

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

21 sierpnia 2018

Spis treści

1	Przedstawienie problemu symulacji N ciał	3
1.1	Szczególne przypadki	3
1.1.1	Problem dwóch ciał	3
1.1.2	Problem trzech ciał	3
1.2	Zastosowania	3
1.3	Implementacja i wykorzystane technologie	4
2	Pierwsze podejście implementacyjne	4
2.1	Sformułowanie problemu	4
2.2	Jednowątkowa wersja naiwnego algorytmu	6
2.3	Zrównoleglenie naiwnego algorytmu	6
2.3.1	Architektura CUDA	6
2.3.2	Thrust	7
2.3.3	Implementacja wielowątkowa	7
2.4	Softening	9
2.5	Złożoność zrównoleglonej naiwnej wersji algorytmu	9
3	Drugie podejście	10
3.1	Drzewa Ósemkowe	10
3.2	Algorytm Barnes Hut z pseudokodem	10
3.3	Tworzenie drzewa	11
3.3.1	Funkcja CreateTree	11
3.3.2	Funkcja InsertNode	12
3.3.3	Złożoność tworzenia drzewa	12
4	Zrównoleglenie algorytmu Barnesa Hut	13
4.1	Kody Mortona	13
4.2	Równoległa implementacja drzew ósemkowych	14
4.3	Pseudokody poszczególnych kroków	15
4.3.1	Liczenie kodów Mortona	15
4.3.2	Sortowanie oraz usunięcie duplikatów w kodach	15
4.3.3	Zliczenie węzłów wewnętrznych oraz łączenie węzłów	16
5	Liczenie sił oddziałujących na ciało	18
6	Wizualizacja	19
6.1	OpenGL	20
6.2	Renderowanie symulacji	20
7	Podsumowanie	21

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 liczbie 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. [18, str. 1-49]

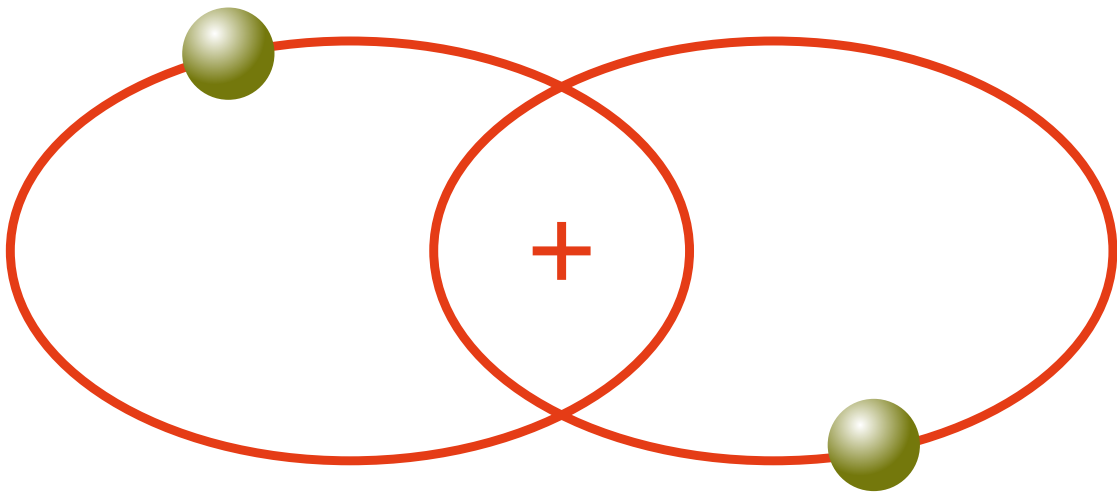
Obydwa ciała poruszają się po krzywych stożkowych, których rodzaj zależy od całkowitej energii układu. Przykładowo, gdy energia jest mała, to ciała nie mogą się od siebie uwolnić, więc krążą wokół siebie po elipsach. (Figure 1)

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



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: [18, 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 inercyjnym.*

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

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

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 [18, 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

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

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 przyspieszony 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 przyspieszenia, 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.

