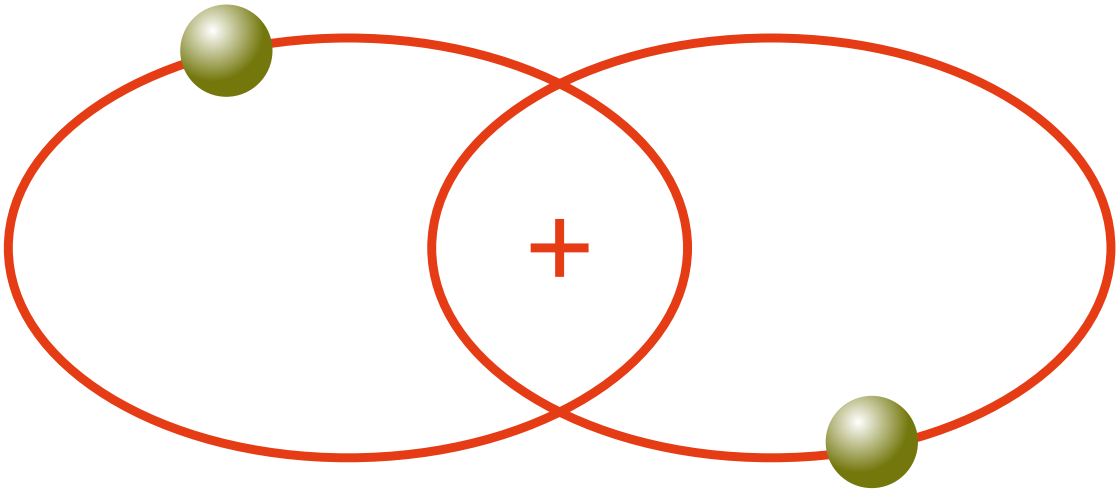
Problem dla dwóch ciał podlegających prawom klasycznej dynamiki Newtona i przyciągających się zgodnie z newtonowskim prawem powszechnego ciążenia został rozstrzygnięty przez J. Bernoulliego przy założeniu, że masa obiektu koncentruje się w jego środku. [19, str. 1-49]
Obydwa ciała poruszają się po krzywych stożkowych, których rodzaj zależy od całkowitej energii układu. Przykładowo, gdy energia jest mała, to ciała nie mogą się od siebie uwolnić, więc krążą wokół siebie po elipsach. (Figure 1)

1.1.2 Problem trzech ciał

Problem trzech ciał wciąż nie został rozwiązany w ogólności. Jednakże istnieją rozwiązania dla szczególnych przypadków, jak na przykład [2, 8]. Inną wariacją tego problemu jest system, w którym masa jednego z ciał jest zaniedbywalnie mała, jest to tak zwany ograniczony problem trzech ciał - przedstawiony przez J. L. Lagrange'a w XVIII wieku. Badał on układ Słońce-Ziemia-Księżyc.

1.2 Zastosowania

Symulacje N ciał są szeroko wykorzystywanymi narzędziami w fizyce oraz astronomii. Problemem, w którym symulacje są użyteczne jest na przykład dynamika systemu z kilkoma ciałami jak układ Słońce-Ziemia-Księżyc [14], co może pomóc w zrozumieniu działania olbrzymich systemów we wszechświecie. [11] W kosmologii symulacje są wykorzystywane do studiowania procesów tworzenia nieliniowych struktur jak galaktyczne halo z wpływem ciemnej materii [7]. Z kolei, bezpośrednie symulacje N ciał są wykorzystywane na przykład do studiowania dynamicznej ewolucji klastrów gwiazd lub do symulacji dynamiki planetozymali. [5]. Symulacje są wykorzystywane również w innych dziedzinach, chociażby w algorytmach rysowania grafu skierowanego siłą [6]



Rysunek 1: Symulacja dwóch ciał poruszających się po elipsach

1.3 Implementacja i wykorzystane technologie

W pierwszej części mojej pracy przedstawię równoległą implementację naiwnego algorytmu symulacji N ciał. W każdym kroku algorytm bezpośrednio wyznacza siły oddziałujące wzajemnie pomiędzy każdymi dwoma ciałami w systemie, przez co oblicza całkowitą siłę oddziałującą na każde ciało w jednym kroku symulacji. W drugiej części zaimplementuję wielowątkowo algorytm Barnes Hut'a, który korzysta z drzew ósemkowych.
Repozytorium jest dostępne pod tym linkiem. Całość została zaimplementowana w C++.
Innymi technologiami wykorzystanymi w pracy są OpenGL, CUDA czy Thrust, których zastosowanie zostanie wspomniane później.
Pełna instrukcja instalacji niezbędnego oprogramowania do uruchomienia symulacji jest zawarta w repozytorium (dla linuxa Ubuntu).

2 Pierwsze podejście implementacyjne

2.1 Sformułowanie problemu

W celu przedstawienia ogólnego sformułowania problemu potrzebujemy przytoczyć trzy prawa dynamiki sformułowane przez Isaaca Newtona: [19, str. 3-4]

Prawo 1. *Każde ciało pozostaje w stanie spoczynku lub ruchu jednostajnego w linii prostej, chyba że jest zmuszone zmienić ten stan przez zewnętrzne oddziaływanie z innymi ciałami, czyli każde ciało jest w układzie inercjalnym.*

Prawo 2. *Szybkość zmiany pędu jest proporcjonalna do siły wywieranej i znajduje się w tym samym kierunku co siła.*

Co oznacza, że w inercjalnym układzie odniesienia zachodzi równość $F = ma$, gdzie F jest wektorem sum sił działających na obiekt, m to masa obiektu, a to jego przyśpieszenie.

