POLITECHNIKA POZNAŃSKA
WYDZIAŁ ELEKTRYCZNY
Instytut Automatyki i Inżynierii Informatycznej

Mateusz Bartkowiak

Praca inżynierska

Współbieżne modelowanie i prezentacja dynamiki obiektów fizycznych

Promotor: dr inż. Andrzej Sikorski

Poznań, styczeń 2017

Streszczenie

Przedmiotem niniejszej pracy jest implementacja symulatora n ciał. Polega ona na zaprogramowaniu układu ruchomych punktów materialnych w przestrzeni trójwymiarowej, które oddziałują na siebie siłą grawitacji. Istotnym elementem pracy jest zrównoleglenie kluczowych dla wydajności symulatora obliczeń oraz prezentacja przebiegu symulacji na trójwymiarowej scenie w czasie rzeczywistym. Korzyści z zastosowania wielowątkowości zbadane zostały pod kątem płynności wyświetlanej animacji. Przedstawiono również implikację parametrów wejściowych na przebieg symulacji.

Abstract

The subject of the following work is the implementation of the n-body simulator. This involves programming of the moving material points in three-dimensional space, interacting under the influence of gravitational forces. The key contribution of this work is parallelisation of the computationally expensive routines which are the key to achieving real-time simulations. The proposed multithreaded implementation was examined in terms of frame rate. The influence of particular input parameters on the simulation itself was also assessed as a part of this thesis.

Spis treści

1 Wstęp **4**
- 1.1 Cel pracy . . . 4
- 1.2 Omówienie rozdziałów . . . 5

2 Problem wielu ciał **6**
- 2.1 Wyznaczanie siły . . . 6
- 2.2 Dynamika punktu materialnego . . . 7

3 Przetwarzanie współbieżne **9**
- 3.1 Biblioteka Intel Threading Building Blocks . . . 9
- 3.2 Dzielenie przestrzeni obliczeń . . . 9
- 3.3 Korzyści ze zrównoleglenia pętli . . . 11

4 Grafika trójwymiarowa **12**
- 4.1 Przestrzeń trójwymiarowa . . . 12
  - 4.1.1 Współrzędne jednorodne . . . 12
  - 4.1.2 Macierze . . . 13
  - 4.1.3 Przekształcenia układu współrzędnych w potoku renderowania 13
- 4.2 Biblioteka OpenGL . . . 13
  - 4.2.1 Biblioteki pomocnicze . . . 14
  - 4.2.2 Obiekty bufora . . . 17
  - 4.2.3 Wierzchołki i ich atrybuty . . . 18
- 4.3 Shadery . . . 18
  - 4.3.1 Shader wierzchołków . . . 19
  - 4.3.2 Shader fragmentów . . . 20

5 Implementacja **22**
- 5.1 Koncepcja i założenia . . . 22
- 5.2 Użyte narzędzia . . . 23
- 5.3 Cząsteczki . . . 23
- 5.4 Wyznaczanie położeń cząsteczek . . . 25
- 5.5 Shadery . . . 27
- 5.6 Kamera . . . 30
- 5.7 Główna funkcja programu . . . 34

6 Użytkowanie i testy **44**
6.1 Środowisko uruchomieniowe 44
6.2 Parametry wejściowe 44
6.3 Sterowanie symulatorem 45
6.4 Działanie symulatora 46
6.5 Wydajność symulatora 49

7 Podsumowanie **51**

Bibliografia **52**

Rozdział 1

Wstęp

Symulacje komputerowe pozwalają na predykcje zachowywania się określonych obiektów w zamodelowanym środowisku. Odtwarzanie określonych zjawisk fizycznych możliwe jest dzięki zastosowaniu odpowiednich metod numerycznych. Pozwalają one na uzyskanie wyników, które z reguły są tylko przybliżeniem świata rzeczywistego. Dokładność symulacji często może być poprawiana poprzez uwzględnianie kolejnych czynników, co z kolei zwiększa kosztowność obliczeń. Jednakże dobierając odpowiednie metody, algorytmy czy podejścia do obliczeń, można znacząco zwiększyć wydajność pracy symulatora. Istotnym zagadnieniem jest również sposób prezentacji uzyskanych wyników. Metoda przedstawienia efektów zależy w głównej mierze od celu wykonywania symulacji i może przybierać wiele form.

1.1 Cel pracy

Celem pracy jest implementacja symulatora wielu ciał, wykorzystując wielowątkowość do obliczania dynamiki obiektów oraz grafikę trójwymiarową do prezentacji przebiegu symulacji. Na przykładzie tej implementacji można opisać kwestie będące motywem przewodnim pracy – współbieżność obliczeń i prezentacja obiektów fizycznych.

Ciała reprezentowane są przez punkty materialne i oddziałują na siebie siłą grawitacji. Ponieważ obliczanie ruchu ciał przy dużej ich ilość zajmuje największy czas procesora, a ruch każdego ciała obliczany jest niezależnie od innych, naturalnym wydaje się więc zastosowanie wielowątkowości.

Oprócz optymalnego wykonania obliczeń, należy zapewnić odpowiednią prezentację działania symulatora. Dla problemu będącego tematem przewodnim pracy, najlepszym sposobem prezentacji, oprócz dostarczania pożądanych informacji liczbowych, jest zastosowanie grafiki trójwymiarowej. Dzięki niej można w czasie rzeczywistym śledzić ruch obiektów znajdujących się w zamodelowanym środowisku.

Ważnym aspektem pracy jest analiza wpływu parametrów wejściowych na przebieg symulacji i wydajność symulatora. Przebieg symulacji można badać, śledząc aktywność ciał, a wydajność mierzyć liczbą klatek na sekundę.

Mnogość zagadnień z jakimi musi się zmierzyć programista motywuję do po-

szerzania swojej wiedzy i kompetencji. A umiejętności nabyte przy projektowaniu i implementacji opisanego symulatora można wykorzystać w wielu dziedzinach pokrewnych.

1.2 Omówienie rozdziałów

Praca ta zaczyna się od przedstawienia problemu wielu ciał. Dalej wspomniane jest prawo powszechnego ciążenia w kontekście obliczania sił działających między parą ciał, po czym opisane są sposoby wyznaczania sił wypadkowych działających na ciała. Następnie scharakteryzowane są podstawowe metody obliczeń numerycznych, które są odpowiedzialne za rozwiązywanie równań ruchu. Rozdział trzeci traktuje o przetwarzaniu równoległym na przykładzie biblioteki Intel Threading Building Blocks. W następnym rozdziale opisane są najważniejsze zagadnienia programowania grafiki trójwymiarowej, których zrozumienie jest niezbędne do implementacji symulatora wielu ciał. Rozdział ten rozpoczyna się od wyjaśnienia podstawowych zagadnień związanych z przestrzenią trójwymiarową i przekształceniami na układach współrzędnych, w tym działanie potoku renderowania. Dalej opisana jest biblioteka OpenGL wraz z bibliotekami pomocniczymi. Następnie opisane są bufory w OpenGL, wierzchołki i ich atrybuty oraz shadery wierzchołków i fragmentów. Dalej opisany jest projekt i implementacja symulatora, począwszy od koncepcji i wymienieniu wykorzystanych technologi, aż do omówienia wszystkich struktur i najważniejszych fragmentów programu. W przedostatnim rozdziale przedstawiony jest sposób użytkowania symulatora, jego działanie oraz omówienie przeprowadzonych na nim testów wydajności. Pracę kończy podsumowanie implementacji symulatora wielu ciał.

Rozdział 2

Problem wielu ciał

Problem wielu ciał jest zagadnieniem mechaniki klasycznej. Polega ono na wyznaczaniu toru ruchów ciał o danych masach, prędkościach i położeniach początkowych w danym układzie n ciał zakładając, że owe ciała oddziałują ze sobą zgodnie z prawem powszechnego ciążenia Newtona.

Symulacja zjawiska opartego na problemie wielu ciał ma zastosowanie głównie w astronomii, dynamice płynów, czy w dynamice molekularnej, w której bada się ruchy atomów i cząsteczek. Przykładem może być predykcja zwijania białka, w której ważnym elementem jest obliczenie oddziaływania międzycząsteczkowego van der Waals'a. Innym przykładem zastosowania symulacji wielu ciał jest symulacja przepływów turbulentnych cieczy i gazów [4].

2.1 Wyznaczanie siły

Kluczowym elementem symulacji wielu ciał jest obliczanie siły oddziałującej na dane ciało w danej chwili. Aby obliczyć siłę z jaką oddziałują na siebie dwa ciała korzysta się z prawa powszechnego ciążenia Newtona brzmiącego: *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*. Prawo te dane jest wzorem:

$$f_{ij} = G\frac{m_i m_j}{|r_{ij}|^2}\hat{r}_{ij} \tag{2.1}$$

gdzie:
f_{ij} – siła z jaką ciało j oddziałuje na ciało i,
G – stała grawitacji,
m_i – masa ciała i,
m_j – masa ciała j,
$|r_{ij}|$ – odległość między środkami mas ciała i oraz ciała j,
$\hat{r}_{ij}$ – wersor wektora łączącego środki mas ciała i oraz ciała j.

Sumując siły oddziaływania ciała i z każdym innym ciałem układu N ciał, otrzymujemy siłę wypadkową F_i działającą na ciało i:

$$F_i = Gm_i \times \sum_{\substack{i \leq j \leq N \\ j \neq i}} \frac{m_j}{|r_{ij}|^2} \hat{r}_{ij} \tag{2.2}$$

W praktyce oznacza to obliczenie sił między wszystkimi istniejącymi parami ciał w układzie N ciał, co skutkuje złożonością obliczeniową $\mathcal{O}(n^2)$. Ponieważ wektor siły z jaką ciało i oddziałuje na ciało j ma przeciwny zwrot do wektora siły z jaką ciało j oddziałuje na ciało i, wystarczy obliczyć siłę f_{ij}, a do siły f_{ji} przypisać ujemną wartość f_{ij}. Dzięki temu redukujemy ilość obliczeń działających sił dwukrotnie. Dokładną liczbę C par ciał, między którymi należy obliczyć działającą siłę, można wyznaczyć korzystając z kombinacji bez powtórzeń danej wzorem:

$$C_2^N = \left(\begin{array}{c} N \\ 2 \end{array} \right) = \frac{N!}{2(N-2)!} \tag{2.3}$$

Istnieją algorytmy, które zmniejszają złożoność obliczeniową symulacji wielu ciał. Przykładem jest algorytm Barnes-Hut o złożoności $\mathcal{O}(n \log n)$. Jest to algorytm aproksymacyjny, co oznacza, że daje on przybliżone wyniki. Idea jego działania opiera się na dzieleniu przestrzeni, co pozwala na grupowanie sąsiednich ciał w jedno ciało o masie równej sumie mas zgrupowanych ciał oraz środku leżącym w środku masy tych ciał.

2.2 Dynamika punktu materialnego

Ciało może być reprezentowane jest jako punkt materialny, czyli posiadające masę, lecz o nieskończenie małych rozmiarach. Do opisu ruchu ciała niezbędna jest wiedza o położeniach ciała w danych momentach czasowych. Ze względu na skończony i określony odstęp czasowy między momentami badania położenia ciała, ruch ten będzie tylko przybliżeniem. Nie jest to jednak istotne dla prostej symulacji, w której wyniki prezentowane są na ekranie w postaci animacji.

W przypadku symulacji wielu ciał istotną role odgrywa związek między siłą a przyspieszeniem, którą opisuje druga zasada dynamiki Newtona, mówiąca że *wektor przyspieszenia $\vec{a}$ jest zawsze równoległy do wektora siły $\vec{F}$, a jego wartość jest proporcjonalna do wartości siły*. Zależność ta dana jest wzorem:

$$\vec{a} = \frac{\vec{F}}{m} \tag{2.4}$$

We wzorze tym współczynnikiem proporcjonalności między działającą siłą a spowodowanym przez nią przyspieszeniem jest masa m, ponieważ im jest ona większa, tym mniejszy efekt powoduje przyłożenie siły.

Ponieważ przyspieszenie ciała w każdej chwili czasu jest znane (wyznacza ja działająca na ciało siła), do wyznaczenia pozostaje zatem położenie i prędkość w kolejnych chwilach czasu. Sprowadza się to do rozwiązywania równań różniczkowych

ruchu[1] odpowiednimi metodami numerycznymi. Jedną z nich jest prosty algorytm Eulera. Innym przykładem może być dokładniejszy algorytm Verleta, czy algorytm Runge-Kutty czwartego rzędu (RK4), który jest wolny, lecz jego dokładność pozwala na zwiększenie kroku czasowego.

Algorytm RK4 jest raczej stosowany w symulacjach, które nie są przeprowadzane w czasie rzeczywistym. Jednak w symulacjach przeprowadzanych na potrzeby grafiki trójwymiarowej dokładność obliczeń nie jest priorytetem, ponieważ różnice, które nie będą widoczne na ekranie, są bez znaczenia.

Algorytm Eulera sprawdza się zaskakująco dobrze w przypadku symulacji prostych układów, jak np. w grach. Wysoka precyzja nie jest wówczas priorytetem. Ponadto przyłożone do ciał siły są często kontrolowane przez gracza, co oznacza nagłe i częste zmiany ich wartości. Wpływ zmian siły będzie więc i tak dominował nad efektami utraty dokładności. Wyznaczenie tą metodą położenia x w następnej chwili $t_0 + \triangle t$ opisuje wzór:

$$x(t_0 + \triangle t) \approx x(t_0) + \dot{x}(t_0)\triangle t + \ddot{x}(t_0)\triangle t^2 \tag{2.5}$$

gdzie:
$\dot{x}$ – prędkość ciała,
$\ddot{x}$ – przyspieszenie ciała,
t_0 – ostatnio badana chwila,
$\triangle t$ – krok czasowy.

Algorytm Verleta jest łatwy do implementacji, dokładny i stabilny. Jego podstawową zaletą jest większa dokładność przy niewiele większym koszcie obliczeniowym w porównaniu z algorytmem Eulera. Położenie w następnej chwili metodą Verleta dane jest wzorem:

$$x(t_0 + \triangle t) \approx -x(t_0 - \triangle t) + 2x(t_0) + \ddot{x}(t_0)\triangle t^2 \tag{2.6}$$

Wzór ten wiąże bezpośrednio położenie i przyspieszenie, a zatem algorytm nie generuje pierwszych pochodnych położenia (prędkości). Należy jednak znać położenie kroku poprzedniego, dlatego w chwili zerowej można obliczyć następne położenie korzystając z algorytmu Eulera.

Stosując wymienione metody, długość kroku czasowego powinna być niewielka, ponieważ zakłada się, że przyspieszenie się nie zmienia i aktualizowane jest dopiero w następnym kroku czasowym.

[1]Problem rozwiązywania równań ruchu należy do szerszej klasy tzw. zagadnień początkowych. Rozwiązywane są w nich równania różniczkowe zwyczajne względem czasu (lub innego parametru) znając rozwiązanie w chwili początkowej [1].

Rozdział 3

Przetwarzanie współbieżne

Aby przyspieszyć wykonywanie się programu, można zrównoleglić obliczenia, które przeprowadzane są niezależnie od siebie. W przypadku procesora z paroma rdzeniami, zrównoleglenie najczęściej polega na zastosowaniu dodatkowych wątków. Wątki te mogą działać równocześnie, każdy na innym rdzeniu. Technika ta nosi nazwę przetwarzania współbieżnego i zastosowana została w niniejszej pracy przy użyciu opisanej w tym rozdziale biblioteki TBB (ang. *Intel Threading Building Blocks*).

