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

26 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.
Podstawowe zagadnienia do symulacji N ciał zostały przedstawione przez **Isaaca Newtona** w 1687 roku, który wprowadził prawa(później zostaną przedstawione dokładniej), które obowiązują przy próbach analitycznego rozwiązania problemu. [19]. Początki symulacji można datować na rok 1941, kiedy to **Erik Holmberg**, prowadził eksperymenty z galaktykami, które 385modelował jako okręgi, w których punkty koncetrowały się wokół wspólnego środka. Co ciekawe, w swoich eksperymentach reprezentował punkty w galaktyce poprzez żarówki. [14] Z kolei pierwsze obliczenia dla symulacji N ciał na komputerze zostały wykonane przez **Sebastian von Hoernera** w 1960 roku. [13]

1.1 Szczególne przypadki

Problem analitycznego 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

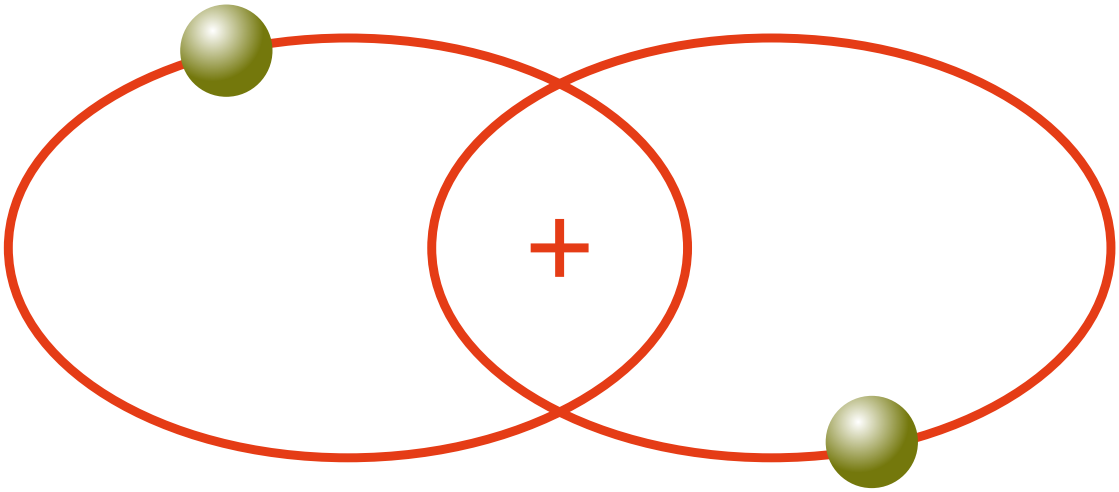
Problem dla dwóch ciał podlegających prawom klasycznej dynamiki Newtona i przyciągających się zgodnie z newtonowskim prawem powszechnego ciążenia został analitycznie rozstrzygnięty przez J. Bernoulliego przy założeniu, że masa obiektu koncentruje się w jego środku. [20, str. 1-49]
Obydwa ciała poruszają 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 analitycznie w ogólności. Jednakże istnieją rozwiązania dla uproszczonych wersji tego problemu, jak na przykład gdy masy obiektów są równe [2] lub w przypadku gdy w układzie mamy trzy równoległe względem siebie linie, a każde z ciał porusza się po jednej z prostych [8], a także gdy masa jednego z ciał obecnych w systemi jest zaniedbywalnie mała(jest to tak zwany ograniczony problem trzech ciał - przedstawiony przez J. L. Lagrange'a w XVIII wieku). [21]

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 [16] lub dynamika systemów gwiezdnych. [11]
W kosmologii symulacje są wykorzystywane do studiowania procesów tworzenia nieliniowych struktur jak galaktyczne halo z wpływem ciemnej materii [7]. Z kolei, bezpośrednie



Rysunek 1: Symulacja dwóch ciał poruszających się po elipsach

symulacje N ciał są wykorzystywane na przykład do studiowania dynamicznej ewolucji klastrów gwiazd lub do symulacji dynamiki planetozymali. [5]. Zastosowania symulacji można zaobserwować również w innych dziedzinach, chociażby w algorytmach rysowania grafu skierowanego siłą. [6] Powyższe przykłady pokazują jak różnorodny jest zakres zastosowań symulacji.

