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

Łukasz Zieliński
nr albumu: 236439

Praca magisterska
na kierunku Informatyka Stosowana

Symulacja cieczy metodą Smoothed Particle Hydrodynamics (SPH). Implementacja wieloplatformowej biblioteki w C#

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski
Instytut Fizyki
Zakład Mechaniki Kwantowej

Toruń 2015

Pracę przyjmuję i akceptuję

.....
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....
data i podpis pracownika dziekanatu

Dziękuje mojemu promotorowi za poświęcony czas i udzieloną pomoc.

Spis treści

Rozdział 1

Wprowadzenie

1.1. Zakres pracy

W roku 1977 R. A. Gingold i J.J. Monaghan oraz L.B. Lucy zaproponowali metodę SPH (smoothed particle hydrodynamics)[1] do obliczeń z dziedziny astrofizyki. Jej główną zaletą było zmniejszenie używanych zasobów komputerowych przy zachowanej dokładności[2]. W krótkim czasie okazało się, że metoda nadaje się również do symulowania zjawisk przepływu płynów. Współcześnie jest powszechnie wykorzystywana do modelowania cieczy. Jednym z wielu przykładów może być przeprowadzona w 2008 roku symulacja modyfikacji terenów 3D przez erozję hydrauliczną w której wykorzystano 25 000 tysięcy cząstek[3].

Metoda SPH należy do grupy bezsiatkowych metod cząstek, gdzie modelowana ciecz jest reprezentowana przez zbiór punktów. Każdemu z nich przypisany jest zestaw właściwości fizycznych cząstek takich, jak położenie, masa, czy prędkość. Na ich podstawie obliczane są oddziaływania między cząstkami, przy pomocy których obliczane są nowe położenia i prędkości. Metody cząstek są przykładem problemu oddziałujących N ciał, gdzie złożoność jest rzędu $O(N^2)$. Z tej przyczyny w celu zmniejszenia złożoności w metodzie SPH wprowadza się promień odcięcia. Przy pomocy którego wyznacza się sąsiedztwo cząstki. Ustalenie sąsiedztwa cząstki, wymaga podziału przestrzeni na komórki, natomiast promień odcięcia wyznacza zbiór komórek, które do sąsiedztwa należą. W tych komórkach znajdują się cząstki będące sąsiedztwem analizowanej cząstki. Oddziaływania między cząstkami są obliczane tylko z cząstkami, które stanowią jej sąsiedztwo. Wprowadzenie takiego sposobu obliczania oddziaływań międzycząstkowych pozwala zmniejszyć złożoność obliczeniową, kosztem złożoności pamięciowej do rzędu O(N).

Symulowanie zjawisk fizycznych niemal zawsze wiąże się ze stosowaniem systemów komputerowych o dużej mocy obliczeniowej i odpowiednio dużych zasobach pamięci operacyjnej. Jednak zgodnie z założeniami niniejszej pracy, obliczenia muszą być przeprowadzone w czasie rzeczywistym i niekoniecznie na maszynach wieloprocesorowych o dużej mocy obliczeniowej. To zmusza do wprowadzania modelowych parametrów

[1] R. A. Gingold, J. J. Monaghan and L.B. Lucy Smoothed particle hydrodynamics: theory and application to non-spherical stars Institute of Astronomy

[2] Monaghan J.J. Smoothed particle hydrodynamics In: Annual review of astronomy and astrophysics.

[3] P. Krištof1, B. Beneš1, J. Křivánek2, O. Šťava1 Hydraulic Erosion Using Smoothed Particle Hydrodynamics Journal

symulacji na relatywnie niewielkiej liczbie cząstek w celu uzyskania oczekiwanego efektu. Przybliżone wyniki są do zaakceptowania np. w grach komputerowych, gdzie mniejsze znaczenie ma odwzorowanie fizycznych parametrów symulacyjnych, a najistotniejszą rolę odgrywa ilość zużytych zasobów i efekt wizualny.

Innym ważnym aspektem tego rodzaju symulacji jest potrzeba wprowadzenia dodatkowych rozwiązań w celu np. obliczania kolizji między cząstkami wody, a innymi obiektami. W tym celu w niniejszej pracy użyto algorytmu SAT (ang. Separating Axis Theorem) do detekcji kolizji między obiektami o wypukłym kształcie. Algorytm pozwala wyeliminować szukanie punktów kolizji dla wszystkich par obiektów i ograniczyć się jedynie do obiektów, dla których istnieje pewność, że zachodzą na siebie. W przypadku symulacji z dużą liczbą potencjalnych obiektów między którymi może dojść do kolizji, takie podejście przełoży się na zmniejszenie liczby wykonywanych operacji, co w większości przypadków przełoży się na zwiększoną szybkość działania aplikacji, przy zachowaniu dużej dokładności.

1.2. Cele pracy

Istnieje wiele metod służących do symulowania cieczy w czasie rzeczywistym. Wiele spośród nich znalazło zastosowanie w różnych dziedzinach nauki i inżynierii. W tej pracy skupiono się na grupie zastosowań dla branży gier komputerowych, w której mniej istotne jest przeprowadzenie dokładnej symulacji fizycznej, a głównym celem jest wizualizacja danego zjawiska fizycznego.

W niniejszej pracy zaproponuję efektywną implementację metody cząstkowej SPH, której głównym przeznaczeniem jest branża rozrywkowa. Stworzenie wydajnej implementacji silnika gry komputerowej opartej na cieczy nie jest trywialne z uwagi na zwiększone wymagania symulacyjne, takie jak praca w czasie rzeczywistym oraz wykrywanie kolizji ze zróżnicowanymi obiektami. W pracy opisano:

- efektywną implementację istniejącego algorytmu SPH, w której zmniejszono złożoność przeprowadzanych obliczeń do minimum,
- rozbudowany silnik gry opartej na modelowaniu cieczy z możliwością zmiany parametrów fizycznych, edycji map,
- implementację algorytmu SAT służącą do wykrywania kolizji między cząstkami cieczy i innymi obiektami,
- analizę biznesową przeprowadzoną pod kątem aktualnie używanych urządzeń na rynku.

Ponadto silnik został zrównoleglony dla obliczeń na procesorach wielordzeniowych i zaimplementowany w taki sposób, aby była możliwa zmiana używanej technologii, a nawet języka programowania np. na Javę używaną powszechnie w urządzeniach z systemem Android, bądź Obiectve – C używany w urządzeniach firmy Apple. Na potrzeby pracy utworzono wieloplatformową bibliotekę napisaną w C#, której użyto do symulowania cieczy w grze „Water your plants”. Stworzoną grę zaimplementowano w C#, natomiast do wizualizacji użyto technologii XNA, którą można zastąpić dowolną inną technologią np. OpenGL, Cocos2d , LibGDX itp. Dodatkowo utworzono aplikację na system Android, zintegrowaną z biblioteką OpenGL, w której wykorzystano stworzoną na potrzeby niniejszej pracy bibliotekę silnika SPH.

Praca dowodzi, że model SPH może zostać efektywnie zaimplementowany, dla celów branży rozrywkowej, gdzie zasoby komputowe są wyraźnie ograniczone.

1.3. Streszczenie

Niniejsza praca magisterska składa się z czterech rozdziałów. W rozdziale pierwszym zawarto informacje dotyczące początków powstania, sposobów wykorzystania, a także opisu poszczególnych algorytmów, które używane są w symulacji, bądź są ściśle związane z metodą SPH. Ponadto w rozdziale znajduje się odpowiedź na zasadnicze pytanie pracy, dotyczące zastosowania symulacji cieczy metodą SPH. W omawianym rozdziale wyróżniono także podstawowe cele pracy magisterskiej.

W rozdziale drugim dokonano analizy, opisu, a także podziału bezsiatkowych metod cząstek. W owym rozdziale w szczegółowy sposób opisano także działanie metody SPH. Opis działania algorytmu wymagał zdefiniowania pudła obliczeniowego, wykorzystywanego w modelu SPH. Opisano strukturę algorytmu, który został podzielony na cztery etapy: „Konfiguracja początkowa ”„Gromadzenie informacji o cząstkach”, „Obliczanie sił miedzy cząstkami” oraz „Rozwiazywanie równań ruchu”. W opisie skupiłem się przede wszystkim na zastosowanych w implementacji metodach rozwiązywania równań, czyli: metodzie Eulera i Verleta. Zaprezentowano równania wykorzystywane w obu algorytmach. Wspomniano także o podstawowych zadaniach algorytmu SPH, dla którego również przedstawiono wzory wykorzystane przy implementacji. Ponadto we wspomnianym rozdziale poruszono temat algorytmu SAT (ang. Separating Axis Theorem), służącego w tej pracy do detekcji kolizji cząstek z innymi obiektami.

Rozdział trzeci został poświęcony w całości kwestiom dotyczącym implementacji, m. in. doborowi kroku czasowego, inicjalizacji struktur danych, czy przebiegu pojedynczego kroku symulacji. Podrodział końcowy omawia detekcję kolizji między cząstkami a innymi obiektami na mapie. Wyjaśniono działanie algorytmu odpowiedzialnego za kolizję, a także określono cel jego działania. Omówione zostały przy tym zagadnienia optymalizacyjne.

W czwartym rozdziale wyjaśniłem sposób działania gry *Water your plants*, a także jej zasady. Przybliżono także technologię, w której została przygotowana. Całość rozdziału podzielono na cztery podrozdziały. W pierwszym z nich zamieszczono precyzyjny opis technologii XNA, która została wykorzystana przy tworzeniu gry. Drugi podrozdział zawiera najistotniejsze informacje dotyczące aplikacji *Water your plants,* takie jak zasady działania, czy edytor map. W szczegółowy sposób opisano interfejs aplikacji. W tym podrozdziale za pomocą ilustracji zobrazowano najciekawsze rundy gry. Przedostatni rozdział zawiera informacje na temat zrównoleglenia operacji. Opisano w nim sposób realizacji wielowątkowości i przedstawiono tabelę porównującą wydajność aplikacji dla

wersji jednowątkowej i wielowątkowej. W ostatnim podrozdziale znajdują się rozważania na temat dalszego rozwoju aplikacji. Zamierzam cały czas ją udoskonalać, a także stworzyć wersję gry na inne systemy operacyjne, takie jak Android, czy telefony komórkowe firmy Apple z systemem iOS.

Rozdział 2

Podstawy teoretyczne

Algorytm SPH należy do grupy algorytmów opierających się na bezsiatkowych metodach cząstek (ang. *meshless methods*). Siły działające pomiędzy cząsteczkami modelowanego płynu mają charakter krótkozasięgowy, ograniczony promieniem odcięcia. W poniższych podrozdziałach omówiono następujące zagadnienia: sposób podziału pudła obliczeniowego, podstawowe kroki algorytmu, czyli inicjalizację podstawowych parametrów, obliczanie sił międzycząstkowych i rozwinięcie równań ruchu. Dodatkowo omówiono sposób, w jaki rozwiązano problem wykrywania kolizji przy pomocy SAT (ang. *Separating Axis Theorem*)

2.1. Bezsiatkowe metody cząstek

W modelach cząstek płyn jest reprezentowany przez cząstki, które posiadają parametry fizyczne takie jak położenie, masa czy objętość. Metody cząstek można podzielić ze względu na wielkość symulowanych układów. Dynamika molekularna[1] (DM) jest używana w biochemii do symulacji białek, kwasów nukleinowych oraz innych biomolekuł. „Dynamika molekularna (DM) służy do obliczania trajektorii w przestrzeni fazowej zbioru molekuł, z których każda oddzielnie podlega klasycznym równaniom ruchu.” (Heermann, 1997). Oddziaływania pomiędzy cząsteczkami zależą jedynie od ich położeń względem siebie.

Kolejną ważną metodą należącą do grupy bezsiatkowych metod cząstek jest DPD (ang. *Dissipative particle dynamics*). Opracowana w 1992 roku przez Hoogerbruggena i Koelmana jest techniką symulacji zjawisk zachodzących zarówno w płynach prostych, jak i złożonych. Inspiracją do powstania tej metody była potrzeba symulowania przepływów płynów, płynów wielofazowych lub złożonych płynów takich jak zawiesiny. Dynamika molekularna okazała się niewystarczająca do przeprowadzenia tego rodzaju symulacji ze względu na liczbę cząstek i długie w układach o wymiarach rzędu 10^{-8} - 10^{-4} metra i skali czasowej rzędu 10^{-5} - 10^{-2} sekundy. W metodzie DPD cząstka nie jest utożsamiana z atomem modelowanego płynu, a reprezentuje pewien ciągły obszar cieczy. W procesie powstawania cząstek DPD wartości opisujące właściwości atomów zastępowane są przez wartości średnie dla coraz to większych objętości.

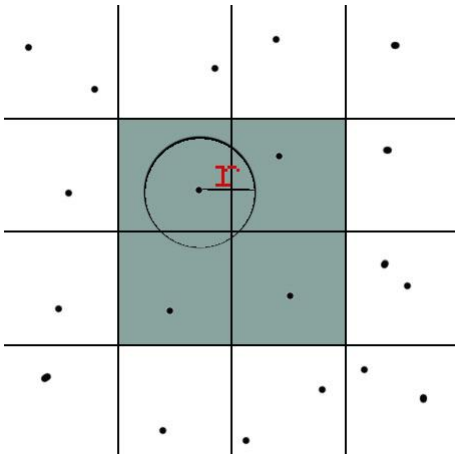
W mojej pracy opisałem metodę SPH (ang. *Smoothed particle hydrodynamics*), która powstała w 1977 na potrzeby symulacji zjawisk astrofizycznych. W kolejnych latach

znalazła ona wiele zastosowań w rożnych dziedzinach nauki. Metoda SPH opiera się na dyskretnym podziale płynu na zestaw cząstek. Cząstki te posiadają długość wygładzenia (ang. *smoothing length*) na obszarze którego ich właściwości fizyczne są wygładzone (ang. *smoothed*) poprzez tzw. funkcje jądra (ang. *kernel function*) Oznacza to, że masa cząstki nie jest skupiona wyłącznie w centrum cząstki, lecz jest rozmyta w jego otoczeniu. Wartość funkcji jądra dąży do zera w nieskończoności, a całka z tej funkcji powinna być równa jedności.

2.2. Podział pudła obliczeniowego na komórki

W bezsiatkowych metodach cząstek złożoność obliczeniowa symulacji osiąga rząd wielkości $O(N^2)$, gdzie N to liczba cząstek w układzie. Tak wysoka złożoność wynika z konieczności obliczenia oddziaływania dla każdej pary cząstek znajdujących się w układzie.

Złożoność obliczeniowa rzędu $O(N^2)$ jest zbyt duża, dlatego aby ją zmniejszyć (kosztem złożoności pamięciowej) stosuje się tzw. promień odcięcia W tym celu należy wyznaczyć odległość między cząstkami i rozdzielić przestrzeń na komórki. W przypadku przestrzeni dwuwymiarowej dzielimy prostokątny obszar na mniejsze prostokąty. Podział jest umowny i nie ma żadnego wpływu na sposób poruszania się cząstek (podział nie stanowi fizycznej bariery). Nie zmienia charakteru bezsiatkowej metody SPH. W momencie wykonywania kroku symulacji używamy promienia odcięcia do określenia sąsiedztwa danej cząstki. Promień odcięcia jest to odległość, która określa jakie komórki są sąsiedztwem danej cząstki. Z każdej komórki pobierane są indeksy znajdujących się w niej cząstek, w ten sposób wyselekcjonowano grupę cząstek stanowiących sąsiedztwo danej cząstki. Między daną cząstką, w konkretnym kroku iteracyjnym, a wybranym sąsiedztwem obliczane są oddziaływania międzycząstkowe. W przypadku pozostałych cząstek w układzie przyjmuje się, że oddziaływania z nimi są zaniedbywalne. Zaprezentowaną koncepcję przedstawiono na rys.1.