3.1 Biblioteka Intel Threading Building Blocks

TBB jest biblioteką firmy Intel napisaną w języku C++ służącą do programowania równoległego. Jest zoptymalizowana do działania na wszystkich architekturach procesorów Intel. Wspiera wiele systemów operacyjnych oraz nie wymaga specjalnego kompilatora. Biblioteka ta automatycznie zarządza wątkami zapewniając najefektywniejsze użycie zasobów procesora jednocześnie dostarczając wysokopoziomowe i proste rozwiązania w języku C++. Biblioteka TBB w dużej części opiera się na paradygmacie programowania generycznego, zapewniając niezależność funkcjonalności od typów, algorytmów czy struktur danych.

3.2 Dzielenie przestrzeni obliczeń

Podstawową metodą zrównoleglenia obliczeń w bibliotece TBB jest funkcja `parallel_for`, która dzieli przestrzeń obliczeń na mniejsze części i uruchamia je na osobnych wątkach. Można ją wykorzystać, aby przyspieszyć wykonywanie się funkcji pokazanej na listingu 3.1.

```
1 void SerialApplyFoo( float a[], size_t n ) {
2     for( size_t i=0; i!=n; ++i )
3         Foo(a[i]);
4 }
```

Listing 3.1: Przykładowa funkcja wykonująca obliczenia sekwencyjne [8].

Pierwszym krokiem, aby przyspieszyć obliczenia jest obudowanie funkcji w klasę i przeciążenie operatora `()`, który będzie odpowiedzialny za przetwarzanie części zadania. Dzięki temu możliwe będzie wywołanie obiektu tak, jakby był on funkcją. Pokazane jest to na listingu 3.2.

```
1  #include "tbb/tbb.h"
2
3  using namespace tbb;
4
5  class ApplyFoo {
6      float *const my_a;
7  public:
8      void operator()( const blocked_range<size_t>& r ) const {
9          float *a = my_a;
10         for( size_t i=r.begin(); i!=r.end(); ++i )
11            Foo(a[i]);
12     }
13     ApplyFoo( float a[] ) :
14         my_a(a)
15     {}
16 };
```

Listing 3.2: Funkcja z listingu 3.1 obudowana w klasę [8].

Konstruktor klasy `ApplyFoo` ustawia pola obiektu. Operator `()` przypisuje zmienną `my_a` do zmiennej lokalnej `a`, co wpływa na bardziej efektywne zoptymalizowanie pętli przez kompilator, który lepiej sobie radzi ze zmiennymi lokalnymi. W przeciążonym operatorze `()` widać klasę szablonową `blocked_range<T>`, która opisuje jednowymiarową przestrzeń obliczeń, iterowaną typem `T`. Biblioteka udostępnia również klasę `blocked_range2d<T>` dla przestrzeni dwuwymiarowej oraz `blocked_range3d<T>` dla przestrzeni trójwymiarowej.

Mając tak przygotowaną klasę, można dokonać obliczeń, wywołując funkcję `parallel_for` (listing 3.3).

```
1  #include "tbb/tbb.h"
2
3  void ParallelApplyFoo( float a[], size_t n ) {
4      parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
5  }
```

Listing 3.3: Wywołanie funkcji `parallel_for` [8].

Wyrażenia lambda[1] pozwalają na prostsze używanie `parallel_for`. Dzięki nim, nie programista, lecz kompilator zajmuje się tworzeniem odpowiednich obiektów funkcji. Listing 3.4 pokazuje zmodyfikowany przykład z listingu 3.1 używając wyrażenia lambda.

```
1  #include "tbb/tbb.h"
2
```

[1]Wyrażenia lambda wprowadzone zostały do standardu języka C++11. Umożliwiają one tworzenie funkcji anonimowych, tzn. takich, które posiadają ciało, lecz niemają nazwy.

```
3  using namespace tbb;
4
5  void ParallelApplyFoo( float* a, size_t n ) {
6     parallel_for( blocked_range<size_t>(0,n),
7        [=](const blocked_range<size_t>& r) {
8                         for(size_t i=r.begin(); i!=r.end(); ++i)
9                             Foo(a[i]);
10                    }
11    );
12 }
```

Listing 3.4: Wywołanie funkcji `parallel_for` używając wyrażenia lambda [8].

Zapis `[=]` rozpoczyna konstrukcję lambda, sugerując, że zmienne lokalne `a` oraz `n`, używane w funkcji, ale zdeklarowane poza nią, są przekazywane przez wartość. Notacja `[&]` sugerowałaby przekazywanie przez referencje. Za zapisem `[=]` znajduje się lista parametrów i definicja operatora `()` generowanego obiektu funkcji.

Funkcja `parallel_for` pozwala również na iterowanie zakresu podanego bezpośrednio w formie liczbowej (listing 3.5).

```
1  #include "tbb/tbb.h"
2
3  using namespace tbb;
4
5  void ParallelApplyFoo(float a[], size_t n) {
6      parallel_for(size_t(0), n, [=](size_t i) {Foo(a[i]);});
7  }
```

Listing 3.5: Funkcja `parallel_for` z wyrażeniem lambda oraz zakresem podanym w formie liczbowej [8].

3.3 Korzyści ze zrównoleglenia pętli

Proces dzielenia przestrzeni obliczeń na części zajmuje znaczący czas procesora. Dlatego też korzyści z używania `parallel_for` są zwykle dostrzegalne, gdy pętla do obliczeń potrzebuje przynajmniej miliona taktów zegara. Przykładowo pętla, która do obliczeń potrzebuje przynajmniej 500 mikrosekund na procesorze 2 GHz, skorzysta na efektywności z użycia funkcji `parallel_for`. Realne korzyści jakie daje zrównoleglenie obliczeń, uwidaczniają się przy badaniu wydajności implementowanego symulatora (podrozdział 6.5).

Aby zrównoważyć obciążenie procesorów, biblioteka TBB od wersji 2.2 automatycznie dzieli przestrzeń obliczeń, i taki sposób jest rekomendowany dla większości przypadków. Lecz są sytuacje, gdzie jawna kontrola podziału przestrzeni wpłynie na korzyść wydajności obliczeń. Dzieje się tak, ponieważ automatyczne próby zoptymalizowania obliczeń są czysto heurystyczne.

Rozdział 4

Grafika trójwymiarowa

Prezentacja wyników symulacji w postaci trójwymiarowej grafiki wiąże się dla programisty ze znajomością wielu zagadnień. Temat ten jest rozległy i obejmuje szereg aspektów. Dlatego w pracy tej przedstawione zostały jedynie podstawowe zasady i mechanizmy pozwalające prezentować w czasie rzeczywistym stan implementowanego symulatora. Zagadnienia poruszone w tym rozdziale opisane zostały na przykładzie biblioteki OpenGL (ang. *Open Graphics Library*).

4.1 Przestrzeń trójwymiarowa

Położenie punktu oraz jego translacja (przesunięcie) w trójwymiarowej przestrzeni mogą być zapisane za pomocą trójwymiarowego wektora. Do opisu przekształceń takich jak: obroty, skalowanie, odbicia czy pochylenie, należy użyć macierzy 3×3. Biblioteka OpenGL korzysta jednak z wektorów i macierzy we współrzędnych jednorodnych (patrz paragraf 4.1.1), które mają cztery wymiary. Za ich pomocą można opisać wszystkie przekształcenia, tzn nie tylko obroty, i skalowanie, ale również translacje i rzutowanie. Zatem wszystkie operacje graficzne na współrzędnych, oprócz podziału perspektywicznego, sprowadzają się do mnożenia przez macierz. Co więcej, wynikiem złożenia kilku przekształceń jest nadal jedna macierz 4×4. Przykładem może być obrót obiektu wokół osi znajdującej się poza nim. Przekształcenie takie należy złożyć z translacji, obrotu i translacji powrotnej, a mimo to może być zapisane jedną macierzą 4×4 [2].

4.1.1 Współrzędne jednorodne

Współrzędne jednorodne, poza trzema współrzędnymi kartezjańskimi x, y i z, obejmują także czwartą współrzędną w, nazywaną współczynnikiem skalowania. Jeżeli współczynnik ten jest równy jedności, pozostałe współrzędne mają takie znaczenie, jak ich odpowiedniki w trójwymiarowym układzie kartezjańskim. Na podobnej zasadzie, macierz 4×4, w której wartości w dolnej i prawej kolumnie równe są zeru, poza elementem diagonalnym równym jedności, działa tak samo, jak jej

podmacierz 3×3 stworzona przez skreślenie ostatniej kolumny i ostatniego wiersza [2].

4.1.2 Macierze

Do operacji na współrzędnych wykorzystuje się macierze modelu (ang. *model matrix*), widoku (ang. *view matrix*) i projekcji (ang. *projection matrix*).

Macierz modelu określa położenie i orientację obiektu względem sceny (świata). Przemnożenie przez tę macierz transformuje lokalny układ współrzędnych (ang. *local space*) obiektu do układu współrzędnych związanego ze światem (ang. *world space*).

Macierz widoku określa ustawienie kamery na scenie. Przemnożenie przez tę macierz skutkuje przekształceniem układu współrzędnych sceny do układu współrzędnych kamery (ang. *view space*).

Macierz rzutowania odpowiada za prezentację trójwymiarowej sceny na dwuwymiarowym ekranie. Przemnożenie przez tę macierz transformuje układ współrzędnych kamery do układu przycinania (ang. *clip space*).

4.1.3 Przekształcenia układu współrzędnych w potoku renderowania

Wierzchołki są przekazywane z aplikacji do karty graficznej poprzez potok renderowania (ang. *rendering pipeline*), który przekształca ich atrybuty w odpowiedni sposób. Współrzędne wierzchołków danego obiektu wyrażone są w jego lokalnym układzie współrzędnych. Mnożone są one przez macierz modelu i macierz widoku. Oznacza to opisanie obiektu we współrzędnych kamery. Następnie współrzędne wierzchołków mnożone są przez macierz rzutowania, co skutkuje przejściem do układu współrzędnych przycinania. Przekształcenia te przeprowadzane są w shaderze wierzchołków (ang. *vertex shader*, patrz paragraf 4.3.1) i muszą być kontrolowane przez programistę.

Dalszym etapem, wykonywanym automatycznie przez kartę graficzną, jest dzielenie perspektywiczne (ang. *perspective division*), w którym wszystkie współrzędne dzielone są przez współrzędną skalowania w. Wierzchołek zostaje więc wyrażony w unormowanych współrzędnych urządzenia (ang. *normalized device coordinates*), mających trzy wymiary.

Położenie uzyskane w shaderze wierzchołków (współrzędne obiektu, sceny, kamery lub przycinania) jest przekazywane do shadera fragmentów (ang. *fragment shader*, patrz paragraf 4.3.2), gdzie wykorzystując interpolację, obliczana jest jego wartość dla każdego piksela.

4.2 Biblioteka OpenGL

OpenGL jest niskopoziomowym i wieloplatformowym interfejsem programistycznym do tworzenia grafiki trójwymiarowej. OpenGL rozwijane jest przez konsorcjum

ARB (ang. *Architecture Review Board*), w skład którego wchodzą firmy zajmujące się oprogramowaniem oraz sprzętem komputerowym. Zapewnia to niezależność od jednej platformy programowej lub sprzętowej przy jednoczesnym uwzględnieniu najnowszych osiągnięć w dziedzinie grafiki komputerowej. Od 2006 roku ARB jest częścią organizacji Khronos Group Inc., która odpowiedzialna jest za rozwijanie otwartych standardów programistycznych (np. OpenCL[1]). Najnowszy standard biblioteki implementują m.in. firmy NVIDIA, AMD, Intel czy Apple. Wartą wspomnienia jest także biblioteka Mesa 3D – otwarta implementacja OpenGL na różne platformy. [3].

4.2.1 Biblioteki pomocnicze

Aby zapewnić wieloplatformowość oraz wysoką wydajność, OpenGL głównie pobiera i zapisuje odpowiednie dane do bufora karty graficznej. Natomiast nie dostarcza innych potrzebnych funkcjonalności takich jak tworzenie kontekstu graficznego, obsługą środowiska okienkowego, obsługę urządzeń wejścia-wyjścia, ładowanie plików do programu, czy działanie na wektorach i macierzach. Dlatego też koniecznym jest używanie dodatkowych bibliotek, które realizują brakujące funkcjonalności.

Przykładem biblioteki pomocniczej jest GLEW (ang. *OpenGL Extension Wrangler Library*). Zarządza ona rozszerzeniami[2] i funkcjami OpenGL. GLEW dostarcza mechanizmy do określania dostępnych rozszerzeń na danej platformie w czasie uruchamiania programu.

```
1 GLenum err = glewInit();
2
3 if (GLEW_OK != err)
4 {
5     fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
6 }
7 fprintf(stdout, "Status: Using GLEW %s\n",
    glewGetString(GLEW_VERSION));
```

Listing 4.1: Załadowanie rozszerzeń oraz funkcji OpenGL za pomocą biblioteki GLEW [10].

GLFW jest biblioteką pomocniczą, służącą do tworzenia oraz zarządzania oknami i kontekstami OpenGL, jak również do obsługi zdarzeń m.in. związanych z klawiaturą czy myszą. Funkcje biblioteki wykorzystywane w niniejszej pracy:

- `glfwInit` – inicjalizacja biblioteki GLFW;

[1]OpenCL (ang. *Open Computing Language*) – platforma programistyczna wspomagająca pisanie aplikacji działających na heterogenicznych platformach składających się z różnego rodzaju jednostek obliczeniowych takich jak np. CPU (ang. *Central Processing Unit*) czy GPU (ang. *Graphics Processing Unit)*.

[2]Rozszerzenia określają dodatkowe możliwości biblioteki i mogą dotyczyć różnych jej elementów. Pozwalają na korzystanie z najnowszych rozwiązań w grafice komputerowej.

- `glfwTerminate` – wyczyszczenie wszystkich zaalokowanych zasobów przez bibliotekę GLFW;
- `glfwCreateWindow` – tworzenie okna i związanego z nim kontekstu OpenGL;
- `glfwSetWindowTitle` – ustawienie tytułu wybranego okna;
- `glfwWindowHint` – ustawienie wybranej opcji na zadaną wartości, przykładowe opcje pokazuje tabela 4.1;
- `glfwMakeContextCurrent` – ustawienie kontekstu do wybranego okna;
- `glfwSetWindowShouldClose` – ustawienie odpowiedniej flagi zamknięcia okna;
- `glfwGetPrimaryMonitor` – pobranie obiektu głównego monitora;
- `glfwGetVideoMode` – pobranie aktualnego trybu wideo wybranego monitora;
- `glfwSwapBuffers` – zamiana przedniego i tylnego bufora wybranego okna, co skutkuje wyświetleniem następnej klatki;
- `glfwPollEvents` – przetworzenie zdarzeń będących w kolejce zdarzeń;
- `glfwSetInputMode` – ustawienie trybu zdarzeń pochodzących z myszy, bądź klawiszy;
- `glfwSetKeyCallback` – ustawienie funkcji wywoływanej przy zdarzeniu związanym z klawiszami;
- `glfwSetCursorPosCallback` – ustawienie funkcji wywoływanej przy zdarzeniu związanym z ruchem myszy;
- `glfwSetScrollCallback` – ustawienie funkcji wywoływanej przy zdarzeniu związanym ze skrolowaniem myszy;
- `glfwGetTime` – pobranie wartości zegara rozpoczynającego swoje działanie przy inicjalizacji GLFW.