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 z całym kodem projektu 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 Teoretyczne podstawy symulacji N ciał

W celu przedstawienia ogólnego sformułowania problemu potrzebujemy przytoczyć trzy prawa dynamiki sformułowane przez Isaaca Newtona: [20, 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 [20, 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 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 to masa ciała na które oddziałują inne ciała, m_j to 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.
Tak zdefiniowana symulacja N ciał spotyka się z następującymi problemami:

- Kiedy ciała są bardzo bliskie siebie, wtedy zadziała na nie ogromna siła, która w rzeczywistości będzie działała na ciało przez krótki czas. To znaczy krok symulacji może trwać znacznie dłużej niż czas, przez który obiekty będą blisko siebie. Przez to na ciało zadziała o wiele za duża siła niż powinna.

- Ciało w pojedynczym kroku symulacji porusza się ruchem jednostajnie przyśpieszonym prostoliniowym, więc ruch w całej symulacji jest jedynie aproksymacją poprzez sklejenie ruchu obiektu w tych odcinkach czasu.

3 Naiwny algorytm symulacji N ciał

3.1 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 następująco:

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 $\mathcal{O}(N^2)$. □

3.2 Zrównoleglenie naiwnego algorytmu

Jednakże ten algorytm jest zbyt wolny dla dużej liczby ciał. W tym podrozdziale zoptymalizujemy go poprzez zrównoleglenie obliczeń. Wyliczenie siły oddziałującej na pewne ciało oraz analityczne wyznaczenie mu nowej pozycji są niezależne względem tych obliczeń dla pozostałych ciał. A to oznacza, że pojedynczy krok algorytmu możemy policzyć równolegle dla każdego obiektu.

3.2.1 Architektura CUDA

W tym celu wykorzystamy architekturę CUDA. Jest to uniwersalna architektura kart graficznych (które dzisiejszych czasach poza zastosowaniem w renderowaniu grafiki, są również często używane do masywnych obliczeń nawet na tysiącach wątków jednocześnie), umożliwiająca wykorzystanie ich mocy obliczeniowej w wielu problemach, które mogą się wykonywać zarówno sekwencyjne i wielowątkowo. Wykorzystałem CUDĘ w wersji v9.2.148 z compute capability 3.0 i wyżej.

Wątki na CUDA są pogrupowane w bloki. W compute capability 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 pozostałych 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.

3.2.2 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 v9.2.148. [9]

3.2.3 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 jest przechowywany 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.

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 każdym kolejnym kernelu $thid$ będzie liczony tak samo). 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.

Dalej mamy część, dla której równoleglimy nasz algorytm. Najpierw w liniach 12-23 mamy pętlę, w której liczymy siłę działającą na obiekt o identyfikatorze, który jest równy wcześniej wyliczonemu $thid$ 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.