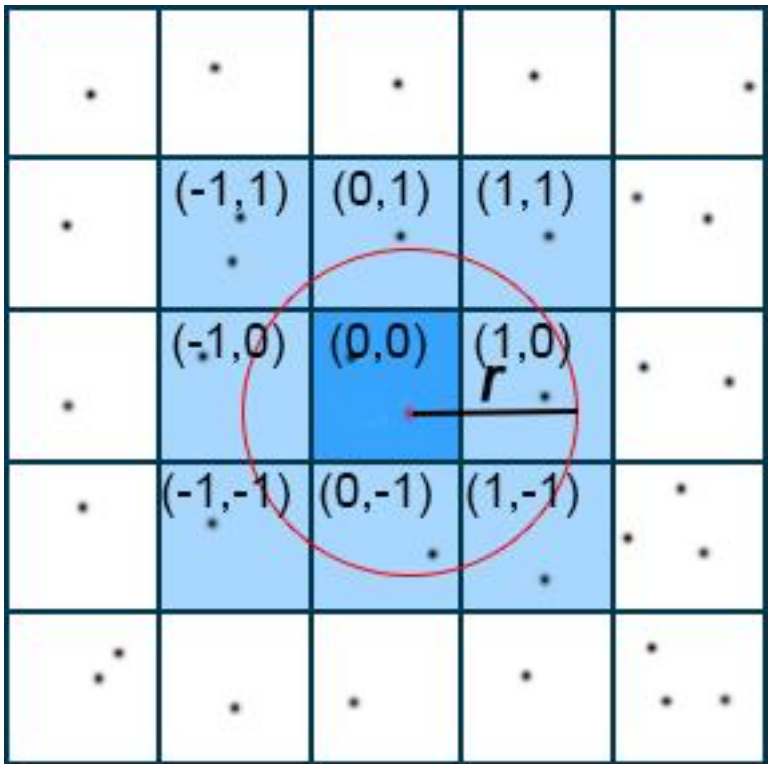
Rysunek 1: Cząstki będące sąsiedztwem dla cząstki i-tej na tle siatki kwadratowej, gdzie r oznacza promień odcięcia

Stosując taki sposób obliczania oddziaływań międzycząstkowych możemy zmniejszyć złożoność obliczeniową z $O(N^2)$ do $O(N)$. Oczywiście należy pamiętać, że odbywa się to kosztem większego użycia pamięci. Jednak z uwagi na stosunkowo niewielki przyrost złożoności pamięciowej i rozmiary pamięci w aktualnie używanych urządzeniach,

ten dodatkowy koszt nie stanowi problemu. W podrozdziale 3.2. pt. „Inicjalizacja struktur danych” szczegółowo omówiono implementację tego rozwiązania oraz udowodniono zmniejszenie złożoności obliczeniowej.

Ze względów implementacyjnych najwygodniejszy jest prostokątny kształt komórek. Implementacja złożonych kształtów naczyń, stanowi problem dla modelu SPH, dlatego zaimplementowano dodatkowy algorytm SAT, który służy do wykrywania kolizji cząstek wody z innymi obiektami. Dokładny opis zagadnienia zamieszczono w podrozdziale 2.5 „Wykrywanie kolizji”.

W celu zmniejszenia złożoności obliczeniowej przyjmuje się również, że promień odcięcia przyjmuje wartość większą od długości dłuższego boku komórki. Dzięki takiemu rozwiązaniu sąsiedztwem cząstki stają się cząstki znajdujące się w komórkach przyległych do komórki w której znajduje się cząstka.



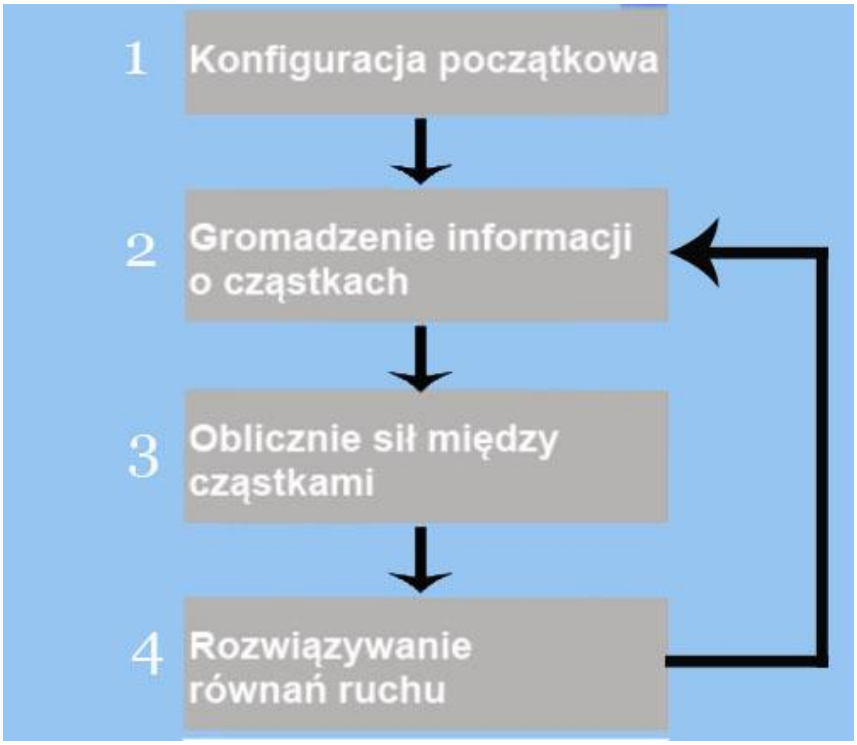
Rysunek 2: Sąsiedztwo cząstki dla promienia odcięcia większego od długości dłuższego boku komórki

Przyjmując, że promień odcięcia ma zawsze wartość większą od długości dłuższego boku komórki, można określić stałe sąsiedztwo komórki, w której znajduje się aktualnie przetwarzana cząstka. Na rys. 2 komórki będące stałym sąsiedztwem oznaczono kolorem niebieskim. Dzięki takiemu rozwiązaniu jesteśmy w stanie zmniejszyć złożoność obliczeniową z O(N*M) do O(N), gdzie N to liczba cząstek w układzie SPH, a M liczba komórek. W większych układach cząstek koszt ten jest pomijalny, ponieważ liczba komórek

jest stała i ustalana przed rozpoczęciem symulacji. Jednak dla małych układów cząstek, może stanowić znaczącym koszt. Przykładowo przy implementacji silnika SPH stworzonego na potrzeby tej pracy, podzielono pudło obliczeniowe na 1000 komórek. Co oznacza, że w każdym kroku iteracyjnym informacje o sąsiedztwie i-tej cząstki otrzymujemy w czasie zbliżonym do O(1), a nie O(1000).

2.3. Struktura algorytmu

W pierwszym etapie algorytmu generowana jest konfiguracja początkowa układu. Oznacza to utworzenie struktur danych, które magazynują i przetwarzają informacje na temat każdej cząstki. W tym etapie następuje także wstępna inicjacja tych struktur. Między innymi ustalone zostają parametry fizyczne każdej z cząstek np. jej położenie, masa, a także globalne parametry np. siła grawitacji, wartość stałej gazowej, gęstość cieczy itd. Po przeprowadzeniu początkowej konfiguracji rozpoczyna się wykonywanie pętli składających się z etapów przedstawionych na rys. 2.



Rysunek 2: Schemat poszczególnych etapów przebiegu symulacji

W drugim etapie algorytmu zbierane są informacje o aktualnym stanie cząstek. Informacje te są niezbędne do przeprowadzenia dalszych obliczeń. Przykładem może być obliczanie ciśnienia i gęstości na podstawie sąsiedztwa danej cząstki. Sposób wyznaczania grupy cząstek będącej sąsiedztwem danej cząstki został opisany w podrozdziale 2.2 pt. „Podział pudła obliczeniowego na komórki".

Głównym etapem całego algorytmu jest obliczanie sił działających na poszczególne cząstki, i obliczenie siły wypadkowej działającej na tę cząstkę. Z uwzględnieniem tej siły, w kolejnym kroku symulacji rozwiązywane są równania ruchu, co prowadzi do nowych

położeń oraz prędkości cząstek.

Jedną z istotnych, jeżeli nie najistotniejszych kwestii, jest dobór metody rozwiązywania równań ruchu. Ma to wpływ na dokładność i efektywność działania całego algorytmu. W niniejszej pracy zaimplementowano metody Eulera i Verleta. Metoda Eulera została stworzona w 1768 roku. Jest metodą pierwszego rzędu. Ponieważ występujące we wzorach symbole i wielkości powtarzają się wielokrotnie zostały one zebrane poniżej:

$r(t) - położenie\ w\ chwili\ t$

$v(t) - prędkość\ \ w\ chwili\ t$

$f(t) - siła\ wypadkowa\ w\ chwili\ t$

$a(t) - przyspieszenie\ w\ chwili\ t$

$\Delta t - krok\ czasowy$

$m - masa\ cząstki$

$\beta - stała\ tłumienia$

Wzory wykorzystywane w metodzie Eulera do znajdowania położeń i prędkości w następnej chwili czasu to:

$$r(t + \Delta t) = r(t) + r(t)'\Delta t = r(t) + v(t)\Delta t$$

$$v(t + \Delta t) = \ v(t) + v(t)'\Delta t = v(t) + a(t)\Delta t$$

$$a(t) = \frac{f(t)}{m}$$

W metodzie Verleta wzory te zastępowane są przez:

$$r(t + \Delta t) = 2r(t) - r(t - \Delta t) + a(t)\Delta t^2$$

$$v(t + \Delta t) = \frac{r(t + \Delta t) - r(t - \Delta t)}{\Delta t}$$

$$a(t) = \frac{f(t)}{m}$$

Wzory te można wspólnie zapisać jako:

$$Euler: r(t + \Delta t) = r(t) + r(t)'\beta\Delta t = r(t) + v(t)\beta\Delta t$$

$$Verlet: r(t + \Delta t) = r(t) + \beta\big(r(t) - r(t - \Delta t)\big) + \frac{f(t)}{m}\Delta t^2$$

Dodatkowym zabiegiem implementacyjnym jest wprowadzenie parametru β, który pozwala na dostosowanie ilości energii kinetycznej znajdującej się w układzie do specyficznej symulacji. Parametr ten pozwala na powolne wygaszenie energii kinetycznej całego układu, w efekcie czego cząstki będą poruszać się wolniej.

Po rozwiązaniu równań ruchu w i-tym kroku symulacyjnym, następuje aktualizacja położenia dla każdej cząstki w układzie. Następnie algorytm przechodzi do kolejnego kroku iteracyjnego. A ponieważ konfigurację początkową wykonuje się tylko raz, algorytm od razu wykonuje się od drugiego etapu, czyli gromadzenia informacji o cząstkach (zob. rys. 2).

2.4. Wybrane aspekty metody SPH

W poniższym podrozdziale zaprezentuję podstawowe wzory stosowane podczas implementacji metody SPH. Ponieważ występujące w nich symbole i wielkości powtarzają się wielokrotnie zostały one zebrane poniżej:

$r_i - wektor\ położenia\ pomiędzy\ i-tej\ cząsteczki$

$r_{ij} = r_i - r_j - wektor\ położenia\ pomiędzy\ cząstkami\ i\ oraz\ j$

$|r_{ij}| = |r_i - r_j| - odległość\ pomiędzy\ cząstkami\ i\ oraz\ j$

$A(r) - podstawowa\ funkcja\ aproksymacjna$

$A_j - dowolna\ wielkość\ skalarna\ dla\ j-tej\ cząstecki$

$W - funkcja\ jądra$

$W_{ij} = W(|r_{ij}|, h) - funkcja\ jądra\ dla\ cząsteczek\ i\ oraz\ j$

$\sum_j - suma\ po\ j-tych\ sąsiadach\ \ i-tej\ cząsteczki$

$\rho - gęstość$

$\Delta t - krok\ czasowy$

$m - masa\ cząstki$

$h - odległość\ rozmycia$

$\beta - stała\ tłumienia$

Jak wspomniano w podrozdziale 2.1 pt. „Bezsiatkowe metody cząstek", podstawową ideą algorytmu SPH jest rozmycie dowolnej wielkości fizycznej realizowane przy pomocy tzw. funkcji jądra. Wartość funkcji dąży do zera w nieskończoności, a całka z tej funkcji równa jedności. Punkty przestrzeni, w których następuje rozmycie nazywane są czastkami modelu SPH. Masa cząstki nie znajduje się w jednym punkcie w przestrzeni, lecz ulega rozmyciu zgodnie z ogólnym wzorem:

$$A(r_i) = \sum_j m_j \frac{A_j}{\rho_j} W(|r_i - r_j|, h),$$

gdzie w i-tym kroku symulacji obliczana jest aproksymowana wielkość fizyczna. Stanowi ona sumę po sąsiadach każdej cząstki, gdzie m; to masa, ρ, gęstość, A to aproksymowana wartość fizyczna, natomiast W to wartość funkcji rozmycia dla j-tego sąsiada. Powyższy wzór wyraża główną ideę metody SPH. W podstawowym ujęciu przyjmuje się, że suma w powyższym wzorze przebiega po wszystkich cząstkach w układzie, choć z uwagi na optymalizację przyjęto bardziej optymalne rozwiązanie bazujące na idei sąsiedztwa i-tej cząstki. Rozwiązanie bazuje na promieniu odcięcia, opisanym w podrozdziale 2.2 „Podział

pudła obliczeniowego", którego głównym założeniem jest wykluczenie jakiekolwiek interakcji w i-tym kroku symulacji z cząstkami układu znajdującymi się poza sąsiedztwem każdej cząstki. Takie podejście pozwala znacznie ograniczyć złożoność obliczeniową.

Podstawowym zadaniem podczas przeprowadzenia symulacji jest obliczenie położenia cząstki w kolejnym kroku iteracyjnym. W tym celu należy najpierw wyliczyć siłę wypadkową oddziałującą na każdą cząstkę, na podstawie której można rozwiązać równania ruchu i obliczyć kolejne położenie cząstki. Poniżej przedstawiono ogólny wzór na siłę wypadkową dla i-tej cząstki:

$$F_i^{resultant\ force} = F_i^{pressure} + F_i^{viscosity} + F_i^{interactive} + F_i^{gravity}$$

gdzie:

$F_i^{resultant\ force}$ – to siła wypadkowa działająca na i – tą cząstkę

$F_i^{pressure}$ – to siła ciśnienia działająca na i – tą cząstkę

$F_i^{viscosity}$ – to siła lepkości działająca na i – tą cząstkę

$F_i^{interactive}$ – to siła zewnętrzna np. wiatr działająca na i – tą cząstkę

$F_i^{gravity}$ – to siła grawitacji działająca na i – tą cząstkę

Korzystając z głównego wzoru aproksymującego możemy przekształcić powyższy wzór do postaci:

$$F_i^{resultant\ force} = \sum_j P_{ij} \nabla W_{ij} + \sum_j V_{ij} \nabla^2 W_{ij} + F_i^{interactive} + g m_i$$

gdzie:

P_{ij} – to wartość ciśnienia pomiędzy cząstkami i oraz j

V_{ij} – to wartość lepkości pomiędzy cząstkami i oraz j

g – wartość przyśpieszenia ziemskiego

W tym wzorze pozostaje do wyprowadzenia wartość ciśnienia i lepkości:

$$P_{ij} = -m_j \left(\frac{p_i}{{\rho_i}^2} + \frac{p_j}{{\rho_j}^2} \right),$$

gdzie:

p – to wartość ciśnienie hydrostatycznej cieczy

ρ – gęstość.

Oraz wzór na lepkość:

$$V_{ij} = \mu m_j \frac{v_j - v_i}{\rho_i \rho_j},$$

gdzie:

p − to wartość ciśnienie hydrostatycznej cieczy

ρ − gęstość.

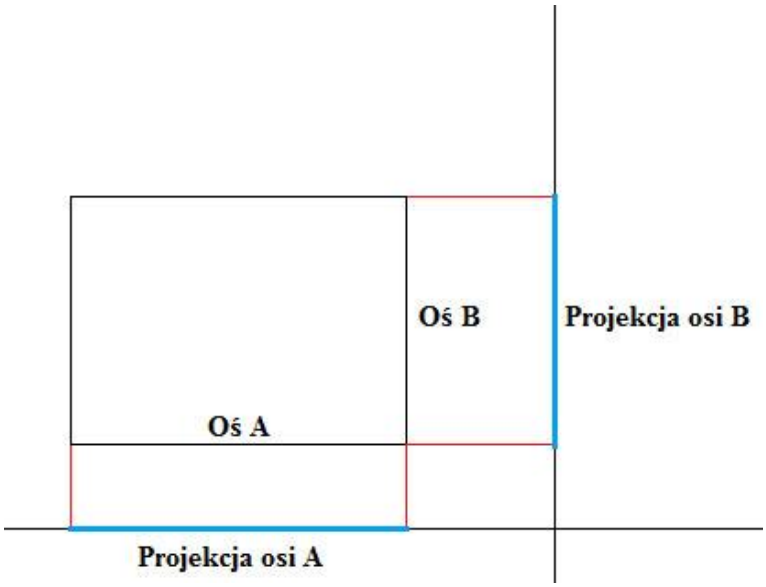
2.5. Wykrywanie kolizji

Algorytm SPH doskonale nadaje się do symulowania cieczy, ale nie nadaje się do wykrywania kolizji z obiektami innymi niż cząstki płynu. Możliwe jest podejście, w którym obiekty inne niż cząstki cieczy są reprezentowane w czasie symulacji przez cząstki (lub zbiory cząstek) o dużej masie i znikomej lepkości. Jednak na podstawie przeprowadzonych przeze mnie testów zrezygnowałem z takiego podejścia, ponieważ takie rozwiązanie wymuszało zwiększenie dokładności obliczeń w celu uzyskania oczekiwanego efektu wizualnego, co byłoby niewłaściwe z uwagi na ograniczone zasoby komputerowe. Ponadto stworzenie niejako sztucznych cząstek o ogromnej masie nieproporcjonalnej do masy innych cząstek w układzie, zakłócało niejednokrotnie poprawny przebieg symulowanego zjawiska, przy zbyt małej dokładności obliczeń.