Opcja	**Przykładowa wartość**	**Opis**
`GLFW_CONTEXT_VERSION_MAJOR`	`4`	Główna wersja biblioteki OpenGL
`GLFW_CONTEXT_VERSION_MINOR`	`5`	Mniejsza wersja biblioteki OpenGL
`GLFW_OPENGL_PROFILE`	`GLFW_OPENGL_CORE_PROFILE`	Profil OpenGL[4]
`GLFW_REFRESH_RATE`	`GLFW_DONT_CARE`	Częstotliwość odświeżania pełnego okna
`GLFW_RESIZABLE`	`GL_FALSE`	Określa, czy użytkownik może zmieniać rozmiar okna

Tabela 4.1: Przykładowe opcje GLFW.

```
1  #include <GLFW/glfw3.h>
2
3  int main(void)
4  {
5      GLFWwindow* window;
6
7      if (!glfwInit())
8          return -1;
9
10     window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
11     if (!window)
12     {
13         glfwTerminate();
14         return -1;
15     }
16     glfwMakeContextCurrent(window);
17
18     while (!glfwWindowShouldClose(window))
19     {
20         glClear(GL_COLOR_BUFFER_BIT);
21         glfwSwapBuffers(window);
22         glfwPollEvents();
23     }
24
25     glfwTerminate();
26     return 0;
27 }
```

Listing 4.2: Zarządzanie oknem oraz kontekstem za pomocą biblioteki GLFW [11].

Innym przykładem biblioteki pomocniczej jest biblioteka matematyczna GLM (ang. *OpenGL Mathematics*). Jest oparta na specyfikacji języka GLSL (ang. *OpenGL Shading Language*, patrz podrozdział 4.3), dzieląc tę samą nomenklaturę i funkcjonalności. Biblioteka zapewnia m.in. działania na wektorach (np. typu `vec3`) i macierzach (np. typ `mat4`). GLM została zaprojektowana do współpracy z OpenGL, aczkolwiek nadaje się również do działania z innymi bibliotekami i środowiskami w których przetwarzane są obrazy, symulowane są zjawiska fizyczne, czy po prostu wykonuje się obliczenia matematyczne. Wybrane funkcje biblioteki wykorzystywane w niniejszej pracy:

- `radians` – zamiana stopni na radiany;
- `normalize` – obliczenie wersora podanego wektora;
- `length` – obliczenie długości podanego wektora;
- `cross` – obliczenie iloczynu wektorowego podanych wektorów;
- `value_ptr` – zwrócenie wskaźnika do podanego obiektu.

```
1  #include <glm/glm.hpp>
2  #include <glm/gtc/matrix_transform.hpp>
3  #include <glm/gtc/type_ptr.hpp>
4
5  void setUniformVector (GLuint Location, glm::vec3 const & Translate,
       glm::vec3 const & Rotate)
6  {
7      glm::mat4 Projection = glm::perspective(45.0f, 4.0f/3.0f, 0.1f,
           100.f);
8      glm::mat4 ViewTranslate = glm::translate(glm::mat4(1.0f),
           Translate);
9      glm::mat4 ViewRotateX = glm::rotate(ViewTranslate, Rotate.y,
           glm::vec3(-1.0f, 0.0f, 0.0f));
10     glm::mat4 View = glm::rotate(ViewRotateX, Rotate.x,
           glm::vec3(0.0f, 1.0f, 0.0f));
11     glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));
12     glm::mat4 MVP = Projection * View * Model;
13
14     glUniformMatrix4fv(Location, 1, GL_FALSE, glm::value_ptr(MVP));
15 }
```

Listing 4.3: Przykładowe użycie biblioteki GLM [12].

SOIL (ang. *Simple OpenGL Image Library*) jest biblioteką pomocniczą do zarządzania plikami graficznymi, pozwalającą ładować tekstury. Może być używana do zapisywania i ładowania obrazów w różnych formatach.

```
1  int width, height, channels;
2
3  unsigned char *ht_map = SOIL_load_image ( "terrain.tga", &width,
       &height, &channels, SOIL_LOAD_L );
4  int save_result = SOIL_save_image ( "new_terrain.dds",
       SOIL_SAVE_TYPE_DDS, width, height, channels, ht_map );
5
6  SOIL_free_image_data(ht_map);
```

Listing 4.4: Załadowanie mapy wysokości i zapisanie jej w innym rozszerzeniu za pomocą biblioteki SOIL [13].

4.2.2 Obiekty bufora

W bibliotece OpenGL wykorzystuje się odpowiednie obiekty, które posiadają zdefiniowane funkcjonalności i najczęściej oznaczone są unikatowymi identyfikatorami typu `GLuint`. Obiekty bufora (ang. *buffer object*) służą do przechowywania i zarządzania danymi w przestrzeni adresowej konkretnej implementacji biblioteki.

Po utworzeniu identyfikatora funkcją `void glGenBuffers(GLsizei n, GLuint* buffers)`, która wpisuje `n` identyfikatorów do tablicy wskazywanej przez wskaźnik `buffers`, należy dokonać tzw. powiązania (ang. *bind*) obiektu z jego identyfikatorem; powiązanie te tworzone jest po wywołaniu funkcji `void glBindBuffer(GLenum target, GLuint buffer)`, w której podaje się rodzaj

obiektu bufora oraz wcześniej utworzony identyfikator. Przykładowym obiektem bufora jest VBO (ang. *vertex buffer object*), który przechowuje atrybuty wierzchołków (parametr `target` powinien wówczas przyjąć wartość równą `GL_ARRAY_BUFFER`). Każde następne wywołanie funkcji `glBindBuffer` z powiązanym wcześniej identyfikatorem, przełącza między różnymi obiektami bufora. Do tak utworzonego obiektu bufora należy załadować dane. Może to być zrealizowane na różne sposoby, aczkolwiek najczęściej używa się do tego funkcji `void glBufferData(GLenum target, GLsizeiptr size, const GLvoid* data, Glenum usage)`, w której dane kopiowane są bezpośrednio z aplikacji OpenGL do bieżącego obiektu bufora. Funkcja ta przyjmuje argumenty określające rodzaj bufora obiektów, rozmiar danych obiektu, miejsce danych w pamięci, oraz spodziewany sposób wykorzystania obiektu bufora.

4.2.3 Wierzchołki i ich atrybuty

Prymitywy geometryczne, z których korzysta OpenGL, składają się z wierzchołków i ich atrybutów. Przykładowymi atrybutami są współrzędne położenia, składowe kolorów, wektory normalne, czy współrzędne tekstury. Atrybuty wierzchołków składowane są we wspomnianych w paragrafie 4.2.2 obiektach bufora wierzchołków VBO. Oprócz nich istnieją także tablice wierzchołków VA (ang. *vertex arrays*), które opisują sposób zorganizowania danych znajdujących się w VBO. Aby zdefiniować tablicę wierzchołków używa się funkcji `void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLsizei stride, const GLvoid* pointer)`, w której podaje się kolejno numer indeksu atrybutu wierzchołka, liczbę składowych atrybutu, typ danych atrybutów wierzchołka, odstęp między danymi poszczególnych atrybutów wierzchołków oraz położenie w pamięci obiektu bufora VBO pierwszego atrybutu definiowanej tablicy wierzchołków. Po zdefiniowaniu tablicy wierzchołków, należy ją aktywować funkcją `void glEnableVertexAttribArray(GLuint index)`.

Tablice wierzchołków VA należy zgrupować do obiektów VAO (ang. *Vertex Array Objects*) i przypisać im unikalne identyfikatory, aby można było nimi zarządzać. Osiąga się to poprzez wywołanie funkcji `void glGenVertexArrays(GLsizei n, GLuint *arrays)`, która generuje obiekty tablicy wierzchołków VAO oraz funkcji `void glBindVertexArray(GLuint array)`, która ustawia tak utworzony obiekt tablicy wierzchołków.

4.3 Shadery

Shadery to programy wykonywane na karcie graficznej. Uruchamiane są w ramach potoku renderowania. Oprócz opisanych i wykorzystywanych w tej pracy shaderów wierzchołków i fragmentów, istnieją inne typy shaderów: teselacji (ang. *tessellation shader*), ewaluacji (ang. *evaluation shader*), geometrii (ang. *geometry shader*) oraz obliczeń (ang. *compute shader*).

Do pisania shaderów w OpenGL 4.5 wykorzystuje się język GLSL, który składniowo przypomina język C. GLSL zawiera typy przydatne w programowaniu grafiki trójwymiarowej wraz z operatorami oraz funkcjami, które na nich działają. Najistotniejsze z nich to typy wektorowe (np. wektor czterowymiarowy `vec4`), macierzowe (np. macierz czterowymiarowa `mat4`) i teksturowe (np. tekstura dwuwymiarowa `sampler2D`).

Shader wierzchołków jest uruchamiany osobno dla każdego wierzchołka z bufora wierzchołków, wykonując dokładnie ten sam kod. Działanie tego shadera w żaden sposób nie zależy od wyników uzyskanych dla innych wierzchołków. To samo dotyczy shadera fragmentów, który niezależnie operuje na poszczególnych pikselach prymitywu. Dzięki temu karty graficzne mogą wykonywać obliczenia równolegle na wielu rdzeniach.

Do każdego wywołania shadera wierzchołków lub shadera fragmentów przesyłane są atrybuty pojedynczego wierzchołka lub piksela. Atrybuty te muszą być zadeklarowane w kodzie shaderów. Ponieważ są to zmienne wejściowe, oznacza się je modyfikatorem `in`. Atrybuty zwracane na wyjściu również muszą być zadeklarowane i oznacza się je modyfikatorem `out`. Ponadto, wyjścia shadera wierzchołków powinny odpowiadać wejściom shadera fragmentów (zakładając użycie tylko tych dwóch shaderów). Działanie shaderów może być kontrolowane przy pomocy parametrów podawanych z poziomu programu wykonywanego na CPU. Parametry te są zmiennymi oznaczonymi modyfikatorem `uniform`. Ich wartość jest taka sama we wszystkich wywołaniach shaderów.

4.3.1 Shader wierzchołków

Shader wierzchołków operuje na wierzchołkach i ich atrybutach. Stanowi więc on połączenie między danymi wierzchołków opisanymi z poziomu aplikacji OpenGL a programowalną częścią potoku renderingu. Shader wierzchołków może być jedynym shaderem zdefiniowanym w programowalnym etapie renderingu, jak też współpracować bezpośrednio z innymi shaderami. Shadery w potoku mogą występować w różnych zestawieniach, lecz w każdym wypadku do prawidłowego działania potoku wymagany jest shader wierzchołków. Posiada on predefiniowane zmienne wyjściowe, których kontrola jest opcjonalna. Przykładem może być zmienna `gl_Position` zawierająca współrzędne przetwarzanego wierzchołka prymitywu w przestrzeni obcinania lub zmienna `gl_PointSize` mówiąca o rozmiarze punktów w pikselach.

Kod przykładowego shadera wierzchołków (listing 4.5) zaczyna się od deklaracji wersji języka. Shader ten przypisuje do predefiniowanej zmiennej wyjściowej `gl_Position` wynik mnożenia czterowymiarowej macierzy przez zmienną wejściową wektora trójelementowego `position_in` uzupełnionego o współrzędną `w`. Macierz `model-view-projection` (która powinna być złożeniem macierzy modelu, widoku i projekcji) jest zmienną oznaczoną modyfikatorem `uniform`, a więc powinna być przygotowana w programie wykonywanym na CPU.

```
#version 450 core

in vec3 position_in;
uniform mat4 model-view-projection;

void main()
{
    gl_Position = model-view-projection * vec4(position_in, 1.0);
}
```

Listing 4.5: Przykładowy shader wierzchołków.

4.3.2 Shader fragmentów

Shader Fragmentów jest odpowiedzialny za wyliczanie koloru pikseli fragmentów będących wynikiem procesu rasteryzacji prymitywów (punktów, odcinków i trójkątów). Rasteryzacja przekształca idealne prymitywy jako fragmenty, które mogą być wyświetlone na ekranie o skończonej rozdzielności. Pokazuje to rysunek 4.1.

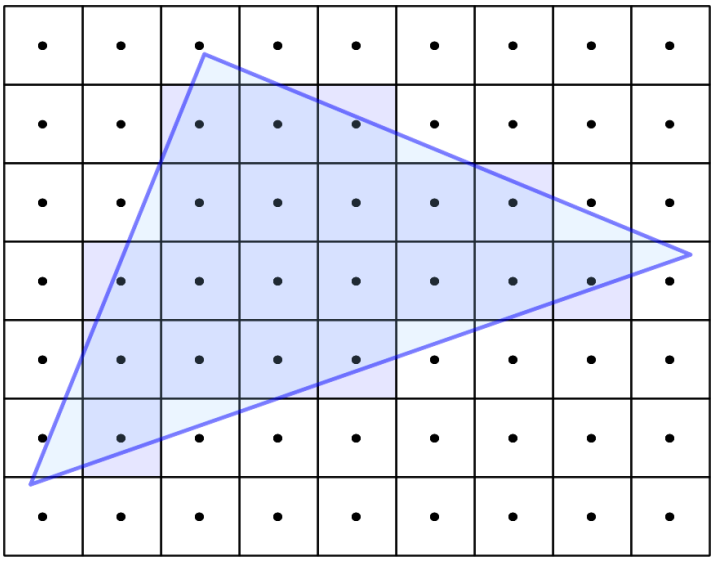
Shadery fragmentów, oprócz wbudowanych zmiennych wejściowych (np. `gl_PointCoord`, która informuje o wartości współrzędnych wewnątrz punktu), mogą przetwarzać zmienne wejściowe zdefiniowane przez programistę, pochodzące z poprzedniego programowalnego etapu renderingu. Są one poddawane procesowi interpolacji, który polega na odpowiednim uśrednieniu ich wartości.

```
#version 450 core

out vec4 color_out;

void main()
{
    color_out = vec4(1, 1, 0, 1);
}
```

Listing 4.6: Przykładowy shader fragmentów, który zwraca wektor określający żółty kolor.



Rysunek 4.1: Rasteryzacja trójkąta [14].

Rozdział 5

Implementacja

Motywem przewodnim pracy jest implementacja symulatora wielu ciał, który do obliczania dynamiki obiektów wykorzystuje obliczenia równoległe, a do prezentacji przebiegu symulacji stosuje grafikę trójwymiarową. W niniejszym rozdziale przedstawiony został sposób implementacji symulatora, omawiając koncepcję i założenia, wymieniając użyte narzędzia, oraz opisując logikę programu, użyte struktury i funkcje.

5.1 Koncepcja i założenia

Symulator przy uruchomieniu przyjmuje opcjonalne parametry wejściowe, które wpływają na przebieg symulacji. Podczas jego działania możliwa jest modyfikacja wybranych parametrów. Dzięki temu w czasie rzeczywistym można obserwować zmiany zachodzące po uaktualnieniu warunków symulacji. Oprócz informowaniu na standardowym strumieniu wyjścia o bieżącym stanie symulacji, można poruszać i rozglądać się po scenie z perspektywy pierwszej osoby (ang. *first-person perspective*, FPP). Ponadto tytuł okna symulatora zawiera aktualną liczbę klatek na sekundę.

Do obliczania sił działających między ciałami wykorzystywane jest prawo powszechnego ciążenia (wzór 2.1). Wyznaczanie przyspieszenia ciał odbywa się za pomocą drugiej zasady dynamiki Newtona (wzór 2.4). Natomiast ruch ciał obliczany jest metodą Eulera (wzór 2.5).

