Uniwersytet Mikołaja Kopernika
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Katedra Informatyki

Bartosz Kowalczyk
nr albumu: 254097

Praca inżynierska
na kierunku Informatyka Stosowana

Symulacje cieczy i gazów metodą Smoothed-Particle Hydrodynamics

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Uniwersytet Mikołaja Kopernika w Toruniu

Toruń 2015

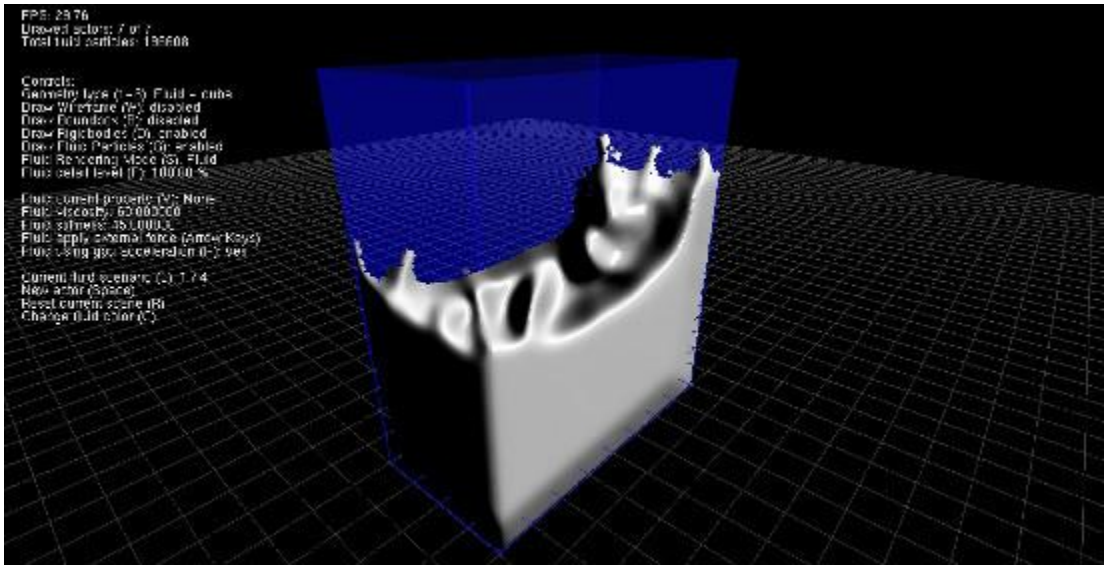
Pracę przyjmuję i akceptuję	Potwierdzam złożenie pracy dyplomowej
……..	……..
data i podpis opiekuna pracy	*data i podpis pracownika dziekanatu*

Dziękuję wszystkim, na których pomoc w tworzeniu tej pracy mogłem liczyć oraz tym, dla których podjąłem się jej realizacji.

1. Spis treści

2. Wstęp

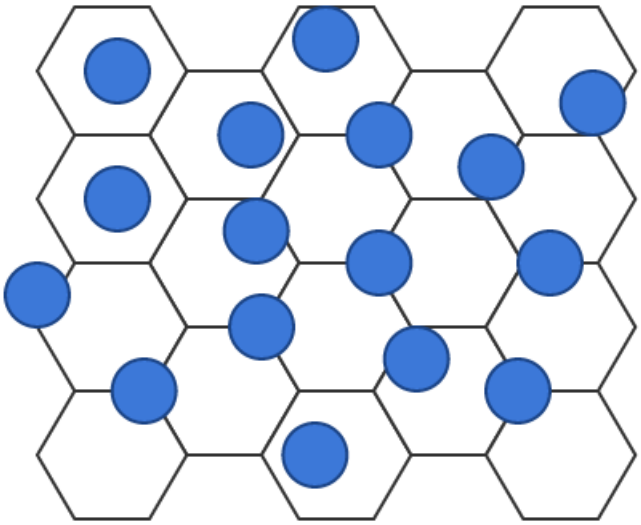
Smoothed-Particle Hydrodynamics *(SPH)* jest metodą numeryczną stosowaną do celu symulacji dynamiki cieczy i gazów. Opracowana została w 1977 roku przez J.J. Monaghana i R.A. Gingolda [4] . Początkowo stworzona z myślą o rozwiązywaniu astrofizycznych problemów. Obecnie używana jest w wielu dziedzinach, niekoniecznie *stricte* naukowych, takich jak tworzenie gier, balistyka, wulkanologia, oceanografia. Najpopularniejsza jej implementacja została opublikowana przez NVIDIA Corporation w



Rysunek 1. Symulacja cieczy przy użyciu PhysX i OpenGl.

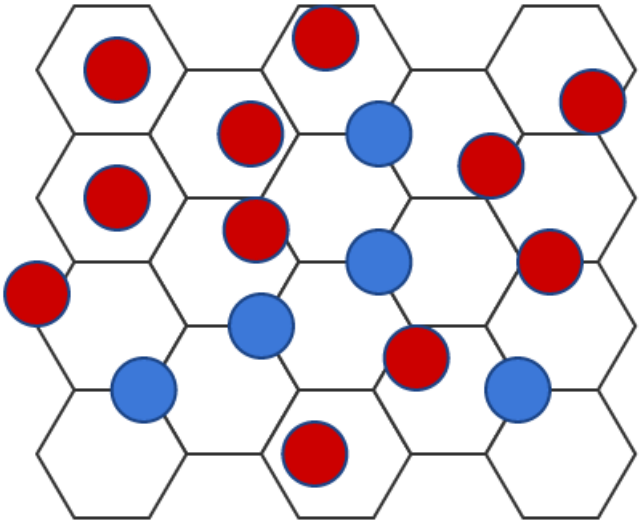
jej silniku fizycznym PhysX [5] (Rysunek 1). Ich implementacja jest wysoce wydajna i wykorzystuje CUDA. CUDA jest opracowaną przez NVIDIA równoległą architekturą obliczeniową, zapewniającą wysoki wzrost wydajności obliczeniowej dzięki wykorzystaniu pełnej mocy układów graficznych GPU. Jako, że historia SPH zaczęła się niespełna 40 lat temu, to zarówno sama metoda, jak i jej przeznaczenie ewoluowały na przestrzeni lat. Najpierw były ściśliwe ciecze nielepkie, a następnie dodawano do niej możliwości symulacji także nieściśliwych cieczy lepkich w polu grawitacyjnym, a nawet zagadnień z dziedziny magnetohydrodynamiki. Największą przeszkodą w implementacji SPH są warunki brzegowe, które nie zostały przewidziane w pierwotnej koncepcji metody. Do celów, w których wykorzystywana jest obecnie były one konieczne – w końcu aby zasymulować zachowanie cieczy w pewnym zbiorniku musimy przyjąć pewnie ograniczenia obszaru, w którym poruszają się cząsteczki cieczy.

Sama metoda działa przez dzielenie cieczy lub gazu, na zbiór elementów, dalej zwanych „cząsteczkami”. U jej podstaw leży równanie Naviera-Stokesa, które dla wymagań SPH przybiera specjalną postać, przedstawioną w rozdziale 3. Uwarunkowane jest to tym, że w metodzie Smoothed-Particle Hydrodynamics używa się opisu Lagrange’a, w którym siatka obliczeniowa porusza się wraz z symulowaną cieczą *(en. mesh-free Langranian method)*. Siatka obliczeniowa jest czymś w rodzaju szkieletu, po którym poruszać mogą się cząsteczki. Różnicę między metodami *mesh* oraz *mesh-free* prezentuje (Rysunek 2) i (Rysunek 3).