Prawo 3. *Każdej akcji towarzyszy reakcja równa co do wartości i kierunku, lecz przeciwnie zwrócona.*

Co oznacza, że jeśli ciało A działa na ciało B siłą F (akcja), to ciało B działa na ciało A siłą (reakcja) o takiej samej wartości i kierunku, lecz o przeciwnym zwrocie.

Niezbędne jest również przytoczenie prawa powszechnego ciążenia Newtona [19, str. 4-5]

Prawo 4. *Każdy obiekt przyciąga każdy inny obiekt z siłą, która jest wprost proporcjonalna do iloczynu ich mas i odwrotnie proporcjonalna do kwadratu odległości między ich środkami.*

Czyli między dowolną parą ciał posiadających masy pojawia się siła przyciągająca, która działa na linii łączącej ich środki, a jej wartość rośnie z iloczynem ich mas i maleje z kwadratem odległości.

Aplikując to prawo do symulacji N ciał, uzyskujemy że na każde i-te ciało działa siła F_i zdefiniowana następująco:

$$F_i = -G \cdot m_i \sum_{j=1, j\neq i}^{N} \frac{m_j(r_i - r_j)}{|r_i - r_j|^3},$$

gdzie m_i masa ciała na które oddziałują inne ciała, m_j masa ciała oddziałującego na i-te ciało, $r_i - r_j$ to różnica wektorów pozycji dwóch ciał, $|r_i - r_j|$ to dystans między ciałami, a G to stała grawitacji i wynosi

$$G = 6,67408(31) \cdot 10^{-11} \frac{m^3}{kgs^2}.$$

Z wykorzystaniem powyższych praw możemy podać następującą definicję

Symulacja N ciał — Dla N ciał mających ustalone masy oraz początkowe położenie i prędkość, ruch każdego obiektu jest symulowany z wykorzystaniem prawa powszechnego ciążenia oraz poprzez wyznaczenie przyspieszenia obiektu korzystając z drugiego prawa dynamiki Newtona.

Potrzebujemy zdefiniować jeszcze jedno pojęcie, jakim jest **ruch jednostajnie przyśpieszony prostoliniowy**. Konkretniej będziemy potrzebować wyprowadzonych wzorów na zmianę prędkości i pokonaną drogę przez obiekt w danym odcinku czasu.

$$v_k = v_p + a \cdot t,$$

gdzie v_k jest prędkością po wykonaniu kroku symulacji, v_p jest prędkością początkową, a oznacza wektora przyśpieszenia, a t to czas kroku symulacji.
Drugim wzorem jest:

$$s = v_p \cdot t + \frac{a \cdot t^2}{2},$$

gdzie s oznacza drogą przebytą w jednym kroku, a pozostałe oznaczenia są identyczne jak powyżej. Zrodlo???

2.2 Jednowątkowa wersja naiwnego algorytmu

Każde ciało na początku symulacji ma pseudolosową pozycję, prędkość oraz masę. W mojej symulacji odległości są wyrażone w metrach, masy ciał są podane w kilogramach, a jednostką prędkości jest metr na sekundę.
Prosty pseudokod dla algorytmu symulującego problem N ciał może wyglądać tak:

Listing 1: Pseudokod naiwnego algorytmu

```
1 ustaw masę oraz początkową pozycję i prędkość dla każdego ciała
2 while(true):
3   for i in {1...N}:
4     for j in {1...N}:
5       if (i!=j):
6         Force[i] += SiłaPomiedzyCiałami(i, j)
7   for i in {1...N}:
8     UaktualnijPozycjeCiała(i)
```

W liniach 3-6 tego algorytmu liczymy bezpośrednio siłę z jaką dwa ciała oddziałują na siebie dla każdej możliwej pary ciał. Jest to najbardziej kosztowna operacja w tym algorytmie, której złożoność to $\mathcal{O}(N^2)$. W linii 7 uaktualniamy pozycję każdego ciała uwzględniając całkowitą siłę, która na nie działa. Siła ta wynika z powyżej przytoczonego wzoru.

Twierdzenie 1. Jednowątkowy naiwny algorytm dla symulacji N ciał złożoność obliczeniową $\mathcal{O}(N^2)$.

Dowód: Dla każdego z N obiektów musimy policzyć siłę oddziałującą nań z każdym innym obiektem, więc musimy $N \cdot (N-1)$ razy policzyć wartość wzoru wynikającego z prawa 4. Z tego wynika, że złożoność tej podoperacji $\mathcal{O}(N^2)$. Następnie dla każdego obiektu musimy wyznaczyć jego przyśpieszenie oraz nową pozycję i prędkość, co jesteśmy w stanie zrobić w czasie $\mathcal{O}(N)$. Poprzez zsumowanie złożoności obu podoperacji, widzimy że złożoność obliczeniowa jednego kroku symulacji naiwnego algorytmu wynosi $\mathcal{O}(N^2)$. □

2.3 Zrównoleglenie naiwnego algorytmu

Jednakże ten algorytm jest zbyt wolny dla dużej liczby ciał. W tym podrozdziale zoptymalizujemy go poprzez zrównoleglenie obliczeń. Siła oddziałująca na pewne ciało oraz wyznaczenie mu nowej pozycji jest niezależne od takich samych obliczeń dla innych ciał. A to oznacza, że pojedynczy krok algorytmu możemy policzyć równolegle dla każdego obiektu.