Cząsteczki zagregowane są w strukturze tablicowej. Wszystkie posiadają taka samą masę, a położenie początkowe każdej cząsteczki jest losowane w zależności od wybranego trybu. Sposób wyznaczania ruchu ciał również jest podzielony na tryby: sekwencyjny oraz równoległy, który wykonuje obliczenia wykorzystując bibliotekę TBB (patrz podrozdział 3.1).

Do prezentacji przebiegu symulacji wykorzystywana jest biblioteka OpenGL (patrz podrozdział 4.2). Ciała reprezentowane są jako punkty, które nie kolidują ze sobą. Na każdy punkt nakładana jest tekstura reprezentująca cząsteczkę. Została ona tak dobrana, aby wykorzystując syntezę addytywną kolorów[1], wyraźnie było widać zwiększone zagęszczenie się cząsteczek w przestrzeni.

[1]Synteza addytywna jest zjawiskiem mieszania barw poprzez ich sumowanie.

5.2 Użyte narzędzia

Do implementacji symulatora wykorzytano język C++ (standard ISO/IEC 14882:2014), język GLSL (wersja 4.50) oraz biblioteki TBB (wersja 4.4), OpenGL (wersja 4.5), GLEW (wersja 2.0.0), GLFW (wersja 3.2.1), GLM (wersja 0.9.8.3) i SOIL (wersja z 7 lipca 2008).

5.3 Cząsteczki

Ciała reprezentowane przez cząsteczki są zgrupowanymi w tablicy obiektami struktury `Particle` pokazanej na listingu 5.1. Struktura ta składa się z następujących pól:

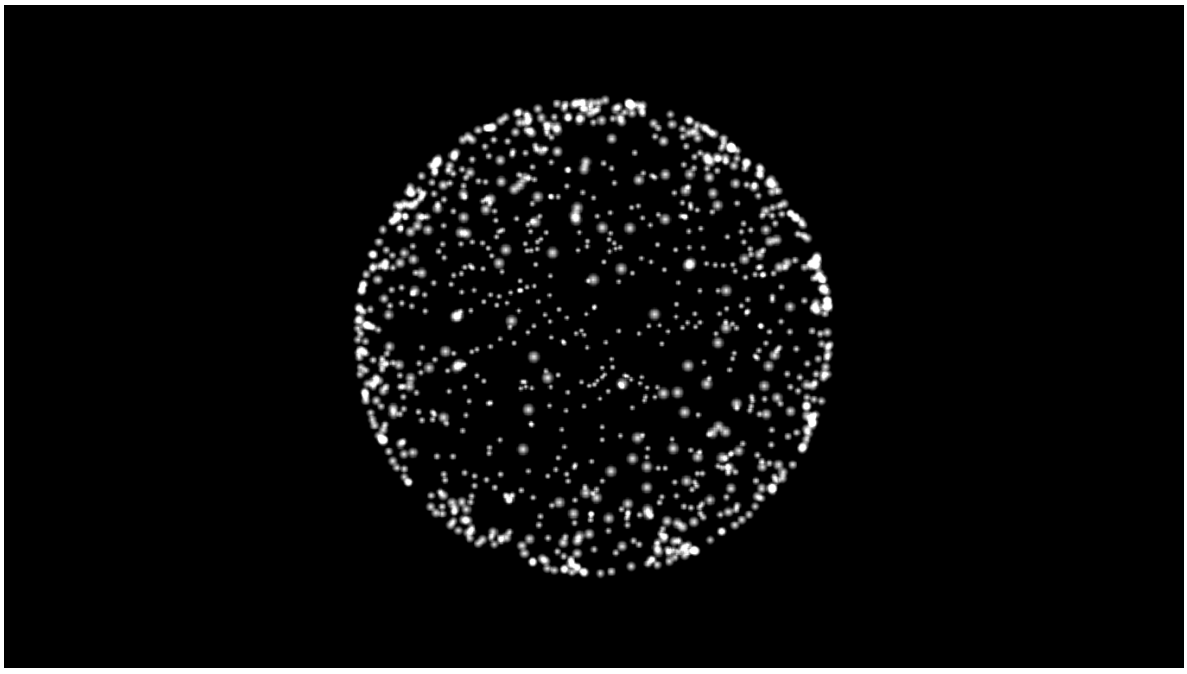
- `float mass` – masa cząsteczki;
- `glm::vec3 location` – położenie cząsteczki w trójwymiarowym układzie współrzędnych;
- `glm::vec3 velocity` – trójwymiarowy wektor prędkości cząsteczki;
- `glm::vec3 acceleration` – trójwymiarowy wektor przyspieszenia cząsteczki;
- `glm::vec3 force` – trójwymiarowy wektor siły wypadkowej działającej na cząsteczkę.

```
1  struct Particle
2  {
3      float mass;
4      glm::vec3 location;
5      glm::vec3 velocity;
6      glm::vec3 acceleration;
7      glm::vec3 force;
8
9      Particle() {}
10
11     Particle(float mass, int initial_location)
12     {
13         this->mass = mass;
14
15         switch(initial_location)
16         {
17             case 0:
18                 location = glm::vec3(glm::sphericalRand(1.5f)); break;
19             case 1:
20                 location = glm::vec3(glm::gaussRand(glm::vec3(0),
                       glm::vec3(1))); break;
21             case 2:
22                 location = glm::vec3(sin(rand()), sin(rand()),
                       sin(rand())); break;
```

```
23            }
24        }
25
26    };
```

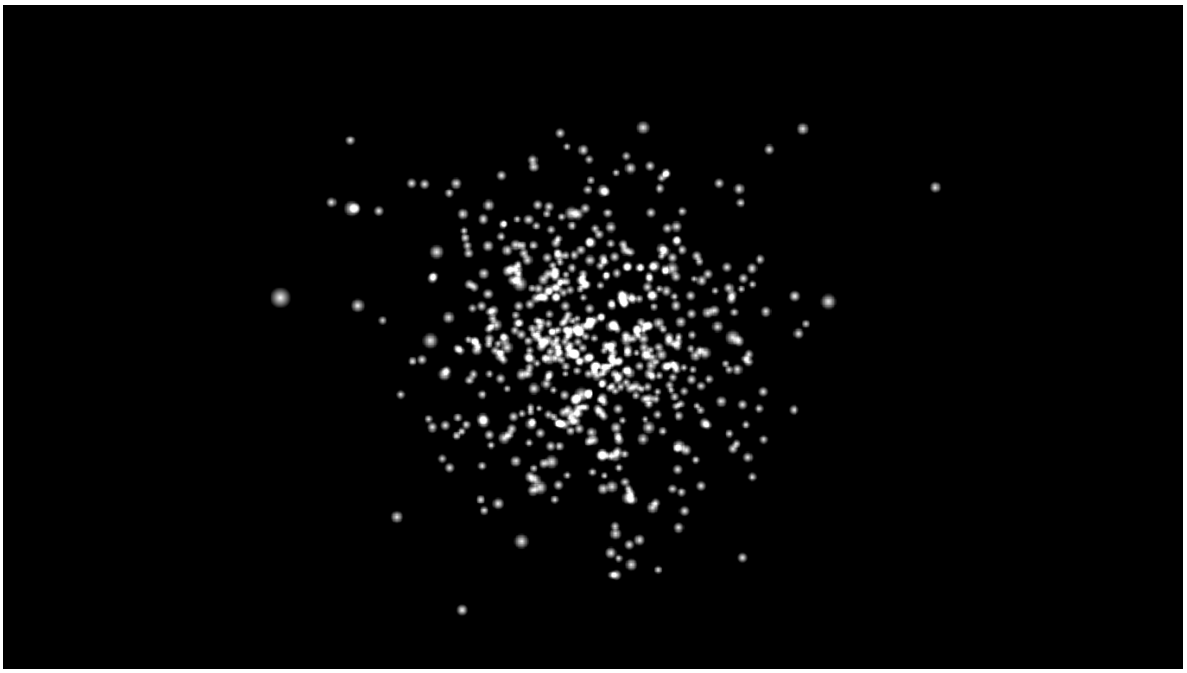
Listing 5.1: Definicja struktury `Particle`.

Oprócz konstruktora domyślnego, nieprzyjmującego parametrów, struktura `Particle` posiada konstruktor przyjmujący jako parametry masę, oraz tryb położenia początkowego cząsteczki. Tryb ten decyduje o sposobie losowania położenia początkowego cząsteczki. Możliwe są trzy tryby do wyboru. Pierwszym jest tryb sferyczny pokazany na rysunku 5.1, w którym położenie początkowe cząsteczki, dzięki funkcji `sphericalRand`, wypada losowo na powierzchni kuli o zadanym promieniu. Drugim trybem jest tryb Gauss'a pokazanym na rysunku 5.2, w którym położenie początkowe cząsteczki losowane jest funkcją `gaussRand` według rozkładu normalnego[2]. Ostatnim trybem jest tryb sześcianu pokazany na rysunku 5.3, w którym każda współrzędna położenia początkowego cząsteczki jest wynikiem funkcji `sin` z losowej wartości.

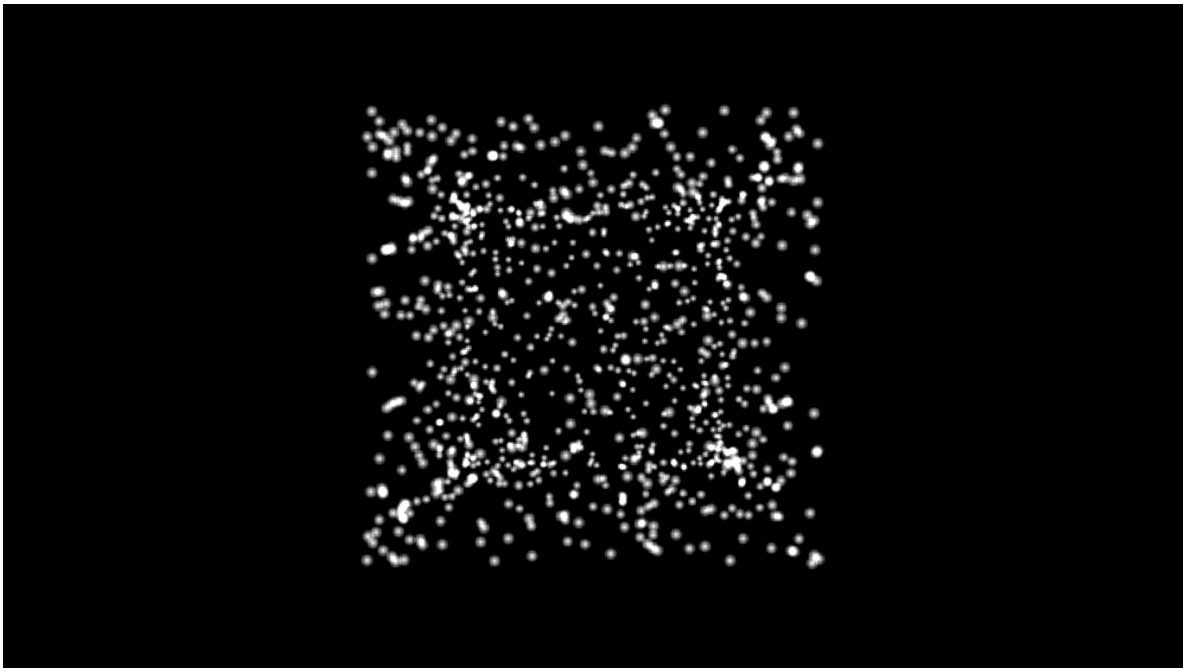


Rysunek 5.1: Tryb sferyczny położenia początkowego cząsteczek.

[2]Rozkład normalny, zwany inaczej rozkładem Gauss'a, jest jednym z najważniejszych rozkładów prawdopodobieństwa. Opisuje on sytuacje w świecie, gdzie większość przypadków jest bliska średniemu wynikowi, a im dany wynik bardziej odchyla się od średniej tym jest mniej reprezentowany [15].



Rysunek 5.2: Tryb Gauss'a położenia początkowego cząsteczek.



Rysunek 5.3: Tryb sześcianu położenia początkowego cząsteczek.

5.4 Wyznaczanie położeń cząsteczek

Wyznaczanie położenia cząsteczek odbywa się w funkcji `ComputeNextPositions` (listing 5.2) i jest obliczane dwuetapowo. Pierwszym etapem jest obliczenie siły wy-

padkowej działającej na każde ciało. Liczbę par ciał, między którymi obliczana jest siła, można wyznaczyć kombinacją bez powtórzeń daną wzorem 2.3. Siła między każda parą ciał obliczana jest za pomocą wzoru 2.1, wykorzystującego stałą grawitacji, masę cząsteczek, dystans między nimi[3], oraz wersor wektora łączącego środki mas tych ciał. Obliczony wektor siły dodawany jest do siły wypadkowej działającej na ciało `i`, a odejmowany od siły wypadkowej działającej na ciało `j` (patrz komentarz do wzoru 2.2 w podrozdziale 2.1). Wykorzystane funkcje biblioteki GLM opisane są w paragrafie 4.2.1. Drugim etapem jest obliczenie następnego położenia każdego ciała na podstawie jego siły wypadkowej. W etapie tym wykorzystywana jest metoda Eulera (wzór 2.5). Oba równania pierwszego stopnia, na prędkość i położenie, wyznaczane są korzystając z ilorazu różnicowego: wartość położenia i prędkości zwiększana jest o wartość ich pochodnej pomnożonej przez krok czasowy [1].

```
1  void ComputeNextPositions(GLuint vbo, Particle* p, int N, GLfloat
       delta_time, float G = 1.0f)
2  {
3      for (int i = 0; i < N - 1; i++)
4      {
5          for (int j = i+1; j < N; j++)
6          {
7              glm::vec3 location_difference = p[j].location -
                   p[i].location;
8              float distance = glm::length(location_difference) + 0.1f;
9              glm::vec3 force = (G * p[i].mass * p[j].mass) / (distance
                   * distance) * glm::normalize(location_difference);
10
11             p[i].force += force;
12             p[j].force -= force;
13         }
14     }
15
16     for (int i = 0; i < N; i++)
17     {
18         p[i].acceleration = p[i].force / p[i].mass;
19         p[i].velocity += p[i].acceleration * delta_time;
20         p[i].location += p[i].velocity * delta_time;
21
22         p[i].force = glm::vec3(0, 0, 0);
23     }
24 }
```

Listing 5.2: Sekwencyjne obliczanie sił wypadkowych i wyznaczanie następnych pozycji ciał.

Oba etapy zostały zrównoleglone w funkcji `ComputeNextPositions_TBB_Outer` (listing 5.3). Wykorzystano do tego funkcję `parallel_for` biblioteki TBB przyjmującej w parametrach zakres liczbowy przestrzeni obliczeń, oraz funkcję anonimową w postaci wyrażenia lambda (patrz podrozdział 3.2).

[3]Z uwagi na stabilność symulacji przyjęto pewien określony rozmiar punktu materialnego ograniczający minimalną odległość. W praktyce oznacza to powiększenie dystansu między cząsteczkami o `0.1`.

```
1  void ComputeNextPositions_TBB_Outer(GLuint vbo, Particle* p, int N,
       GLfloat delta_time, float G = 1.0f)
2  {
3      tbb::parallel_for(0, N-1, [&](int i) {
4          for (int j = i+1; j < N; j++)
5          {
6              glm::vec3 location_difference = p[j].location -
                   p[i].location;
7              float distance = glm::length(location_difference) + 0.1f;
8              glm::vec3 force = (G * p[i].mass * p[j].mass) / (distance
                   * distance) * glm::normalize(location_difference);
9
10             p[i].force += force;
11             p[j].force -= force;
12         }
13     });
14
15     tbb::parallel_for(0, N, [&](int i)
16     {
17         p[i].acceleration = p[i].force / p[i].mass;
18         p[i].velocity += p[i].acceleration * delta_time;
19         p[i].location += p[i].velocity * delta_time;
20
21         p[i].force = glm::vec3(0, 0, 0);
22     });
23 }
```

Listing 5.3: Funkcja `ComputeNextPositions_TBB_Outer` odpowiedzialna za równoległe wyznaczanie następnych pozycji ciał.