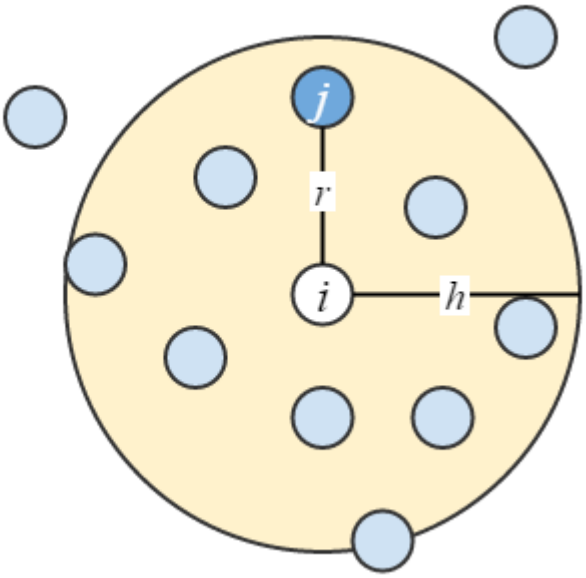
Rysunek 2. Wizualizacja mesh-free - metody bezsiatkowej

Różnice są oczywiste – metody siatkowe wymuszają na cząsteczce konkretne położenia, które równoznaczne są z węzłami siatki. Ponadto metody siatkowe wiążą się z innymi ograniczeniami. Dla każdej symulacji należy wygenerować siatkę. Jest to dodatkowy koszt. Aby zwiększyć dokładność symulacji należy przebudować (zagęścić) siatkę, co jest problematyczne z oczywistych względów – są to kolejne koszty. Przewagą metod bezsiatkowych jest przede wszystkim brak ograniczenia położenia do węzłów siatki. Dzięki temu można pozwolić sobie na zwiększenie gęstości cząsteczek, nie przejmując się przy tym przebudową siatki, aby odpowiadała im swoją gęstością węzłów. Położenia cząsteczek są, do granic błędów numerycznych, dowolne.



Rysunek 3. Wizualizacja mesh - metody siatkowej

Metoda SPH opiera się na teorii interpolacji – ciągłe rozkłady parametrów zastępowane są *(przybliżane numerycznie)* przez estymatory. Estymuje się parametry równania Naviera-Stokesa za pomocą wzorów omówionych szczegółowo w rozdziale 3. Estymatory, a konkretnie ich wartości, są parametrami obliczonymi dla pewnej próby, na podstawie których szacuje się realne wartości tego parametru. Są to oczywiście estymatory obciążone pewnym pomijalnym, dla efektów tej pracy, błędem statystycznym. Aby przybliżyć każdą zmienną np. gęstość, ciśnienie, *i-tej* cząsteczki należy wykonać pętlę sumującą po wszystkich cząsteczkach ją otaczających. Gęstość w estymatorach wartości dla SPH nie jest gęstością cieczy, jest to pseudo-gęstość liczona dla każdej cząsteczki z osobna. Ponieważ metoda opiera się na teorii interpolacji oprócz estymowania odpowiednich wartości konieczne jest założenie pewnego jądra interpolacji, które zwykle oznaczane jest jako *W*, opisane dokładniej w rozdziale 3. Jednym z parametrów tej funkcji jest *h*, nazywany długością wygładzania (en. *Smoothing length)*. Określa on maksymalną odległość w jakiej j-ta cząsteczka może oddziaływać na cząsteczkę i-tą, co przedstawia (Rysunek 4). W mojej pracy wartość tego parametru przyjmowana jest jako stała.



Rysunek 4. Parametr h – Smoothing length

Powyższy rysunek obrazuje sens istnienia funkcji jądra interpolacji. Widać, że dla obecnie liczonej cząsteczki *i*, cząsteczka *j* znajduje się w odległości *r* mniejszej od założonej stałej wielkości *h*.

3. Wyprowadzenia wzorów

Do wyświetlenia efektów symulacji potrzebna jest lista położeń wszystkich cząsteczek cieczy, w każdej klatce symulacji. Należy zatem tak przekształcić równanie Naviera-Stokesa (1), aby otrzymać z niego przyśpieszenie dla *i-tej* cząsteczki. Następnie z otrzymanego przyśpieszenia *i-tej* cząsteczki należy wyliczyć jej położenie.

1. Wyprowadzenie wzoru na przyśpieszenie

(1) $$\frac{d\vec{v}}{dt} = -\frac{\nabla p}{\rho} + \vec{a}_{visc} + \vec{\phi},$$

gdzie:

v – wektor prędkości

t – czas

p – ciśnienie

ρ – gęstość

a – przyśpieszenie wynikające z istnienia sił lepkości

Φ – przyśpieszenie wynikające z istnienia sił grawitacyjnych

Wartości gradientu ciśnienia oraz gęstości przybliżyć należy wspomnianymi we wstępie estymatorami – wartości oraz gradientu wartości – odpowiednio wyrażającymi się wzorami:

(2) $$A_i = \sum_{j=1}^{N} m_j \frac{A_j}{\rho_j} W_{ij}$$

oraz

(3) $$\nabla_i A_i = \sum_{j=1}^{N} m_j \frac{A_j}{\rho_j} \nabla_i W_{ij},$$

gdzie:

m – masa j-tej cząsteczki

ρ – gęstość j-tej cząsteczki

Wspomniane we wstępie jądro interpolacji:

$$W(r,h) = \frac{1}{h^{\nu}} f(u), \tag{4}$$

gdzie:

r – odległość miedzy cząsteczkami

h – parametr długości wygładzania

v – liczba wymiarów

u – stosunek odległości między cząsteczkami do liczby wymiarów

parametryzowane jest funkcją *f(u)*, która musi spełniać pewne warunki określone wzorami:

$$\int f(u) dV = 1\,, \tag{5}$$

gdzie:

dV – element objętości, w trzech wymiarach równy $4\pi u^2\, du$ [4]

oraz

$$\lim_{h \to 0} f(u) = \delta(r). \tag{6}$$

Ponadto *h* musi być dobrane tak, aby w jego promieniu od cząsteczki *i-tej* znalazło się odpowiednio wiele innych cząsteczek. Autor metody sugeruje zakres $h \in \langle 22, 110 \rangle$ dla 3 wymiarów [4].

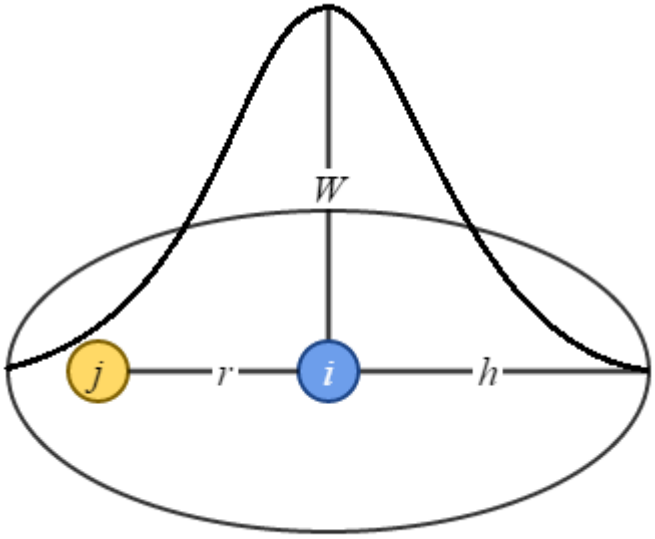
W mojej pracy przyjąłem funkcję jądra interpolacji zasugerowane przez Monaghana [3], które przyjmuje postać:

(7) $$W(r,h) = \frac{1}{\pi h^{v}} \begin{cases} 1 - 3u^2\left(\frac{1}{2} - \frac{u}{4}\right) & dla\ 0 \le u \le 1 \\ \frac{1}{4}(2-u)^3 & dla\ 1 < u \le 2 \\ 0 & dla\ u > 2 \end{cases}$$

a jej gradient:

(8) $$\nabla W(r,h) = \frac{1}{\pi h^{v+1}} \frac{\vec{r_i} - \vec{r_j}}{|\vec{r_i} - \vec{r_j}|} \begin{cases} -3u(1 - \frac{3}{4}u) & dla\ 0 \le u \le 1 \\ \frac{-3}{4}(2-u)^2 & dla\ 1 < u \le 2 \\ 0 & dla\ u > 2 \end{cases}$$

Wykres funkcji jądra interpolacji przedstawiony został na rysunku 5. Jak widać funkcja ta jest przybliżoną krzywą Gaussa obróconą wokół osi *OY*.