Wykrywanie kolizji z obiektami innymi niż cząstki cieczy zostało zatem zrealizowane przy pomocy algorytmu SAT (ang. *Separating Axis Theorem*)[4] , który na podstawie kształtu i położenia dwóch obiektów wypukłych jest w stanie stwierdzić czy się one przecinają. Dzięki zastosowaniu takiego rozwiązania, wykrywanie kolizji w układzie można ograniczyć jedynie do cząstek cieczy i obiektów, z którymi te cząstki kolidują. Algorytm opiera się na stwierdzeniu, według którego jeżeli między dwoma obiektami wypukłymi można ustawić płaszczyznę, która nie będzie przecinać żadnego z obiektów, to obiekty te nie nachodzą na siebie. Aby dokładnie przeanalizować sposób działania algorytmu, należy zaznajomić się z pojęciem rzutowania. „Rzutowanie można przyrównać do cienia rzucanego przez obiekt. Jeśli skierujesz latarkę, której światło będzie padać prostopadle na pewną płaszczyznę to na płaszczyźnie tej pojawi się cień. Ten cień będzie właśnie projekcją oświetlonego obiektu”[5]. Przykładowa projekcja obiektu dwuwymiarowego jest widoczna na rys 3.

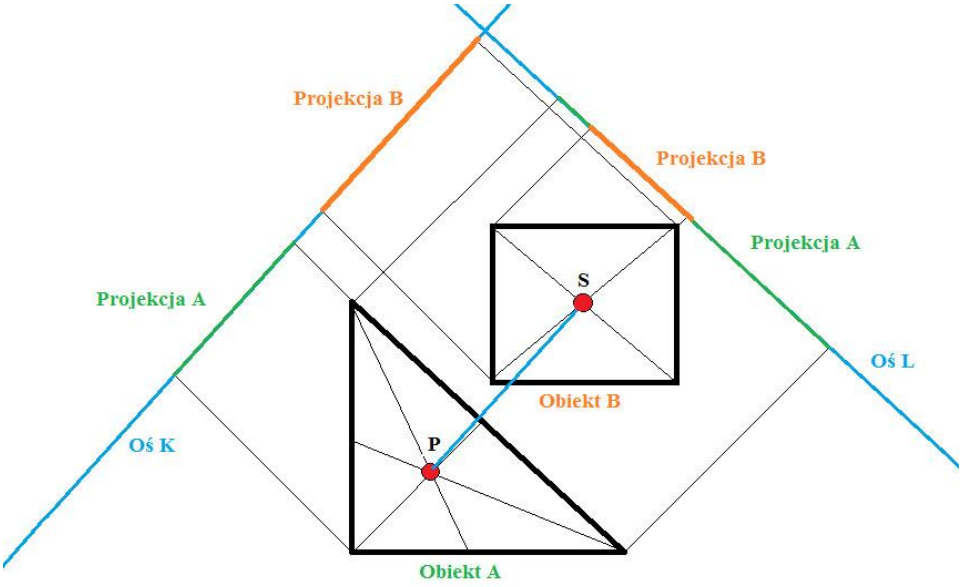
[4] S. Gottschalk, M. Lin, and D. Manocha. “OBB-Tree: A Hierarchical Structure for Rapid Interference Detection.” ACM SIGGRAPH. 1996.

[5] M. Zając, „Algorytm SAT”, http://zajacmarek.com/wp-content/uploads/2012/12/Algorytm-SAT.pdf 2012.



Rysunek 4: Projekcja prostokąta na linii

Na podstawie rys. 4 i 5 można wywnioskować stwierdzenie: Jeśli dla dwóch obiektów o wypukłym kształcie istnieje oś, na której projekcje tych dwóch obiektów nie nachodzą na siebie, to obiekty nie kolidują ze sobą.



Rysunek 5: Rzutowanie ścian na oś K i L

Na rys. 5 widać, że istnieje oś K, gdzie projekcja obiektu A i B nie nachodzi na siebie, stąd można wnioskować, że obiekty nie nachodzą na siebie. Oś K i L zostały wyznaczone na podstawie odcinka łączącego środki ciężkości P i S. Oś K jest równoległa do tego odcinka, natomiast oś L prostopadła. Poszczególne wierzchołki obiektów A i B są rzutowane pod kątem prostym na osie K i L tworząc w ten sposób rzuty krawędzi tych

obiektów.

Przeprowadzenie testu kolizji polegało na wyznaczeniu środków ciężkości, osi oraz projekcji dla obu obiektów. Ostatnim krokiem było sprawdzenie czy wyznaczone odcinki nachodzą na siebie. Jeśli tak, to obiekty kolidują ze sobą. W przeciwnym przypadku sprawdzano kolejną oś. Jeśli nie było więcej osi do sprawdzenia, wskazywało to, że obiekty nie nachodzą na siebie.

Głównymi zaletami opisanego algorytmu jest duża dokładność oraz stosunkowo wysoka wydajność. Mankamentem jest brak możliwości badania obiektów niewypukłych. Można je jednak podzielić na mniejsze obiekty wypukłe.

W pracy cząstki cieczy zamknięto w kwadratowe boksy. Dla każdego z nich przeprowadzana jest detekcja kolizji, przy pomocy SAT z innymi obiektami na mapie. Dodatkowo aby ograniczyć ilość badanych osi przyjęto, że cząstka jest reprezentowaną przez punkt ciężkości i długość boku boksu. Takie założenie pozwala na badanie co najwyżej dwóch osi, podczas testu kolizji. Dzięki tak zaprojektowanemu rozwiązaniu uzyskano wysoką wydajność i dużą dokładność wykrywania kolizji

Rozdział 3

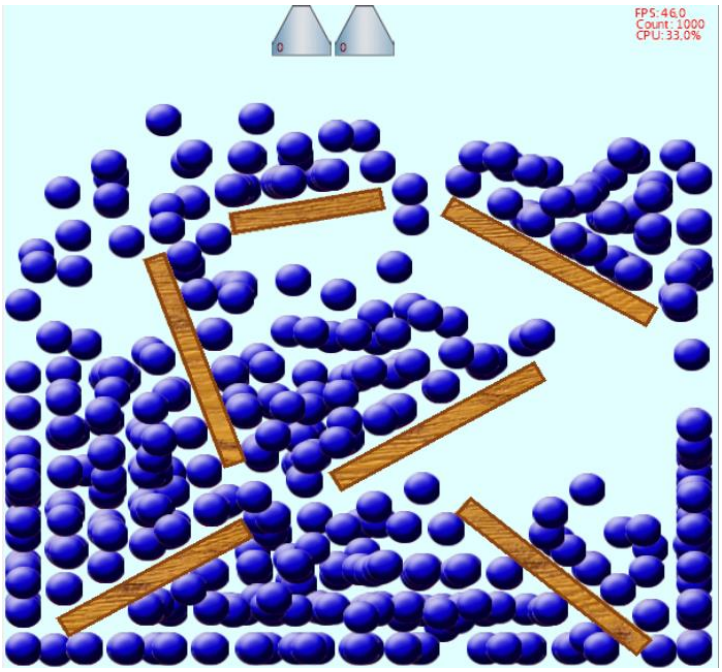
Implementacja silnika SPH.

W poprzednim rozdziale przedstawiono algorytm SPH. Głównym celem niniejszej pracy jest jednak jego wydajna implementacja. W niniejszym rozdziale umieszczono fragmenty kodu, w których używano nazw zmiennych, metod, klas itp. w języku angielskim, natomiast komentarze zostały zamieszczone w języku polskim. Użytym językiem programistycznym jest C# 4.0. Zagadnienia implementacyjne przedstawione w tym rozdziale dotyczą doboru struktur, instrukcji programu, i doboru wartości parametrów fizycznych symulacji.

3.1. Dobór kroku czasowego

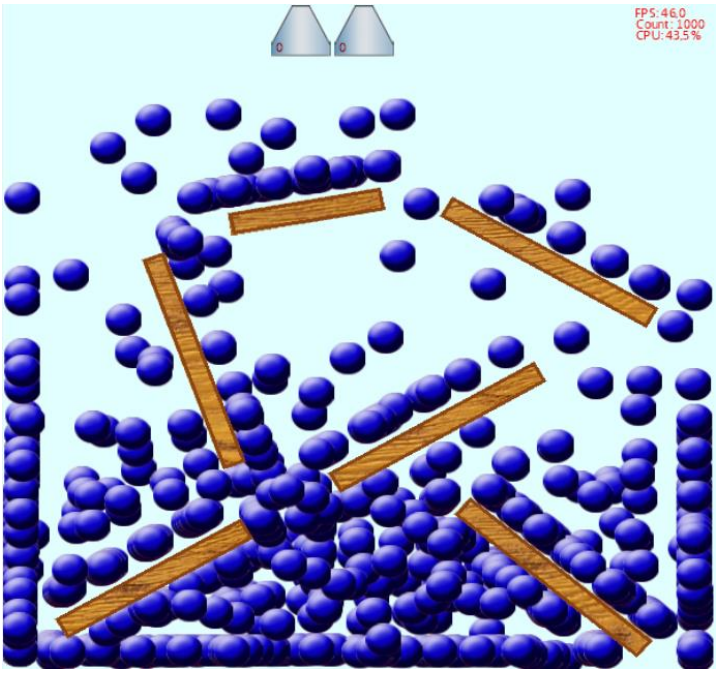
Kluczowym parametrem symulacji jest długość kroku czasowego. Im większa jest jego wartość, tym mniej czasu zajmuje symulacja, ale kosztem jej dokładności. Jednym z głównych założeń niniejszej pracy jest ustalenie kroku tak, aby możliwe było uruchomienie symulacji na urządzeniach o ograniczonych zasobach. Z uwagi na to, że obliczenia muszą być przeprowadzone w czasie rzeczywistym, poziom dokładności obliczeń staje się kwestią drugorzędną, przynajmniej do momentu wystąpienia artefaktów rzucających się w oczy. Przykładem tego rodzaju artefaktu może być przeskakiwanie obiektów z miejsca na miejsce lub „przenikanie” przez przeszkody.

W tej pracy przyjęto metodę wielokrotnych testów w celu wyznaczenia optymalnego kroku czasowego dla danego urządzenia. Poniżej zamieszczono kilka przykładów działania aplikacji dla różnego kroku czasowego. Testy przeprowadzono dla stałej liczby cząstek równej 1000.



Rysunek 6:Testowy przebieg dla bardzo małego kroku czasowego 0,0005s.

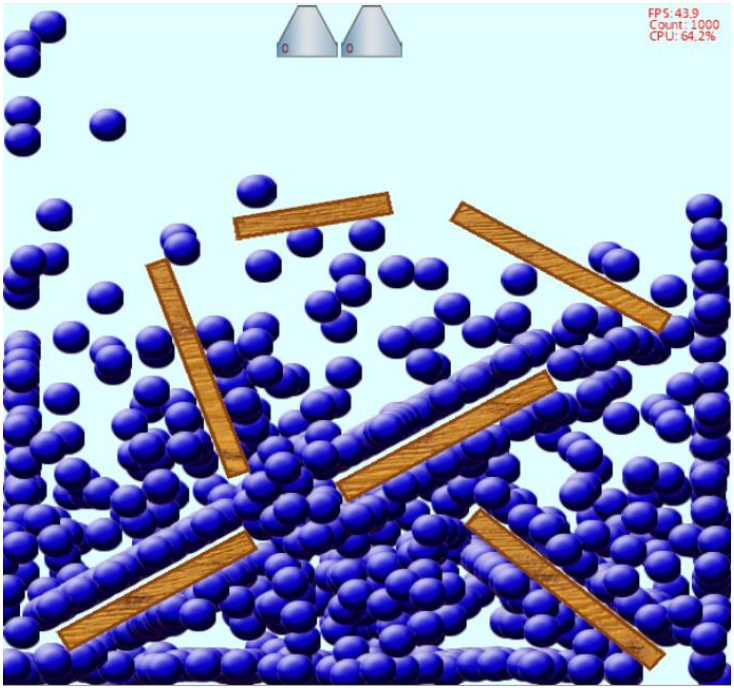
Na rys. 6 zaprezentowano wynik symulacji gdy zastosowano zbyt mały krok czasowy, w wyniku czego cząstki poruszają się bardzo wolno. Dodatkowo widać zbyt duży wpływ siły lepkości, w wyniku czego cząstki „przywierają” do siebie. Taki efekt może być pożądany podczas symulacji cieczy o dużej gęstości.



Rysunek 7:Testowy przebieg dla małego kroku czasowego 0,00015s.

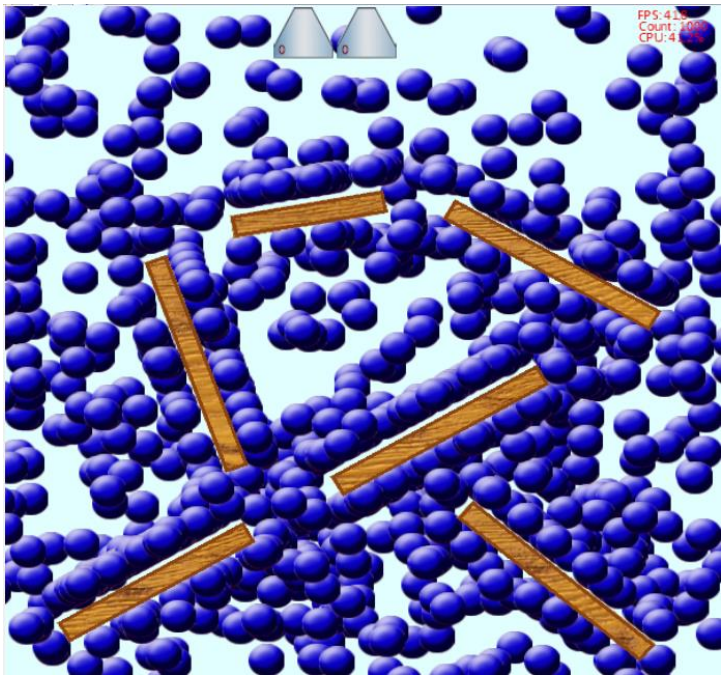
W wyniku symulacji zaprezentowanej na rys. 7 otrzymano ciecz, której cząstki

poruszały się wyraźnie zbyt wolno (jak na cząsteczki wody). Aczkolwiek oddziaływanie sił lepkości i grawitacji był na prawidłowym poziomie.



Rysunek 8:Testowy przebieg dla kroku czasowego 0,00025s.

Symulacja przeprowadzona dla kroku czasowego 0,00025s , zaprezentowana na rys. 8 moim zdaniem najlepiej oddaje zachowanie cząstek wody. Cząstki cieczy poruszały się z należytą prędkością, przy zachowaniu odpowiedniej lepkości. Widoczny jest taże wpływ sił grawitacji na układ cząstek.



Rysunek 9:Testowy przebieg dla zbyt dużego kroku czasowego 0,00040s.

Na rys. 9 zademonstrowano wpływ, jaki ma zbyt duży krok czasowy na symulację układu. W wyniku przeprowadzonej symulacji cząstki poruszały się bardzo szybko, niemal całkowicie nie było widać wpływu siły grawitacji na układ. Większe znaczenie miała siła lepkości.

Na powyższych rysunkach zademonstrowano jak zmiana kroku czasowego wpływa na przebieg symulacji. Wraz z jego wzrostem zwiększała się również prędkość poruszania cząstek. Jak widać na rys. 9, po przekroczeniu pewnej granicy kroku czasowego symulacja przestała być uporządkowana i coraz bardziej przestała przypominać symulację cieczy, a zaczęła przypominać symulację gazu. Znacząco wzrosła energia kinetyczna układu, w wyniku czego zmniejszył się wpływ grawitacji na cząstki. W efekcie cząstki zaczęły ,,odrywać się" od podłoża.

3.2. Inicjalizacja struktur danych

Pierwszym etapem symulacji jest inicjowanie niezbędnych struktur danych oraz stworzenie pudła obliczeniowego, dzięki któremu można w możliwie najdokładniejszy sposób określać sąsiedztwo i-tej cząstki. Metoda inicjująca widoczna jest na listingu 1.