2.3.1 Architektura CUDA

W tym celu wykorzystamy architekturę CUDA. Jest to uniwersalna architektura kart graficznych umożliwiająca wykorzystanie ich mocy obliczeniowej w wielu problemach, które mogą się wykonywać zarówno sekwencyjne i wielowątkowo. Wykorzystałem CUDĘ w wersji v9.1.85 z compute compability 3.0 i wyżej. W dzisiejszych czasach poza zastosowaniem

w renderowaniu grafiki, jest również często używana do masywnych obliczeń nawet na tysiącach wątków jednocześnie.
Wątki są pogrupowane w bloki. W compute compability 3.0 każdy blok może mieć do 1024 wątków. Z kolei bloki są ułożone w gridzie, który może być nawet trójwymiarowy. W wymiarze x może być aż $2^{31}-1$ wątków, w dwóch kolejnych 65535. [3, str. 238-242] Dodatkowo wątki są grupowane w mniejsze grupy niż bloki, zwane warpami, które liczą po 32 wątki. Wątki w jednym warpie są uruchamiane jednocześnie i zarządzane przez warp scheduler [3, str. 70]

Warto, także wspomnieć o podziale pamięci w programach pisanych w tej architekturze. Możemy wyróżnić trzy rodzaje pamięci :

- lokalna - osobna dla każdego wątku, czyli wątki mogą mieć zmienne lokalne na swój użytek,
- współdzielona - pamięć dzielona przez wszystkie wątki w bloku(ale jest jej tylko 48kb [3, str. 238-242]),
- globalna - najwolniejsza ze wszystkich rodzajów pamięci, ale wspólna dla wszystkich bloków.

2.3.2 Thrust

Thrust jest szablonową biblioteką dla CUDA bazująca na bibliotece STL z C++. Thrust umożliwia implementację aplikacji wielowątkowych za pośrednictwem interfejsu wysokiego poziomu, który jest w pełni zgodny z CUDA C. Korzystałem z wersji v9.2.88. [9]

2.3.3 Implementacja wielowątkowa

W implementacji wykorzystuje dwie szablonowe struktury z biblioteki Thrust. **host_vector**[9] jest odpowiednikiem **std::vector**. Rezyduje w pamięci hosta powiązanego z równoległym devicem. **device_vector**[9] różni się tym, że pamięć związana z nim jest w pamięci device'a.
Implementacje podzielimy na dwie funkcje.

Funkcja NaiveSimBridge, przedstawiona w listingu 2, przyjmuje jako argument **host_vector** z pozycjami wszystkich elementów symulacji, a następnie kopiuje go do **device_vector** dla odowiedzialnego za transport pozycji ciał do pamięci device'a.
Następnie w liniach 3-5 konwertuje **device_vector** dla pozycji, prędkości i masy do raw pointerów. Wykorzystywane są w 6 linii, w której wywołujemy kernel NaiveSim. W potrójnych nawiasach specyfikujemy liczbę bloków oraz liczbę wątków w każdym bloku(jest to składnia z CUDA Runtime API [4]). W linii 7 kopiuje nowe pozycje z device'a do hosta.

Listing 2: Bridge pomiędzy główną pętlą a kernelem

```
1 void NaiveSimBridge(host_vector &pos, int numberOfBodies, float dt) {
```

```
2   thrust::device_vector<float> posD = pos;
3   float *d_positions = thrust::raw_pointer_cast(posD.data());
4   NaiveSim<<<numberOfBlocks, threadsPerBlock>>>(d_positions,
5         d_velocities, d_weights, numberOfBodies, dt);
6   pos = posD;
7 }
```

Kernelem nazywamy funkcję, którą możemy uruchomić dla wielu wątków w pamięci device'a. W tym przypadku kernel NaiveSim, ukazany w listingu 3, jest odpowiedzialny za wyznaczenie nowej pozycji dla każdego ciała.

Listing 3: Kernel NaiveSim

```
1  const double G = 6.674 * (1e-11);
2  template <typename T>
3  __global__ void
4  NaiveSim(T *pos, T *velo, T *weigh, int numberOfBodies, double dt)
5  {
6    int thid = blockIdx.x * blockDim.x + threadIdx.x;
7    if(thid>=numberOfBodies) return;
8    double pos[3] = {pos[thid*3], pos[thid*3+1], pos[thid*3+2]};
9    double weighI = weigh[thid];
10   double force[3] = {0.0, 0.0, 0.0};
11
12   for (int j = 0; j < numberOfBodies; j++) {
13     if (j != thid) {
14       double d[3];
15       for(int k=0; k<3; k++)
16         d[k] = pos[j*3 + k] - pos[k];
17       float dist = (d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
18       dist = dist*sqrt(dist);
19       float F = G * (weighI * weigh[j]);
20       for(int k=0; k<3; k++)
21         force[k] += F * d[k] / dist;
22     }
23   }
24   for(int k=0; k<3; k++) {
25     float acc = force[k] / weighI;
26     pos[thid*3+k] += velo[thid*3+k]*dt + acc*dt*dt/2;
27     velo[thid*3+k] += acc*dt;
28   }
29 }
```

W pierwszej linii mamy stałą G, której wartość oznacza wcześniej zdefiniowaną stałą grawitacji. W sygnaturze kernela widzimy, że przyjmuje trzy raw_pointery do odpowiednio pozycji, prędkości oraz mas. A poza tym dostaje również odcinek czasu, w którym wykonywany jest dany krok oraz całkowitą liczbę ciał w symulacji. W 6 linii wyznaczamy indeks obiektu, dla którego będziemy prowadzić obliczenia. Jako że korzystamy tylko z jednowymiarowego schematu bloków, to wystarczy wymnożyć identyfikator bloku z rozmiarem pojedynczego bloku oraz dodać identyfikator wątku w bloku. W linii 7 mamy zabezpiecznie na wypadek gdyby obiekt o takim indeksie nie istniał, gdyż czasami liczba wywołanych wątków jest większa niz rzeczywista liczba obiektów.