Rysunek 5. Wykres funkcji jądra interpolacji

Następnym krokiem jest dodanie przyśpieszenia – a w praktyce opóźnienia - wynikającego z istnienia sił lepkości. W metodzie SPH jest ono przybliżane za pomocą sztucznej lepkości określonej wzorem:

(9) $$\vec{a}_i = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla W_{ij} ,$$

gdzie:

(10) $$\Pi_{ij} = \frac{-c\mu_{ij}+2{\mu_{ij}}^2}{\rho_i+\rho_j},$$

w którym:

(11) $$\mu_{ij} = \begin{cases} \frac{h^2(\vec{v}_i-\vec{v}_j)(\vec{r}_i-\vec{r}_j)}{|\vec{r}_i-\vec{r}_j|^2+0.01h^2} & dla\ (\vec{v}_i-\vec{v}_j)(\vec{r}_i-\vec{r}_j) < 0 \\ 0 & dla\ (\vec{v}_i-\vec{v}_j)(\vec{r}_i-\vec{r}_j) \geq 0 \end{cases}$$

oraz

(12) $$c = \sqrt{\frac{pc_p}{\rho c_v}}$$

jest równaniem stanu, przyjmującym za argumenty stosunek c_p/c_v, w programie wyrażonym przez stałą *k* oraz ciśnienie i gęstość.

Ostatecznie, z pominięciem warunków brzegowych i sił grawitacyjnych, po przekształceniach, otrzymujemy wzór na przyśpieszenie *i-tej* cząstki, w danym kroku, mający postać

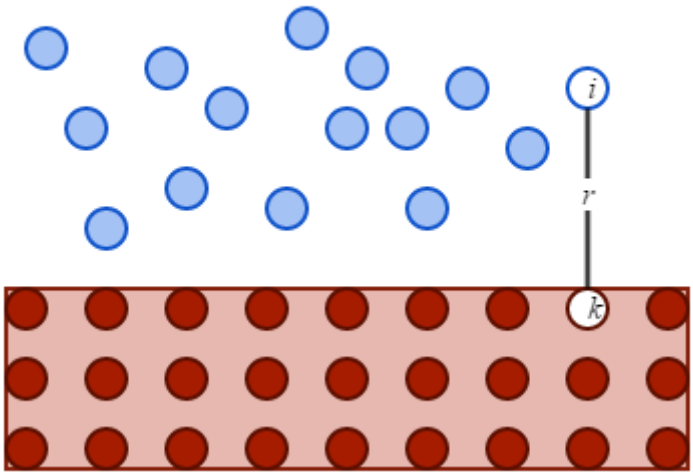
(13) $$\vec{a}_i = -\frac{c^2 c_v}{c_p}\sum_{j=1}^{N} m_j \left(\frac{1}{\rho_i}+\frac{1}{\rho_j}+\Pi_{ij}\right)\nabla W_{ij}$$

Oczywiście interesuje nas symulacja cieczy w kontakcie z naczyniem. Koniecznym jest zatem uwzględnienie warunków brzegowych. To stanowi temat następnego podrozdziału.

2. Warunki brzegowe

Warunki brzegowe to w SPH temat najbardziej problematyczny. Tworząc tą metodę Monaghan nie myślał, że będzie ona wykorzystywana do symulacji zderzeń cząsteczek z naczyniem. Tym samym ich implementacja rodzi wiele problemów. Zaproponowano wiele sposobów ich rozwiązywania, między innymi przez Monaghana [3] oraz w publikacji [6].

W swojej pracy zaimplementowałem warunki brzegowe, które są najczęściej sugerowane do zastosowań mających na celu jak największy realizm rezultatów symulacji [6]. Sposób polega na wprowadzeniu do symulacji nowych cząsteczek, o wielokrotnie większej masie od cząsteczek symulowanej cieczy, które nie reagują między sobą, lecz pozostają w stałej odległości od siebie. Aby utworzyć obszar wzbroniony dla cząsteczek cieczy liczone są siły kontaktowe między cząsteczką i-tą – aktualnie liczoną – a wszystkimi cząsteczkami naczynia. Następnie z tych sił wyznacza się wektor przyśpieszenia i dodaje do wcześniej wyznaczonego wektora przyśpieszenia (14). Istotę tego rozwiązania obrazuje (Rysunek 6).



Rysunek 6. Działanie warunków brzegowych

Po dodaniu przyśpieszenia wynikającego z sił kontaktowych wzór na przyśpieszenie *i-tej* cząsteczki prezentuje się następująco:

(14) $$\vec{a}_i = -\frac{c^2 c_v}{c_p}\sum_{j=1}^{N} m_j \left(\frac{1}{\rho_i}+\frac{1}{\rho_j}+\Pi_{ij}\right)\nabla W_{ij} + \frac{1}{m_v}\sum_{k=1}^{M}\vec{F}_{ik}$$

Idea wprowadzenia warunków brzegowych jest już znana, należy teraz wybrać odpowiednią funkcję, która obliczy siły kontaktowe. W mojej pracy zdecydowałem się na użycie funkcji na siły kontaktowe zaproponowanej w publikacji [6], wyrażającej się wzorem:

(15) $$\vec{F}_{ik} = \begin{cases} \dfrac{\left[\left(\frac{r_0}{|r_{ik}|}\right)^{12} - \left(\frac{r_0}{|r_{ik}|}\right)^{4}\right] K\vec{r}_{ik}}{|r_{ik}|^2} & dla\ \dfrac{|r_{ik}|}{r_0} \le 1 \\ 0 & dla\ \dfrac{|r_{ik}|}{r_0} > 1 \end{cases}$$

Gdzie *K* to parametr zależny od implementacji, jednak zgodnie ze źródłem [6] wybrany jako kwadrat największej prędkości cząsteczki cieczy w układzie. r_0 jest natomiast dystansem odcięcia, w mojej pracy równym 0.015, obranym metodą prób i błędów.

3. Siły grawitacyjne

W przeciwieństwie do warunków brzegowych, procedura dodania przyśpieszenia wynikającego z sił grawitacyjnych jest prosta. Do wektora przyśpieszenia dodajemy skierowany pionowo w dół wektor o długości ziemskiego przyspieszenia $g = 9.8$. Ostateczny wzór na przyśpieszenie *i-tej* cząsteczki jest następujący:

(16) $$\vec{a}_i = -\frac{c^2 c_v}{c_p}\sum_{j=1}^{N} m_j \left(\frac{1}{\rho_i}+\frac{1}{\rho_j}+\Pi_{ij}\right)\nabla W_{ij} + \frac{1}{m_v}\sum_{k=1}^{M}\vec{F}_{ik} + \vec{g}$$

4. Wyznaczanie prędkości i położenia

W celu wyznaczenia prędkości z przyspieszenia należy scałkować przyśpieszenie po czasie. Jest to w mojej pracy realizowane *metodą Eulera*. Metoda ta jest stosunkowo wydajna, problemem jest jednak często mała dokładność. Dla zastosowań obecnych w mojej pracy nadaje się jednak bardzo dobrze. Wymaga ona założenia warunków początkowych, ponieważ każdy następny krok korzysta z wartości poprzedniego, co ilustruje wzór:

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{17}$$

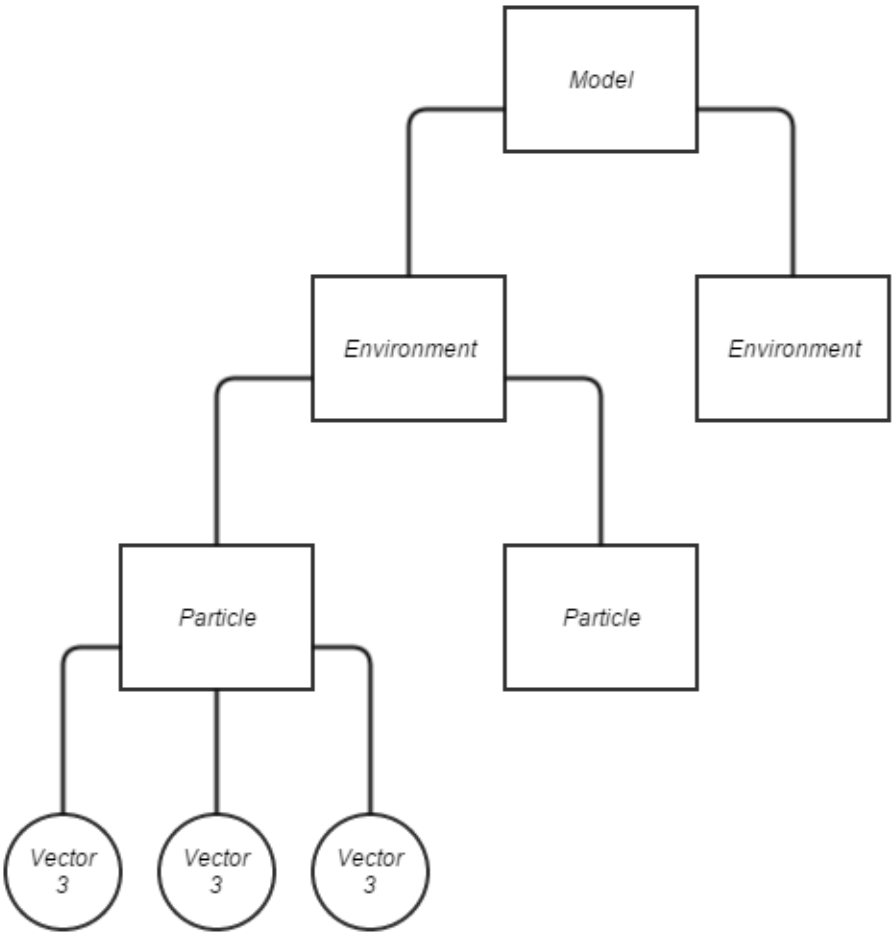
gdzie y_{n+1} jest krokiem następnym, y_n obecnym, h jest krokiem czasowym. W analogiczny sposób z otrzymanej prędkości obliczamy położenie.

4. Implementacja

1. Platforma deweloperska

Implementacja metody Smoothed-Particles Hydrodynamics została wykonana w języku programowania *C++*, w środowisku *Microsoft Visual Studio 2013*. Obejmuje ona dwa projekty – pierwszy – silnik obliczeniowy, czyli właściwą implementację SPH oraz drugi – program napisany przy użyciu biblioteki *OpenGL* służący wizualizacji wyników.

2. Specyfikacja silnika obliczeniowego



Rysunek 7. Struktura klas projektu

Struktura projektu podzielona została na klasy, które zawierają się wzajemnie. Ułatwia to modyfikację kodu oraz pozwala na jego późniejszy rozwój. struktura klas

zastosowana w projekcie przedstawiona została na diagramie (Rysunek 7). Omawianie poszczególnych klas rozpocznę od dołu tego schematu.

1. Klasa Vector3

Klasa `Vector3` jest trójwymiarowym wektorem. Zdecydowałem się na stworzenie własnej klasy, a nie korzystanie ze standardowej biblioteki <*vector*>, ponieważ zależało mi na prostocie funkcjonalności i możliwości pełnej kontroli nad zachowaniem instancji tej klasy. Elementy tej klasy – zarówno pola jak i metody – obrazuje (Tabela 1).

`Vector3<T>`		
		Pola
Typ	*Nazwa*	*Komentarz*
T	**x**	Składowa x
T	**y**	Składowa y
T	**z**	Składowa z
		Metody
	Vector3()	Konstruktor bezargumentowy, zeruje współrzędne.
	Vector3(*T _x, T _y, T _z*)	Konstruktor wieloargumentowy, ustawiający współrzędne wg podanych argumentów.
void	**zeroIt**()	Metoda zerująca współrzędne wektora.
float	**length**()	Metoda zwracająca długość wektora.
ostream	**operator** <<	Drukuje współrzędne wektora.
Vector3	**operator** */arytm./*	Przeciążone operatory zwracające wartości logiczne dla działań algebraicznych na wektorach, włączając w to działania wektora z wektorem, jak i również wektora z wartością skalarną.
float	**operator ***	Zwraca iloczyn skalarny dwóch wektorów.

Tabela 1 Klasa `Vector3`

2. Klasa Particle

Klasa `Particle` jest odzwierciedla cząsteczkę w symulacji. Zawiera wszystkie dane charakterystyczne dla niej oraz korzysta z klasy `Vector3` w celu przechowywania informacji na temat położenia, prędkości i przyśpieszenia. Jej struktura przedstawiona została w (Tabela 2).

`Particle<T>`		
		Pola
Typ	*Nazwa*	*Komentarz*
Vector3<T>	**r**	Wektor położenia
Vector3<T>	**v**	Wektor prędkości
Vector3<T>	**a**	Wektor przyśpieszenia
T	**density**	'Gęstość' cząsteczki
		Metody
	Particle()	Konstruktor bezargumentowy, zeruje pola.
	Particle (*T _x, T _y, T _z, T _d*)	Konstruktor wieloargumentowy, ustawiający współrzędne położenia oraz gęstość wg podanych argumentów.
void	**setPosition**(*T _x, T _y, T _z*)	Ustawia położenie cząsteczki na podane w argumentach współrzędne wektora.
ostream	**operator** <<	Drukuje współrzędne wszystkich wektorów.
Particle	**operator** =	Zwraca cząsteczkę równą podanej jako argument.

Tabela 2 Klasa `Particle`

3. Klasa Environment

Klasa `Environment` opisuje środowisko, w jakim znajdują się „cząsteczki” cieczy. Symbolizuje ona ośrodki - np. H_2O, O_2 . Jest to inaczej mówiąc zbiór cząsteczek cieczy lub gazu. W tej klasie generowane są położenia cząsteczek. Stworzona została z myślą o rozwoju aplikacji, gdzie możliwe będzie symulowanie mieszania się różnych ośrodków – np. symulowanie kieszeni powietrznych, mieszanie się substancji o różnej gęstości. W obecnej implementacji istnieje jedno środowisko – woda. Klasa `Environment` przechowuje w swoich polach informacje charakterystyczne dla tego ośrodka.

`Environment<T>`		
		Pola
Typ	*Nazwa*	*Komentarz*
`Particle<T>[]`	**`particle`**	Tablica cząsteczek
`float`	**`mass`**	Masa cząsteczki w tym ośrodku
`float`	**`density`**	Gęstość ośrodka
`float`	**`pressure`**	Ciśnienie panujące w ośrodku
`float`	**`k`**	Stosunek c_p/c_v
`long int`	**`quantity`**	Ilość cząsteczek w ośrodku
`string`	**`name`**	Nazwa ośrodka
		Metody
	`Environment``()`	Konstruktor bezargumentowy.
	`Environment``(`*`long int q, float m, float d, float p, float kk, string n`*`)`	Konstruktor wieloargumentowy, ustawiający wszystkie stałe dla danego ośrodka oraz jego nazwę.
`ostream`	**`operator`** `<<`	Drukuje informacje o środowisku.
`Environment`	**`operator`** `=`	Zwraca środowisko równe podanemu jako argument.