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 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 przyspieszenie 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 $\mathcal{O}(N^2)$. \square

2.3 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 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.

2.3.1 Architektura CUDA

W tym celu wykorzystamy architekturę CUDA. Jest to uniwersalna architektura kart graficznych (które dzisiaj 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 CUDE_x w wersji v9.1.85 z compute compability 3.0 i wyżej.

Wątki na CUDA 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 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.

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 device'm. **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 odpowiedzialnego 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 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 zabezpieczenie na wypadek gdyby obiekt o takim indeksie nie istniał, gdyż czasami liczba wywołanych wątków jest większa niż 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(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 pojedynczym kroku porusza się ruchem jednostajnym przyspieszonym, 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 przyspieszenia 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 przyspieszyć 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.

2.4 Softening

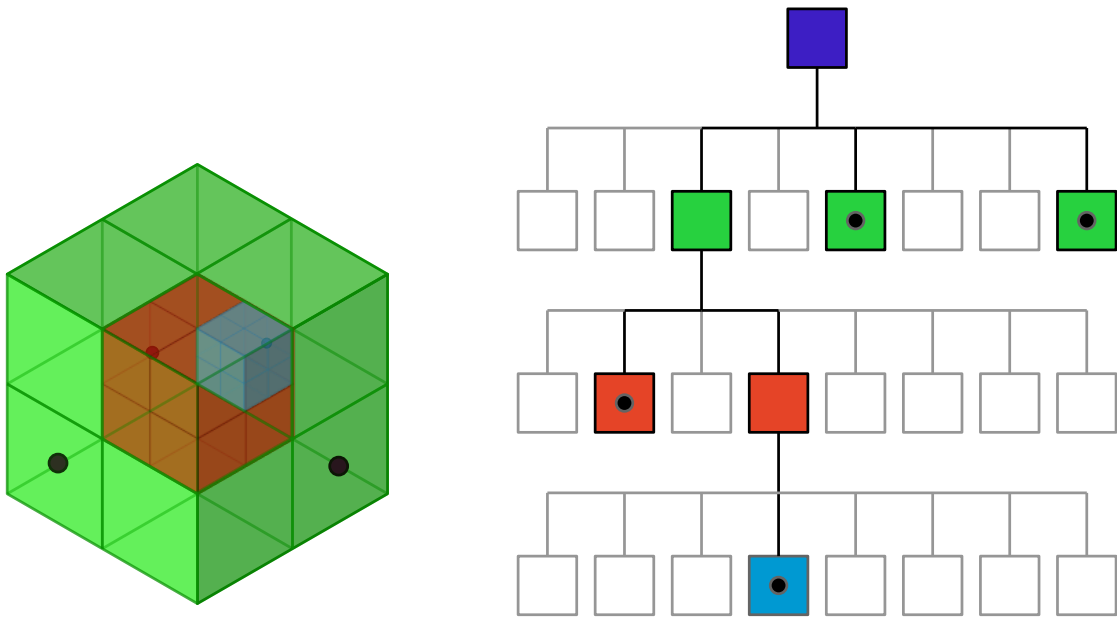
Optymalizacją dla tego problemu jest, tak zwany **softening**[1, str. 21]. Polega na modyfikacji wzoru wprowadzonego w Prawie 4, do następującej postaci:

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

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

2.5 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.



Rysunek 2: Przykład drzewa ósemkowego z czterema węzłami

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 drugi algorytm, zwany algorytmem Barnes'a Hut. [12, str. 446-449], [16]

3.1 Drzewa Ósemkowe

Algorytm jest oparty na drzewach, a dokładniej drzewach ósemkowych [17]. Struktura drzewa ósemkowego jest następująca (rys. Figure 2):

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

3.2 Algorytm Barnes-Hut z pseudokodem

Algorytm Barnes-Hut możemy podzielić na dwie części:

- pierwszą z nich jest stworzenie drzewa ósemkowego, które będzie reprezentowało 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.