Listing 1: Metoda inicjalizująca niezbędne struktury danych

```
1        public void Initialization(int numberRound=0)
2       {
3         _loader = new Loader();//Inicjalizacja map opisano w rozdziale
             4.1 Ogólne zasady gry
4          ParticlePositions = new List<SPHVector>();//Aktualne położenie cząstek
5          ParticlePositionsOld=new List<SPHVector>();//Poprzednie położenie cząstek
6          _grid = new Grid(ParticlePositions);//Inicjalizacja pudła obliczeniowego
7          Particles =new List<Particle>();//Inicjalizacja listy cząstek
8          Polygons = new List<Polygon>();//Inicjalizacja listy prostokątów w
            których umieszczone są cząstki opisano w podrozdziale 3.4
             Implementacja detekcji kolizji z innymi obiektami
9        }
```

W metodzie tej inicjalizowane są podstawowe struktury danych. W linii 3 zainicjalizowano mapy gry, której opis znajduje się w podrozdziale 4.2. pt. „Menu, zasady działania i edytor poziomów gry *Water your plants*". W liniach 4 – 7 zainicjalizowano struktury danych używane przez algorytm SPH, czyli m. in., listy położeń cząstek w aktualnie przetwarzanym i w poprzednim kroku symulacji. Następnie zainicjalizowano siatkę, która stanowi pudło obliczeniowe oraz listę zawierającą dane na temat aktualnego stanu parametrów fizycznych każdej cząstki. W linii 8 nastąpiło zainicjalizowanie struktur przechowujących informacje na temat obszaru kolizji danej cząstki. Każda z cząstek w celu detekcji kolizji z innymi obiektami niż cząstki cieczy została umieszczona w bryle typu OBB (ang. *oriented boxing box*). Detekcja kolizji między cząstkami cieczy, a innymi obiektami na mapie zrealizowana została z użyciem algorytmu SAT (podrozdział 3.4).

W kolejnym fragmencie pracy zostaną kolejno omówione struktury użyte do implementacji metody SPH. W celu zapamiętania położeń cząstek, została utworzona klasa `SPHVector`, która przechowuje współrzędne wektora i wykonuje obliczenia na wektorze. Tego rodzaju klasy znajdują się w każdej bibliotece graficznej np. `OpenTK.Math.Vector2`, lub `Microsoft.Xna.Framework.Vector2`. W niniejszej pracy, z uwagi na przenaszalność, zdecydowano się na napisanie własnej klasy zajmującej się przetwarzaniem wektorów. Takie podejście dotyczy także wielu innych struktur użytych w programie, dzięki czemu ułatwione jest przeniesienie kodu do innej technologii. Na listingu 2 widoczny jest fragment kodu klasy `SPHVector`.

Listing 2:Klasa SPHVector

```
1    public class SPHVector
2    {
3        public float X { get; set; }
4        public float Y { get; set; }
         //Lista operatorów umożliwiająca wygodne operacje na wektorach
5        public static SPHVector operator -(SPHVector V1, SPHVector V2)
6        public static SPHVector operator -(SPHVector V1, float s)
7        public static SPHVector operator -( float s,SPHVector V1)
8        public static SPHVector operator +(SPHVector V1, float s)
9        public static SPHVector operator +(float s,SPHVector V1)
10        public static SPHVector operator +(SPHVector V1, SPHVector V2)
11       public static SPHVector operator *(SPHVector V1, float s)
12       public static SPHVector operator *(float s, SPHVector V1)
13       public static SPHVector operator *(SPHVector V1, SPHVector V2)
14       public static SPHVector operator /(SPHVector V1, float s)
         //Lista metod umożliwiająca wygodne przeprowadzanie obliczań na wektorach
         //Obliczanie odległości v1 od v2
15       public static float Distance(SPHVector V1, SPHVector V2)
         //Obliczanie długości
16       public float Distance()
         //Wyznaczanie wektora prostopadłego
17       public SPHVector Perpendicular()
         //Wyznaczanie odległości kwadratowej
18       public float LengthSquared()
         //Wyznaczanie odległości kwadratowej v1 i v2
19       internal static float Dot(SPHVector V1, SPHVector V2)
         //Normalizacja
20       internal SPHVector Normalize()
21   }
```

Kolejną klasą umożliwiającą przechowywanie danych jest `Grid`. Jej głównym zadaniem jest stworzenie listy cząstek w komórkach, na listingu 3 widoczne są interesujące elementy klasy `Grid`.

Listing 3: Klasa Grid implementująca pudło obliczeniowe

```
1    public class Grid
2    {
3        private List<int>[,] _grid; //Tablica list identyfikatorów cząstki
4        public int Width { get; set; } //Liczba kolumn
5        public int Height { get; set; } //Liczba wierszy

6        public Grid(List<SPHVector> particlesPosition)
7        {
8            this.Width = (int)(Constants.GAMEWIDTH/ Constants.CELLSPACE);
             //Liczba kolumn = stała szerokość okna / stałą szerokość jednej komórki
9            this.Height = (int)(Constants.GAMEHEIGHT /Constants.CELLSPACE);
             //Liczba wierszy = stała szerokość okna / stałą szerokość jednej
             komórki
10           this.Refresh(particlesPosition); //Przypisanie cząstek do komórek
11       }
12       public void Refresh(List<SPHVector> particlesPosition)
13       {
14           _grid = new List<int>[this.Width, this.Height]; //Inicjalizacja
15           for (int i = 0; i < particlesPosition.Count; i++)
16           {
17               int gridIndexX = GetGridIndexX(particlesPosition[i]); //index X
18               int gridIndexY = GetGridIndexY(particlesPosition[i]); //index Y
19               if (_grid[gridIndexX, gridIndexY] == null)
```

```
20                {
21                    _grid[gridIndexX, gridIndexY] = new List<int>();
22                }
23                _grid[gridIndexX, gridIndexY].Add(i); //Dodaj identyfikator
cząstki
24            }
25        }
26        private int GetGridIndexX(SPHVector particlePosition)
27        {
28            int gridIndexX = (int)(particlePosition.X / Constants.CELLSPACE);
29            if(gridIndexX < 0)
30            {
31                gridIndexX = 0;
32            }
33            if (gridIndexX >= this.Width)
34            {
35                gridIndexX = this.Width - 1;
36            }
37            return gridIndexX;
38        }
39        private int GetGridIndexY(SPHVector particlePosition)
40        {
41            int gridIndexY = (int)(particlePosition.Y / Constants.CELLSPACE);
42            if (gridIndexY < 0)
43            {
44                gridIndexY = 0;
45            }
46            if (gridIndexY >= this.Height)
47            {
48                gridIndexY = this.Height - 1;
49            }
50            return gridIndexY;
51        }
52        public IEnumerable<int> GetNeighbourIndex(SPHVector particlePosition)
53        {
54            int x0 = GetGridIndexX(particlePosition);
55            int y0 = GetGridIndexY(particlePosition);
56            for (int xOff = -1; xOff < 2; xOff++)
57            {
58                for (int yOff = -1; yOff < 2; yOff++)
59                {
60                    int x = x0 + xOff;
61                    int y = y0+ yOff;
62                    if (x > -1 && x < this.Width && y > -1 && y < this.Height)
63                    {
64                        List<int> idxList = _grid[x, y];
65                        if (idxList != null)
66                        {
67                            foreach (var idx in idxList)
68                            {
69                                yield return idx;
70                            }
71                        }
72                    }
73                }
74            }
75        }
76    }
```

Na listingu 3 zaprezentowano kod jednej z najistotniejszych struktur programu.

Pozwala ona na efektywne przetwarzanie cząstek, dlatego każda z metod zostanie poddana analizie pod względem funkcjonalnym i złożonościowym. W liniach 3 – 11 widoczne są pola klasy i konstruktor. W linii 10 przy pomocy metody `Refresh` (linie 12-25) zapisywane są identyfikatory cząstek do tablicy dwuwymiarowej `_grid`. Dla uproszczenia przyjęto, że dynamicznie przypisany indeks tablicy (bądź listy elementów) stanowi identyfikator. Tego rodzaju rozwiązanie zastosowano także w innych miejscach kodu, co pozwoliło zmniejszyć złożoność algorytmu. Oczywistym faktem jest, że taka mała poprawka optymalizacyjna nie może natychmiast znacząco wpłynąć na poprawę jakości działania całego silnika SPH, ale przykładów tego rodzaju działań jest więcej.

Złożoność algorytmu niejako z definicji metody SPH zbiega do O(N), aczkolwiek w momencie gdy zasoby komputerowe są dość ubogie, a wymagana jest praca w czasie rzeczywistym, należy (niczym na mikrokontrolerach) przeliczać każdy niepotrzebny przebieg. W pierwszej wersji algorytmu udało się symulować do 700 cząstek (wraz ze wszystkimi metodami renderującymi, sprawdzającymi kolizje itp.). Aktualnie algorytm jest w stanie utrzymać płynne działanie aplikacji nawet dla 1500 cząstek na tej samej maszynie testowej (laptopa ProBook 4730s):

Procesor: Intel®Core™ i5-2410M CPU @ 2.30GHz ,

RAM: 4,00 GB

Typ systemu: 64 - bitowy system operacyjny

DYSK: Blaze Patriot SSD 240 GB

Dynamicznie przypisany indeks tablicy stanowiący identyfikator zastosowano w linii 23. Kluczowe dla implementacji pudła obliczeniowego są metody `GetGridIndexX` i `GetGridIndexY`, które pobierają indeksy cząstki znajdującej się w komórce (złożoność O(1)). Na podstawie jej indeksów możliwe jest, pobieranie jej sąsiedztwa, także w złożoności zbiegającej do O(1), co zostało zrealizowane w metodzie `GetNeighbourIndex` (linie 52 -75). Idea tego rozwiązania została przedstawiona w podrozdziale 2.2 pt. „Podział pudła obliczeniowego”. Ostatnią omawianą strukturą danych jest klasa `Particle`, przy pomocy której przechowywane są parametry fizyczne cząstki modelu SPH (listing 4).

Listing 3: Klasa Particle przechowująca i przetwarzająca cząstekę modelu SPH.

```
1    public class Particle
2    {
3        public float H = Constants.CELLSPACE; //Odległość rozmycia
4        public int Index = -1; //Index na liście cząstek
5        public float Mass=1.0f; //Masa cząstki
6        public SPHVector Velocity= new SPHVector(); //Prędkość
```

```
7         public SPHVector Force = new SPHVector(10,0); //Siła wypadkowa
8         public float Density = 1; //Gęstość
9         public float Pressure = 0; //Cisnienie
10        public SPHVector SumAcceleration = new SPHVector(); //Wypadkowe
przyspieszenie
11        public int TimeLife = Constants.TIMELIFE; //Czas życia cząstki
wykorzystywany w procesie rederowania
12         public Polygon Polygon; //Prostokąt w którym umieszczona jest cząsteka w
celu wykrywania kolizji

14         internal static IEnumerable<Particle> CreateParticles(int count,int
                                                                  startIndex)
15        {
16            float sizeBox = Constants.CELLSPACE * 0.25f;
17            for (int i = 0+startIndex; i < count+startIndex; i++)
18            {
19                yield return new Particle(){Index = i,Polygon=new Point(new
                  SPHVector (sizeBox, sizeBox),new SPHVector())};
              }
20        }
21    }
```

W liniach 1-12 następuje wstępna inicjalizacja prędkości, ciśnienia itp. W metodzie `CreateParticles`, zainicjalizowano obiekt typu `Polygon`, który służy do wykrywania kolizji między obiektami innymi niż cząstki cieczy.

3.3. Przebieg kroku symulacji

Poniższy rozdział przybliża działanie głównej pętli algorytmu SPH. W każdym kroku obliczane są nowe pozycje cząstek. Główną metodę `Update` wykonywaną w każdym kroku czasowym zaprezentowano na listingu 4.

Listing 4:Metoda aktualizująca położenie cząstek w układzie SPH

```
1        public void Update()
2        {
3            _grid.Refresh(ParticlePositions); // Odświeżanie pozycji w pudle
4            obliczeniowym
5            CarculateDensityPressure();// Obliczanie ciśnienia i gęstości
6            CarculateResultantForce(Constants.GRAVITY * Constants.PARTICLEMASS); //
7            Obliczanie siły wypadkowej
8            RefreshParticlePositions();// Odświeżanie pozycji cząstek
9            RefreshPolygonPositions();// Odświeżanie pozycji prostokątów do których
10           przypisane są cząstki
11           CollisionDetection();// Detekcja kolizji cząstek z innymi obiektami
12       }
```

Z metody `Update` wywoływane są metody przeprowadzające najważniejsze obliczenia algorytmu SPH. W tym rozdziale poszczególne fragmenty kodu zostaną poddane analizie. Metoda `Refresh(ParticlePositions)` wywoływana na rzecz obiektu `_grid` została omówiona w podrozdziale 3.2 pt. „Inicjalizacja struktur danych". Odpowiada ona za odświeżenie pozycji cząstki w pudle obliczeniowym. Na listingu 6 następuje obliczenie ciśnienia i gęstości dla każdej cząstki w układzie.

Listing 5:Metoda obliczająca ciśnienie i gęstość cząstek

```
1        private void CarculateDensityPressure()
2        {
3            foreach (var particle in Particles)
4            {
5                particle.Velocity = Constants.VELOCITY;
6                particle.Density = Constants.DENSITY;
7                List<Particle> particlesNeighborhoods = FindNeighborhood(particle);
8                foreach (var particlesNeighborhood in particlesNeighborhoods)
9                {
10                    SPHVector Rij = ParticlePositions[particle.Index] -
11                    ParticlePositions[particlesNeighborhood.Index];
12                    particle.Density += particlesNeighborhood.Mass * W(Rij,
13                    particle.H);
14                }
15                particle.Pressure = Constants.K*(particle.Density -
16                Constants.DENSITY);
17            }
18       }
```

W tej metodzie obliczenia przeprowadzane są dla wszystkich cząstek w układzie zgodnie z pętlą zamieszczoną w linii nr 3. Kolejnym krokiem jest korekta gęstości i prędkości polegająca na nadaniu stałej wartości. Efektem tego jest utrzymanie energii kinetycznej układu na pewnym stałym poziomie, co pozwala na uzyskanie lepszego efektu wizualnego „rozbryzgu" cieczy. Nierzadko w grach opartych na cieczy taki efekt wizualny

jest pożądany, choć niekoniecznie jest on zgodny z prawami fizyki. W linii 7 pobierane jest sąsiedztwo *i*-tej cząstki. Warto zwrócić uwagę na to, że złożoność tej operacji jest rzędu O(1), co zostało opisane w poprzednim podrozdziale. W liniach 8-14 następuje obliczenie ciśnienia i gęstości zgodne ze wzorem omówionym w podrozdziale 2.4 pt. „Wybrane aspekty metody SPH", gdzie najpierw jest obliczana odległość pomiędzy cząstkami, a następnie wynikająca z niej gęstość. Na podstawie gęstości cząstki wyliczane jest ciśnienie (linie 15-16), realizując w ten sposób wzór na gęstość z podrozdziału 2.4. Kolejną czynnością jest obliczenie siły wypadkowej działającej na każdą cząstkę. Za realizację tego zadania odpowiada metoda `CalculateResultantForce(Constants.GRAVITY * Constants.PARTICLEMASS)`, która widoczna jest na listingu 6.