W liniach 8-10 tworzymy zmienne lokalne dla dotychczasowych pozycji oraz masy naszego ciała, żeby zdecydowanie ograniczyć odwołania do droższej pamięci globalnej oraz tworzymy tablicę, w której będziemy liczyć siły działające na ciało.

Dalej mamy część, dla której równoleglimy nasz algorytm. Najpierw w liniach 12-23 mamy pętlę, w której liczymy siłę działającą na obiekt o identyfikator, której jest równe wcześniej wyliczonemu thid(threadId) poprzez każdy inny obiekt w symulacji. Postępujemy zgodnie ze wzorem z Prawa 4. W liniach 15-16 wyliczamy wektory odległości między naszym obiektem a każdym innym. W następnych dwóch liniach wyliczamy odległość między tymi dwoma obiektami i podnosimy ją do potęgi trzeciej, tak jak we wzorze. Z kolei w ostatnich liniach tej pętli podstawiamy wszystkie wartości do wzoru, aby obliczyć siłę działającą na ciało.

Mając policzoną siłę oddziałującą na obiekt, przechodzimy do wyliczenia nowej pozycji dla obiektu. Jako, że ruch obiektu aproksymujemy poprzez przyjęcie że obiekt w pojedynczmy kroku porusza się ruchem jednostajnym przyśpieszonym, a następnie łączenie ze sobą kolejnych kroków symulacji, to możemy w tym celu wykorzystać wcześniej wprowadzone wzory.
Najpierw w linii 25, korzystając z prawa 2, wyliczamy wektor przyśpieszenia ciała. W linii 26 do aktualnej pozycji dodajemy wektor drogi, o którą przesuwamy ciało po obecnym kroku. I w końcu, linia 27 aktualizuje wektor prędkości.

Wykorzystując architekturę CUDA, mogliśmy przyśpieszyć nasz algorytm. Jednakże, symulacja implementowana tym algorytmem ma jeszcze jeden duży problem. Gdy obiekty są bardzo blisko siebie, to znaczy odległość między nimi jest bliska zera, to wtedy siła, którą na siebie działają jest ogromna. Wtedy ciała wyraźnie się oddalają od siebie, co jest nienaturalne. W dwóch następnych podrozdziałach podam dwie optymalizacje, które ograniczają skutki takich sytuacji.

2.4 Softening

Pierwszą optymalizacją jest, tak zwany **softening**[1, str. 21]. Polega na modyfikacji wzoru wprowadzonego w Prawie 4, do następującej postaci:

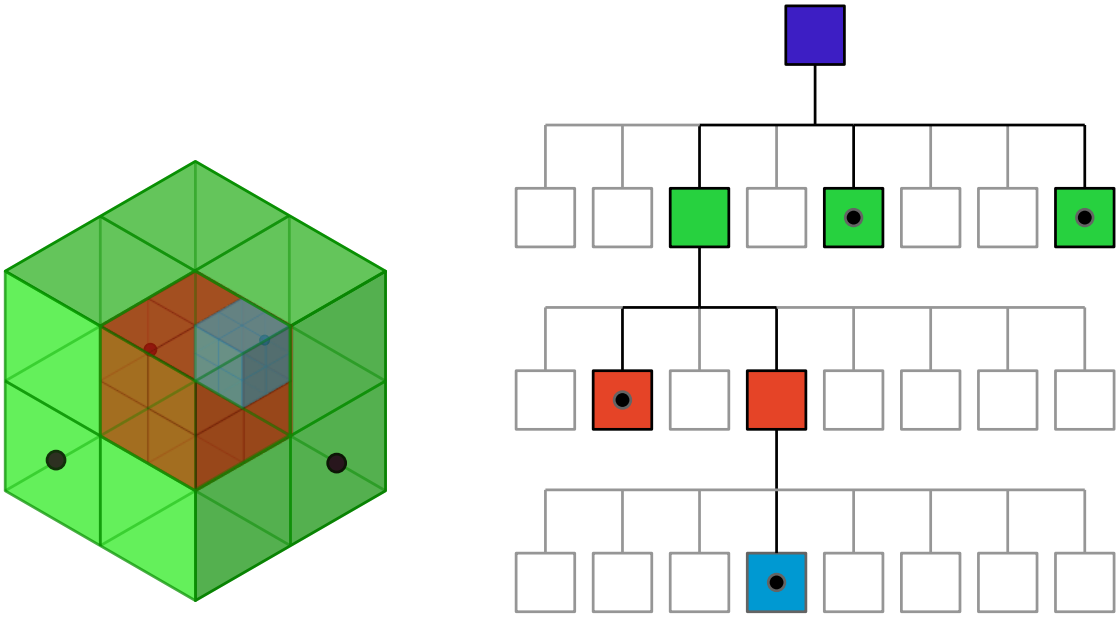
$$F_i = -G \cdot m_i \sum_{j=1, j\neq i}^{N} \frac{m_j \cdot (r_i - r_j)}{(|r_i - r_j|^2 + \epsilon^2)^{\frac{3}{2}}},$$

gdzie wprowadzony ϵ ma za zadanie nie dopuścić do bardzo małych odległości w mianowniku wzoru. W pracy przyjąłem, że $\epsilon = 0.01$.