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 j in {0..numberOfBodies-1}
13   {
14     if (j != thid) {
15       double d[3];
16       for(int k=0; k<3; k++)
17         d[k] = pos[j*3 + k] - pos[k];
18       float dist = (d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
19       dist = dist*sqrt(dist);
20       float F = G * (weighI * weigh[j]);
21       for(int k=0; k<3; k++)
22         force[k] += F * d[k] / dist;
23     }
24   }
25   for k in {0..2}
26   {
27     float acc = force[k] / weighI;
28     pos[thid*3+k] += velo[thid*3+k]*dt + acc*dt*dt/2;
29     velo[thid*3+k] += acc*dt;
30   }
31 }
```

Mając policzoną siłę oddziałującą na obiekt, przechodzimy do wyliczenia nowej pozycji dla obiektu. Jako, że ruch obiektu aproksymujemy poprzez przyjęcie, że obiekt w pojedynczym kroku porusza się ruchem jednostajnym przyśpieszonym, a następnie łączymy ze sobą kolejne kroki 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. Tak jak wspomnieliśmy wcześniej, gdy obiekty są bardzo blisko siebie, to znaczy odległość między nimi jest bliska zera, to wtedy siła, którą na siebie działają jest ogromna. Wtedy działa ona na nie przez dłuższy czas niż w rzeczywistości, co jest nienaturalne.

3.3 Softening

Optymalizacją dla tego problemu 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 $\epsilon = 0.01$.

3.4 Złożoność zrównoleglonej naiwnej wersji algorytmu

Równoległa wersja naiwnego algorytmu wciąż wykonuje pracę rzędu $\mathcal{O}(N^2)$, jednakże dzięki temu, że dla każdego obiektu siły na niego działające oraz nową pozycję liczymy na osobnym wątku, to czas działania algorytmu to $\mathcal{O}(N)$, przy założeniu, że mamy do dyspozycji N procesorów.

4 Symulacja N ciał algorytmem Barnes Hut

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 drugi algorytm, zwany algorytmem Barnesa Hut. [15, str. 446-449]

4.1 Drzewa Ósemkowe

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

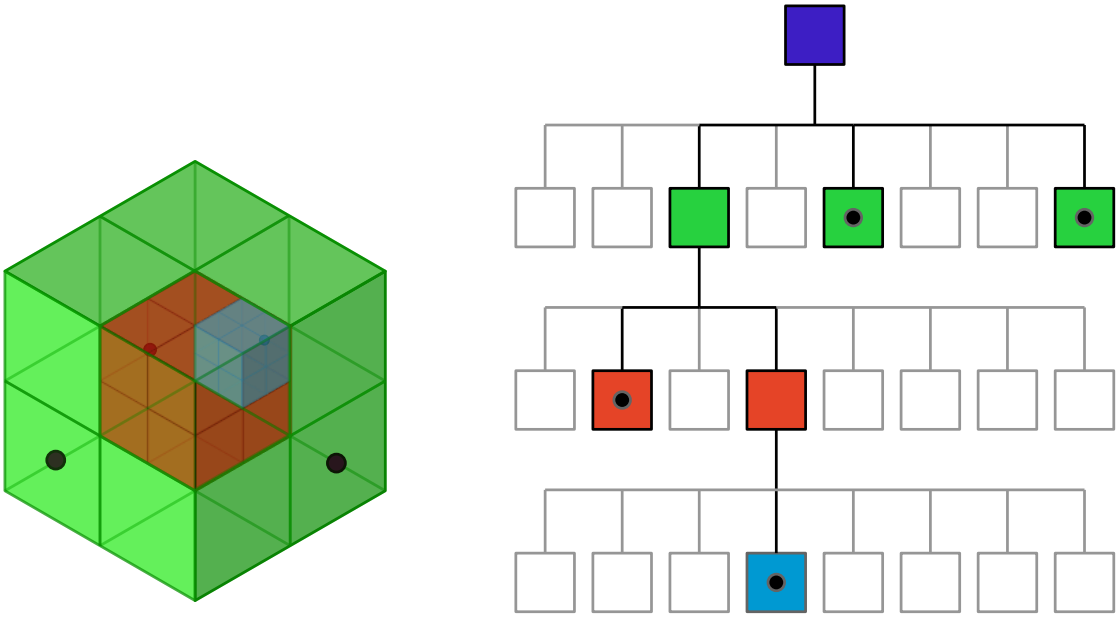
- w naszym przypadku cała przestrzeń trójwymiarowa jest przedstawiana przez sześcian, który jest reprezentowany przez korzeń drzewa
- w klasycznej definicji korzeń drzewa ma ośmioro następników, z których każdy reprezentuje jeden z 8 podsześcianów, na które dzielimy duży sześcian reprezentowany przez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

4.2 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 możemy zaimplementować rekurencyjnie.



Rysunek 2: Przykład drzewa ósemkowego z czterema węzłami

4.3 Rekurencyjne tworzenie drzewa

Poniżej pokażemy pseudokod jednowątkowego tworzenia drzewa ósemkowego. Najpierw przedstawmy strukturę węzła w naszym drzewie. Jest ona ukazana w poniższym listingu.

Listing 4: Struktura węzła w drzewie ósemkowym

```
1  struct NodeBH {
2      double mass;
3      double totalMass;
4      bool hasPoint;
5      bool childrenExists;
6      double boundaries[6];
7      double pos[3];
8      double centerOfMass[3];
9      NodeBH* quads[8];
10 };
```

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, czyli jak duży 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).

4.3.1 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 i in {0..numberOfBodies-1}
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 }
```