Listing 6: Metoda odpowiedzialna za obliczanie siły wypadkowej działającej na i-tą cząstekę

```
1       private void CalculateResultantForce(SPHVector globalForce)
2       {
3           foreach (var particle in Particles)
4           {
5               float scalar=0;
6               SPHVector viscosity = new SPHVector();
7               SPHVector force = new SPHVector();
8               SPHVector Rij = new SPHVector();
9
10               particle.Force += globalForce;
11               foreach (var particlesNeighborhood in FindNeighborhood(particle))
12               {
13                   if (particlesNeighborhood.Index < particle.Index)
14                   {
15                       Rij = ParticlePositions[particle.Index] –
16                            ParticlePositions[particlesNeighborhood.Index];
17                       if (Rij.X != 0 || Rij.Y != 0)
18                       {
19                           scalar = particle.Mass*(particle.Pressure +
20                              particlesNeighborhood.Pressure)/
22                         (2.0f*particlesNeighborhood.Density);
23                           force = scalar*GradientW(Rij);
24                           particle.Force -= force;
25                           particlesNeighborhood.Force += force;
26
27                           scalar = particlesNeighborhood.Mass*
                              Constants.VISCOSITYSCALAR*1/particlesNeighborhood.Density*
                              LaplacianW(Rij, particle.H);
28                       if (particlesNeighborhood.Viscosity!= particle.Viscosity)
29                   viscosity= particlesNeighborhood.Viscosity- particle. Viscosity;
30                           else
31                           {
32                               velocity = Constants.VISCOSITY;
33                           }
34                           force = velocity*scalar;
35                           particle.Force += new SPHVector((float) force.X,
36                             (float) force.Y);
37                           particlesNeighborhood.Force -= force;
38                       }
39                   }
40               }
41           }
51       }
```

Powyższa metoda przyjmuje jako parametr wartość siły zewnętrznej działającej na układ cząstek. W naszym przypadku jedyną siłą zewnętrzną jest siła grawitacji, równa

iloczynowi cząstek i przyśpieszenia ziemskiego. W linii 10 aktualna siła wypadkowa *i*-tej cząstki jest powiększana o siłę zewnętrzną działającą na układ. Następnie wybierane jest sąsiedztwo *i*-tej cząstki, które pozwala przeprowadzić obliczenia zgodnie z wzorami zawartymi w podrozdziale 2.4. Głównymi składowymi siły wypadkowej działającej na cząstkę układu SPH jest siła ciśnienia wynikająca z (linie 16 – 25) oraz siła lepkości (linie 27-37). W powyżej metodzie zmienne przyjmują następujące znaczenie: `force` jest aktualną wartością siły np. siła ciśnienia i siła lepkości, `scalar`, jest wartością tymczasową używaną do obliczenia siły wypadkowej. Pozostałe zmienne i wynikające z nich wartości są odwzorowaniem zmiennych ze wzorów zamieszczonych w podrozdziale 2.4, gdzie:

`Viscosity` – wektorowa siła lepkości

`particle.Force` – siła wypadkowa działająca na *i*-tą cząstkę

`Rij` – wektor położenia pomiędzy *i*-tą cząstką, a jej *j*-tym sąsiadem

`particle.Pressure` – ciśnienie *i*-tej cząstki

`particlesNeighborhood.Density` – gęstość *j*-tego sąsiada *i*-tej cząstki

`GradientW(Rij)` – gradient funkcji jądra dla wektora położenia

`particlesNeighborhood.Mass` – masa *j*-tego sąsiada *i*-tej cząstki

W każdym kroku symulacji obliczane są siły działające na każdą cząstkę. Dodatkowo w celach optymalizacyjnych wyliczane są siły działające na *j*-tego sąsiada. Realizując trzecią zasadę dynamiki Newtona, z której wiadomo że *„Jeżeli ciało A działa na ciało B siłą* $\overrightarrow{F_{AB}}$, *to ciało B działa na ciało A siłą* $\overrightarrow{F_{BA}}$, *o takim samym kierunku i wartości jak* $\overrightarrow{F_{AB}}$, *ale przeciwnym zwrocie”*. Dlatego w linii 13 sprawdzane jest czy index *i*-tej cząstki jest większy od indeksu *j*-tego sąsiada. W ten sposób eliminowane są zbędne obliczenia (redukcja o połowę). Sumowywania sił *j*-tych sąsiadów *i*-tej cząstki przeprowadzone są odpowiednio w liniach 35 i 37. Kolejną wywoływaną metodą jest `RefreshParticlePositions`, która jest zwieńczeniem wszystkich obliczeń przeprowadzonych w kroku symulacji (listing 8).

Listing 7: Metoda aktualizująca położenie cząstek w układzie na podstawie wcześniej wyliczonych wartości

```
1     private void RefreshParticlePositions()
2     {
3          Verlet verlet = new Verlet();
4          foreach (var particle in Particles)
5          {
6              particle.SumAcceleration = particle.Force / particle.Mass;
7              SPHVector position=ParticlePositions[particle.Index];
8              SPHVector positionOld=ParticlePositionsOld[particle.Index];
9              CheckBordersPosition(ref position);
```

```
10              verlet.Solve(ref position, ref positionOld, particle,
                Constants.TIMESTEP);
11              ParticlePositions[particle.Index] = position;
12              ParticlePositionsOld[particle.Index] = positionOld;
13          }
```

Głównym zadaniem tej metody jest rozwiązanie równań ruchu, a następnie aktualizacja położenia cząstki. Dodatkowym zadaniem tej metody jest sprawdzenie czy cząstka nie znalazła się poza granicami układu cząstek i ewentualne ustawienie jej w odpowiedniej pozycji. Odpowiedzialna jest za to metoda `CheckBordersPosition(ref position)`. W linii 3 następuje utworzenie instancji klasy implementującej metodę, przy pomocy której nastąpi rozwiązanie układu równań ruchu. W powyższym przykładzie użyto metody Verleta, która została opisana w podrozdziale 2.3. W liniach 7 – 9 odczytane są położenia cząstek w bieżącym i poprzednim kroku, które muszą zostać przekazane do metody rozwiązującej równania ruchu. Metoda ta widoczna jest na listingu 9.

Listing 8: Metoda opowiadająca za rozwinięcie równań ruchu metodą Verleta

```
1       public void Solve(ref SPHVector position,ref SPHVector positionOld,
2        Particle particle, float timeStep)
3       {
4           SPHVector positionTemp = position;
5           position = position + (1.0f - Constants.DAMPING) * (position –
6           positionOld) + timeStep * timeStep * particle.SumAcceleration;
7           positionOld = positionTemp;
8           particle.Velocity = (position - positionOld) / timeStep;
```

Kod na listingu nr 7 jest odzwierciedleniem wzorów opisanych w podrozdziale 2.3. W liniach 11 – 12 następuje ostateczna aktualizacja położeń co kończy przebieg jednego kroku czasowego algorytmu SPH. Kolejne działania to odświeżenie pozycji komórek do których przypisane są cząstki i detekcja kolizji przy pomocy algorytmu SAT, które zostaną przedstawione w kolejnym podrozdziale.

3.4. Detekcja kolizji między cząstkami cieczy, a innymi obiektami na mapie

Za detekcję kolizji między cząstkami, a innymi obiektami na mapie odpowiada algorytm SAT, który gwarantuje dużą wydajność i dokładność obliczeń. Dokładny opis działania algorytmu przedstawiono w podrozdziale 2.5. pt. „Wykrywanie kolizji" . Na listingu 10 widoczna jest metoda odpowiadającą za detekcję kolizji.

Listing 10; Detekcja kolizji pomiędzy i-tą cząstką, a innym obiektem znajdującym się na mapie

```
1        private void PolygonDetetcion(Polygon polygon,Particle particle)
2        {
3            SPHVector penetrationNormal = new SPHVector();
4            float penetrationLength = float.MaxValue;
5            bool hasCollided = false;
6            if (polygon != particle.Polygon && polygon.Visable)
7            {
8                hasCollided = true;
9                foreach (SPHVector axis in polygon.Axes)
10               {
11                   float minP1, maxP1, minP2, maxP2;
12                   if (!polygon.TestSeparatingAxis(polygon.Position,
13                    particle.Polygon.Position, axis,
14                   particle.Polygon, out minP1, out maxP1, out minP2, out maxP2))
15                   {
16                       hasCollided = false;
17                       break;
18                   }
19                   float difference=Math.Min(maxP2,maxP1)-Math.Max(minP2, minP1);
20                   if (difference < penetrationLength)
21                   {
22                       penetrationLength = difference;
23                       penetrationNormal = axis;
24                   }
25               }
26               if (hasCollided)
27               {
28                   if (polygon.Index.Contains("WaterDestination"))
29                   {
30                       particle.TimeLife = 0;
31                       polygon.AbsorbedWater++;
32                       if (polygon.AbsorbedWater == Constans.MaxAbsorbedWater)
33                       polygon.Visable = false;
34                   }
35                   SPHVector penetration = penetrationLength * penetrationNormal;
36                   ParticlePositions[particle.Index] -= penetration;
37               }
38           }
39       }
```

Metoda `PolygonDetetcion` przyjmuje jako argumenty obiekt `polygon`, który reprezentuje prostokąt obiektu nie będący cząstką cieczy oraz `particle`, czyli cząstkę modelu SPH. Zmienna `penetrationNormal` to oś na którą zostaną zrzutowane ściany tych obiektów, a zmienna `penetrationLength` reprezentuje odległość między

środkami ciężkości tych obiektów. W linii 6 sprawdzamy czy oba utworzone wielokąty są różne od siebie i czy obiekt ma być widoczny na mapie. Następnie w liniach od 8 do 26 przeprowadzany jest test separacji rzutów tych obiektów. Jeśli którakolwiek z projekcji pierwszego obiektu przecina się z którąkolwiek projekcją drugiego obiektu, to wiadomo że obiekty uległy kolizji. Tego rodzaju warunek zapisano w postaci kodu w linii 12. Jeśli doszło do kolizji, zapisujemy na jakiej osi pokrywały się i jaka była odległość miedzy środkami przyrównywanych obiektów. Początkowo sprawdzamy czy kolizja nie nastąpiła z obiektem, który pochłania cząstki cieczy (linia 28). Jeśli tak, to usuwamy cząstkę z mapy i powiększamy licznik pochłoniętych cząstek. Następnie wyliczamy wektor przesunięcia pierwszego obiektu względem drugiego tak, by obiekty nieco oddaliły się od siebie. W linii 36 modyfikowana jest pozycja cząstki cieczy tak, by nie wnikała w inny obiekt.

Pozostała do prezentacji najważniejsza metoda algorytmu SAT, czyli test separacji obiektów (listingu 11).

Listing 9: Test separacji

```
1        public bool TestSeparatingAxis(SPHVector axis, Polygon polygon, out float
2                   minP1, out float maxP1, out float minP2, out float maxP2)
3        {
4            this.Project(axis, out minP1, out maxP1);
5            polygon.ProjectPoint(axis, out minP2, out maxP2);
6            if ((minP1 >= minP2 && minP1 < maxP2) ||
7           (maxP1 >= minP2 && minP1 < maxP2))
8            {
9                return true;
10            }
11            return false;
12        }
```

Metoda przyjmuje parametry potrzebne do obliczenia rzutu krawędzi badanego obiektu na oś. Idea SAT została opisana w podrozdziale 2.5. W linii 4 i 5 obliczane są rzuty dla pierwszego i drugiego obiektu. Jeśli odcinki nie nakładają się na siebie nie doszło do kolizji względem analizowanej osi. W pozostałych przypadkach wystarczy, gdy wykryto kolizję na jednej osi, co oznacza, że doszło do kolizji między obiektami. Powyżej opisany warunek znajduje się w linii 6 i 7. Zmienne `minP1` i `maxP1` oznaczają odpowiednio punkt początkowy i końcowy pierwszego odcinka, analogicznie zmienne `minP2` i `maxP2` drugiego.

Rozdział 4

Opis aplikacji „Water your plants”

W tym rozdziale umieszczono przykładową aplikację opartą na silniku SPH. Aplikację stworzono w formie gry zręcznościowej. Do wizualizacji obiektów użyto technologii XNA 4.0. Opis instalacji, zasad gry i wykorzystywanych narzędzi będzie treścią poniższego rozdziału.

4.1. Instalacja komponentów potrzebnych do uruchomienia aplikacji

Całość kodu aplikacji została napisana w języku C# w wersji 4.0. Jednak z uwagi na przenaszalność kodu starano się całkowicie unikać wszelkich dodatkowych bibliotek, które ściśle wiążą się z platformą Windows. Aplikacja została podzielona na trzy podstawowe elementy: logika biznesowa, warstwa bazodanowa i warstwa prezentacji, która siłą rzeczy musi być kompatybilna ze środowiskiem, na którym działa aplikacja. W niniejszej pracy do stworzenia warstwy prezentacji zastosowano XNA Framework. Dodatkowo na potrzeby demonstracji, stworzono aplikacje na system Android, zintegrowaną z biblioteką OpenGL, w której zastosowano bibliotekę SPH stworzoną w ramach tej pracy. Aplikacja demonstracyjna, w której nie został zaimplementowany interfejs, została utworzona przy pomocy narzędzi udostępnionych przez firmę Xamarin. Projekt Mono, o którym mowa, umożliwia uruchamianie aplikacji stworzonych dla platformy .NET Framework, na różnych platformach, a także na większości dystrybucji systemu Linux.

Zgodnie z przyjętą filozofią, kod aplikacji został podzielony tak, aby oddzielić biblioteki odpowiedzialne za przetwarzanie danych od warstwy prezentacyjnej i warstwy bazodanowej. W wypadku warstwy logiki biznesowej silnik SPH zajmuje się wszystkim, co związane z przygotowaniem modelu cząstek do wyświetlenia. Natomiast z punktu widzenia warstwy prezentującej, silnik SPH jest czarną skrzynką, która co zadany krok czasowy udostępnia kolejne położenia cząstek modelu SPH. Jedynym zadaniem warstwy prezentacyjnej jest wyświetlenie wyniku na ekranie. Zgodnie z tą filozofią dla silnika SPH nie jest istotne przy pomocy jakiej biblioteki graficznej dane zostaną wyświetlone. Ostatnim elementem jest warstwa bazodanowa, która w owej implementacji występuje częściowo, ponieważ nie istnieje baza danych, do której byłyby zapisane dane. Wielu programistów uważa, że jeśli aplikacja nie magazynuje danych w bazie danych, (jak w wypadku wykonanej w niniejszej pracy gry zręcznościowej), to warstwa bazodanowa nie powinna być ujęta w architekturze aplikacji. Moim zdaniem każda aplikacja powinna posiadać bibliotekę

tworzącą ewentualny model użytych encji w aplikacji. Takie podejście stwarza możliwość rozwoju aplikacji, pozwala na dodanie bazy danych do aplikacji choćby przy pomocy przyjaznych frameworków, takich jak Entity Framework. Poza tym takie rozwiązanie pozwala zachować wszystkie niezbędne kontenery danych w jednym miejscu.

Dodatkową technologią wielokrotnie używaną w implementacji opisywanej pracy jest Language INtegrated Query (LINQ), który pozwala na traktowanie obiektów, tak jakby były one elementami bazy danych SQL. Technologia LINQ umożliwia przygotowanie zapytań na obiektach, natomiast składnia języka LINQ jest prosta i przypomina SQL. Pozwala to na zaoszczędzenie czasu przeznaczonego na tworzenia kodu i zwiększenie jego optymalności.

Wróćmy jednak do XNA Framework biblioteki odpowiedzialnej za wyświetlanie i animację interfejsu użytkownika. XNA jest zbiorem narzędzi firmy Microsoft pozwalającym na tworzenie gier przeznaczonych dla systemu Windows, konsoli Xbox 360, jak również telefonów z systemem operacyjnym Windows Phone 7. Zawiera ona bogaty zestaw bibliotek klas, które są przeznaczone specjalnie do tworzenia gier komputerowych. Biblioteki te są skonstruowane w taki sposób, aby umożliwić łatwe przeniesienie programu na inną platformę sprzętowo-systemową, bez dokonywania wielu poprawek w kodzie. Tworzone aplikacje mogą działać na systemach Windows XP, Windows Vista, Windows 7, Windows Phone 7 i Xbox 360. Programy mogą być pisane właściwie w każdym języku programowania zgodnym z .NET, jednak używa się jedynie C# oraz Visual Basic. Aplikacja „Water your plants” wykorzystuje XNA do stworzenia interfejsu użytkownika i animacji cząstek. Na listingu 10 umieszczono fragment kodu odpowiedzialnego za cykliczne odświeżanie obiektów modelu SPH od strony warstwy prezentacji.

Listing 10: Cykliczne odświeżanie modelu SPH przez XNA

```
1        protected override void Update(GameTime gameTime)
2        {
3            base.Update(gameTime);
4            if (!_waterModel.IsPause)
5              {
6                    SPH.Update();
7                    SPH.CheckTimeLife();
8                }
9        }
```

W linii trzeciej aktualizujemy czas gry. W kolejnej linii sprawdzamy, czy aplikacja jest aktualnie w stanie wstrzymania. Jeśli nie silnik SPH ma zaktualizować położenia cząstek oraz sprawdzić czy istnieją cząstki układu, które powinny zostać usunięte. W grze przyjęto, że każda cząstka może znajdować się na mapie określoną liczbę iteracji, po upływie których

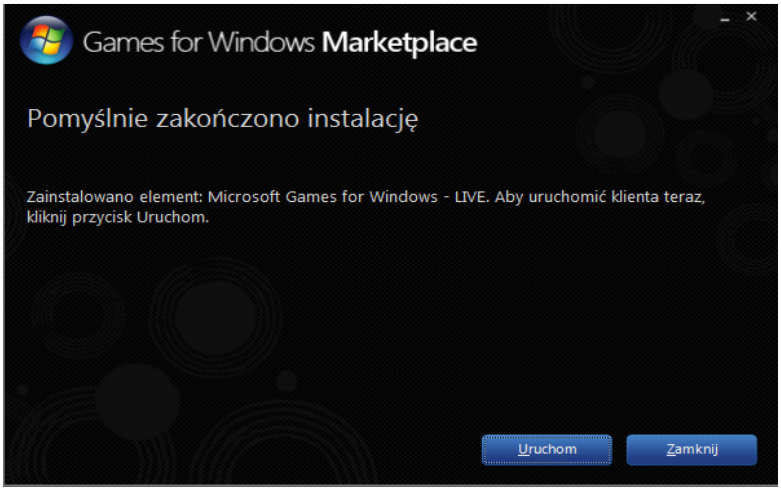
zostanie usunięta z mapy. Tego rodzaju rozwiązanie wynika jasno z zasad gry omówionych w kolejnym podrozdziale. Oprócz metody aktualizującej układ cząstek, istotną metodą z punktu warstwy prezentacyjnej jest metoda renderująca obiekty (listing 11). Głównym zadaniem metody renderującej jest wyświetlanie cząstek cieczy oraz wszystkich innych elementów widocznych na ekranie.