3.3 Tworzenie drzewa

Poniżej pokażemy pseudokod jednowątkowego tworzenia drzewa ósemkowego. Najpierw przedstawmy strukturę węzła w naszym drzewie. Jest to ukazane 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     std::array<double, 6> boundaries;
7     std::array<double, 3> pos;
8     std::array<double, 3> centerOfMass;
9     std::array<NodeBH*, 8> quads;
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 sześciangu, 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 określonej 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).

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)

3.3.3 Złożoność tworzenia drzewa

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 }

```

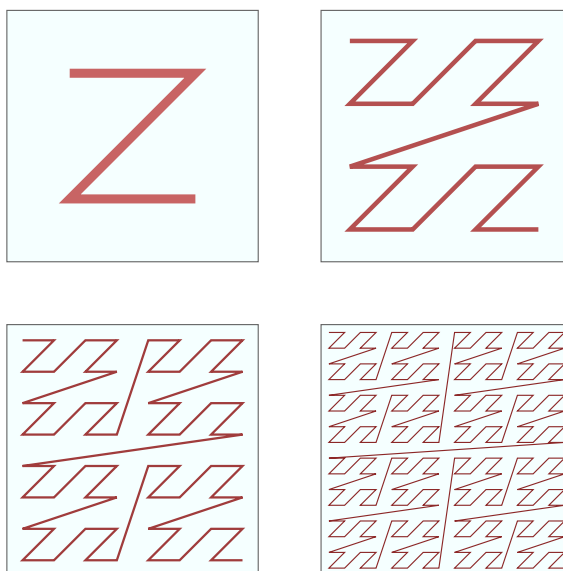
4 Zrównoleglenie algorytmu Barnesa Hut

W celu przyspieszenia algorytmu poprzez wielowątkową implementację, 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.

4.1 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 interesować 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

4.2 Równoległa implementacja drzew ósemkowych

Z wykorzystaniem powyżej zdefiniowanych kodów będziemy implementować algorytm wspomniany w pracy Tero Karrasa [14], z tym wyjątkiem, że będzie lekko zmodyfikowany, 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 dotyczą 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.

4.3 Pseudokody poszczególnych kroków

W tym rozdziale przedstawimy pseudokody do kolejnych kroków algorytmu.

4.3.1 Liczenie kodów Mortona

Listing 7: Krok 1

```

1 __global__
2 void calculateMortonCodes(T* pos, long long* codes, int numberOfBodies, T* mins, T* maxs) {
3     float start[3] = {mins[0], mins[1], mins[2]};
4     float middle[3] = {(maxs[0] - mins[0])/2, (maxs[1] - mins[1])/2, (maxs[2] - mins[2])/2};
5     unsigned long long code = 0;
6     for(int i = 0; i < K; ++i) {
7         for(int j = 0; j < 3; ++j) {
8             code <<= 1;
9             if(start[j] + middle[j] < pos[3*thid + j]) {
10                 code |= 0x1;
11                 start[j] += middle[j];
12             }
13             middle[j] /= 2;
14         }
15     }
16     codes[thid] = code;
17 }

```

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 głębokości. 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 8: Kroki 2-4

```

1 thrust::device_vector<int> sortedNodes(numberOfBodies);
2 int* d_sortedNodes = thrust::raw_pointer_cast(sortedNodes.data());
3 fillNodes<<<blocks, THREADS_PER_BLOCK>>>(d_sortedNodes, numberOfBodies);
4
5 thrust::sort_by_key(mortonCodes.begin(), mortonCodes.end(), sortedNodes.begin());
6
7 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**.

4.3.3 Zliczenie węzłów wewnętrznych oraz łączenie węzłów

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

Listing 9: Struktura OctreeNode

```
1 struct OctreeNode {
2     int children[8];
3     int position;
4     float totalMass;
5     float centersOfMass[3];
6 };
```

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 (wejściowy), gdy węzeł jest liściem. Jak widzimy niżej, początkowo tworzymy **device_vector** dla węzłów o liczbie elementów, którą wyznacza liczba punktów z wejścia, które mają unikalne kody Mortona.