4.3.2 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 }
```

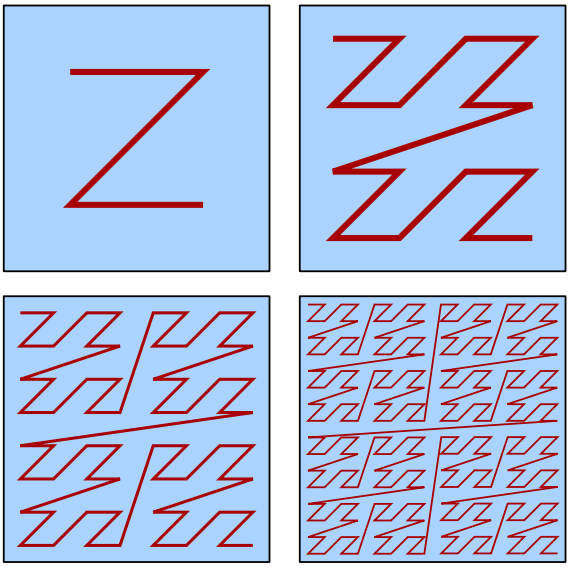
5 Zrównoleglenie algorytmu Barnes Hut

W celu przyśpieszenia algorytmu poprzez wielowątkową implementaję, musimy zmienić nasze podejście w tworzeniu drzewa. Obecny pomysł się nie sprawdzi, ponieważ poszczególne wierzchołki 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.

5.1 Kody Mortona

W tym celu zastosujemy **kody Mortona**. Kodami Mortona, nazywamy mapowanie punktu w przestrzeni do listy liczb(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 siebie w drzewie, to w tym porządku ta bliskość jest zachowana w ułożeniu w pamięci. 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 jedną z dwóch części, w których punkt kolejno ląduje, gdy umieszczamy go w coraz mniejszych sześcianach, aż do przyjętej głębokości równej 20.



Rysunek 3: Przykład Z-curve

5.2 Równoległa implementacja drzew ósemkowych

Z wykorzystaniem powyżej zdefiniowanych kodów będziemy implementować algorytm wspomniany w pracy Tero Karrasa [17], z tym wyjątkiem, że nasz algorytm będzie lekko zmodykifowany, gdyż nie będziemy korzystać z wprowadzonych tam **binary radix tree**. 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 (znajdujemy dolną i górną granicę w każdym z wymiarów), wyznacza przez 20 poziomów coraz to dokładniejsze położenie poprzez ustawianie kolejnych bitów w liczbie binarnej.

- 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 (zidentyfikowanie i usunięcie duplikatów). W naszej implementacji załatwiamy to kolejną funkcją z biblioteki thrust, to jest **unique_by_key**.
- 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.

5.3 Pseudokody poszczególnych kroków

W tym rozdziale przedstawimy pseudokody do kolejnych kroków algorytmu.

5.3.1 Liczenie kodów Mortona

Listing 7: Krok 1

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

5.3.2 Sortowanie oraz usuwanie duplikatów w kodach

Listing 8: Kroki 2-4

```
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());

auto iterators = thrust::unique_by_key(mortonCodes.begin(), mortonCodes.end(), sortedNodes.begin());
mortonCodes.erase(iterators.first, mortonCodes.end());
sortedNodes.erase(iterators.second, sortedNodes.end());
```

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**. Ostatecznie musimy użyć **erase** do usunięcia znalezionych duplikatów.

5.3.3 Zliczenie węzłów wewnętrznych oraz łączenie węzłów

Dwie ostatnie części algoorytmu budowania drzewa opiszemy w jednym podrozdziale. Zacznijmy od przedstawienia struktury odpowiedzialnej za węzły w naszym drzewie:

Listing 9: Struktura OctreeNode

```
struct OctreeNode {
  int children[8] = {-1, -1, -1, -1, -1, -1, -1, -1};
  int position = -1;
  float totalMass = 0.0;
  float centersOfMass[3] = {0.0, 0.0, 0.0};
};
```

W powyższej strukturze przechowujemy indeksy następników w globalnej tablicy węzłów, a także standardowo całkowitą masę poddrzewa oraz środek masy. Dodatkowo mamy pole position, które przyjmuje -1, gdy węzeł jest wewnętrzny oraz indeks węzła(na tablicę globalną), gdy węzeł jest liściem. Jak widzimy niżej, początkowo tworzymy **device_vector octree** dla węzłów o liczbie elementów, którą wyznacza liczba punktów z wejścia, które mają unikalne kody Mortona.
Tworzymy również wektor **parentsNumbers**, którego zadaniem jest przechowywanie informacji o rodzicu każdego z wezłów (informacja ta będzie niezbędna przy łączeniu wierzchołków) na każdym z poziomów (jak wspomnieliśmy nasze drzewo ma głębokość 20). Poniższy fragment kodu zawiera również trzy istotne zmienne:

- **childrenCount** - zlicza liczbę nowych węzłów na każdym z poziomów drzewa,
- **previousChildrenCount** - przechowuje liczbę węzłów z poprzedniego poziomu,
- **allChildrenCount** - zlicza liczbę wszystkich węzłów w drzewie ósemkowym.

Listing 10: Przechowywanie węzłów drzewa