Listing 11: Renderowanie obiektów przy pomocy XNA

```
protected override void Draw(GameTime gameTime)
        {
            _waterModel.SpriteBatch.Begin(); // Inicjacja obszaru renderowania
            if (!LoadScreenActive) // Jeśli screen początkowy nie aktywny
            {
                foreach (var particlePosition in SPH.ParticlePositions) // Odśwież
                pozycje wszystkich cząstek w układzie
                {
                    _waterModel.SpriteBatch.Draw(_waterModel.ParticleWater.Texture,
                        new Vector2(particlePosition.X, particlePosition.Y),
                       _waterModel.ParticleWater.Rectangle,
                        _waterModel.ParticleWater.Color);
                }
             foreach (var polygon in SPH.Polygons.Where(x =>
                            x.Visable))/ Odśwież pozycje wszystkich elementów nie
             będących cząstkami cieczy
                {
                   polygon.Position = GetCursorPos();
                   foreach (string index in _waterModel.ActivePoligonIndexes)
                   {
                        WaterSource waterSource=_waterModel.WaterSources.Where(x =>
                         x.Index == index).FirstOrDefault();
                    }
                    Texture2D texture2D = _waterModel.GetTexture(polygon); //
                   pobierz teksturę 2D i narysuj obiekty inne od cząstek cieczy
                    Vector2 p1 = GetBeginPoit(polygon) + new
                   Vector2(Constants.CELLSPACE, Constants.CELLSPACE);
                    if (polygon.Index.Contains("WaterDestination")) p1 -= new
                   Vector2(20, 10);
                    _waterModel.SpriteBatch.Draw(texture2D, new Vector2((int) p1.X,
                   p1.Y), Color.White);
                    if (polygon.Index.Contains("WaterSource"))
                    {
                        WaterSource waterSource =
                            _waterModel.WaterSources.Where(x => x.Index ==
                          polygon.Index).FirstOrDefault();
                    }
                }
                if (!_waterModel.VisableSelectScreenControl)
                {
                    _waterModel.WaterForm.VisableSelectScreenControl(false);
                    _waterModel.VisableSelectScreenControl = true;
                }
            }
            else
            {
                if (_waterModel.SelectScreenActive) //
                   jeśli aktywny screen z gwizdaki
                {
                    foreach (Star star in _waterModel.Stars)
                    {
```

```
                _waterModel.SpriteBatch.Draw(star.Texture,star.Position,
                star.Rectangle, Color.White);
                 }
            }
        }
        _waterModel.SpriteBatch.End();
        base.Draw(gameTime);
    }
```

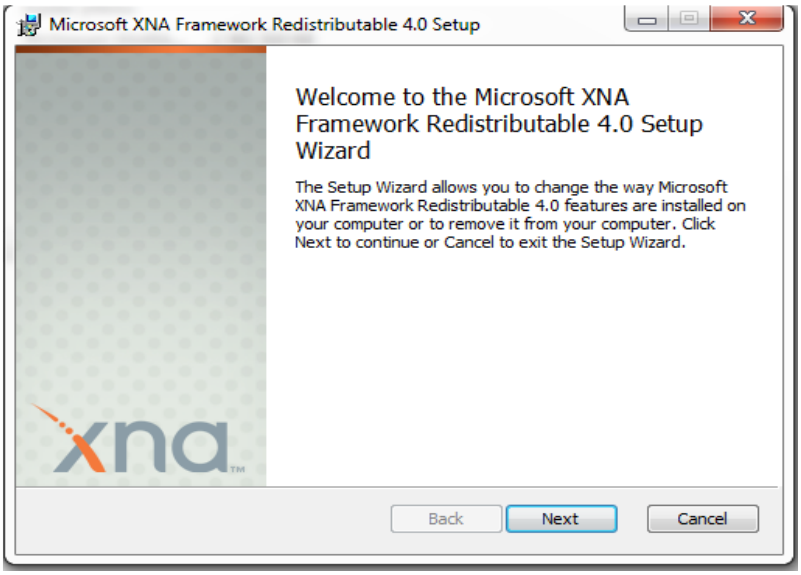
Instalację rozpoczynamy od zainstalowania Games for Windows Marketplace:



Rysunek 10: Instalacja Games for Windows Marketplace

Po ukończeniu instalacji możemy przejść do instalacji Microsoft XNA Framework Redistributable 4.0 (rysunek 10).

Po zakończeniu jej instalacji można uruchamiać wszystkie programy napisane w technologii XNA. Nie oznacza to jednak że posiadamy niezbędne środowisko do kompilacji i wytwarzania oprogramowania.



Rysunek 11: Instalacja Microsoft XNA Framework Redistributable 4.0

W niniejszej pracy użyto do tego środowiska Visual Studio 2010 z SP 1. Niestety Microsoft zaprzestał rozwoju technologii XNA. Odpowiedzią dość licznych użytkowników

i zwolenników tej technologii było utworzenie biblioteki Monogame w pełni kompatybilnej z XNA, ale typu open source. Założeniem twórców było powstanie wieloplatformowej prostej biblioteki opartej na idei Microsoft XNA 4 Framework. Teoretycznie biblioteka obsługuje Mac OS, Linux, iOS, Android, PlayStation Mobile oraz konsole Ouya. Niestety posiada nadal sporo błędów, co powoduje trudności i nierozwiązywalne problemy z kompilacją programu. Z tego powodu kod owego programu nie został przekompilowany dla Monogame.

Z drugiej strony pozostawienie programu w niepostępowej technologii XNA jest skazaniem go na porażkę. Dlatego na bazie silnika SPH powstała wieloplatformowa biblioteka SPH przygotowana w technologii firmy Xamarin. Bibliotekę SPH wykorzystano w testowej aplikacji na system Android, do realizacji której wykorzystano projekt Mono firmy Xamarin, który zakłada możliwość przeniesienia aplikacji stworzonych dla platformy Microsoft .NET na różne systemy, takie jak Android, IOS czy większość dystrybucji Linuxa. Wspierane są również konsole do gier, takie jak: PlayStation 3, Nintendo Wii i Xbox 360. Aktualnie Mono wykorzystuje domyślnie C# 4.0 oraz graficzny interfejs programowania aplikacji Windows Forms 2.0. Kolejnym celem Mono jest zapewnienie wsparcia dla.NET 4.0, co tym dla WF(Windows Workflow Foundation) oraz WCF (Windows Communication Foundation).

4.2. Menu, zasady gry i edytor poziomów

Rysunek 12: Powitalny screen aplikacji

Na Rys. 12 przedstawiono powitalny screen aplikacji.

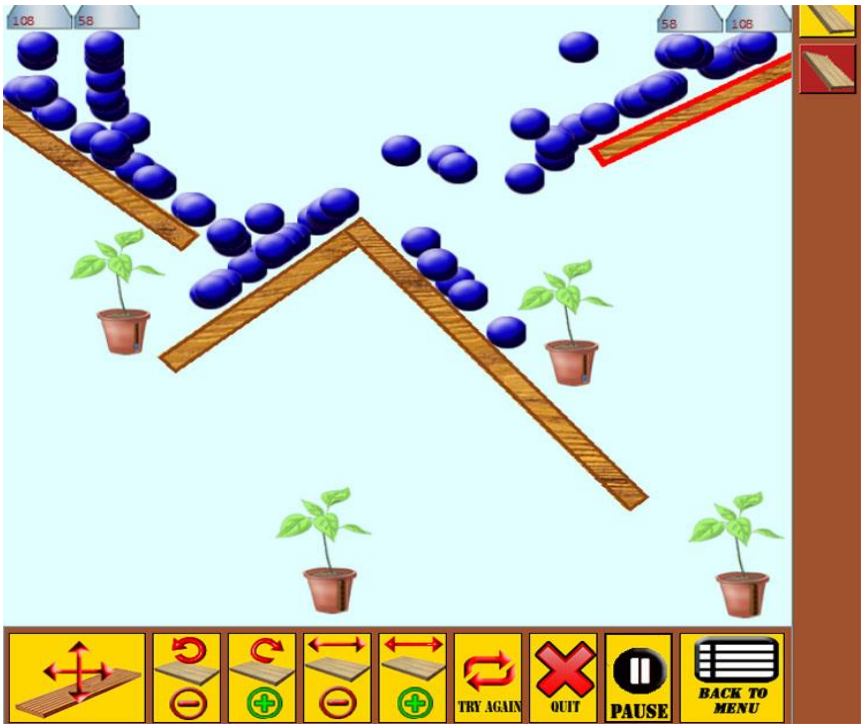
Gra nazywa się „Water your plants”, jej główną ideą jest odpowiednie ułożenie elementów na mapie danego poziomu tak, aby nawodnić wszystkie roślinki znajdujące się na planszy. Po kliknięciu przycisku „START” pojawia się menu wyboru planszy wraz ze spisem naszych dotychczasowych osiągnięć w postaci liczb zebranych gwiazdek, (maksymalnie trzy). Brak jakiejkolwiek gwiazdki świadczy, że mapy nie udało się dotychczas przejść. Menu wyboru mapy zaprezentowano na Rys. 13.

Rysunek 13: Menu wyboru planszy

Do wyboru mamy dziewięć różnych map, które różnią się poziomem trudności. Każdą z rund trzeba przejść w inny sposób, co sprawia, że gra nie jest nużąca. Pierwszy i najprostszy poziom widoczny jest na rysunku 14. Można go traktować jako formę samouczka.

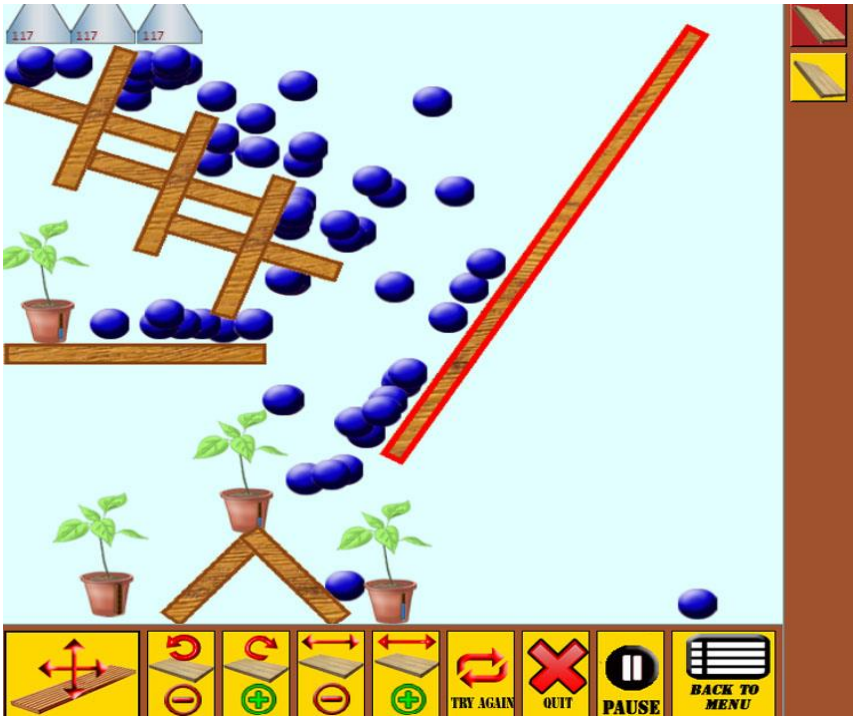
Rysunek 14: Screen z pierwszej rundy

Niebieskimi cyframi oznaczono poszczególne elementy mapy oraz przyciski interfejsu użytkownika przy pomocy, których można sterować aplikacją. Numerami od 1 do 3 oznaczono elementy mapy: 1 to źródło cieczy o pojemności 150 cząstek, 2 to ujście cieczy, w postaci roślinki z animacją wypełniającej się doniczki, 3 to deski, po których mogą poruszać się cząstki. Celem gry jest doprowadzenie cząstek wody do roślin przy pomocy desek umieszonych na prawym panelu. W wypadku pierwszego poziomu mamy do dyspozycji dwie deski oznaczone numerem 13. W dolnej części menu znajduje się pasek z przyciskami, przy pomocy którego gracz może odpowiednio ustawić elementy z prawego panelu. Numerem 4 został oznaczony przycisk przy pomocy, którego po wybraniu z prawego panelu deski można zmieniać jej położenie. Numer 5 służy do obracania obiektu w lewo, natomiast numer 6 obraca obiekt w prawo. Analogicznie 7 i 8 służy odpowiednio do modyfikowania wielkości obiektu. Numer 9 pozwala graczowi rozpocząć grę od początku, zaś dzięki numerowi 10 mamy możliwość wyjścia z gry. Wreszcie przycisk 11 uruchamia symulację. W czasie trwania symulacji interfejs użytkownika nie jest zablokowany. Aby uzyskać dobry wynik podczas trwania symulacji, gracz musi ingerować w położenie desek i odpowiednio kalkulować rozkład cząstek wody na mapie, ponieważ ich ilość jest ograniczona i każda cząstka ma ograniczony „czas życia”. Przy pomocy przycisku numer 12 można wrócić do ekranu wyboru mapy. Gra została tak pomyślana, aby z jednej strony przeprowadzać częściową symulację zjawiska przepływu cieczy, a z drugiej dostarczyć możliwie jak największej rozrywki użytkownikowi. Na rysunkach 15-18 widoczne są pozostałe poziomy gry.



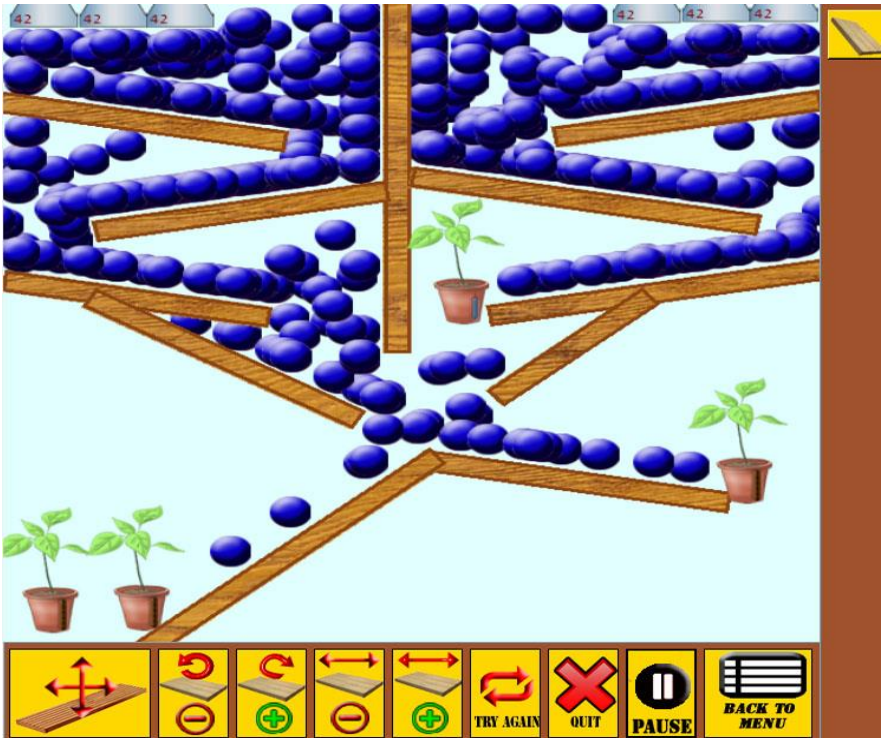
Rysunek 15: Pierwszy poziom gry po uruchomieniu symulacji

Na rys. 15 zaprezentowano przykład trwającej symulacji cieczy. Warto wspomnieć, że wszystkie obliczenia są prowadzone w czasie rzeczywistym, w związku z tym użytkownik może dowolnie przesuwać elementy wybrane z prawego panelu. Na Rys. 16 zademonstrowano wybór górnej deseczki, która na mapie została obrysowana czerwonym kolorem oraz zaznaczona na czerwono na prawym panelu.