Tabela 3 Klasa `Environment`

4. Klasa Model

Klasa `Model` odpowiada za wszystkie obliczenia, to w niej wykonywany jest cały algorytm Smoothed-Particle Hydrodynamics, zapisywanie do plików wynikowych i czytanie z plików konfiguracyjnych. Ponadto, w tej klasie, następuje deklaracja stałych parametrów używanych w SPH. Zaliczają się do nich między innymi – krok czasowy *dt*, stała *g*, parametr długości wygładzania *h*. Skład klasy `Model` ilustruje (Tabela 4 Klasa `Model`).

`Model<T>`		
		Pola
Typ	*Nazwa*	*Komentarz*
const T	**g**	Stała grawitacyjna
const T	**dt**	Krok czasowy
const T	**n**	Parametr eta
const T	**beta**	Parametr beta
const T	**alpha**	Parametr alfa
const T	**h**	Parametr długości wygładzania
int	**qVessel**	Ilość cząsteczek naczynia
Vector3<T>[]	**vessel**	Tablica wektorów położeń cząsteczek naczynia
int	**quantity**	Ilość ośrodków w symulacji
Environment<T>[]	**environment**	Tablica ośrodków
ofstream	**positionFile**	Strumień wyjściowy dla pliku z pozycjami cząsteczek
ofstream	**vesselFile**	Strumień wyjściowy dla pliku z pozycjami cząsteczek naczynia

ofstream	**configFile**	Strumień wejściowy dla pliku z konfiguracją
ofstream	**vFile**	Strumień wyjściowy dla pliku z prędkościami cząsteczek
ofstream	**aFile**	Strumień wyjściowy dla pliku z przyśpieszeniami cząsteczek
Vector3<T>	**gravity**	Wektor przyśpieszenia wynikającego z sił grawitacji
		Metody
	Model(*int environmentsQuantity, long int* particlesQuantity, T* mass, T* density, T* pressure, T* k, string* name, int frames, int _qVessel*)*	Konstruktor, który wypełnia pola, tworzy ośrodki o podanych ilościach cząsteczek, masach, gęstościach, ciśnieniach, c_p/c_v i nazwach, tworzy pliki wyjściowe, wywołuje główne metody liczące, a po zakończonej symulacji zamyka pliki.
void	**calculateEnergy()**	Liczy energię wszystkich cząsteczek cieczy/gazu
T	**W***(int envI, int envJ, long int i, long int j)*	Funkcja jądra interpolacji.
ostream	**operator** <<	Drukuje informacje o modelu.
Vector3<T>	**gradW**(*int envI, int envJ, long int i, long int j)*	Gradient funkcji jądra interpolacji.

void	**updateDensity**(*int env, long int i)*	Ponownie przelicza gęstość dla cząsteczki cieczy/gazu.
void	**calculateDensity()**	Przelicza gęstość wszystkich cząsteczek cieczy/gazu.
void	**createVessel()**	Tworzy naczynie.
T	**maxVelocity()**	Wybiera największą prędkość cząsteczki w układzie.
Vector3<T>	**F**(*int env, long int i, int j*)	Liczy siłę kontaktową cząsteczki cieczy/gazu z cząsteczką naczynia.
Vector3<T>	**calculateBoundaryConditions**(*int env, long int i)*	Liczy wypadkową siłę z jaką cząsteczki naczynia działają na cząsteczkę cieczy/gazu.
void	**calculateAcceleration()**	Główna metoda SPH – liczy przyśpieszenia dla wszystkich cząsteczek cieczy/gazu.
void	**calculateNextVelocityAndPosition()**	Liczy prędkości i położenia dla następnego kroku symulacji.

Tabela 4 Klasa `Model`

3. Omówienie głównych metod

Za obliczanie przyśpieszeń cząsteczek cieczy – bądź gazu – odpowiada metoda `calculateAcceleration`. Zaimplementowany w niej jest wzór (16) na przyśpieszenie *i-tej* cząsteczki układu omówiony w rozdziale 3. Funkcja ta wykonuje pętlę po wszystkich środowiskach w układzie (indeks *i*), w niej zawarta jest pętla po wszystkich cząsteczkach danego ośrodka (indeks *j*), a w tejże pętla, której zadaniem jest liczenie oddziaływań ze wszystkimi cząsteczkami w układzie (także w innych ośrodkach). Zakładając, że symulujemy jeden ośrodek osiągamy tutaj złożoność obliczeniową $O(n^2)$.

```
...
environment[i].particle[j].a.zeroIt();
 for (long int l = 0; l < environment[i].quantity; l++)
 {
  if (l != j)
  {
   Type my = 0;
   Vector3<Type> v = environment[i].particle[j].v -
                     environment[i].particle[l].v;
   Vector3<Type> r = environment[i].particle[j].r -
                     environment[i].particle[l].r;
   if (v*r < 0)
    my = (v*r) / ((r.lenght()*r.lenght() / h) + h*n*n);

   Type density = (environment[i].particle[j].density +
                   environment[i].particle[l].density) / 2;
   Type product = (-alpha*c*my + beta*my*my) / density;
   Vector3<Type> tempA;
   tempA = gradW(i, i, j, l);
   tempA *= environment[i].mass * (1 /
            environment[i].particle[j].density+ 1 /
            environment[i].particle[l].density + product);
   environment[i].particle[j].a += tempA;
  }
 }
 environment[i].particle[j].a *= -c*c*(1 / environment[i].k);
 environment[i].particle[j].a += gravity;

 Vector3<Type> boundaryForces = calculateBoundaryConditions(i, j);
 boundaryForces *= 0.2;
 environment[i].particle[j].a += boundaryForces;

...
```

Listing 1. Fragment metody calculateAcceleration()

Na przedstawionym w Listing 1 fragmencie metody zostały wykonane następujące czynności: wyzerowanie wektora przyśpieszenia dla cząsteczki *j-tej*, w środowisku i-tym, obliczenie wektorów wypadkowych dla prędkości i położenia, wartości μ_{jl}, gęstości uśrednionej, wartości Π_{jl}, funkcji jądra interpolacji, następnie dodanie obliczonego wektora tymczasowego `tempA` do wektora `a`. Kolejnym krokiem jest pomnożenie przez stałe oraz dodanie wektora grawitacji. Wektor `boundaryForces` przechowuje wypadkowy wektor sił kontaktowych zachodzących między cząsteczką *j-tą* cieczy i wszystkimi cząsteczkami naczynia. Obliczony w ten sposób wektor przyśpieszenia wynikający z warunków brzegowych dodawany jest do wektora przyśpieszenia cząsteczki *j-tej*. Jest to jeden krok pętli po wszystkich cząsteczkach cieczy/gazu – wszystkie te obliczenia wykonywane są *n* razy, gdzie *n* to ilość cząsteczek symulowanych.

```
Type r0 = 0.015;
Vector3<Type> r = environment[env].particle[i].r - vessel[j];
```

```
Type dr = r.lenght();
if (dr == 0)
 cout << "\b ERROR" << endl;

Vector3<Type> force = Vector3<Type>();

if (dr / r0 <= 1)
{
 Type K = maxVelocity();
 force = r;
 force *= (K / powf(dr, 2));
 force *= (powf(r0 / dr, 12) - powf(r0 / dr, 4));
}
return force;
```

Listing 2. Funkcja licząca siły kontaktowe.