2.5 Aproksymacja kroku

3 Drugie podejście

Zrównoleglony naiwny algorytm jest szybszy niż wersja jednowątkowa, jednakże wciąż jest zbyt wolny do symulacji układów z bardzo dużą liczbą ciał. W tym celu zaimplementujemy



Rysunek 2: Przykład drzewa ósemkowego z czterema węzłami

drugi algorytm, zwany algorytmem Barnesa Hut. [13, str. 446-449], [17]

3.1 Drzewa Ósemkowe

Algorytm jest oparty na drzewach, a dokładniej drzewach ósemkowych [18]. Struktura drzewa ósemkowego jest następująca(rys. Figure 2):

- w naszym przypadku cała przestrzeń trójwymiarowa jest przedstawiana przez sześcian, który jest reprezentowany przez korzeń drzewa
- w klasycznej definicji korzeń drzewa ma ośmioro następników, z których każdy reprezentuje jeden z 8 podsześcianów, na które dzielimy duży sześcian reprezentowany przez korzeń.
- każdy z kolejnych wierzchołków w drzewie możemy mieć 0 następników, je śli nie zawiera żadnego punktu lub 8 następników, gdy zawiera co najmniej jeden punkt

3.2 Algorytm Barnes Hut z pseudokodem

Algorytm Barnes-Hut możemy podzielić na dwie części:

- pierwszą z nich jest stworzenie drzewa ósemkowego, które będzie reprezenotwało przestrzeń, w której obecne będą wszystkie obiekty poddawane symulacji

- druga część opiera się na przejściu drzewa dla każdego z obiektów i policzenie siły działającej na niego

W podstawowej wersji algorytmu obie te części zaimplementujemy rekurencyjnie.

3.3 Tworzenie drzewa

Najpierw przedstawmy strukturę węzła w naszym drzewie. Jest to ukazane w listingu poniżej.

Listing 4: Struktura węzła w drzewie ósemkowym

```
struct NodeBH {
    double mass;
    double totalMass;
    bool hasPoint;
    bool childrenExists;
    std::array<double, 6> boundaries;
    std::array<double, 3> pos;
    std::array<double, 3> centerOfMass;
    std::array<NodeBH*, 8> quads;
};
```

Żeby nie tworzyć dwóch osobnych struktur dla węzłów wewnętrznych i zewnętrznych połączyliśmy wszystko w jedną strukturę.
Przez to nasza struktura musi zawierać masę ciała oraz jego położenie w przestrzeni(tablica pos). Dodatkowo posiada także pola, które mówią o tym czy jest węzłem wewnętrznym czy zewnętrznym oraz czy posiada punkt(czy jest pusty), a także tablicę z granicami sześcianu, to jest jaki fragment przestrzeni jest ograniczony przez węzeł.

Jednakże najważniejszymi polami tej struktury są tablica kwadrantów, czyli ośmiu mniejszych węzłów następników na które dzielimy naszą przestrzeń.
A także środek masy oraz całkowita masa punktów w przestrzeni określanej przez ten węzeł. **Środek masy(barycentrum) ciała** jest punktem w przestrzeni, który zachowuje się tak, jak gdyby w nim skupiona była cała masa układu ciał. Jest zadany następującym wzorem:

$$x_{srm} = \frac{\sum_{j=1}^{N} m_j \cdot x_j}{\sum_{j=1}^{N} m_j},$$

gdzie m_j oznacza masę j-tego ciała, x_j oznacza jego x-ową współrzędną (dla dwóch pozostałych wzory są analogiczne).

W naszej symulacji przyjmujemy jedno założenie: jeśli obiekt wyląduje poza najbardziej zewnętrznym sześcianem wtedy znika z naszej symulacji, nie jest uwzględniany w kolejnym kroku.
Początkowe pozycje ciał są losowane z pewnego zakresu $[-init, init]$ w każdym wymiarze.
W celu ograniczenia wypadania obiektów z symulacji korzeń drzewa określa w każdym wymiarze przestrzeń, która jest osiem razy większa od początkowej, to znaczy $[-8 \cdot init, 8 \cdot init]$.

Poniżej przedstawimy pseudokod rekurencyjnego tworzenia drzewa ósemkowego, co będzie wprowadzeniem nas do iteracyjnej implementacji tego problemu w architekturze CUDA w późniejszej części pracy.

3.3.1 Funkcja CreateTree

Poniższy listing przedstawia główną funkcję, która odpowiada za stworzenie drzewa ósemkowego dla naszego algorytmu.

Funkcja ta przyjmuje jako argument tablice z pozycjami wszystkich obiektów. Początkowo w linii 2 tworzy następniki dla korzenia drzewa. Następnie w pętli uaktualnia środek masy korzeniowi (linia 6), a potem sprawdza czy punkt nie wyszedł poza całą przestrzeń naszego algorytmu (linia 8). W przypadku, gdy nie wyszedł, wyszukujemy, do którego z następników możemy wstawić nasz punkt (linia 9), a następnie po stworzeniu węzła dla nowego punktu, wstawiamy go do drzewa w linii 11.

Listing 5: Pseudokod algorytmu tworzenia drzewa ósemkowego

```
1  void CreateTree(vector<double>& positions) {
2      root->addQuads(root->getBoundaries());
3      for(unsigned i=0; i<numberOfBodies; i++)
4      {
5          auto pos = copy(positions[i*3], ..., positions[i*3+2]);
6          root->addPointToCenterOfMass(weights[i], pos);
7          if(root->pointIsInSpace(pos))
8          {
9              int index = root->getIndexOfSubCube(pos);
10             NodeBH* node = new NodeBH(weights[i], pos);
11             InsertNode(node, root->getSubQuad(index));
12         }
13     }
14 }
```

3.3.2 Funkcja InsertNode

W powyższej funkcji wywołujemy funkcję **InsertNode** dla każdego węzła wstawianego do drzewa.