Tworzymy również wektor, którego zadaniem jest przechowywanie informacji o rodzicu każdego z węzłów (informacja ta będzie niezbędna przy łączeniu wierzchołków) na każdym z poziomów (jak wspomnieliśmy nasze drzewo ma 20 poziomów). 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

```
1 thrust::device_vector<OctreeNode> octree(uniquePointsCount);
2 OctreeNode* d_octree = thrust::raw_pointer_cast(octree.data());
3
4 thrust::device_vector<int> parentsNumbers(uniquePointsCount);
5 int* d_parentsNumbers = thrust::raw_pointer_cast(parentsNumbers.data());
6 int childrenCount = uniquePointsCount;
7 int allChildrenCount = uniquePointsCount;
8 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 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, które są poprzednikami dla więcej niż kilku węzłów na niższym poziomie). Kernel ustawia w wektorze 1, jeśli rodzice dwóch kolejnych węzłów są różni (wystarczy sprawdzić dwa kolejne, gdyż dzięki sortowaniu dzieci jednego węzła są obok siebie), a następnie *scan* robi sumę prefiksową tej tablicy, aby wyznaczyć te, które są takie same (mają przypisane takie same liczby po *scanie*).

Listing 11: Kernel calculateDuplicates

```

1 __global__
2 void calculateDuplicates(unsigned long long* mortonCodes, int* result, int N) {
3     int thid = blockIdx.x*blockDim.x + threadIdx.x;
4     if(thid >= N || thid == 0) return;
5     unsigned long long code = mortonCodes[thid];
6     unsigned long long previous_code = mortonCodes[thid-1];
7     code >>= 3;
8     previous_code >>= 3;
9     result[thid] = (code != previous_code);
10 }

```

Po wyznaczeniu nowych wierzchołków na kolejnym poziomie, wstawiamy je do wektora **octree**, a następnie przesuwamy im indeks, aby miały indeksy zaczynające się od pierwszego numeru po ostatnim węźle z poziomu niżej. 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 niższym poziomem. Relizujemy to poprzez kernel **connectChildren**.

Listing 12: Kernel connectChildren

```

1 __global__
2 void connectChildren(unsigned long long int* mortonCodes, int* parentsNumbers, OctreeNode* octree,
3     int N, int previousChildrenCount, int* sortedNodes, int level) {
4     unsigned long long childNumber = mortonCodes[thid]&0x7;
5     octree[parentsNumbers[thid]].children[childNumber] = thid+previousChildrenCount;
6     octree[thid+previousChildrenCount].position = level == 0 ? sortedNodes[thid] : -1;
7     octree[parentsNumbers[thid]].totalMass += octree[childNumber].totalMass;
8     octree[parentsNumbers[thid]].centersOfMass = updateCentersOfMass(parentsNumbers[thid],
9         childNumber);
10 }

```

Cele, które ten kernel realizuje, są następujące:

- ustawienie odpowiedniego dziecka (informacja z kodu Mortona),
- ustawienie pola position, to znaczy -1 gdy węzeł wewnętrzny, w przeciwnym przypadku identyfikator węzła,
- aktualizacja totalMass dla węzła,
- aktualizacja centersOfMass dla węzła.

Całą pętlę realizującą kroki piąty oraz szósty, przedstawia poniższy pseudokod:

Listing 13: Kroki 5-6