5.5 Shadery

Do obsługi shaderów służy struktura `Shader` pokazana na listingu 5.4. Posiada ona pole `GLuint program` z identyfikatorem obiektu programu z którym wiązane są obiekty shaderów. Obiekty te są tworzone, kompilowane oraz linkowane w konstruktorze, który przyjmuje jako argumenty ścieżki do plików źródłowych shadera wierzchołków i shadera fragmentów.

```
1  struct Shader
2  {
3      GLuint program;
4
5      Shader(const GLchar* vertex_path, const GLchar* fragment_path);
6  };
```

Listing 5.4: Struktura Shader

Listing 5.5 pokazuje wpisywanie shaderów do tablic znaków za pomocą klas `string`, `ifstream` i `stringstream` z biblioteki standardowej.

```
1  std::string vertex_code;
2  std::string fragment_code;
3  std::ifstream vertex_file;
4  std::ifstream fragment_file;
5
6  vertex_file.exceptions(std::ifstream::badbit);
7  fragment_file.exceptions(std::ifstream::badbit);
8  try
9  {
10     vertex_file.open(vertex_path);
11     fragment_file.open(fragment_path);
12     std::stringstream vertex_stream, fragment_stream;
13
14     vertex_stream << vertex_file.rdbuf();
15     fragment_stream << fragment_file.rdbuf();
16
17     vertex_file.close();
18     fragment_file.close();
19
20     vertex_code = vertex_stream.str();
21     fragment_code = fragment_stream.str();
22 }
23 catch (std::ifstream::failure e)
24 {
25     std::cout << "ERROR - read file failed" << std::endl;
26 }
27
28 const GLchar* vertex_code_chars = vertex_code.c_str();
29 const GLchar* fragment_code_chars = fragment_code.c_str();
```

Listing 5.5: Fragment konstruktora struktury `Shader`, wpisujący shadery do tablic znaków.

Kompilację shaderów pokazuje listing 5.6. Na początku tworzony jest obiekt shadera wierzchołków funkcją `glCreateShader`, wskazywany jest kod źródłowy dzięki funkcji `glShaderSource`. Tak utworzony obiekt jest kompilowany funkcją `glCompileShader`. Na końcu sprawdzana jest poprawność procesu funkcją `glGetShaderiv`. Shader fragmentów tworzony jest w analogiczny sposób.

```
1  GLuint vertex, fragment;
2  GLint success;
3  GLchar info_log[512];
4
5  vertex = glCreateShader(GL_VERTEX_SHADER);
6  glShaderSource(vertex, 1, &vertex_code_chars, NULL);
7  glCompileShader(vertex);
8
9  glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
10 if (!success)
11 {
12     glGetShaderInfoLog(vertex, 512, NULL, info_log);
13     std::cout << "ERROR - vertex shader compilation failed:\n" <<
          info_log << std::endl;
```

```
14 }
15
16 fragment = glCreateShader(GL_FRAGMENT_SHADER);
17 glShaderSource(fragment, 1, &fragment_code_chars, NULL);
18 glCompileShader(fragment);
19
20 glGetShaderiv(fragment, GL_COMPILE_STATUS, &success);
21 if (!success)
22 {
23     glGetShaderInfoLog(fragment, 512, NULL, info_log);
24     std::cout << "ERROR - fragment shader compilation failed:\n" <<
          info_log << std::endl;
25 }
```

Listing 5.6: Fragment konstruktora struktury `Shader` odpowiedzialny za kompilacje shaderów.

Wiązanie shaderów z programem pokazane jest na listingu 5.7. Funkcja `glCreateProgram` tworzy obiekt programu, z którym następnie wiązane są funkcją `glAttachShader` obiekty shaderów. Linkowanie programu odbywa się za pomocą funkcji `glLinkProgram`. Na końcu sprawdzana jest poprawność linkowania funkcją `glGetProgramiv`.

```
1  this->program = glCreateProgram();
2  glAttachShader(this->program, vertex);
3  glAttachShader(this->program, fragment);
4  glLinkProgram(this->program);
5
6  glGetProgramiv(this->program, GL_LINK_STATUS, &success);
7  if (!success)
8  {
9      glGetProgramInfoLog(this->program, 512, NULL, info_log);
10     std::cout << "ERROR - shader link failed:\n" << info_log <<
           std::endl;
11 }
```

Listing 5.7: Fragment konstruktora struktury `Shader`. Wiązanie shaderów z programem oraz linkowanie programu.

Kod shadera wierzchołków używany w symulacji pokazuje listing 5.8. Shader ten przyjmuje na wejściu pozycje cząsteczki `position_in` w trójwymiarowym układzie współrzędnych. Argumentem podawanym z poziomu aplikacji jest macierz modelu-widoku-pozycji `mvp` o wymiarach 4×4 (utworzenie macierzy oraz jej przesłanie do shadera opisane jest w podrozdziale 5.7). Macierz ta mnożona jest przez pozycję cząsteczki, co zmienia położenie ciała we współrzędnych ekranu (lecz nie we współrzędnych świata). Następnie w zależności od współrzędnej `z` cząsteczki (czyli jej odległości od kamery) ustawiana jest wielkość punktu. Dzięki temu obiekty są tym mniejsze, im dalej leżą od kamery, co daje efekt głębi. `gl_Position` oraz `gl_PointSize` są predefiniowanymi zmiennymi wyjściowymi i opisane są w paragrafie 4.3.1.

```
#version 450 core

in vec3 position_in;

uniform mat4 mvp;

void main()
{
    vec4 pos = mvp * vec4(position_in, 1.0f);
    gl_Position = pos;
    gl_PointSize = 50 * (1/pos.z);
}
```

Listing 5.8: Shader wierzchołków.

Kod shadera fragmentów używany w symulacji pokazuje listing 5.9. Parametrami podawanym z poziomu aplikacji OpenGL są: dwuwymiarowa tekstura cząsteczki `tex` i trójwymiarowy wektor koloru `color`. Wyjściem shadera jest kolor piksela zależny od tekstury (a konkretnie koloru piksela tekstury na współrzędnych przekazanych przez predefiniowaną zmienną wejściową `gl_PointCoord`) i podanego koloru. Efektem mnożenia koloru piksela tekstury i podanego koloru jest podany kolor z różną, zależną od współrzędnych tekstury przezroczystością. Jest tak ponieważ wszystkie piksele w teksturze są białe i różnią się tylko przezroczystością (tekstura cząsteczki pokazana jest na rysuneku 5.4).

Skutkiem działania shadera fragmentów jest nałożenie dwuwymiarowej grafiki o wybranym kolorze na punkt reprezentujący cząsteczkę. Technika ta, nazywana po angielsku *point sprites*, świetnie nadaje się do systemów cząsteczek (ang. *particle systems*).

```
#version 450 core

out vec4 color_out;

uniform sampler2D tex;
uniform vec3 color;

void main()
{
    color_out = texture(tex, gl_PointCoord) * vec4(color, 1);
}
```

Listing 5.9: Shader fragmentów.

5.6 Kamera

Za odpowiednie wyświetlanie sceny w głównej mierze odpowiada obiekt kamery zdefiniowany strukturą `Camera` (listing 5.10). Struktura ta zawiera następujące pola:

- `glm::vec3 world_up` – kierunek osi y układu współrzędnych świata;
- `glm::vec3 position` – pozycja kamery na scenie;
- `glm::vec3 front` – wersor osi z układu współrzędnych kamery w układzie świata;
- `glm::vec3 up` – wersor osi y układu współrzędnych kamery w układzie świata;
- `glm::vec3 right` – wersor osi z układu współrzędnych kamery w układzie świata;
- `GLfloat yaw` – kąt odchylenia kamery;
- `GLfloat pitch` – kąt nachylenia kamery;
- `GLfloat movement_speed` – szybkość ruchu kamery;
- `GLfloat mouse_sensitivity` – czułość ruchów myszy, które wpływają na orientację kamery;
- `GLfloat fov` – kąt widzenia kamery (ang. *field of view*).

```
1  struct Camera
2  {
3      glm::vec3 world_up;
4      glm::vec3 position;
5      glm::vec3 front;
6      glm::vec3 up;
7      glm::vec3 right;
8
9      GLfloat yaw;
10     GLfloat pitch;
11
12     GLfloat movement_speed;
13     GLfloat mouse_sensitivity;
14     GLfloat fov;
15
16     Camera();
17
18     glm::mat4 GetViewMatrix();
19
20     void ProcessKeyboard(DIRECTION direction, GLfloat delta_time);
21
22     void ProcessMouseMovement(GLfloat x_offset, GLfloat y_offset);
23
24     void ProcessMouseScroll(GLfloat y_offset);
25
26     void UpdateCameraVectors();
27 };
```

Listing 5.10: Struktura `Camera`.

Przypisanie wartości domyślnych obiektowi kamery odbywa się w jej konstruktorze (listing 5.11).

```
Camera()
{
    this->world_up = glm::vec3(0.0f, 1.0f, 0.0f);
    this->position = glm::vec3(0.0f, 0.0f, 3.0f);
    this->yaw = 0.0f;
    this->pitch = 0.0f;
    this->fov = 60.0f;
    this->movement_speed = 3.0f;
    this->mouse_sensitivity = 0.05f;

    this->UpdateCameraVectors();
}
```

Listing 5.11: Konstruktor struktury `Camera`.

Parametry potrzebne do zdefiniowania kamery to jej pozycja w układzie współrzędnych świata oraz trzy wersory w układzie współrzędnych świata wskazujące przód kamery, jej prawą i górną oś. Ustawianie wersorów kamery następuje wewnątrz funkcji `UpdateCameraVectors` pokazanej na listingu 5.12. Orientacja kamery zdefiniowana jest przez odpowiednie przekształcenia trygonometryczne kąta odchylenie (ang. *yaw*) i kąta nachylenie (ang. *pitch*). Dzięki wyznaczeniu kierunku osi z (wektor `front`) układu współrzędnych kamery i znajomości wersora osi y układu współrzędnych świata (wektor `world_up`) można wyznaczyć kierunek osi x (wektor `right`) oraz kierunek osi y (wektor `up`) układu współrzędnych kamery. Wykorzystane funkcje biblioteki GLM opisano w paragrafie 4.2.1.

```
void UpdateCameraVectors()
{
    glm::vec3 temp;
    temp.x = sin(glm::radians(this->yaw)) *
        cos(glm::radians(this->pitch));
    temp.y = sin(glm::radians(this->pitch));
    temp.z = -cos(glm::radians(this->yaw)) *
        cos(glm::radians(this->pitch));
    this->front = glm::normalize(temp);
    this->right = glm::normalize(glm::cross(this->front,
        this->world_up));
    this->up = glm::normalize(glm::cross(this->right, this->front));
}
```

Listing 5.12: Funkcja `UpdateCameraVectors` ustawiająca orientację kamery w układzie współrzędnych świata.

Kąt nachylenia oraz kąt odchylenia przy ruchu myszą aktualizowany jest wewnątrz funkcji `ProcessMouseMovement` pokazanej na listingu 5.13. Szybkość zmiany kątów zależna jest od zmiennej `mouse_sensitivity`. Ponadto kąt nachylenia nie może być mniejszy niż `-89` ani przekroczyć wartości `89`. Po zmianie kątów, funkcją `UpdateCameraVectors` aktualizowane są parametry kamery mówiące o jej pozycji i orientacji na scenie.

```
void ProcessMouseMovement(GLfloat x_offset, GLfloat y_offset)
{
    x_offset *= this->mouse_sensitivity;
    y_offset *= this->mouse_sensitivity;

    this->yaw += x_offset;
    this->pitch += y_offset;

    if (this->pitch > 89.0f)
        this->pitch = 89.0f;
    if (this->pitch < -89.0f)
        this->pitch = -89.0f;

    this->UpdateCameraVectors();
}
```

Listing 5.13: Funkcja `ProcessMouseMovement` interpretująca ruch myszy w kontekście obiektu kamery.

Zmiana pozycji kamery następuje przy wciśnięciu odpowiedniego klawisza (patrz podrozdział 6.3). Odpowiedzialna jest za to funkcja `ProcessKeyboard` pokazana na listingu 5.14. Kierunek ruchu określa zmienna `direction` typu enumeracyjnego `DIRECTION`, przyjmującego wartości `FORWARD`, `BACKWARD`, `LEFT` lub `RIGHT`, oznaczające odpowiednio ruch do przodu, do tyłu, w lewo lub w prawo.

```
void ProcessKeyboard(DIRECTION direction, GLfloat delta_time)
{
    GLfloat distance = this->movement_speed * delta_time;

    switch(direction)
    {
        case FORWARD:
            this->position += this->front * distance; break;
        case BACKWARD:
            this->position -= this->front * distance; break;
        case LEFT:
            this->position -= this->right * distance; break;
        case RIGHT:
            this->position += this->right * distance; break;
    }
}
```

Listing 5.14: Funkcja `ProcessKeyboard` odpowiedzialna za ruch kamery.

Kąt widzenia `fov` może być zmieniany poprzez skrolowanie myszą, za co odpowiedzialna jest funkcja `ProcessMouseScroll` pokazana na listingu 5.15. Najmniejsza wartość kąta może wynosić `1`, natomiast największa `120`.

```
void ProcessMouseScroll(GLfloat y_offset)
{
    if (this->fov >= 1.0f && this->fov <= 120.0f)
        this->fov -= y_offset * 5;
    if (this->fov <= 1.0f)
```

```
6           this->fov = 1.0f;
7       if (this->fov >= 120.0f)
8           this->fov = 120.0f;
9   }
```

Listing 5.15: Funkcja `ProcessMouseScroll` odpowiedzialna za zmianę kąta widzenia kamery.

Funkcja `GetViewMatrix` zwraca macierz widoku (patrz paragraf 4.1.2), która tworzona jest za pomocą funkcji `lookAt` biblioteki GLM, przyjmującą jako parametry pozycję kamery, punkt na który kamera jest skierowana oraz wersor osi y kamery.

```
1   glm::mat4 GetViewMatrix()
2   {
3       return glm::lookAt(this->position, this->position + this->front,
            this->up);
4   }
```

Listing 5.16: Funkcja `GetViewMatrix` zwracająca macierz widoku kamery.