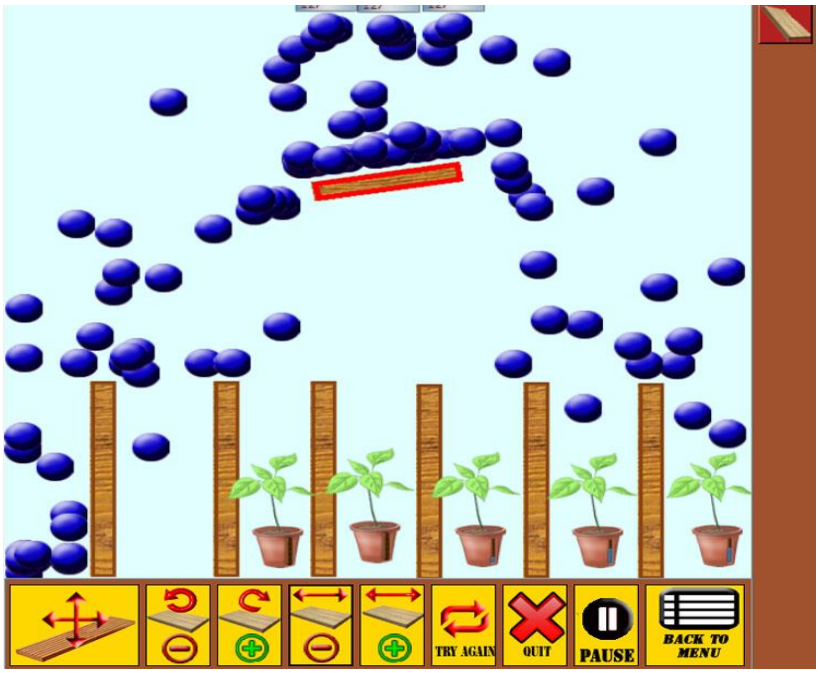
Rysunek 16: Screen z piątej rundy po uruchomieniu symulacji

Na Rys. 16 zaprezentowano przebieg gry na poziomie 5. Należy zwrócić uwagę na „młynki”, czyli trzy obiekty znajdujące się po lewej stronie u góry. Jak widać na rysunku Rys. 16 silnik gry może symulować pracę młynów wodnych.



Rysunek 17: Screen z ósmej rundy po uruchomieniu symulacji

Runda 8 zaprezentowana na rys. 17 jest interesująca z uwagi na dużą liczbę cząstek znajdujących się jednocześnie w symulowanym układzie. Poziom ten stanowi test wydajności na różnych urządzeniach.



Rysunek 18: Screen z dziewiątej rundy po uruchomieniu symulacji

Poziom nr 9, również może okazać się zajmujący, z uwagi na fakt, że wymaga od użytkownika ingerencji w czasie rzeczywistym w symulację. W przeciwnym razie nie można przejść tej rundy.

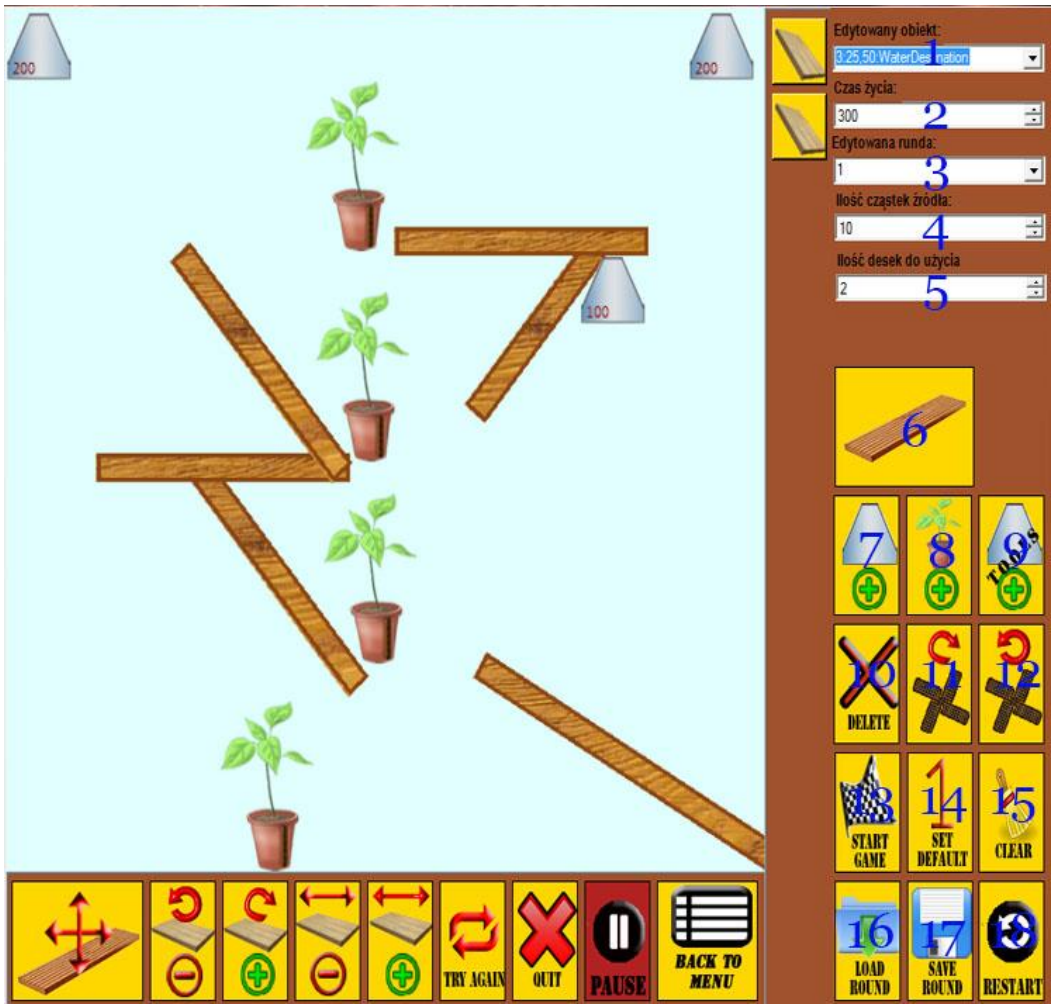
Jak widać z rysunków 14-18, gra „Water your plants” pozwala na przeprowadzenie wielu symulacji cieczy. Nie można przewidzieć jak układ cząstek zachowa się w danej rundzie, co dodatkowo wpływa na korzyść gry, ponieważ dzięki temu gra nie jest nużąca. Kolejne dwa zrzuty ekranu (19-20) prezentują ekran rundy wygranej i przegranej.

Rysunek 19: Screen z wygranej rundy

Rysunek 20: Screen z przegranej rundy

Ponieważ wygenerowanie tak wielu rund byłoby dość kłopotliwe, gdyby umieszczać dane na temat położenia w statycznej klasie w kodzie, to dla uproszczenia powstało wygodne narzędzie do generowania nowych map, edycji starych i zmiany w menu głównym na dowolnie wybrany zbiór map.

Położenie obiektów na mapach przechowywane jest w plikach typu XML. Ich tworzenie i edycja możliwe jest dzięki edytorowi widocznemu na rysunku 21. Jest on dostępny z jako dodatkowa aplikacja służąca do edycji map. Pozwala na całkowitą reorganizacje map oraz na tworzenie zupełnie nowych, a także na ustalenie niektórych reguł gry związanych z poszczególnymi obiektami.



Rysunek 21: Screen z edytora map aplikacji

Na rys. 21 zaprezentowano edytor map wraz z licznymi elementami menu oznaczonymi numerami od 1 do 18. Numerami od 1 do 5 oznaczono listy wyboru wartości, gdzie :

1 - oznacza wybór aktualnie edytowanego elementu na mapie,

2 - czas życia cząstki na mapie,

3 - edytowana rundę,

4 - ilość cząstek tworzonych przez źródło,

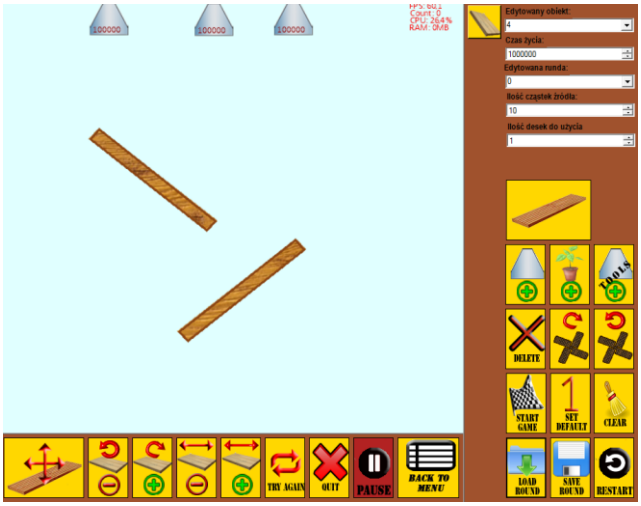
5 - liczbę desek, którą użytkownik będzie mógł ustawić na mapie.

Elementy interfejsu oznaczone numerami od 6 do 12 dodają nowe obiekty do mapy, bądź usuwają aktualnie zaznaczone. Numerem 6 oznaczono nieedytowalną przez użytkownika deskę, 7 oznacza dodanie nowego źródła na mapę, numer 8 tworzy nowe ujście cieczy, 9 nowe źródło cieczy, lecz dostępne tylko dla użytku użytkownika, zaś 10 usuwa aktualnie edytowany obiekt z mapy wybrany w liście rozwijanej oznaczonej numerem 1. Numery 11 i 12 dodają nowy obiekt typu „młynek", który będzie się obracał w prawą bądź lewą stronę. Przycisk oznaczony numerem 13 pozwala na wyjście z edytora i rozpoczęcie normalnego trybu gry, dzięki czemu zaraz po skonfigurowaniu mapy można przetestować jej działanie w aplikacji. Numer 14 ustawia mapę jako domyślną, 15 usuwa wszystkie cząstki z mapy, natomiast 18 usuwa wszystkie elementy z mapy (w tym cząstki). Przyciski 16 i 17 służą do odczytu i zapisu map w pliku.

Cały interfejs został wykonany samodzielnie. W związku z tym od strony graficznej zarówno przyciski, jak i elementy umieszczane na mapie mogą wydawać się mało atrakcyjne. Z biznesowego punktu widzenia dopracowany interfejs jest istotnym elementem. Należy jednak zwrócić uwagę na fakt, że zaprezentowane oprogramowanie ma służyć głównie do testów, w tym testów wydajności i jest zaledwie wstępną wersją gry, która mogłaby powstać na bazie zaimplementowanego przeze mnie silnika SPH.

4.3. Analiza wydajności działania aplikacji „Water your plants”

Głównym celem analizy jest zbadanie granic możliwości silnika SPH stworzonego w ramach niniejszej pracy magisterskiej. W tym celu przeprowadzono testy wydajnościowe dla grupy urządzeń o różnych parametrach technicznych. W grupie testowej znalazły się urządzenia aktualnie dostępne do kupienia, ale także nieco starsze oraz takie które już nie są dostępne w sprzedaży. Tak dobrana grupa urządzeń ma w jak najlepszy sposób odzwierciedlić sprzęt aktualnie znajdujący się w rękach użytkowników. Aby przeprowadzić porównanie wydajności każdego z tych urządzeń, ustalono parametry symulacji. Podstawowej ocenie będzie podlegać płynność działania aplikacji, mierzona na podstawie liczby klatek na sekundę oraz procent zużytych zasobów procesora. Aby wygenerować odpowiednie obciążenie stworzono specjalną testową mapę, zaprezentowaną na rys. 22.



Rysunek 22:Mapa testująca

Dodatkowo parametry przebiegu symulacji (użycie CPU, FPS i liczba cząstek) co sekundę zapisywane są do pliku w formacie csv, co umożliwia wygenerowanie wykresów przebiegu symulacji na podstawie, których będzie można dokładnie przeanalizować jej przebieg. Poniżej zaprezentowano wyniki analizy dla laptopa ProBook 4730s o parametrach technicznych:

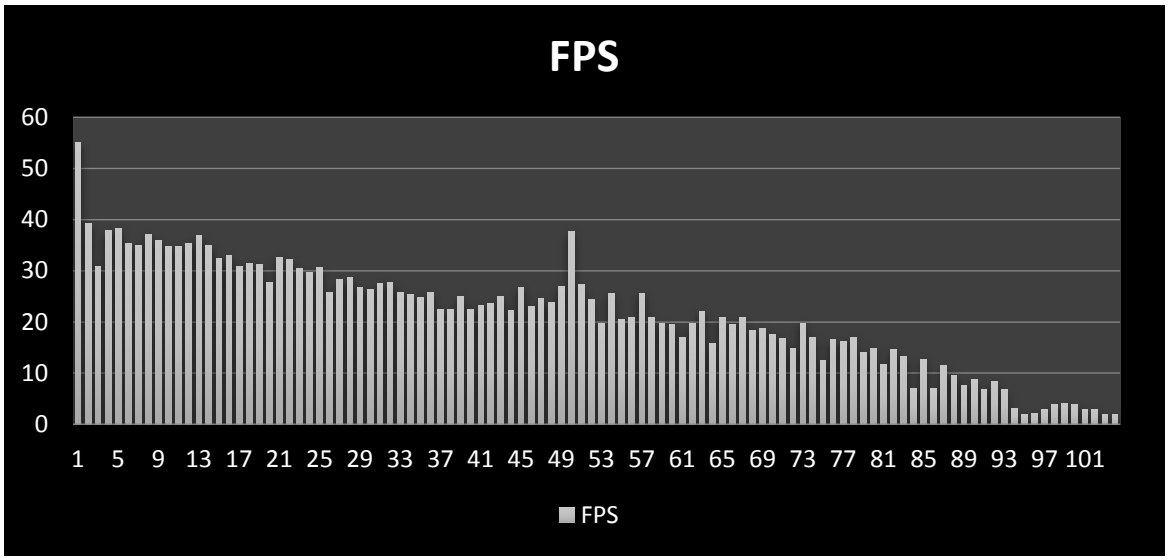
Procesor: Intel®Core™ i5-2410M CPU @ 2.30GHz ,

RAM: 4,00 GB

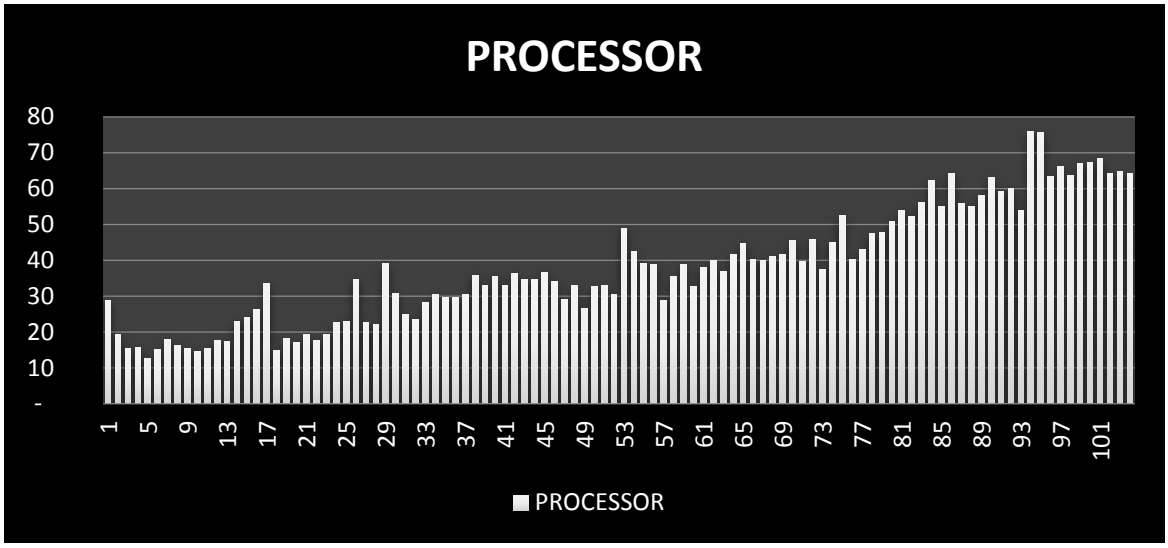
Typ systemu: 64 - bitowy system operacyjny

DYSK: Blaze Patriot SSD 240 GB

Na wykresach osie y reprezentują liczbę klatek na sekundę, procentowe zużycie procesora oraz liczbę cząstek, natomiast oś x przedstawia upływający czas w sekundach.