```

1 for(int i = 0; i < K; ++i) {
2     blocks = (childrenCount+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK;
3     thrust::fill(parentsNumbers.begin(), parentsNumbers.end(), 0);
4     calculateDuplicates<<<blocks, THREADS_PER_BLOCK>>>(
5         d_codes, d_parentsNumbers, childrenCount);
6
7     thrust::inclusive_scan(parentsNumbers.begin(), parentsNumbers.end(), parentsNumbers.begin());
8
9     octree.insert(octree.end(), parentsNumbers[childrenCount-1]+1, OctreeNode());
10    d_octree = thrust::raw_pointer_cast(octree.data());
11
12    thrust::for_each(parentsNumbers.begin(), parentsNumbers.end(),
13        thrust::placeholders::_1 += allChildrenCount);
14
15    connectChildren<<<blocks, THREADS_PER_BLOCK>>>(
16        d_codes, d_parentsNumbers, d_octree, childrenCount,
17        previousChildrenCount, d_sortedNodes, i
18    );
19
20    thrust::for_each(mortonCodes.begin(), mortonCodes.end(), thrust::placeholders::_1 >= 3);
21    auto it = thrust::unique(mortonCodes.begin(), mortonCodes.end());
22    childrenCount = thrust::distance(mortonCodes.begin(), it);
23    previousChildrenCount = allChildrenCount;
24    allChildrenCount += childrenCount;
25 }

```

5 Liczenie sił oddziałujących na ciało

Drzewo zostało stworzone w celu szybszego liczenia siły jaka działa na ciało.

Wersje jednowątkowa oraz wielowątkowa do liczenia siły różnią się jedynie tym, że ta druga robi to na osobnym wątku dla każdego ciała. 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 5.1. 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 14: Liczenie siły oddziałującej na ciało w układzie

```

1  __global__
2  void computeForces(OctreeNode* octree, float* forcesGlobal, float* velocities, float* weights,
3      float* pos, int AllNodes, int N, float dt)
4  {
5      float forces[3] = {0.0, 0.0, 0.0};
6      OctreeNode* node;
7      float p = {pos[3*thid], pos[3*thid + 1], pos[3*thid + 2]};
8      forces[thid] = 0.0;
9      unsigned long long int stack[64];
10     int top = -1;
11     stack[++top] = AllNodes-1;
12     while(top>=0)
13     {
14         node = octree[idx];
15         int idx = stack[top--];
16         if(noe->position == -1)
17         {
18             double d = dist(node->getCenterOfMass(), p);
19             double s = boundaries[1] - boundaries[0];
20             bool isFarAway = (s/d < theta);
21             if(isFarAway)
22             {
23                 for(int k=0; k<3; k++)
24                     forces[3*thid + k] += forcesBetween(node->getCenterOfMass(), p);
25             }
26             else
27             {
28                 for(int i=0; i<numberOfChlds; i++)
29                 {
30                     stack[++top] = node->children[i];
31                 }
32             }
33         }
34         else
35         {
36             if(thid == octree[idx].position)
37                 continue;
38             for(int k=0; k<3; k++)
39                 forces[3*thid + k] += forcesBetween(node->getPoint(), p);
40         }
41     }
42     for(int k=0; k<3; k++)
43         forcesGlobal[3*thid+k] = forces[k];
44     UpdatePositionAndVelocity(forces, positions, velocities, weights, N, dt);
45 }

```

6 Wizualizacja

W celu wizualizacji symulacji wykorzystałem OpenGL3.

6.1 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.

6.2 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). 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 ratio**, czyli proporcje pomiędzy szerokością i wysokością obrazu
- **far clipping plane** oraz **near clipping plane**, które wyznaczają płaszczyzny, które ograniczają obszar renderowania

7 Podsumowanie

References

- [1] Sverre J. Aarseth. *Gravitational 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 Language 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] P. Hut J. Barnes. "A hierarchical $O(N \log N)$ force-calculation algorithm". In: *Nature* 324 (1986), pp. 446–449.
- [13] J.Eiland. *N-Body Simulation of the Formation of the Earth-Moon System from a Single Giant Impact*.
- [14] Tero Karras. *Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees*. 2012.
- [15] Jan Prins Lars Nyland Mark Harris. "GPU Gems 3". In: 2007. Chap. Fast N-Body Simulation with CUDA. Chapter 31, pp. 677–694.
- [16] Keshav Pingali Martin Burtcher. "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.
- [17] Donald Meagher. "Geometric Modeling Using Octree Encoding". In: *Computer Graphics and Image Processing* 19 (1981), pp. 129–147.
- [18] 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.