Funkcja ta przyjmuje jako argumenty węzeł $node$, który chcemy wstawić do podprzestrzeni reprezentowanej przez drugi argument, czyli węzeł $quad$.
W funkcji tej obsługujemy trzy następujące przypadki:

- Gdy węzeł jest pusty, wtedy po prostu wstawiamy nowy węzeł w to miejsce. (linie 3-7)
- Jeśli natknęliśmy się na zewnętrzny wierzchołek drzewa, to musimy zamienić go na węzeł wewnętrzny, stworzyć dla niego osiem następników, a następnie przepchnąć punkt, który do tej pory zawierał do jednego z nich. W dalszej części tego przypadku, musimy wstawić nowy wierzchołek do tej podprzestrzeni, co robimy uaktualniając środek masy dla korzenia poddrzewa oraz wstawiając węzeł rekurencyjnie do odpowiedniego poddrzewa. (linie 8-21)

- Ostatni przypadek obejmuje wewnętrzne węzły, dla których musimy uaktualnić masę o nowy wierzchołek, a potem wstawić go rekurencyjnie do odpowiedniego następnika. (linie 22-33)

Listing 6: Wstawianie pojedynczego węzła do drzewa

```
1  void InsertNode(NodeBH* node, NodeBH* quad)
2  {
3      if(!quad->isPoint() and !quad->wasInitializedSubQuads())
4      {
5          // pusty węzeł
6          quad.insertHere(node);
7      }
8      else if(quad->isPoint() and !quad->wasInitializedSubQuads()
9      {
10         // zewnętrzny węzeł
11         quad->setPoint(false);
12         quad->pushPointFromHereLower();
13         quad->addPointToCenterOfMass(node->getMass(),
14             node->getPositions());
15         if(quad->pointIsInQuad(node->getPositions());
16         {
17             int index = int index = quad->getIndexOfSubCube(
18                 node->getPositions());
19             InsertNode(node, quad->getSubQuad(index));
20         }
21     }
22     else if(!quad->isPoint() && quad->wasInitializedSubQuads())
23     {
24         // wewnętrzny węzeł
25         quad->updateCenterOfMass(node->getMass(),
26             node->getPositions());
27         if(quad->pointIsInQuad(node->getPositions());
28         {
29             int index = int index = quad->getIndexOfSubCube(
30                 node->getPositions());
31             InsertNode(node, quad->getSubQuad(index));
32         }
33     }
34 }
```

3.3.3 Złożoność tworzenia drzewa

3.4 Spacer po drzewie

Drzewo zostało stworzone w celu szybszego liczenia siły jaka działa na ciało.
Najprostszym podejściem byłoby przejście po drzewie, a kiedy napotkamy punkt w którymś z kwadrantów, to liczymy siłę z jaką oddziałuje na nasz obecny punkt. Jednakże w tym przypadku wciąż musielibyśmy przejść wszystkie węzły w drzewie, więc złożoność nie uległaby poprawie.

Z tego powodu wprowadzamy następującą definicję[1] :

Definicja 3.1. Ratio Niech s będzie szerokością obszaru obejmowanego przez sześcian oraz niech d będzie odległością między środkiem masy obszaru a punktem, to jeśli $s/d < \Theta$, wtedy liczymy siłę między środkiem masy obszaru a punktem. W przeciwnym przypadku wywołujemy się rekurencyjnie na ośmiu podsześcianach.

Listing 7: Liczenie siły oddziałującej na ciało w układzie

```
1  void computeForceForBody(NodeBH* r, std::array<double, 3>& pos, int i)
2  {
3      if(r->isPoint() && !r->wasInitializedSubQuads())
4      {
5          if(r->getIndex() == i) return;
6                  // węzeł zewnętrzny
7          for(int k=0; k<3; k++)
8            forces[i*3+k] += forcesBetweenTwoNodes(r, pos);
9      }
10     else if(!r->isPoint() && r->wasInitialized())
11     {
12         if(r->pointIsInQuad(pos))
13         {
14             for(auto* child : r->getQuads())
15             {
16                 computeForceForBody(child, pos, i);
17             }
18             return;
19         }
20
21         std::array<double, 6>& boundaries = r->getBoundaries();
22         double d = distanceBetweenTwoNodes(pos,
23                 r->getSelectedCenterOfMass());
24         double s = boundaries[1] - boundaries[0];
25         bool isFarAway = (s/d < theta) ? true : false;
26         if(isFarAway)
27         {
28             for(int k=0; k<3; k++)
29                 forces[i*3+k] += forcesBetweenTwoNodes(pos,
30                     r->getSelectedCenterOfMass());
31         }
32         else
33         {
34             for(auto* child : r->getQuads())
35             {
36                 computeForceForBody(child, pos, i);
37             }
38         }
39     }
40 }
```

Tutaj dodać opis do algo, można zrobić potem, gdyż mało istotne. Dodatkowo dołożyć cytacje i poprawić bibliografie

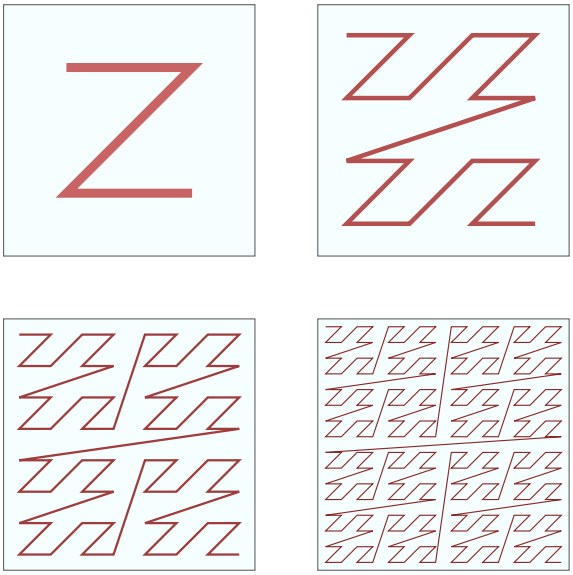
4 Zrównoleglenie algorytmu Barnesa Hut

Jednakże aby zaimplementować konstruowanie drzewa ósemkowego wielowątkowo, to musi zmienić nasze podejście. Obecny pomysł się nie sprawdzi, ponieważ poszczególne wierz-

chołki są doklejane sekwencyjnie jako dzieci do innych wierzchołków, przez co nie możemy tego zrealizować równolegle. Aby podjąć wyzwanie równoległej implementacji drzew ósemkowych, musimy przed rozpoczęciem konstruowania drzewa mieć informację o położeniu każdego wierzchołka w drzewie.