```
thrust::device_vector<OctreeNode> octree(uniquePointsCount);
OctreeNode* d_octree = thrust::raw_pointer_cast(octree.data());

thrust::device_vector<int> parentsNumbers(uniquePointsCount);
int* d_parentsNumbers = thrust::raw_pointer_cast(parentsNumbers.data());

int childrenCount = uniquePointsCount;
int allChildrenCount = uniquePointsCount;
int previousChildrenCount = 0;
```

Kluczowym punktem tych dwóch części jest poniższa pętla. Iteruje się ona $K = 20$ razy, zaczynając od dzieci drzewa (podejście **bottom-up**), oznacza to także, że zaczyna od węzłów ograniczających najmniejszy obszar, czyli obszaru ograniczającego nasze punkty najdokładniej.
W każdej z iteracji chcemy wyznaczyć rodziców dla węzłów.
W tym celu pomocny jest kernel **calculateDuplicates** w połączeniu z **thrust::inclusive_scan**, które pozwalają wyznaczyć duplikaty wśród rodziców(czyli węzły na wyższym poziomie, które mają identyczne kody Mortona, więc mają takie same następniki). Kernel ustawia wartość 1 w danej komórce wektora, jeśli rodzice dwóch kolejnych węzłów są różni (wystarczy sprawdzić dwa kolejne, gdyż dzięki sortowaniu dzieci każdego z węzłów są obok siebie), a następnie **inclusive_scan** robi sumę prefiksową tej tablicy, aby wyznaczyć te, które są takie same(mają przypisane takie same liczby po obliczeniu sum prefiksowych).

Listing 11: Kernel calculateDuplicates

```
__global__
void calculateDuplicates(unsigned long long* mortonCodes, int* result, int N) {
  int thid = blockIdx.x*blockDim.x + threadIdx.x;
  if(thid >= N || thid == 0) return;
  unsigned long long code = mortonCodes[thid];
  unsigned long long previous_code = mortonCodes[thid-1];
  code >>= 3;
  previous_code >>= 3;
  result[thid] = (code != previous_code);
}
```

Po wyznaczeniu nowych wierzchołków na kolejnym poziomie, wstawiamy je do wektora **octree**, a następnie przesuwamy im indeks, aby miały indeksy zaczynające się od pierwszego wolnego (po ostatnim węźle z niższego poziomu). Widzimy, że tym sposobem **root** drzewa będzie ostatnim węzłem w naszej tablicy.
Kiedy stworzymy węzły na danym poziomie, to następnym zadaniem jest połączenie ich z węzłami będącymi na niższym poziomie. Relizujemy to poprzez kernel **connectChildren**.

Dodatkowo w tym kernelu nie liczymy całkowicie środka masy dla punktów, a jedynie liczniki ze wzoru, czyli sumę iloczynów mas i pozycji ciał w układzie. W celu dokończenia wyznaczania barycentrum po utworzeniu całego drzewa, odpalamy kernel, który dla każdego z węzłów podzielimy obecne środki mas przez $totalMass$ (które również wyliczyliśmy).

Listing 12: Kernel connectChildren

```
__global__
void connectChildren(unsigned long long int* mortonCodes, int* parentsNumbers, OctreeNode* octree,
    int N, int previousChildrenCount, int* sortedNodes, double* pos, double* wei, int level)
{
    unsigned long long int childNumber = mortonCodes[thid]&0x7;
    octree[parentsNumbers[thid]].children[childNumber] = thid+previousChildrenCount;
    octree[parentsNumbers[thid]].position = -1;
    octree[thid+previousChildrenCount].position = level == 0 ? sortedNodes[thid] : -1;
    int childId = sortedNodes[thid];
    if(level == 0) {
        octree[thid].totalMass = wei[childId];
        for k in {0..2}
        {
            octree[thid].centersOfMass[k] = (wei[childId] * pos[3*childId + k]) / wei[childId];
        }
    }
    int pthid = parentsNumbers[thid];
    atomicAdd(&octree[pthid].totalMass, octree[thid+previousChildrenCount].totalMass);
    for k in {0..2}:
    {
        atomicAdd(&octree[pthid].centersOfMass[k],
                octree[thid+previousChildrenCount].centersOfMass[k]);
    }
}
```

Cele, które ten kernel realizuje, są następujące:

- ustawienie odpowiedniego dziecka dla rodzica(informacja z kodu Mortona),
- ustawienie pola position, to znaczy -1 gdy węzeł wewnętrzny, w przeciwnym przypadku identyfikator węzła,
- aktualizacja totalMass oraz centersOfMass dla węzła,
- dodatkowo, gdy funkcja przetwarza liście drzewa, to ustawia im początkowe wartości dla totalMass oraz centersOfMass.

Całą pętlę realizującą kroki piąty oraz szósty, przedstawia pseudokod w listingu 13.

6 Liczenie sił oddziałujących na ciało

Drzewo zostało stworzone w celu szybszego liczenia siły jaka działa na ciało.
Wersje jednowątkowa oraz wielowątkowa algorytmu liczenia siły różnią się jedynie tym, że ta druga robi to na osobnym wątku dla każdego ciała, a także została zaimplementowa iteracyjnie. Przedstawimy więc jedynie wielowątkowe podejście.
Algorytm polega na przejściu drzewa od korzenia i na sprawdzaniu czy węzeł jest wewnętrzny czy zewnętrzny. Gdy węzeł jest liściem, to zwyczajnie liczymy siłę między naszym punktem a tym węzłem (jak w naiwnym algorytmie). Gdy węzeł jest wewnętrzny, to potrzebujemy następującej definicji [1]:

Definicja 6.1. 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.

Powyższa definicja wyjaśnia postępowanie dla wewnętrznego węzła. Dodatkowo musimy rostrzygać czy nasz punkt i przetwarzany węzeł nie są tym samym węzłem, aby nie liczyć siły między nimi.

Listing 13: Kroki 5-6