5.7 Główna funkcja programu

Główna funkcja programu `main` zaczyna swoje działanie od przetworzenia argumentów podanych na wejściu programu. Pokazuje to listing 5.17. Szczegółowy opis parametrów wejściowych znajduje się w podrozdziale 6.2.

```
1   for (int i = 1; i < argc; i++)
2   {
3       if (strcmp(argv[i], "-n") == 0)
4           N = atoi(argv[++i]);
5       else if (strcmp(argv[i], "-m") == 0)
6           mass = atof(argv[++i]);
7       else if (strcmp(argv[i], "-r") == 0)
8       {
9           color.r = atof(argv[++i]);
10          color.g = atof(argv[++i]);
11          color.b = atof(argv[++i]);
12      }
13      else if (strcmp(argv[i], "-l") == 0)
14          initial_location_mode = atoi(argv[++i]);
15      else if (strcmp(argv[i], "-c") == 0)
16          compute_mode = atoi(argv[++i]);
17      else if (strcmp(argv[i], "-d") == 0)
18      {
19          win_width = atoi(argv[++i]);
20          win_height = atoi(argv[++i]);
21      }
22      else if (strcmp(argv[i], "-f") == 0)
23          fullscreen = true;
24      else if (strcmp(argv[i], "-h") == 0)
```

```
25      {
26          std::cout << "\t-n [count] : number of particles; default =
                600"
27                    "\n\t-m [mass] : mass of each particle; default =
                          0.001"
28                    "\n\t-r [r] [g] [b] : color of each particle; default
                          = 1.0 0.3 0.1"
29                    "\n\t-l [initial location mode] : initial location
                          mode (0-2); default = 0"
30                    "\n\t-c [compute mode] : compute mode (0-1); default =
                          1"
31                    "\n\t-d [width] [height] : window width and height;
                          default = 800 600"
32                    "\n\t-f : fullscreen mode"
33                    "\n\t-h : show this help";
34          return 0;
35      }
36
37  }
```

Listing 5.17: Przetwarzanie argumentów wejściowych funkcji `main`.

Po przetworzeniu argumentów następuje inicjalizacja okna poprzez wywołanie funkcji `InitGLFW`, pokazanej na listingu 5.18. Użyte w niej funkcje biblioteki GLFW opisane zostały w paragrafie 4.2.1.

```
1   GLFWwindow* InitGLFW()
2   {
3       glfwInit();
4       glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
5       glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
6       glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
7       glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
8       glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
9
10      GLFWwindow* window;
11
12      if(fullscreen)
13      {
14          const GLFWvidmode* mode =
                glfwGetVideoMode(glfwGetPrimaryMonitor());
15          win_width = mode->width;
16          win_height = mode->height;
17          window = glfwCreateWindow(win_width, win_height, "OpenGL",
                glfwGetPrimaryMonitor(), nullptr);
18      }
19      else
20          window = glfwCreateWindow(win_width, win_height, "OpenGL",
                nullptr, nullptr);
21
22      if (window == nullptr)
23      {
24          std::cout << "Failed to create GLFW window" << std::endl;
25          glfwTerminate();
```

```
26      }
27      glfwMakeContextCurrent(window);
28      glfwSetKeyCallback(window, KeyHandler);
29      glfwSetCursorPosCallback(window, MouseHandler);
30      glfwSetScrollCallback(window, ScrollHandler);
31      glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
32
33      return window;
34  }
```

Listing 5.18: Funkcja `InitGLFW` użyta w głównej funkcji `main`.

W funkcji tej zdarzenia związane z klawiaturą i myszą zostają skojarzone z odpowiednimi funkcjami wywoływanymi podczas danego zdarzenia. Funkcja `KeyHandler`, uruchamiana przy zdarzeniu związanym z klawiszem, pokazana jest na listingu 5.19. Funkcję `MouseHandler` uruchamianą podczas ruchu myszą pokazuje listing 5.20. Natomiast skrolowanie myszy uruchamia funkcję `ScrollHandler` pokazaną na listingu 5.21.

```
1   void KeyHandler(GLFWwindow* window, int key, int code, int action, int
        mods)
2   {
3       if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
4           glfwSetWindowShouldClose(window, GL_TRUE);
5       else if (key == GLFW_KEY_Z && action == GLFW_PRESS)
6           initial_location_mode = 0;
7       else if (key == GLFW_KEY_X && action == GLFW_PRESS)
8           initial_location_mode = 1;
9       else if (key == GLFW_KEY_C && action == GLFW_PRESS)
10          initial_location_mode = 2;
11      else if (key == GLFW_KEY_F1 && action == GLFW_PRESS)
12          compute_mode = 0;
13      else if (key == GLFW_KEY_F2 && action == GLFW_PRESS)
14          compute_mode = 1;
15      else if (key == GLFW_KEY_F3 && action == GLFW_PRESS)
16          compute_mode = 2;
17      else if (key == GLFW_KEY_F4 && action == GLFW_PRESS)
18          compute_mode = 3;
19      else if (key == GLFW_KEY_Q && action == GLFW_PRESS)
20          for(int i=0; i<N; i++)
21              particles[i].mass /= 2;
22      else if (key == GLFW_KEY_E && action == GLFW_PRESS)
23          for(int i=0; i<N; i++)
24              particles[i].mass *= 2;
25      else if (key == GLFW_KEY_SPACE && action == GLFW_PRESS)
26          for(int i=0; i<N; i++)
27              particles[i] = Particle(mass, initial_location_mode);
28      else if (key == GLFW_KEY_I && action == GLFW_PRESS)
29          PrintInfo();
30
31      if (key >= 0 && key < 1024)
32      {
33          if (action == GLFW_PRESS)
```

```
34              keys[key] = true;
35          else if (action == GLFW_RELEASE)
36              keys[key] = false;
37      }
38  }
```

Listing 5.19: Funkcja `KeyHandler` odpowiedzialna za przetworzenia zdarzenia związanego z klawiaturą.

```
1  void MouseHandler(GLFWwindow* window, double x_pos, double y_pos)
2  {
3      if (first_mouse)
4      {
5          last_x = x_pos;
6          last_y = y_pos;
7          first_mouse = false;
8      }
9
10     GLfloat xoffset = x_pos - last_x;
11     GLfloat yoffset = last_y - y_pos;
12
13     last_x = x_pos;
14     last_y = y_pos;
15
16     camera.ProcessMouseMovement(xoffset, yoffset);
17 }
```

Listing 5.20: Funkcja `MouseHandler` odpowiedzialna za przetworzenia zdarzenia związanego z ruchem myszy.

```
1  void ScrollHandler(GLFWwindow* window, double x_offset, double
       y_offset)
2  {
3      camera.ProcessMouseScroll(y_offset);
4  }
```

Listing 5.21: Funkcja `ScrollHandler` odpowiedzialna za przetworzenia zdarzenia związanego ze skrolem myszy.

Funkcja `PrintInfo` (listing 5.22), wywoływana w przypadku naciśnięcia odpowiedniego klawisza (listing 5.19), drukuje na standardowym wyjściu aktualne informacje o symulacji.

```
1  void PrintInfo()
2  {
3      printf("Simulation info:\n");
4      printf("\tTime: %0.2fs\n", glfwGetTime());
5      printf("\tAverage FPS: %0.2f\n",
           average_fps_sum/average_fps_count);
6      printf("\tCamera position: [%.2f, %.2f, %.2f]\n",
           camera.position.x, camera.position.y, camera.position.z);
7      printf("\tField of view: %f\n", camera.fov);
```

```
8     printf("\tNumber of particles %d\n", N);
9     printf("\tMass of each particle: %f\n", mass);
10    printf("\tColor: [%.2f, %.2f, %.2f]\n", color.r, color.g, color.b);
11    printf("\tInitial location: %d\n", initial_location_mode);
12    printf("\tCompute mode: %d\n", compute_mode);
13 }
```

Listing 5.22: Funkcja `PrintInfo` drukująca aktualne informacje o symulacji.

Następnie, w głównej funkcji `main`, inicjowana jest funkcjonalność biblioteki GLEW (patrz paragraf 4.2.1) za pomocą funkcji `InitGLEW` pokazanej na listingu 5.23.

```
1 void InitGLEW()
2 {
3     glewExperimental = GL_TRUE;
4     if (glewInit() != GLEW_OK)
5         std::cout << "Failed to initialize GLEW" << std::endl;
6 }
```

Listing 5.23: Funkcja `InitGLEW` inicjująca funkcjonalność biblioteki GLEW.

Wywoływana jest również funkcja `InitOpenGL` (listing 5.24), która ustawia wysokość i szerokość renderowanego obrazu oraz kolor tła. Ponadto umożliwia zmianę rozmiaru punktu i deklaruje syntezę addytywną kolorów.

```
1 void InitOpenGL()
2 {
3     glViewport(0, 0, win_width, win_height);
4     glClearColor(0, 0, 0, 1.0f);
5     glBlendFunc(GL_SRC_ALPHA, GL_ONE);
6     glEnable(GL_BLEND);
7     glEnable(GL_PROGRAM_POINT_SIZE);
8 }
```

Listing 5.24: Funkcja `InitOpenGL` odpowiedzialna m.in. za ustawienie koloru tła.

W dalszej kolejności tworzony zostaje obiekt typu `Shader` i wskazywany jest program shaderów do użycia (listing 5.25). Następnie drukowane są informacje o wersji biblioteki (listing 5.26).

```
1 Shader shader("vertex.glsl", "fragment.glsl");
2 glUseProgram(shader.program);
```

Listing 5.25: Fragment funkcji `main` odpowiedzialny za wskazanie shaderów do użycia.

```
1 printf("Version: %s\nVendor: %s\nRenderer: %s\n",
      glGetString(GL_VERSION), glGetString(GL_VENDOR),
      glGetString(GL_RENDERER));
```

Listing 5.26: Wydrukowanie informacji o bibliotece OpenGL i jednostce renderującej.

Dalej następuje inicjalizacja tablicy, w której zgrupowane są cząsteczki. Następnie funkcją `glUniform3fv` przesyłany jest do shadera fragmentów parametr mówiący o kolorze cząsteczek, po czym tworzone i ustawiane są bufory do komunikacji z kartą graficzną. Pokazane jest to na listingu 5.27. Funkcje biblioteki OpenGL użyte w tym fragmencie opisane zostały w paragrafach 4.2.2 i 4.2.3.

```
1  particles = new Particle[N];
2  for(int i=0; i<N; i++)
3      particles[i] = Particle(mass, initial_location_mode);
4
5  glUniform3fv(glGetUniformLocation(shader.program, "color"), 1,
       glm::value_ptr(color));
6
7  GLuint vbo, vao;
8  glGenVertexArrays(1, &vao);
9  glGenBuffers(1, &vbo);
10 glBindVertexArray(vao);
11 glBindBuffer(GL_ARRAY_BUFFER, vbo);
12 SendParticlesToBuffer(vbo, particles, N);
13 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
       (GLvoid*)0);
14 glEnableVertexAttribArray(0);
```

Listing 5.27: Inicjalizacja obiektów cząsteczek oraz przygotowanie buforu z danymi w funkcji `main`.

Po utworzeniu bufora należy przesłać do niego dane dotyczące pozycji cząsteczek. Dzieje się to poprzez wywołanie funkcji `SendParticlesToBuffer` pokazanej na listingu 5.28. W funkcji tej współrzędne położenia wszystkich cząsteczek zostają przesłane do struktury tablicowej o typie `float`. Następnie wskazywany jest obiekt bufora funkcją `glBindBuffer`, po czym dane zostają wpisane do niego poprzez wywołanie funkcji `glBufferData`.

```
1  void SendParticlesToBuffer(GLuint vbo, Particle* p, int N)
2  {
3      GLfloat* all_floats = new GLfloat[N * 3];
4
5      for (int i = 0, j = 0; i < N; i++)
6      {
7          all_floats[j] = p[i].location.x; j++;
8          all_floats[j] = p[i].location.y; j++;
9          all_floats[j] = p[i].location.z; j++;
10     }
11
12     glBindBuffer(GL_ARRAY_BUFFER, vbo);
13     glBufferData(GL_ARRAY_BUFFER, 3 * N * sizeof(GLfloat), all_floats,
           GL_STATIC_DRAW);
14 }
```

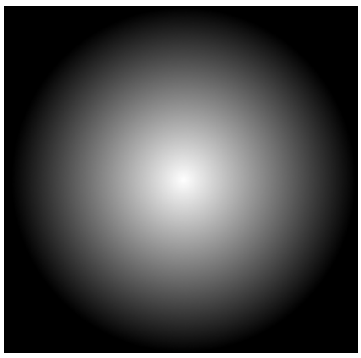
Listing 5.28: Funkcja `SendParticlesToBuffer` odpowiedzialna za przesłanie informacji o położeniach cząsteczek do bufora karty graficznej.

Dalej funkcją `glTexImage2D` określana zostaje dwuwymiarowa tekstura, którą jest używana w shaderze fragmentów. Generowana jest również jej mipmapa[4] dzięki funkcji `glGenerateMipmap`. W ładowaniu tekstury do programu pomaga biblioteka SOIL (patrz paragraf 4.2.1). Pokazane jest to na listingu 5.29.

```
1 int width, height;
2 unsigned char* image = SOIL_load_image("Data/particle.png", &width,
     &height, 0, SOIL_LOAD_AUTO);
3 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
     GL_UNSIGNED_BYTE, image);
4 glGenerateMipmap(GL_TEXTURE_2D);
5 SOIL_free_image_data(image);
```

Listing 5.29: Załadowanie tekstury cząsteczki w funkcji `main`.

Grafikę tekstury cząsteczki na czarnym tle pokazuje rysunek 5.4. Tekstura ta korzysta z kanału alfa w ten sposób, że im dalej od środka grafiki, tym większa wartość przezroczystości. Pozostałe kanały (R, G i B) mają wartość maksymalną.



Rysunek 5.4: Tekstura cząsteczki na czarnym tle.

Pętlę główną programu pokazuje listing 5.30. Wykonuje się ona dopóki użytkownik nie zatrzyma symulacji odpowiednim klawiszem (patrz podrozdział 6.3). Pętla zaczyna swoje działanie od przetworzenia zdarzeń funkcją `glfwPollEvents`. Dalej czyszczony zostaje bufor okna dzięki funkcji `glClear`. Następnie wyznaczany jest skok czasowy `delta_time` między poszczególnymi klatkami funkcją `UpdateDelta` pokazanej na listingu 5.31.

Funkcją `MoveCamera` sprawdzany jest stan klawiszy odpowiedzialnych za ruch kamery (listing 5.32). Jeżeli dany klawisz jest wciśnięty, następuje wywołanie metody zmieniającej pozycję kamery. Metody te nie są wywoływane bezpośrednio w funkcji `KeyHandler`, aby ruch kamery był możliwie ciągły podczas trzymania klawisza.

Aktualizowana jest następnie liczba klatek na sekundę w tytule okna. Dzieje się to w funkcji `ShowFPS` pokazanej na listingu 5.33. Funkcja ta sumuję również liczby

[4]Mipmapa jest zmniejszoną wersją danej grafiki, tym samym zawierającą mniej szczegółów. Stosowanie mipmap jest uzasadnione odciążeniem procesora wykonującego obliczenia na teksturach znajdujących się na tyle daleko od kamery, że monitor, ze względu na ograniczoną rozdzielczość, nie byłby w stanie wyświetlić szczegółów.

klatek na sekundę do zmiennej `average_fps_sum` oraz liczbę sumowań do zmiennej `average_fps_count`, dzięki czemu możliwe jest obliczenie średniej liczby klatek na sekundę przez cały czas pracy symulatora (np. w funkcji `PrintInfo` pokazanej na listingu 5.22). Aby pominąć początkowe niestabilności pracy symulatora, sumowanie te odbywa się dopiero po dziesięciu sekundach od jego uruchomienia.