4.1 Kody Mortona

W tym celu zastosujemy **kody Mortona**. Porządkiem Mortona, nazywamy mapowanie punktu w przestrzeni do liniowej list liczby(w naszym przypadku mapujemy punkty w 3-wymiarowej przestrzeni do liczb całkowitych, a konkretniej interesowć nas będzie ich bitowa reprezentacja). Porządek Mortona jest definiowany przez **space-filling curve**, która jest w kształcie **Z**, więc czasami jest nazywana **Z-curve**. Krzywa ta ma ważną własność - jeśli elementy są blisko są blisko siebie w drzewie, to w tym porządku ta bliskość jest zachowana. W pracy traktujemy kod Mortona jako 64-bitową liczbę całkowitą, w której 20 bitów (łącznie 60 dla wszystkich trzech wymiarów) oznacza kolejne rozgałęzienia dla naszego węzła, czyli w których częściach kolejno ląduje, gdy umieszczamy go w coraz mniejszych sześcianach, aż do przyjętej głębokości równej 20.



Rysunek 3: Przykład Z-curve

4.2 Równoległa implementacja drzew ósemkowych

Z wykorzystaniem powyżej zdefiniowanych kodów będziemy implementować algorytm wspomniany w pracy Tero Karrasa, z tym wyjątkiem, że będzie lekko zmodykifowany, gdyż nie będziemy korzystać z **binary radix tree** [15].
Zmodyfikowany algorytm (skrótowo) składa się z sześciu kroków :

- Obliczenie kodów Mortona dla każdego węzła. Realizujemy to poprzez kernel, który na podstawie przedziałów, w których zawierają się punkty w całym układzie dla każdego z wymiarów, wyznacza przez 20 poziomów coraz to dokładniejsze położenie poprzez liczbę bitową.
- Sortujemy kody Mortona. W tym celu wykorzystujemy bibliotekę Thrust, a dokładniej funkcję **sort_by_key**.
- Kody liczymy do głębokości 20, więc istnieje możliwość pojawienia się duplikatów, których tyczą się dwie kolejne części naszego algorytmu (zidentyfikowanei i usunięcie duplikatów). W naszej implementacji załatwiamy to kolejną funkcją z biblioteki thrust, to jest **unique**.
- Liczymy ile wierzchołków będzie miało nasze drzewo(znamy jedynie liczbę liści, potrzebujemy policzyć również liczbę węzłów wewnętrznych. Dodatkowo tworzymy wierzchołki.
- Łączymy węzły według relacji (rodzic, dziecko). W tym celu musimy odpowiednio przyporządkować każdemu węzłowi jego węzeł nadrzędny.

4.3 Implementacja

W tym rozdziale przedstawimy pseudokody implementacji kolejnych kroków algorytmu.

4.3.1 Kody Mortona

Listing 8: Krok 1

```
__global__
void calculateMortonCodes(T* positions, unsigned long long* codes, int numberOfBodies, T* mins, T* maxs) {
  float start[3] = {mins[0], mins[1], mins[2]};
  float p[3] = {positions[3*thid], positions[3*thid+1], positions[3*thid+2]};
  float middle[3] = {(maxs[0] - mins[0])/2, (maxs[1] - mins[1])/2, (maxs[2] - mins[2])/2};
  unsigned long long code = 0;
  for(int i = 0; i < K; ++i) {
    for(int j = 0; j < 3; ++j) {
      code <<= 1;
      if(start[j] + middle[j] < p[j]) {
        code |= 0x1;
        start[j] += middle[j];
      }
      middle[j] /= 2;
    }
  }
  codes[thid] = code;
}
```

W przypadku liczenia kodów Mortona, potrzebujemy minimalnego oraz maksymalnego punktu w każdym z wymiarów. Dzięki temu jesteśmy w stanie wyznaczyć początek oraz środek przedziału dla obiektu na danej głębkości. Poprzez to możemy stwierdzić, w której połowie znajduje się nasz obiekt(czyli na danej głębokości przydzielić 0 lub 1).

4.3.2 Sortowanie oraz usunięcie duplikatów w kodach

Listing 9: Krok 2

```
thrust::device_vector<int> sortedNodes(numberOfBodies);
int* d_sortedNodes = thrust::raw_pointer_cast(sortedNodes.data());
fillNodes<<<blocks, THREADS_PER_BLOCK>>>(d_sortedNodes, numberOfBodies);

thrust::sort_by_key(mortonCodes.begin(), mortonCodes.end(), sortedNodes.begin());

iterators = thrust::unique_by_key(mortonCodes.begin(), mortonCodes.end(), sortedNodes.begin());
```

Rolą wektora sortedNodes jest zachowanie początkowej numeracji punktów, to znaczy sortujemy pary **(kodMortona, początkowyIndeks)**. Zmienna iterators dostaje parę iteratorów na posortowane wektory jako wynik z **thrust::unique_by_key**.