Powyżej na Listing 2 przedstawiono algorytm liczenia wektora sił kontaktowych. Zaczyna się od definicji dystansu odcięcia `r0`, który bezpośrednio wpływa na odległość, na jaką cząsteczka cieczy może zbliżyć się do cząsteczki naczynia. Następnie liczony jest wektor położenia między cząsteczką cieczy a naczynia. Tworzony jest wektor siły, wynoszący domyślnie `(0, 0, 0)`, po czym wykonywane są obliczenia ze wzoru (15); funkcja `maxVelocity` wybiera największą prędkość cząsteczki w układzie – na koniec zwracana jest wartość wektora `force`. Funkcję tę wywołuje się w pętli, dla każdej *i-tej* cząsteczki cieczy z każdą *j-tą* cząsteczką naczynia.

```
Type norm = 1 / (3.14 * powf(h, 3));

Vector3<Type> r = environment[envI].particle[i].r -
                  environment[envJ].particle[j].r;
Type u = r.lenght() / h;

if (u >= 0 && u <= 1)
 return norm*(1 - 3 / 2 * u*u + 3 / 4 * powf(u, 3));
else if (u > 1 && u <= 2)
 return norm*(1 / 4 * powf((2 - u), 3));
else
 return 0;
```

Listing 3 Funkcja jądra interpolacji.

Powyższy listing przedstawia implementację funkcji jądra interpolacji. Zmienna `norm` jest parametrem, który wynosi $\frac{1}{\pi h^3}$. Potęga nad parametrem długości wygładzania zależy od ilości wymiarów w symulacji. Zmienna `u` natomiast jest stosunkiem długości wypadkowego wektora między cząsteczkami i parametru `h`. Podobnie wygląda gradient funkcji jądra interpolacji (Listing 4).

```
Type norm = 1 / (3.14 * powf(h, 4));

Vector3<Type> r = environment[envI].particle[i].r -
                  environment[envJ].particle[j].r;
Type u = r.lenght() / h;

r /= r.lenght();

Type endW = 0;

if (u >= 0 && u <= 1)
 endW = -3 * u + 9 / 4 * u*u;
else if (u > 1 && u <= 2)
 endW = -3 / 4 * powf((2 - u), 2);

r *= norm * endW;

return r;
```

Listing 4 Gradient funkcji jądra interpolacji.

Drugą najważniejszą wielkością do obliczenia, po przyśpieszeniu cząsteczki, jest jej położenie i prędkość. To wektory położenia posłużą do przygotowania wizualizacji. Jak wcześniej wspomniałem do obliczenia położenia użyłem metody Eulera. Jej implementacja używana zarówno do obliczenia prędkości, jak i położenia, jest przeprowadzona w metodzie `calculateNextVelocityAndPosition` (Listing 5).

```
for (long int j = 0; j < environment[i].quantity; j++)
{
 environment[i].particle[j].v += environment[i].particle[j].a *dt;
 environment[i].particle[j].r += environment[i].particle[j].v *dt;

 positionFile.precision(5);
 positionFile << environment[i].particle[j].r << endl;

 vFile << environment[i].particle[j].v << endl;
 aFile << environment[i].particle[j].a << endl;
}
```

Listing 5 Implementacja metody Eulera

Dla każdego *i-tego* ośrodka liczymy prędkość i położenie w następnym kroku. Całość zapisujemy wszystko do pliku. Położenia w celu wyświetlenia symulacji, a prędkość i przyśpieszenia w celu weryfikacji poprawności zachowania układu.

Wymienione fragmenty kodu są najważniejszymi w całej aplikacji. Korzystają one z dużej ilości parametrów, których odpowiednie dobranie było jednym z największych problemów implementacyjnych.

4. Działanie programu

Program podzielony został na dwie części. Pierwsza z nich – obliczeniowa – została omówiona szczegółowo powyżej. Odpowiada ona za przeprowadzenie symulacji i zapisanie jej wyników do pliku. Rozpoczyna swoje działanie od wczytania pliku konfiguracyjnego *config.cfg*, który zawiera w konkretnej kolejności – liczbę klatek symulacji, liczbę cząsteczek cieczy, liczbę cząsteczek naczynia. Wyniki obliczeń zapisywane są do plików wynikowych – *pozycje.dat, naczynie.dat, przyspieszenia.dat, predkosci.dat*. Druga część, realizowana przez program korzystający z biblioteki OpenGL, ma za zadanie wyświetlenie wyników symulacji. Wczytuje ten sam plik konfiguracyjny co poprzednio oraz wynikowy plik z położeniami cząsteczek.

Położenia zapisywane są do pliku w postaci wektorów, a przykładowy fragment takiego pliku przedstawiony został na Listingu 6.

```
...
0.01  0.033866    -0.03
0.01  0.030357    -0.03
0.02  0.022922    -0.03
0.02  0.029544    -0.03
0.02  0.032428    -0.03
0.02  0.031422    -0.03
0.02  0.030817    -0.03
0.03  0.026541    -0.03
0.03  0.025519    -0.03
0.03  0.033513    -0.03
0.03  0.037615    -0.03
0.03  0.039466    -0.03
...
```

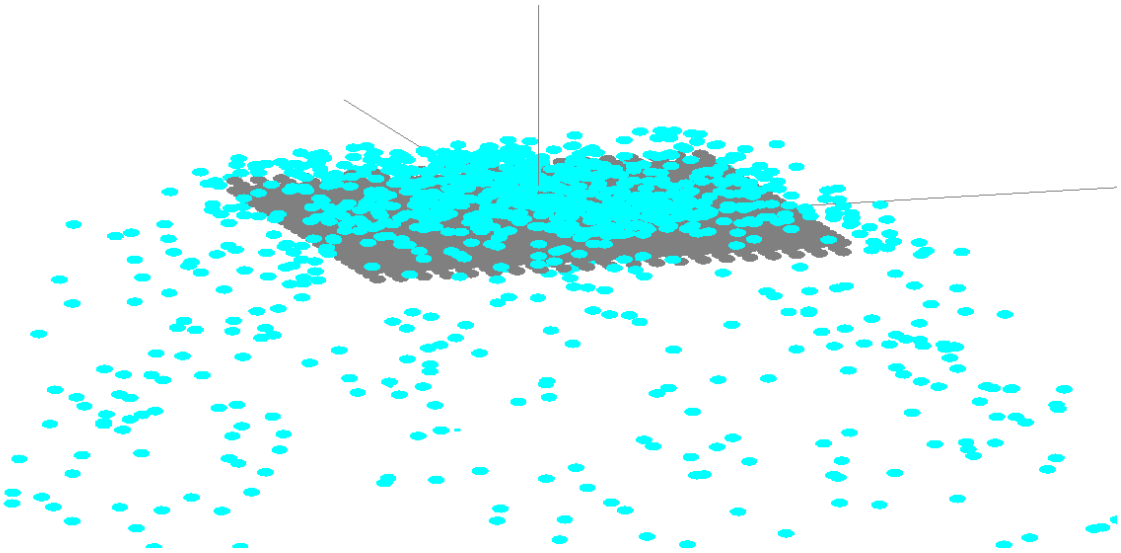
Listing 6 Fragment pliku wynikowego - pozycje.dat

Zrealizowane zostało to w ten sposób, ponieważ obecna implementacja programu działa na CPU, który do obliczeń tego typu – wielu powtarzalnych operacji arytmetycznych - jest mało wydajny. To natomiast skutkowałoby małą liczbą klatek na sekundę.

W planach rozwoju programu jest użycie technologii Nvidia CUDA, która pozwala na zrównoleglenie obliczeń i przeniesienie ich na układ GPU.