Następnie instrukcją `switch` wybierany jest tryb obliczenia następnych pozycji cząsteczek. Po obliczeniu nowych pozycji, przesyłane są one do bufora. W dalszej kolejności wyznaczane zostają macierze `model`, `view` oraz `projection` (więcej o macierzach modelu, widoku i projekcji w paragrafie 4.1.2). Macierz modelu `model`, dzięki konstruktorowi domyślnemu typu `glm::mat4`, jest macierzą jednostkową; i taką pozostaje, ponieważ cząsteczki reprezentowane są przez punkty, a współrzędne punktów podawane są we współrzędnych sceny. Macierz widoku `view` tworzona jest dzięki metodzie `GetViewMatrix` obiektu kamery (listing 5.16). Macierz projekcji `projection`, służąca do transformacji położenia punktu we współrzędnych kamery do współrzędnych przycinania, tworzona jest dzięki funkcji biblioteki GLM `perspective`, która przyjmuje parametry określające stożek ścięty widzenia (ang. *view frustum*). Złożenie tych macierzy reprezentuje jedna macierz 4×4 `mvp`, która zostaje przesłana do shaderów funkcją `glUniformMatrix4fv`. Ostatnim etapem przebiegu pętli głównej jest narysowanie przygotowanych punktów funkcją `glDrawArrays` oraz zamiana buforów okna funkcją `glfwSwapBuffers`.

```
1  while (!glfwWindowShouldClose(window))
2  {
3      glfwPollEvents();
4      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
5      UpdateDelta();
6      ShowFPS(window);
7      MoveCamera();
8
9      switch(compute_mode)
10     {
11         case 0:
12             ComputeNextPositions(vbo, particles, N, delta_time); break;
13         case 1:
14             ComputeNextPositions_TBB_Outer(vbo, particles, N,
                   delta_time); break;
15     }
16
17     SendParticlesToBuffer(vbo, particles, N);
18
19     glm::mat4 model;
20     glm::mat4 view = camera.GetViewMatrix();
21     glm::mat4 projection = glm::perspective(glm::radians(camera.fov) ,
           (float)win_width / (float)win_height, 0.01f, 1000.f);
22
23     glm::mat4 mvp = projection * view * model;
24     glUniformMatrix4fv(glGetUniformLocation(shader.program, "mvp"), 1,
           GL_FALSE, glm::value_ptr(mvp));
25
26     glDrawArrays(GL_POINTS, 0, N);
```

```
27      glfwSwapBuffers(window);
28  }
```

Listing 5.30: Pętla główna programu w funkcji `main`.

```
1   void UpdateDelta()
2   {
3       GLfloat current_frame = glfwGetTime();
4       delta_time = current_frame - last_frame;
5       last_frame = current_frame;
6   }
```

Listing 5.31: Funkcja `UpdateDelta` aktualizująca krok czasowy.

```
1   void MoveCamera()
2   {
3       if (keys[GLFW_KEY_W])
4           camera.ProcessKeyboard(FORWARD, delta_time);
5       if (keys[GLFW_KEY_S])
6           camera.ProcessKeyboard(BACKWARD, delta_time);
7       if (keys[GLFW_KEY_A])
8           camera.ProcessKeyboard(LEFT, delta_time);
9       if (keys[GLFW_KEY_D])
10          camera.ProcessKeyboard(RIGHT, delta_time);
11  }
```

Listing 5.32: Funkcja `MoveCamera` sprawdzająca stan klawiszy ruchu.

```
1   void ShowFPS(GLFWwindow* window)
2   {
3       double current_time = glfwGetTime();
4       frame_count++;
5
6       double time_interval = current_time - previous_time;
7       if (time_interval > 1.0f)
8       {
9           double fps = frame_count / (time_interval);
10          previous_time = current_time;
11
12          char title[256];
13          title[255] = '\0';
14          snprintf(title, 255, "%.2f", fps);
15          glfwSetWindowTitle(window, title);
16          frame_count = 0;
17          if(current_time > 10.0f)
18          {
19              average_fps_sum += fps;
20              average_fps_count += 1;
21          }
22      }
23  }
```

Listing 5.33: Funkcja `ShowFPS` aktualizująca informacje o liczbie klatek na sekundę.

Po wyjściu z pętli głównej zwalniane są zasoby zajęte przez obiekty cząsteczek i bibliotekę GLFW, po czym następuje wyjście z programu i zamknięcie symulatora. Pokazane jest to na listingu 5.34

```
1 glfwTerminate();
2 delete [] particles;
3 return 0;
```

Listing 5.34: Wyjście z funkcji `main`.

Rozdział 6

Użytkowanie i testy

Na pracę zaimplementowanego symulatora wpływają parametry wejściowe oraz interakcja użytkownika. Rozdział ten traktuje o możliwościach kontroli pracy symulatora, o przebiegu symulacji w zależności od parametrów wejściowych oraz o wynikach badań wydajności symulatora.

6.1 Środowisko uruchomieniowe

Symulator testowany był na komputerze z procesorem Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz o architekturze x86_64 i czterech rdzeniach, kartą graficzną Nvidia GeForce 710M i system operacyjnym Arch Linux 4.8.8-1-ARCH. Symulator kompilowany był z plików źródłowych do pliku wykonywalnego za pomocą kompilatora G++ (wersja 6.2.1) komendą: `g++ main.cpp -lGL -lGLEW -lglfw -ltbb -lSOIL -o nbody`. Aby uzyskać liczbę klatek na sekundę większą niż 60, wyłączona została synchronizacja pionowa poprzez przypisanie do zmiennej środowiskowej `vblank_mode` wartości `0`.

6.2 Parametry wejściowe

Symulator ma możliwość przyjęcia opcjonalnych parametrów wejściowych, wpływających na działanie symulatora i przebieg symulacji: liczba cząsteczek, masa cząsteczek, kolor cząsteczek, tryb położenia początkowego, tryb obliczania, szerokość i wysokość okna. Ponadto można ustawić tryb pełnoekranowy. Przy braku argumentów, przyjmowane są odpowiednie wartości domyślne. Szczegółowe informacje przedstawia tabela 6.1.

Parametr	Opis	Możliwe wartości	Domyślna wartość
`-n`	Liczba cząsteczek	Liczba naturalna	`600`
`-m`	Masa cząsteczek	Liczba zmiennoprzecinkowa	`0.01`
`-r`	Kolor cząsteczek w modelu RGB	Trzy liczby zmiennoprzecinkowe mieszczące się w zakresie `0.0` – `1.0`	`1.0 0.3 0.1`
`-l`	Tryb położenia początkowego	`0` – Tryb sferyczny, `1` – tryb Gauss'a, `2` – Tryb sześcianu	`0`
`-c`	Tryb obliczania następnych pozycji cząsteczek	`0` – tryb sekwencyjny, `1` – tryb równoległy	`0`
`-d`	Szerokość i wysokość okna w pikselach	Dwie liczby naturalne	`800 600`
`-f`	Ustawienie trybu pełnoekranowego	-	-
`-h`	Wyświetlenie pomocy i zapobiegnięcie uruchomienia symulatora	-	-

Tabela 6.1: Opcjonalne parametry wejściowe symulatora.

6.3 Sterowanie symulatorem

Dzięki urządzeniom wejścia-wyjścia użytkownik może poruszać się po scenie oraz zmieniać orientację i kąt widzenia. Możliwe jest to dzięki zmienianiu parametrów kamery podczas trwania symulacji. Ponadto istnieje możliwość zmieniania masy cząsteczek układu, co wpływa na ich zachowanie. Cząsteczki można również zastąpić nowymi, co w połączeniu ze zmianą trybu położenia początkowego, ułatwia badania aktywności cząsteczek. W trakcie trwania symulacji użytkownik może również zmieniać tryb obliczania oraz drukować informacje o bieżącym stanie symulacji: czas pracy symulatora, średnia liczba klatek na sekundę, pozycja kamery w układzie świata, kąt widzenia, liczba, masa i kolor cząsteczek, tryb położenia początkowego oraz tryb obliczania następnych położeń cząsteczek (listing 5.22). Szczegóły dotyczące możliwości interakcji użytkownika z symulatorem pokazują tabele 6.2 i 6.3.

Klawisz	Opis
W	Ruch kamery w przód
S	Ruch kamery w tył
A	Ruch kamery w lewo
D	Ruch kamery w prawo
SPACE	Zastąpienie wszystkich cząsteczek nowymi
Z	Ustawienie trybu sferycznego położenia początkowego cząsteczek
X	Ustawienie trybu Gauss'a położenia początkowego cząsteczek
C	Ustawienie trybu sześcianu położenia początkowego cząsteczek
F1	Ustawienie trybu sekwencyjnego obliczania
F2	Ustawienie trybu równoległego obliczania
Q	Dwukrotne zmniejszenie masy wszystkich cząsteczek
E	Dwukrotne zwiększenie masy wszystkich cząsteczek
I	Wydrukowanie aktualnych informacji o symulacji
ESC	Zakończenie pracy symulatora

Tabela 6.2: Komunikacja symulatora z użytkownikiem za pomocą klawiatury.

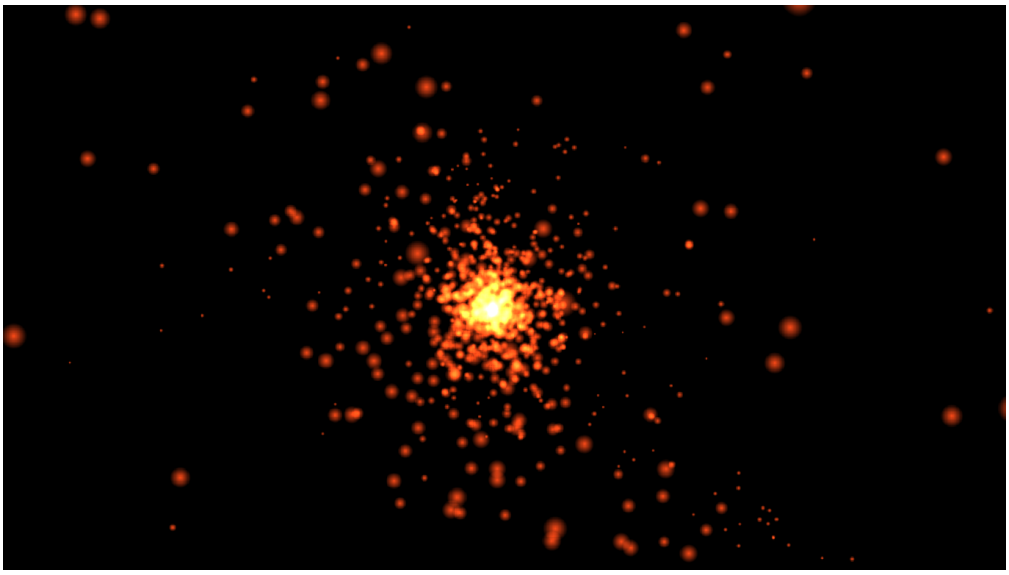
Akcja	Opis
Skrolowanie	Zmiana kąta widzenia kamery
Ruch myszy	Zmiana orientacji kamery

Tabela 6.3: Komunikacja symulatora z użytkownikiem za pomocą myszy.

6.4 Działanie symulatora

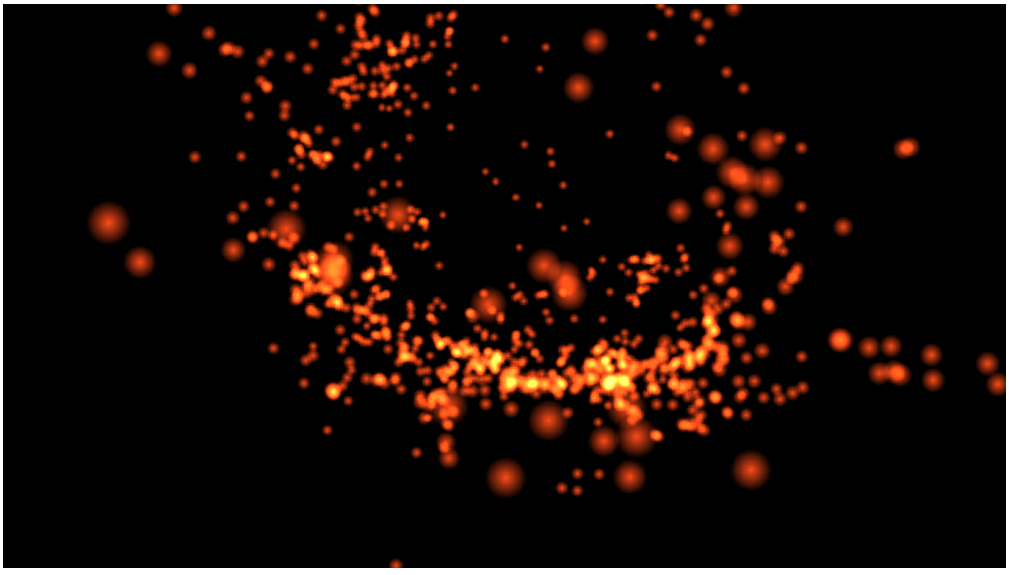
Przykładowe stany stany symulacji przedstawione są na poniższych rysunkach. We wszystkich przypadkach obliczenia odbywały się w trybie równoległym. Informacje o stanie symulatora uzyskiwano za pomocą klawisza I.

Rysunek 6.1 przedstawia stan symulacji po 18.3 sekundach działania symulatora. Pozycja kamery jest bez zmian, a kąt widzenia zmniejszony do 40 stopni. Masa i kolor cząsteczek mają wartości domyślne. Ruch cząsteczek, których jest 1200, rozpoczął się od trybu położenia początkowego Gauss'a.



Rysunek 6.1: Przykładowy stan symulacji rozpoczęty w trybie Gauss'a położenia początkowego.

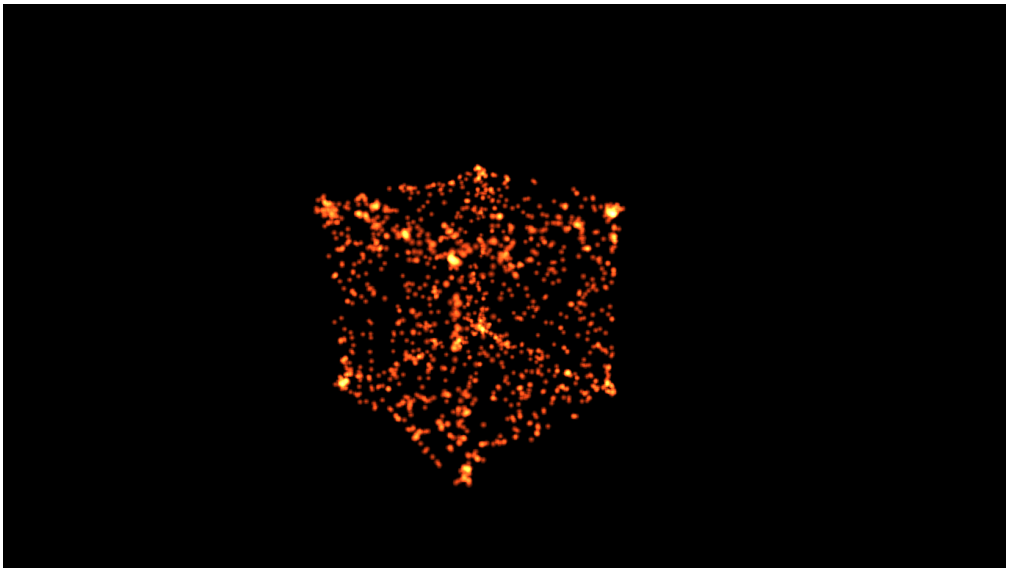
Na rysunku 6.2 przedstawiony jest stan symulatora w `3.26` sekundzie działania, z przybliżoną kamerą w kierunku osi z o `0.71`. Masa cząsteczek, których jest `1000`, została zwiększona dwukrotnie. Tryb położenia początkowego, kolor cząsteczek i kąt widzenia kamery pozostają bez zmian.



Rysunek 6.2: Przykładowy stan symulacji po ponad trzech sekundach działania symulatora.