5 Wizualizacja

W celu wizualizacji symulacji wykorzystałem OpenGL3.

5.1 OpenGL

OpenGL [10] jest API, które jest przeznaczone głównie do tworzenia grafiki. Wykorzystuje on kartę graficzną (GPU), więc tworzenie grafiki następuje szybciej niż innymi sposobami. Ten efekt nazywamy przyspieszeniem sprzętowym. OpenGL wykorzystywany jest często przez gry komputerowe i wygaszacze ekranu.

5.2 Renderowanie symulacji

Najistotniejszym punktem kodu OpenGL-owego jest rysowanie symulacji w każdym kroku, za które odpowiada funkcja **Render**:

Listing 10: Pseudokod renderowania symulacji

```
void Render()
{
    glUseProgram(program);
    glm::mat4 view = glm::lookAt(
        glm::vec3(
        camera_radius*sin(camera_theta),
        camera_radius*cos(camera_theta)*cos(camera_phi),
        camera_radius*cos(camera_theta)*sin(camera_phi)
        ),
        glm::vec3(0, 0, 0),
        glm::vec3(0, 1, 0)
    );
    glm::mat4 projection = glm::perspective(
        glm::radians(45.0f),
        1.0f*width/height,
        0.1f,
        100.0f
    );
```

```
19      glm::mat4 mvp = projection * view;
20      GLuint MatrixID = glGetUniformLocation(program, "MVP");
21      glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &mvp[0][0]);
22
23      UpdateBuffers();
24      DrawPoints();
25  }
```

Render polega głównie na wykorzystaniu macierzy Model-View-Projection (MVP). [12] W linii 19 wymnażamy wszystkie macierze ze sobą (odpowiedzialnego za każdy z kroków MVP), uzyskując wyrenderowaną pozycję naszych obiektów).
Bardziej szczegółowo, najpierw ustawiamy kamerę, do czego służy funkcja **lookAt**[10], która jako pierwszy argument przyjmuje współrzędne kamery, następnie punkt na który patrzymy, a w ostatnim parametrze ustawiamy, że patrzymy z góry do dołu.
Z kolei funkcja **perspective**[10] odpowiada za rzutowanie, a w argumentach przyjmuje:

- kąt z jakiego patrzymy na punkt podany w radianach
- **aspect rato**, czyli proporcje pomiędzy szerokością i wysokością obrazu
- **far clipping plane** oraz **near clipping plane**, które wyznaczają płaszczyzny, które ograniczają obszar renderowania

6 Podsumowanie

References

[1] Sverre J. Aarseth. *Gravitional N-Body Simulations. Tools and Algorithms 1 edition*. Cambridge University Press, 2003.

[2] Richard Montgomery Alain Chenciner. “A remarkable periodic solution of the three-body problem in the case of equal masses”. In: *Annals of Mathematics* 152 (2000), pp. 881–901.

[3] NVIDIA Corporation. *CUDA C Programming Guide*. v9.1.85, 2018. url: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 03/05/2018).

[4] NVIDIA Corporation. *NVIDIA CUDA Runtime API*. v9.1.85, 2018. url: http://docs.nvidia.com/cuda/cuda-runtime-api/index.html (visited on 03/05/2018).

[5] J. Stadel D. Richardson T. Quinn and G. Lake. *Direct Large-Scale N-Body Simulations of Planetesimal Dynamics*. 1998.

[6] A.Bezgodov D.Egorov. *Improved Force-Directed Method of Graph Layout Generation with Adaptive Step Length*. 2015.

[7] G.Chincarini E.D’Onghia C.Firmani. *The Halo Density Profiles with Non-Standard N-body Simulations*. 2002.

[8] M.Kramer G.Lodge J. A. Walsh. “A Trilinear Three-Body Problem”. In: *International Journal of Bifurcation and Chaos* 13 (2003), pp. 2141–2155.

[9] GOOGLE. *THRUST*. v9.2.88, 2018. url: https://docs.nvidia.com/cuda/thrust/index.html (visited on 05/15/2018).

[10] Khronos Group. *OpenGL API, OpenGL Shading Lanugage and GLX Specifications*. OpenGL 4.6. 2017. url: https://www.khronos.org/registry/OpenGL/index_gl.php (visited on 07/30/2017).

[11] Douglas C. Heggie. *CHAOS IN THE N-BODY PROBLEM OF STELLAR DYNAMICS*. 1991.

[12] ??? *http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/*.

[13] P. Hut J. Barnes. “A hierarchical O(N log N) force-calculation algorithm”. In: *Nature* 324 (1986), pp. 446–449.

[14] J.Eiland. *N-Body Simulation of the Formation of the Earth-Moon System from a Single Giant Impact*.

[15] Tero Karras. *Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees*. 2012.

[16] Jan Prins Lars Nyland Mark Harris. “GPU Gems 3”. In: 2007. Chap. Fast N-Body Simulation with CUDA. Chapter 31, pp. 677–694.

[17] Keshav Pingali Martin Burtscher. “GPU Computing Gems Emerald Edition”. In: NVIDIA Corporation, Wen-mei W. Hwu, 2011. Chap. An Efficient CUDA Implementation of the Tree-Based Barnes HUT N-Body Algorithm. Chapter 6, pp. 75–92.

[18] Donald Meagher. “Geometric Modeling Using Octree Encoding”. In: *Computer Graphics and Image Processing* 19 (1981), pp. 129–147.

[19] Jerry E. White Roger R. Bate Donald D. Mueller. “Fundamentals of astrodynamics”. In: DOVER PUBLICATIONS, 1971. Chap. 1 TWO-BODY ORBITAL MECHANICS, pp. 1–49.