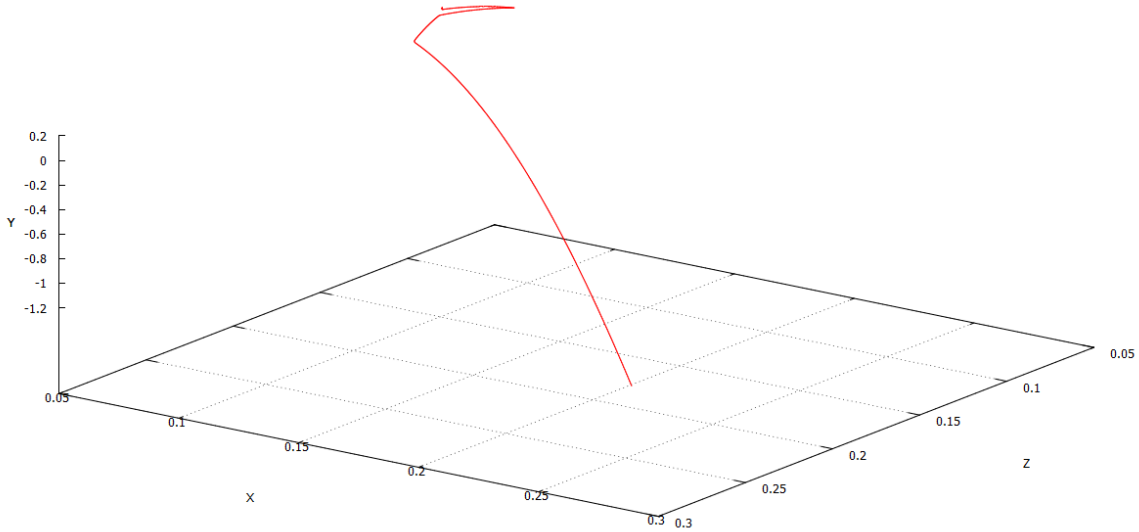
5. Przykładowa symulacja

Celem przeprowadzonej przeze mnie przykładowej symulacji było wygenerowanie położeń cząsteczek wody opadających z pewnej wysokości na płaskie naczynie. Prezentowana symulacja została przeprowadzona dla 1125 cząsteczek wody, 882 cząsteczek naczynia, w 400 klatkach (krokach czasowych).



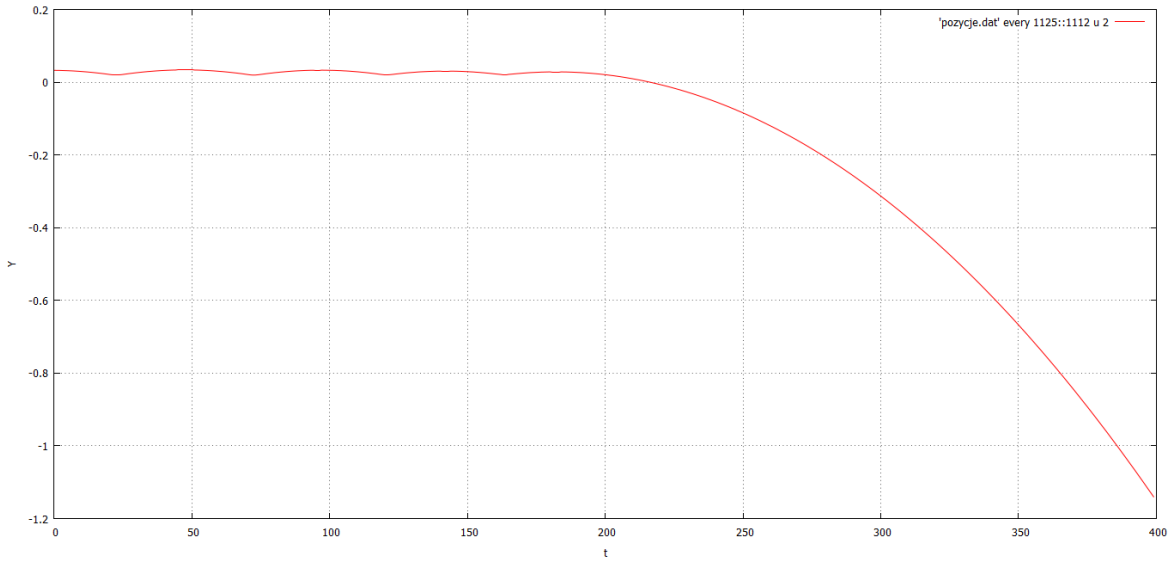
Rysunek 8. Klatka z symulacji.

Na rysunku 8 zaprezentowano jedną z 400 klatek symulacji. Symulacja pokazuje cząsteczki wody spływające z płaskiej, kwadratowej powierzchni. Na wykresie 1 przedstawiono zmiany położenia wybranej losowo cząsteczki na przestrzeni 400 klatek symulacji.



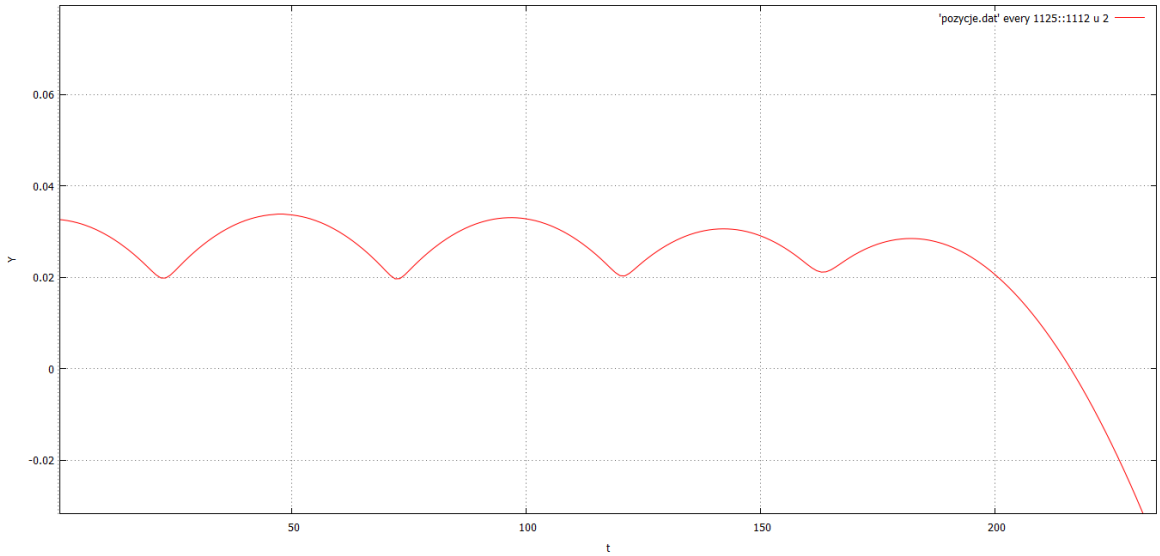
Wykres 1. Zmiana położenia losowej cząsteczki

Największe zmiany obserwujemy oczywiście od momentu wyjścia poza powierzchnię naczynia, wtedy to przyśpieszenie g może bez przeszkód działać na cząsteczkę. Wykres 2 prezentuje zmianę wysokości cząsteczki względem ustalonego układu współrzędnych. Zaobserować można na nim wszystkie odbicia od powierzchni naczynia, po czym spadek swobodny po przekroczeniu jego granicy. Cząsteczka zderzyła się dokładnie