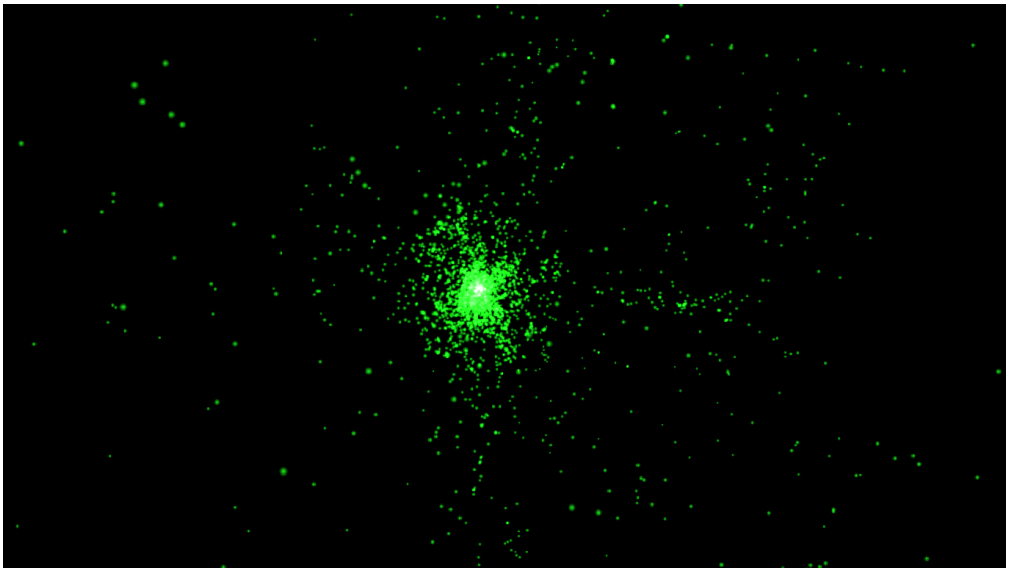
Stan symulacji pokazany na rysunku 6.3 został osiągnięty w trybie sześcianu położenia początkowego, po `11.51` sekundach działania symulatora, z kamerą ustawioną na pozycji `[x=3.02, y=1.73, z=2.52]`. Masa cząsteczek została zmniej-

szona tysiąckrotnie, co znacząco spowolniło ruchy cząsteczek. Liczba cząsteczek wynosi `1500`, a kąt widzenia `50`.

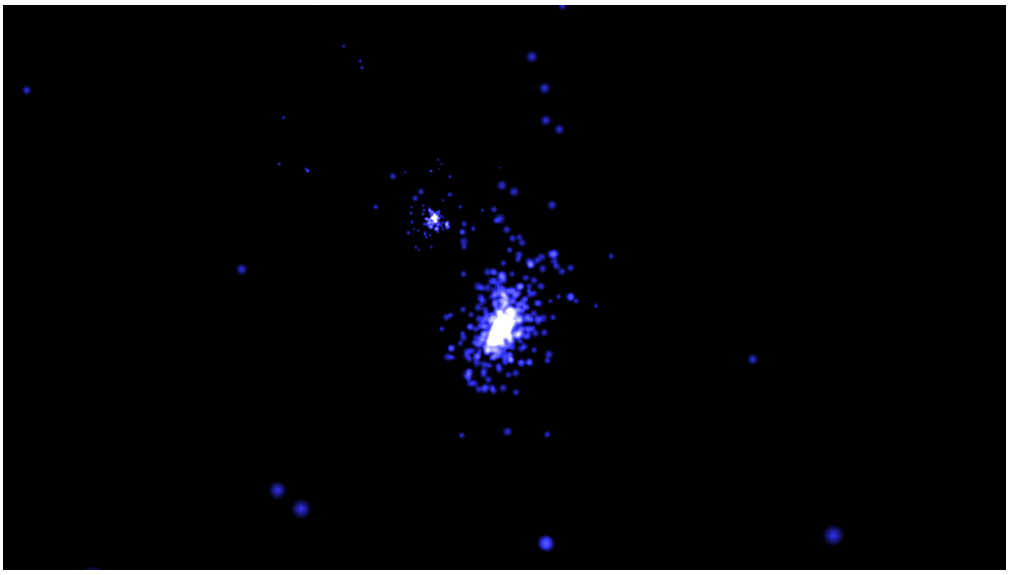


Rysunek 6.3: Przykładowy stan symulacji w trybie sześcianu położenia początkowego.

Rysunek 6.4 przedstawia stan symulacji po `19.18` sekundach działania. Liczba cząsteczek wynosi `3000`. Kamera została oddalona w osi z o `4.12`. Masa cząsteczek zmniejszona została do `0.0005`, a kąt widzenia do `45` stopni. Cząsteczki mają kolor zdefiniowany wektorem `[r=0.1, g=1, b=0.1]`. Średnia liczba klatek na sekundę podczas trwania symulacji wyniosła `4.36`.



Rysunek 6.4: Przykładowy stan symulacji trzech tysięcy cząsteczek.



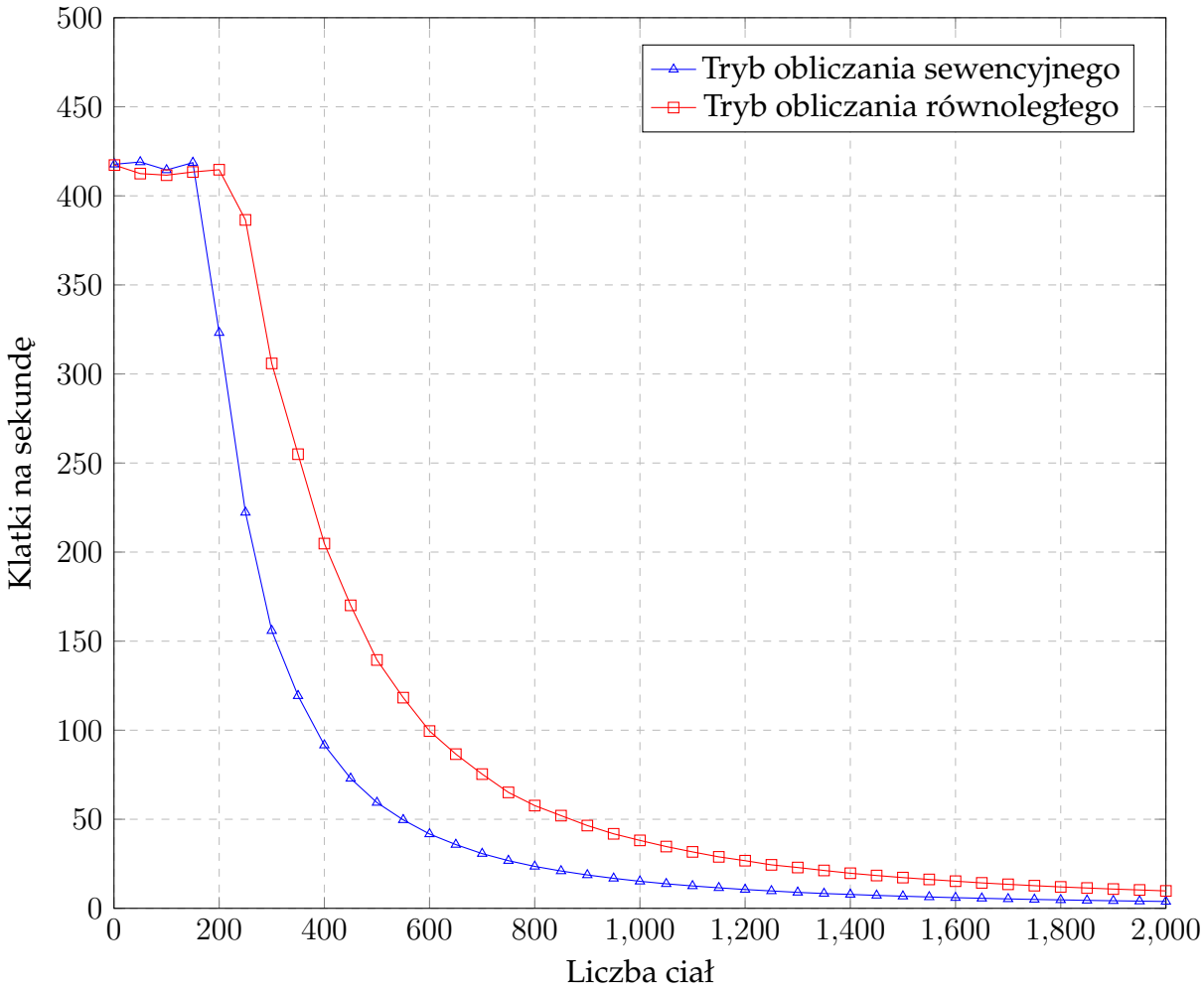
Rysunek 6.5: Przykładowy stan symulacji przy maksymalnym kącie widzenia kamery.

Na rysunku 6.5 przedstawiony jest stan symulacji w `13.81` sekundzie działania symulatora. Położenie kamery wynosi `[x=2.98, y=-0.78, z=2.43]`, a kąt widzenia `120` stopni. Układ zawiera `1000` cząsteczek, każda o masie `0.002` i kolorze `[r=0.2, g=0.2, b=1]`.

6.5 Wydajność symulatora

Symulator został zbadany pod kątem wydajności mierzonej w liczbie klatek na sekundę. Porównane zostały dwa tryby obliczania następnych pozycji ciał: tryb sekwencyjny działający na jednym wątku oraz tryb równoległy wykorzystujący wielowątkowość zarządzaną przez bibliotekę TBB.

Symulator uruchamiany był z różnymi parametrami wejściowymi, aby zbadać wpływ trybu obliczania i liczby ciał w układzie na liczbę klatek na sekundę. Każdy pomiar odbywał się po 110 sekundach pracy symulatora. Dbał o to skrypt, który automatycznie uruchamiał symulator i zbierał wyniki po upływie tego czasu, po czym zmieniając parametry uruchamiał go ponownie. Skrypt zwiększał liczbę ciał co każde uruchomienie o pięćdziesiąt, począwszy od pięćdziesięciu, aż do dwóch tysięcy ciał. Przetestowano również wariant z jednym ciałem w układzie.



Wykres 6.1: Wydajność symulatora.

Wyniki zebrane dla trybu sekwencyjnego oraz trybu równoległego pokazane są na wykresie 6.1. Widać na nim, że korzyści wynikające z zastosowania wielowątkowości zaczynają być znaczące od dwustu ciał w układzie, utrzymując ponad dwukrotną przewagę wydajności nad trybem obliczania sekwencyjnego.

Rozdział 7

Podsumowanie

Na przykładzie implementacji symulatora wielu ciał zaprezentowano modelowanie współbieżne oraz kwestie prezentacji obiektów fizycznych w przestrzeni trójwymiarowej. Oba te zagadnienia, będące motywem przewodnim pracy, są na tyle rozległe, że pracę tę winno się traktować jako kiepski wstęp do nich.

Podczas implementacji symulatora sprawdzano różne sposoby zrównoleglenia obliczeń. Zdecydowano się jednak zrównoleglić pętlę zewnętrzną, co dzięki trafnemu doborowi metody i miejsca jej użycia, okazało się być proste, acz bardzo efektywne. Najwięcej uwagi wymagała prezentacja systemu cząsteczek na scenie trójwymiarowej. Aspekty programowania grafiki trójwymiarowej wymagają sporej pracy wejścia, obejmują wiele zagadnień i sposobów dojścia do pożądanych rezultatów. Jednak kwestie te udało się rozwiązać w sposób prosty i skuteczny.

Symulator został tak skonstruowany, aby rozszerzenie go o nowe funkcjonalności i sposoby działania nie stanowiło problemów. Naturalnym wydaje się być dodanie nowych trybów obliczania, zaimplementowanie innych algorytmów wyznaczania sił między ciałami (jak np. wspomniany algorytm Barnes-Hut), czy zastosowanie innych metod numerycznych do obliczania ruchu (np. przytoczona metoda Verleta). Ponadto można dorobić elementy, które mniej wpływają na samą pracę symulatora, a bardziej na przebieg symulacji, jak np. nowe tryby położenia początkowego czy wprowadzenie aspektu różnorodności cząsteczek pod względem masy lub prędkości początkowej.

Krokiem milowym przy rozbudowie symulatora może być przerzucenie obliczeń z CPU na GPU. Systemy cząsteczek charakteryzuje mnogość i prostota obliczeń. Dzięki temu w znacznym stopniu można wykorzystać zalety architektury procesorów graficznych nastawionej na masowe obliczenia równoległe. Systemy cząsteczek wykorzystujące do obliczeń karty graficzne, mogą się składać z relatywnie dużej liczby cząsteczek, jednocześnie zachowując wysoką wydajność.

Implementacja symulatora wielu ciał okazała się nie być skomplikowana. Projekt został zrealizowany pomyślnie, acz pozostawia miejsce dla efektywniejszych i bardziej wyrafinowanych metod.

Bibliografia

[1] Jacek Matulewski, Tomasz Dziubak, Marcin Sylwestrzak, Radosław Płoszajczak, *Grafika, Fizyka, Metody numeryczne. Symulacje fizyczne z wizualizacją 3D*, Wydawnictwo naukowe PWN, Warszawa 2010

[2] Jacek Matulewski, *Grafika 3D czasu rzeczywistego. Nowoczesny OpenGL*, Wydawnictwo naukowe PWN, Warszawa 2014

[3] Janusz Ganczarski, *OpenGL. Podstawy programowania grafiki 3D*, Helion, Gliwice 2015

[4] Lars Nyland, Mark Harris, Jan Prins, *GPU Gems 3*, Chapter 31. Fast N-Body Simulation with CUDA, [online] `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html` [dostęp: 01-01-2017]

[5] Srinivas Aluru, John Gustafson, G. M. Prabhu, *Truly Distribution-Independent Algorithms for the N-body Problem*, [online] `http://hint.byu.edu/documentation/Gus/N-Body/N-Body.html` [dostęp: 01-01-2017]

[6] Tancred Lindholm, *N-body algorithms*, [online] `http://www.cs.hut.fi/~ctl/NBody.pdf` [dostęp: 01-01-2017]

[7] Mark Segal, Kurt Akeley, *The OpenGL Graphics System: A Specification (Version 4.5 (Core Profile) - October 24, 2016)*, [online] `https://www.opengl.org/registry/doc/glspec45.core.pdf` [dostęp: 01-01-2017]

[8] Intel Threading Building Blocks Documentation, `https://software.intel.com/en-us/tbb-documentation` [dostęp: 01-01-2017]

[9] OpenGL – The Industry Standard for High Performance Graphics, `https://www.opengl.org/` [dostęp: 01-01-2017]

[10] GLEW: The OpenGL Extension Wrangler Library, `http://glew.sourceforge.net/` [dostęp: 01-01-2017]

[11] GLFW – An OpenGL library, `http://www.glfw.org/` [dostęp: 01-01-2017]

[12] OpenGL Mathematics, `http://glm.g-truc.net/` [dostęp: 01-01-2017]

[13] Simple OpenGL Image Library, `http://www.lonesock.net/soil.html` [dostęp: 01-01-2017]

[14] Cg Programming/Rasterization – Wikibooks, open books for an open world, https://en.wikibooks.org/wiki/Cg_Programming/Rasterization [dostęp: 01-01-2017]

[15] Naukowiec.org, http://www.naukowiec.org/wiedza/statystyka/rozklad-normalny-rozklad-gaussa_710.html [dostęp: 01-01-2017]

[16] Learn OpenGL, extensive tutorial resource for learning Modern OpenGL, https://learnopengl.com/ [dostęp: 01-01-2017]