```
for(int i = 0; i < K; ++i) {
    blocks = (childrenCount+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK;
    thrust::fill(parentsNumbers.begin(), parentsNumbers.end(), 0);
    calculateDuplicates<<<blocks, THREADS_PER_BLOCK>>>(
        d_codes,
        d_parentsNumbers,
        childrenCount
    );

    thrust::inclusive_scan(parentsNumbers.begin(), parentsNumbers.end(), parentsNumbers.begin());
    octree.insert(octree.end(), parentsNumbers[childrenCount-1]+1, OctreeNode());
    d_octree = thrust::raw_pointer_cast(octree.data());

    thrust::for_each(parentsNumbers.begin(), parentsNumbers.end(),
        thrust::placeholders::_1 += allChildrenCount);

    connectChildren<<<blocks, THREADS_PER_BLOCK>>>(
        d_codes,
        d_parentsNumbers,
        d_octree,
        childrenCount,
        previousChildrenCount,
        d_sortedNodes,
        d_positions,
        d_weights,
        i
    );

    thrust::for_each(mortonCodes.begin(), mortonCodes.end(), thrust::placeholders::_1 >>= 3);
    auto it = thrust::unique(mortonCodes.begin(), mortonCodes.end());
    mortonCodes.erase(it, mortonCodes.end());
    d_codes = thrust::raw_pointer_cast(mortonCodes.data());
    childrenCount = thrust::distance(mortonCodes.begin(), it);
    previousChildrenCount = allChildrenCount;
    allChildrenCount += childrenCount;
}
```

W poniższej iteracyjnej implementacji algorytmu liczenia siły korzystamy z dwóch tablic, które będzie nam symulować stos. Gdy przetwarzamy jeden wierzchołek i okazało się, że musimy w rekurencyjnej wersji wywoływalibyśmy się rekurencyjnie na mniejszym obszarze, tutaj wrzucimy na stos dwie pary. Pierwszą będzie **(idObecnegoWęzła, nrDziecka+1)**, czyli po przetworzeniu całego poddrzewa dziecka o nrDziecka, przejdziemy do przetwarzania kolejnego dziecka. Druga para to **(idDziecka, 0)**, czyli wrzucamy na stos pierwszego

dziecko obecnego węzła (to znaczy węzeł na którym będziemy wywoływać się rekurencyjnie).
Z kolei w celu przechowywania granic wyznaczających obszar, który ogranicza dany węzeł trzymamy jedynie tablice z minimalnymi i maksymalnymi punktami w każdym z wymiarów dla korzenia. Następnie, gdy jesteśmy na poziomie i, to dzielimy długość boku sześcianu korzenia przez 2^i, co pozwala nam uzyskać bok węzła na poziomie i.

Listing 14: Liczenie siły oddziałującej na ciało w układzie

```
__global__
void computeForces(OctreeNode* octree, double* velocities, double* weights,
    double* pos, double* mins, double* maxs, int AllNodes, int N, double dt)
{
    float forces[3] = {0.0, 0.0, 0.0};
    float p = {pos[3*thid], pos[3*thid + 1], pos[3*thid + 2]);
    unsigned int stack[20], child[20];
    float widthCube = maxs[0]-mins[0];
    int top = -1;
    stack[++top] = AllNodes - 1, child[top] = 0;

    while(top>=0)
    {
        int prevTop = top, nextChild = child[top], idx = stack[top--];
        if(idx == -1) continue;

        if(octree[idx].position == -1) {
            double d = distanceBetween(octree[idx].centerOfMass(), p);
            double s = widthCube/(1<<prevTop);
            bool isFarAway = (s/d < theta);
            if(isFarAway) {
                forces += computeForcesBetween(octree[idx].centerOfMass, p);
            }
            else {
                if(nextChild==numberOfChilds) continue;
                stack[++top] = idx, child[top] = nextChild + 1;
                stack[++top] = octree[idx].children[nextChild], child[top] = 0;
            }
        }
        else {
            if(thid == octree[idx].position) continue;
                forces += computeForcesBetween(octree[idx].getPoint(), p);
        }
    }
    UpdatePositionAndVelocity(forces, positions, velocities, weights, N, dt);
}
```

7 Wizualizacja

W celu wizualizacji symulacji wykorzystałem OpenGL3.

7.1 OpenGL

OpenGL [10] jest API, które jest przeznaczone głównie do tworzenia grafiki. OpenGL wykorzystuje 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.

7.2 Renderowanie symulacji

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

Listing 15: Pseudokod renderowania symulacji