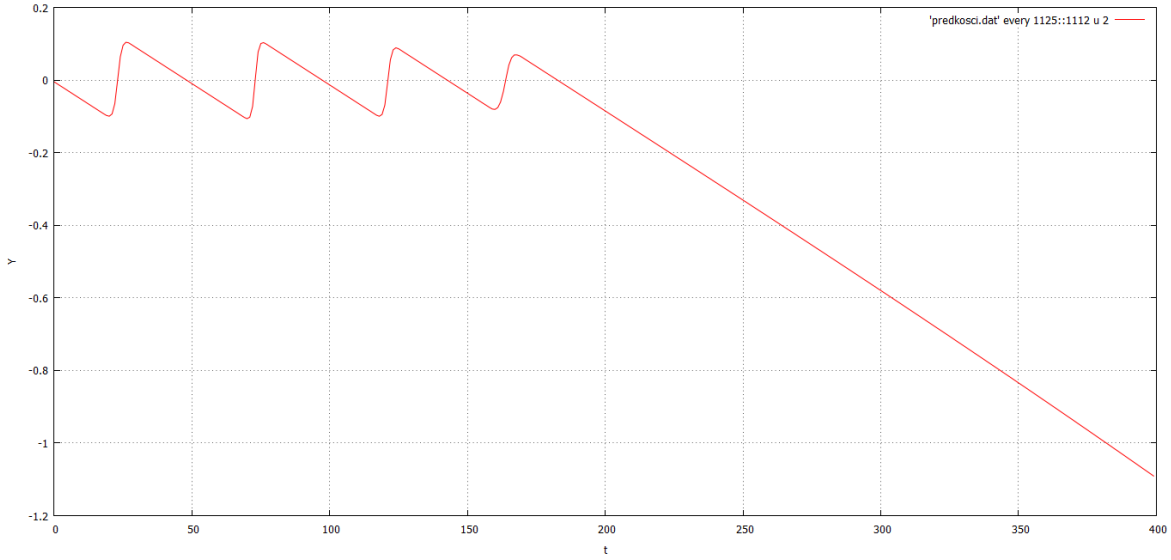
Wykres 2. Zmiana położenia cząsteczki względem osi OY

cztery razy z naczyniem, po czym spadła z jego powierzchni. Wykres zgodny jest z wartościami teoretycznymi. Po zmianie zakresu zmiennej Y (Wykres 3) zaobserwować można, że wysokość maleje z każdym odbiciem. Oznacza to, że odbicia nie są sprężyste.



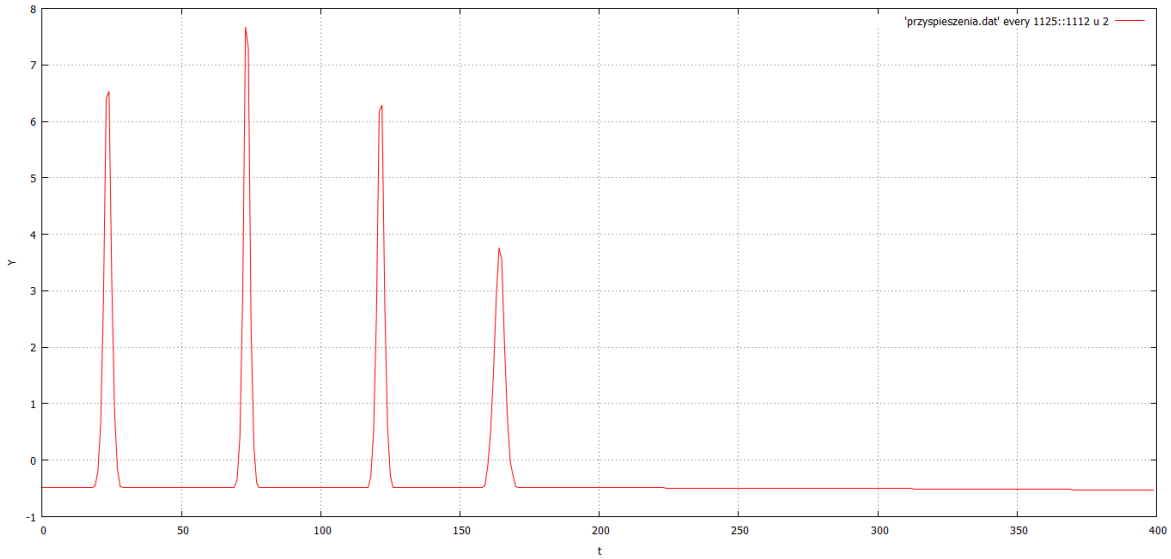
Wykres 3. Zmiana położenia cząsteczki względem osi OY w skali 200 klatek

Na wykresie 4 przedstawiono natomiast pierwszą pochodną po czasie położenia czyli prędkość cząsteczki względem osi OY. Widać na nim, że prędkość maleje z każdym odbiciem od powierzchni naczynia, gdy jednak przekroczy jego krawędź gwałtownie rośnie.



Wykres 4. Zmiana prędkości cząsteczki względem osi OY

Najważniejszy jednak, w kontekście samej metody Smoothed-Particles Hydrodynamics, wykres 5 obrazujący zmiany przyśpieszenia względem osi OY na przestrzeni 400 klatek symulacji. Wartości na nim przedstawione są bezpośrednim rezultatem działania metody SPH.



Wykres 5. Zmiana przyśpieszenia wybranej cząsteczki w czasie na osi OY

6. Podsumowanie

Smoothed-Particles Hydrodynamics okazała się trudną w implementacji metodą symulacji cieczy. Trudności spowodowane były głownie niejasno określonymi sposobami wprowadzenia warunków brzegowych oraz ogromną liczbą parametrów, które nieokreślone w żadnym oficjalnym i wiarygodnym dokumencie w większości musiały być ustalone metodą prób i błędów. Niejasnym przez długi czas było dla mnie także pojęcie „cząsteczki", bowiem to właśnie zmiana podejścia – z faktycznej cząsteczki na kroplę cieczy – zaowocowała postępami w pracy nad programem.

Dużo do życzenia pozostawia także wydajność metody – dla każdej cząsteczki musimy przeprowadzić obliczenia jej reakcji z innymi cząsteczkami – tutaj zaletą jest funkcja jądra interpolacji i ograniczenie parametru długości wygładzania. Dzięki nim nie trzeba wykonywać wszystkich obliczeń, jeśli cząsteczki są zbyt oddalone od siebie.

Metoda daje realistyczne wyniki dla liczby cząsteczek cieczy powyżej 1000. Poniżej tej wartości jest za mało oddziaływań międzycząsteczkowych stabilizujących cały układ. Skutkuje to zazwyczaj zbyt dużymi przyśpieszeniami cząsteczek po kontakcie z podłożem. Na procesorze *Intel Core i5 2410M* symulacja przedstawiona w rozdziale 5 zajęła około 15 minut. Czas ten pokazuje, że implementowanie takich rozwiązań na procesorach CPU może stanowić jedynie materiał badawczy, nie ma jednak zastosowań praktycznych dla symulacji w czasie rzeczywistym. Moim planem na rozwój aplikacji jest implementacja technologii CUDA, która pozwoli na przeniesienie obliczeń na GPU, które przy takich zadaniach – wielokrotnie powtarzanie takich samych czynności – sprawuje się dużo lepiej. Wzrost wydajności przy użyciu CUDA w porównaniu do zwykłego CPU może być nawet 20-krotny [5]. Implementacja metody SPH na GPU oraz optymalizacja obliczeń i klas pozwoli na symulację cieczy w czasie rzeczywistym, co jest moim głównym celem na przyszłość w rozwoju tej aplikacji. Dodatkowo w planach jest rysowanie wody jako powierzchni, a nie zbioru kropel.

Mimo wielu trudności cel pracy został osiągnięty. Z powodzeniem przeprowadzono symulację wody opadającej na płaskie podłoże, która przypomina realne zachowanie wody. Umiejętności przyswojone w trakcie implementacji metody SPH pomogą mi w przyszłości w pisaniu kodu wydajnego oraz odpowiednim dobieraniu narzędzi programistycznych.

7. Literatura

[1] Dullemond, C. P. i Wang, H. H. (2009). Numerische Strömungsmechanik.

[2] Manenti, D. S. (2009). A Smoothed Particle Hydrodynamics: Basics and Applications. Dipartimento di Meccanica Strutturale - Università degli Studi di Pavia.

[3] Monaghan, J. (2005). Smoothed particle hydrodynamics. Australia.

[4] Monaghan, J. J. i Gingold, R. A. (1977). Smoothed particle hydrodynamics: theory and application to non-spherical stars.

[5] NVIDIA Corporation. (2012). *NVIDIA PhysX SDK Documentation*.

[6] Yildiz, M., Rook, R. A. i Suleman, A. (2008). SPH with the multiple boundary tangent method. *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*.