```
1  void Render()
2  {
3      glUseProgram(program);
4      glm::mat4 view = glm::lookAt(
5          glm::vec3(
6          camera_radius*sin(camera_theta),
7          camera_radius*cos(camera_theta)*cos(camera_phi),
8          camera_radius*cos(camera_theta)*sin(camera_phi)
9          ),
10         glm::vec3(0, 0, 0),
11         glm::vec3(0, 1, 0)
12     );
13     glm::mat4 projection = glm::perspective(
14         glm::radians(45.0f),
15         1.0f*width/height,
16         0.1f,
17         100.0f
18     );
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:

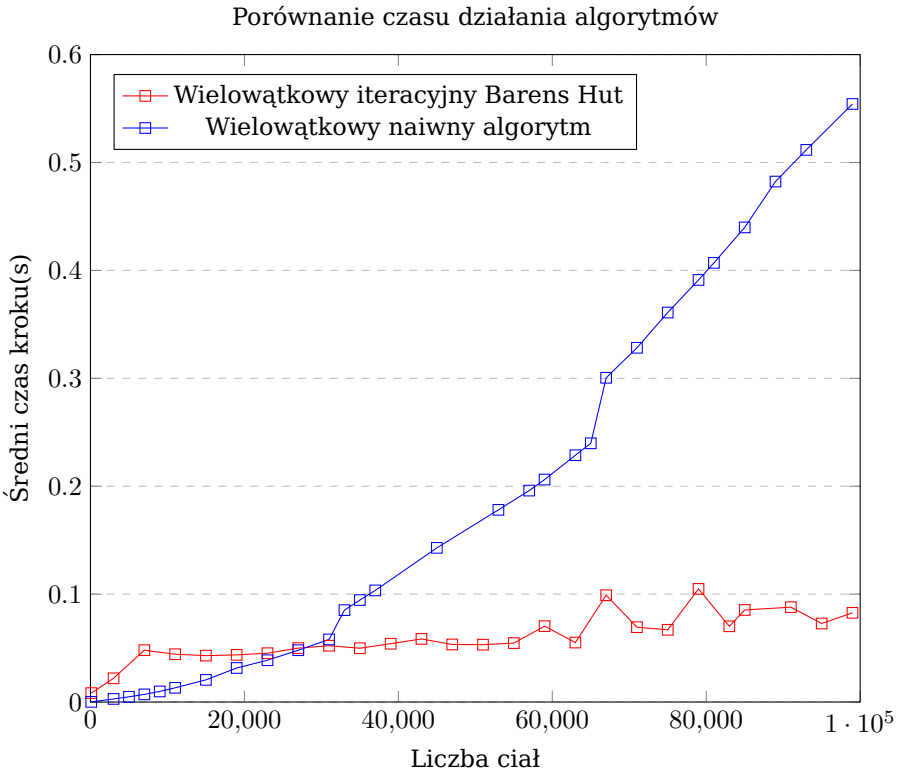
- kąt z jakiego patrzymy na punkt podany w radianach
- **aspect rato**, czyli proporcje pomiędzy szerokością i wysokością obrazu
- **far clipping plane** oraz **near clipping plane**, które wyznaczają płaszczyzny, które ograniczają obszar renderowania

8 Podsumowanie

W pracy zaimplementowałem cztery algorytmy:

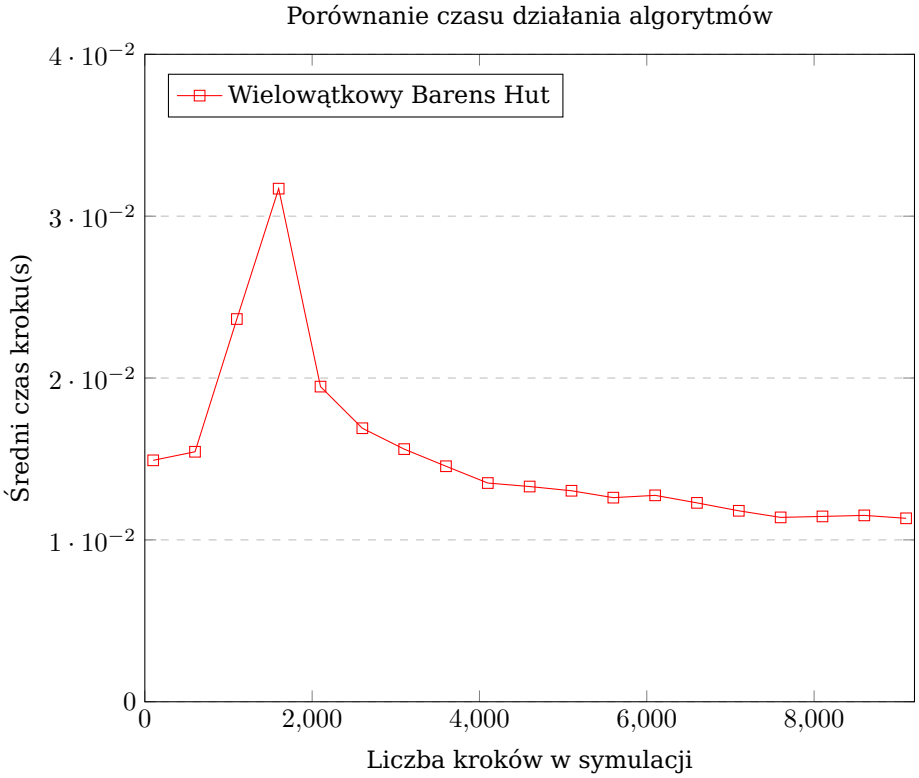
- Jednowątkowy naiwny algorytm
- Wielowątkowy naiwny algorytm
- Jednowątkowy algorytm Barnes Hut
- Wielowątkowy algorytm Barnes Hut

Poniżej dokonam porównania algorytmów zaimplementowanych na CUDA z dwoma dalej opisanymi kryteriami.
Symulacje, których wyniki zostały zebrane w poniższych wykresach były uruchamiane na karcie NVIDIA GeForce GTX 980.
Poniżej widać wykres przedstawiający czas wykonywania się poszczególnych algorytmów przy jednakowych danych wejściowych oraz przez identyczną liczbę kroków. W testach algorytmy były odpalane przez 500 kroków, a następnie całkowity czas działania algorytmu był dzielony przez liczbę kroków. Wykres przedstawia czasy działania trzech ostatnich programów z powyższej listy dla wybranych liczb obiektów z zakresu $\{0, 100000\}$.



Rysunek 4: Porównanie czasu działania algorytmów

Z tego widzimy, że osiągnęliśmy zamierzony efekt, czyli dla coraz większej liczby ciał, równoległa implementacja algorytmu Barnes Hut jest o wiele szybsza niż wielowątkowy algorytm liczący dla każdego ciała siłę działającą na niego przez każde inne ciało.
Zachodzi również zależność, że im więcej tur, tym algorytm Barnes Hut spisuje się lepiej, gdyż część obiektów oddala się względem innych obiektów i przez to często grupka obiektów jest traktowana jako jeden obiekt i nie musimy liczyć siły z każdym ciałem z osobna, a za to liczymy siłę tylko raz z środkiem masy tej grupki ciał.
Widzimy to na drugim wykresie, gdzie mamy ustalone, że ciał w symulacji jest 1000, a zmienna jest liczba symulowanych kroków.



Rysunek 5: Porównanie czasu działania algorytmów

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*. v9.2.148, 2018. url: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 08/01/2018).

[4] NVIDIA Corporation. *NVIDIA CUDA Runtime API*. v9.2.148, 2018. url: http://docs.nvidia.com/cuda/cuda-runtime-api/index.html (visited on 08/01/2018).

[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*. v9.2.148, 2018. url: https://docs.nvidia.com/cuda/thrust/index.html (visited on 08/01/2018).

[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 05/14/2018).

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

[12] Sam Hocevar. *Tutorial 3 : Matrices*. version 2. url: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/ (visited on 12/01/2004).

[13] S. von Hoerner. "Die numerische Integration des n-Körper-Problemes für Sternhaufen. I". In: *Zeitschrift fur Astrophysik* 50 (1960), pp. 184–214.

[14] Erik Holmberg. "On the Clustering Tendencies among the Nebulae." In: *Astrophysical Journal* 94 (1941), p. 385.

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

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

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

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

[19] Isaac Newton. *Philosophiae Naturalis Principia Mathematica*. 1687.

[20] 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.

[21] C. F. Peters V.Szebehely. "A New Periodic Solution of the Problem of Three Bodies". In: *The Astronomical Journal* 72 (1967), pp. 1187–1190.