Wykres 1: Liczba klatek na sekundę dla laptopa ProBook 4730s

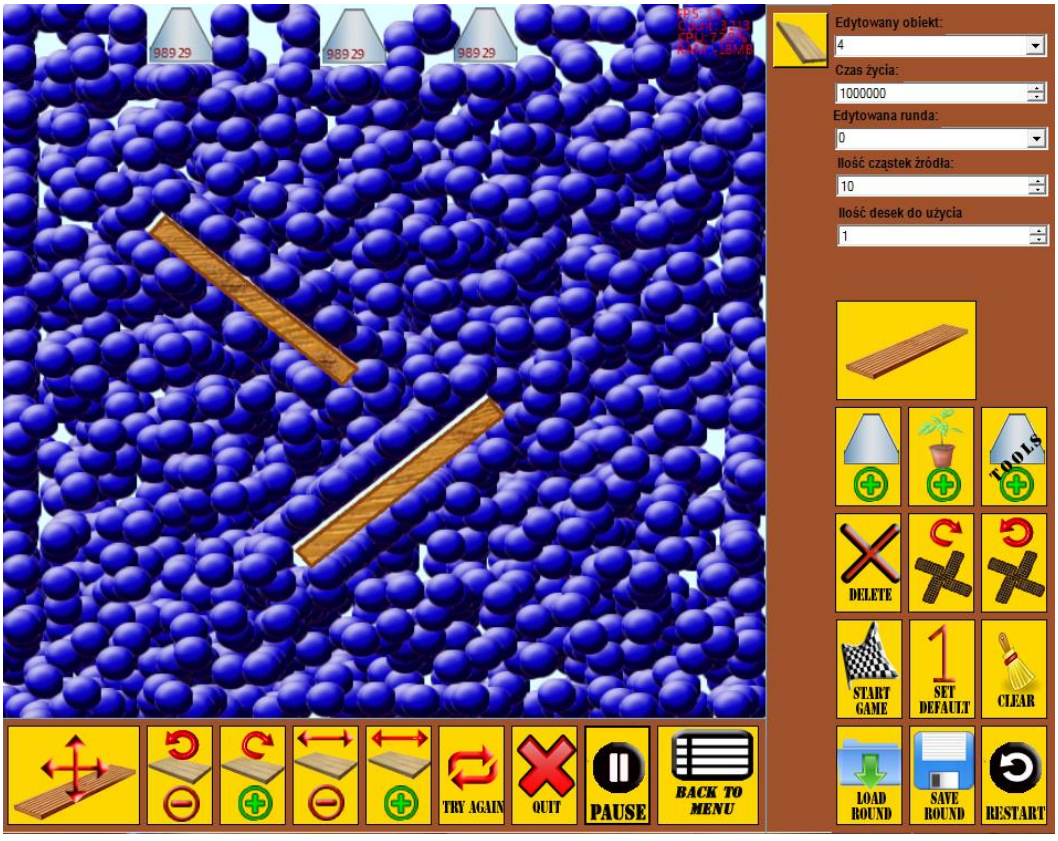


Wykres 2: Procentowe użycie procesora dla laptopa ProBook 4730s

Wykres 3: Liczba cząstek w modelu SPH dla laptopa ProBook 4730s

Z powyższych wykresów wyraźnie wynika to, jaki wpływ ma na obciążenie procesora ma liczba cząstek w modelu. Porównując wykresy 2 i 3 widoczna jest proporcjonalna zależność użycia CPU do liczby cząstek oraz odwrotnie proporcjonalna zależność liczby klatek na sekundę. Zaprezentowane powyżej wykresy są zgodne z teoretycznymi założeniami, iż wydajność metody SPH kształtuje się na poziomie O(N). Liniowe zwiększenie liczby cząstek powoduje proporcjonalny wzrost zużycia procesora i odwrotnie porównywalny spadek liczby klatek na sekundę. Na podstawie wykresów można wywnioskować, że złożoność obliczeniowa jest rzędu O(N).

W przypadku laptopa ProBook 4730s maksymalna liczba cząstek to 3 tysiące. Dla takiej wartości liczba klatek na sekundę spadła poniżej 10 i aplikacja przestała płynnie działać. Oczywiście granica 10 klatek na sekundę jest umowna. Na rys. 23 zaprezentowano wynik przeprowadzanej symulacji dla 3 tys cząstek. Widać, że to liczba znacznie wyższa niż potrzebna do tworzenia gry.



Rysunek 23: Symulacja dla laptopa ProBook 4730s przy 3213 cząstek jednocześnie znajdującymi się na mapie

Tego rodzaju analizę i testy przeprowadzono dla reprezentatywnej grupy urządzeń. Wyniki umieszczono w tabeli 1.

Tabela 1: Analiza optymalności działania silnika SPH na różnych urządzeniach

Model	Procesor	RAM	Liczba cząstek: (OPT)	Liczba cząstek (MAX)	Maksymalne użycie CPU %
laptop ProBook 4730s	Intel(R) Core(TM) i5-2410 M 2.30 GHz	4,00 GB	1992	3105	76 %
PC	Intel(R) Core(TM)2 Duo 2.33 GHz	2,00 GB	1614	2724	93 %
PC z wbudowanym ekranem dotykowym	Genuine Intel® CPU 575 2,00 GHz	1,93 GB	1521	1761	100%
PC	Intel(R) Core(TM) i5-3210M 2.50 GHz	8,00 GB	2732	3909	52%
laptop NP550P7C – S04PL	Intel(R) Core(TM) i7-4790K 4.40 GHz	12,00 GB	3675	4779	34%
Laptop z win 8	AMD E1-1500 1.48 GHz	4,00 GB	768	1512	82%

Legenda:

Model – nazwa urządzenia lub model

Procesor – nazwa procesora i częstotliwość taktowania

RAM – zainstalowana pamięć flash

Ilość cząstek (MAX) – liczba cząstek w przy spadku klatek na sekundę do dziesięciu, gdzie zaczynają się problemy z płynnością

Ilość cząstek (OPT) – liczba cząstek w przy spadku klatek na sekundę do dwudziestu, gdzie jest zachowana pełna płynność działania

Maksymalne użycie CPU – maksymalna procentowa wartość osiągnięta w czasie trwania testowej symulacji

Na podstawie Tab. 1 można wywnioskować, że badany silnik SPH współpracuje z wszystkimi testowanymi urządzeniami. Biorąc pod uwagę ich różnorodność (tablet, PC, laptop) oraz procesowy i7 – 4310M, AMD E1-6010 można stwierdzić, że wydajność stworzonego oprogramowania jest zadowalająca. Na każdym z przedstawionych urządzeń można przeprowadzić symulacje nawet do 1500 cząstek, których renderowanie i badanie kolizji przebiega w czasie rzeczywistym. Tak duża liczba cząstek, która może być jednocześnie wyświetlona, w zupełności wystarczy do stworzenia gry. Wiele gier opartych na symulacji cieczy używa mniejszej liczby cząstek, nadrabiając to odpowiednim jej renderowaniem. Przykładem tego rodzaju działań jest aplikacja: „Where's My Water", którą zaprezentowano na rys 24.

Rysunek 24: Przykład gry opartej o fizykę cieczy: „Where's My Water"

Takie renderowanie cieczy nie jest jednak zadaniem trywialnym i wymaga sporych nakładów pracy. Najprawdopodobniej aplikacja „Water your plants", stworzona na potrzeby niniejszej pracy magisterskiej, w kolejnych etapach rozwoju również zostanie rozwinięta pod tym kątem, jednak budowa odpowiedniej szaty graficznej nie była tematem niniejszej pracy.

4.4. Zrównoleglenie operacji na maszynach wielordzeniowych

W celu zrównolegnienia operacji na urządzeniach wielordzeniowych użyto nowoczesnego rozwiązania programistycznego Task Parallel Library (TPL), które pozwala na obsługę równoległą pętli i regionów. Przestrzeń nazw Parallel została wprowadzona do .NET Framework 4.5, aczkolwiek jest kompatybilna wstecz z wersją .NET Framework 4.0. Do prawidłowego działania wymaga, aby operacje wykonywane w pętli w każdym elemencie zbioru były niezależne od siebie. Na listingu 14, zaprezentowano przykładowe użycie konstrukcji `Parallel.ForEach()`.

Listing 12: Przykładowy sposób użycia konstrukcji Parallel.ForEach()

```
1        private void CarculateDensityPressure()
2        {
3            Parallel.ForEach(Particles,
4                (particle) =>
5            {
6                particle.Velocity = Constants.VELOCITY;
7                particle.Density = 0.001f;
8                List<Particle> particlesNeighborhoods = FindNeighborhood(particle);
9                foreach (var particlesNeighborhood in particlesNeighborhoods)
10                {
11                    SPHVector Rij = ParticlePositions[particle.Index] –
12                    ParticlePositions[particlesNeighborhood.Index];
13                    particle.Density += particlesNeighborhood.Mass*W(Rij,
14                    particle.H);
15                }
16                particle.Pressure = Constants.K*(particle.Density –
17                Constants.DENSITY);
18            }
19                );
```

W linii 3 i 4 powyższego fragmentu kodu zainicjalizowano pętlę foreach dla listy obiektów `Particles`, następnie przy pomocy wyrażenia lambda expression zdefiniowano operacje, które mają zostać wykonywane równolegle. Operacje te znajdują się w liniach od 5 do 18. Obiekt `particle` jest abstrakcyjną reprezentacją *i*-tego elementu listy `Particles`, z którego możemy korzystać w każdym kroku. W linii 19 zamknięto konstrukcję `Parallel.ForEach()`. W tak zdefiniowanej pętli każdy kolejny przebieg pętli jest wykonywany asynchronicznie. Pozwala to na wykonanie operacji znajdujących się w lambda w czasie zbiegającym do O(1), zakładając, że posiadamy maszynę o N lub więcej rdzeniach, gdzie N to liczba elementów zbioru, na którym wykonywane są operacje w pętli foreach. Głównymi zaletami tego rozwiązania jest prostota zastosowania. `TPL` nie wymaga od programisty „dbania” o operacje wielowątkowe, czyli np. badana zakleszczeń, lokowania zasobów, czy synchronizacji operacji wykonywanych równolegle. Natomiast główną wadą powyższego rozwiązania jest brak możliwości pełnej kontroli nad przetwarzaniem wielowątkowym.

Oczekiwana wydajność wzrasta wraz z ilością rdzeni. Dla maszyn dwurdzeniowych powinniśmy osiągnąć dwukrotny wzrost wydajności, przy trzech rdzeniach trzykrotny itd. Niestety nie jest to możliwe z uwagi na dwie zasadnicze kwestie. Po pierwsze przełączenie i synchronizacja jest kosztowna. Drugim problemem jest to, że nie wszystkie operacje są możliwe do wykonania równolegle. Przykładem tego rodzaju operacji jest wyświetlanie cząstek, które musi być wykonane przez główny wątek. W tabeli 2 zaprezentowano wyniki porównania pracy jednowątkowej i wielowątkowej na maszynach wielordzeniowych:

Tabela 2:Porównanie wydajności wersji jednowątkowej i wielowątkowej

Model	Procesor	RAM	Ilość cząstek: (OPT)	Ilość cząstek (MAX)	Maksymalne użycie CPU %	Ilość rdzeni
laptop ProBook 4730s	Intel(R) Core(TM) i5-2410 M 2.30 GHz	4,00 GB	J:1483 W: 1992 R: 34,3%	J:2231 W: 3105 R:39,2%	J:76 % W: 30% R:46%	4
PC	Intel(R) Core(TM)2 Duo 2.33 GHz	2,00 GB	J:1266 W:1614 R:27,5%	J:1978 W:2724 R:37,7%	J:55% W:93 % R:38%	2
PC	Intel(R) Core(TM) i5-3210M 2.50 GHz	8,00 GB	J:2054 W:2732 R:33 %	J:2735 W:3909 R: 42,9%	J:26% W:52% R:26%	4

Legenda:

Model – nazwa urządzenia lub model

Procesor – nazwa procesora i częstotliwość taktowania

RAM – zainstalowana pamięć flash

Ilość cząstek (MAX) – liczba cząstek w przy spadku klatek na sekundę do dziesięciu, gdzie zaczynają się problemy z płynnością

Ilość cząstek (OPT) – liczba cząstek przy spadku klatek na sekundę do dwudziestu, gdzie

zachowana jest pełna płynność działania

Maksymalne użycie CPU – maksymalna procentowa wartość osiągnięta w czasie trwania testowej symulacji

Ilość rdzeni – liczba rdzeni w procesorze danego urządzenia

J – testy przeprowadzone dla aplikacji w wersji jednowątkowej

W– testy przeprowadzone dla aplikacji wielowątkowej

R – różnica w procentach względem aplikacji jednowątkowej, a wielowątkowej

Z powyższej tabeli wynika, że pomimo zrównolegnienia, nie uzyskaliśmy znaczącego wzrostu liczby wykonywanych operacji. Powodem tak niskiego wzrostu (40%) są m. in. rozwiązania programistyczne opisane powyżej, czyli kosztowne przełączenie wątków oraz brak możliwości zrównoleglenia wszystkich operacji. Kolejną przyczyną jest fizyczna budowa procesorów wielordzeniowych, tzn. w których część układów są współdzielone, np. magistrala danych i cache procesora. Pomimo, że teoretycznie jednostka arytmetyczna mogłaby przetwarzać więcej informacji, to RAM procesora, bądź magistrala danych przydzielona do niego nie jest w stanie obsłużyć większej ilości danych. Należy również pamiętać o tym, że układ po osiągnięciu pewnej granicznej temperatury zmniejsza swoją wydajność. Chroni się w ten sposób przed przegrzaniem.

Podsumowując, pomimo, że utrzymamy procesora wielordzeniowego nie jesteśmy w stanie zwielokrotnić wydajności przetwarzania operacji. W wypadku aplikacji stworzonej na potrzeby mojej pracy udało się uzyskać zaledwie 40% wzrostu liczby przetwarzanych równocześnie cząstek.

4.5. Perspektywy dalszego rozwoju aplikacji „Water your plants”

Aplikacja i stworzony kod będą dalej rozwijane, także po obronie niniejszej pracy magisterskiej.

Kolejnym etapem rozwoju będzie wykorzystanie dostępnych bibliotek graficznych LibGDX i Cocos2d w celu stworzenia interfejsu graficznego aplikacji. Niewykluczone, że kolejne etapy będą wymagały współpracy z osobą doświadczoną w tworzeniu grafiki komputerowej, która pomoże zaprojektować bardziej przejrzysty i przyjazny interfejs graficzny.

Inną możliwością rozwoju aplikacji może być stworzenie dodatkowych obiektów interaktywnych takich, jak „kaktus”, który pochłaniałby ciecz, lecz po przekroczeniu pewnej liczby pochłoniętych cząstek powodowałby przegraną. Prócz dodawania kolejnych obiektów, można by postarać się o stworzenie kolejnych modeli cieczy. Ciecz zaprezentowana w niniejszej pracy miała charakter wody dzięki dobraniu odpowiednich parametrów fizycznych. Możliwa jest zmiana tych paramentów i stworzenie innych rodzajów cieczy takich, jak ciecz o dużej lepkości tzw. „szlam”, bądź ciecz, do której odwrotnie przyłożymy siłę grawitacji oraz ustawimy duży promień odcięcia, co spowoduje rozczepianie cząstek. Taką właśnie ciecz można porównać do pary wodnej.

Rozdział 5

Podsumowanie

Głównym celem pracy było stworzenie efektywnej implementacji silnika SPH oraz aplikacji opartej na tym silniku, która spełniałaby wymagania gry zręcznościowej. Silnik ten został zaimplementowany w postaci wieloplatformowej biblioteki SPH. Z pomocą tej biblioteki została stworzona gra zręcznościowa „Water your plants" spełniająca założenia funkcjonalne i wydajnościowe. Tym samym gra potwierdza, że model płynu SPH może zostać efektywnie zaimplementowany w grach. W silniku dodatkowo wdrożono wykrywanie kolizji z obiektami innymi niż cząstki modelu SPH przy pomocy algorytmu SAT. Zapewniając tym samym wysoką wydajność i dokładność przy wykrywaniu kolizji z obiektami innymi niż cząstki cieczy.

W pracy udało się zrealizować następujące cele:

- stworzono efektywną implementację istniejącego algorytmu SPH. Jego wydajność potwierdzono testami wydajnościowymi,
- stworzono rozbudowany silnik gry opartej na cieczy z możliwością zmiany parametrów fizycznych symulacji,
- zaimplementowano algorytm SAT służący do wykrywania kolizji między cząstkami cieczy, a innymi obiektami,
- silnik zaimplementowano tak, by można było zmienić używaną technologię, a nawet język programowania np. na język Java używany powszechnie w urządzeniach z systemem Android bądź Objective - C używany w urządzeniach firmy Apple,
- zaimplementowano zrównoleglenie obliczeń na procesorach wielordzeniowych,
- silnik SPH został przepisany do postaci wieloplatformowej biblioteki, korzystającej z narzędzi Mono firmy Xamarin
- przeprowadzono testy wydajnościowe na reprezentatywnej grupie urządzeń, które potwierdzały, iż implementacja silnika SPH może zostać użyta do tworzenia gier opartych o użycie cieczy (obliczenia przeprowadzane w czasie rzeczywistym).

Bibliografia

[1] Smoothed particle hydrodynamics: theory and application to non-spherical stars R. A. Gingold and J. J. Monaghan Institute of Astronomy, Madingtey Road, Cambridge

[2] Smoothed particle hydrodynamics Monaghan J.J. In: Annual review of astronomy and astrophysics.

[3] Hydraulic Erosion Using Smoothed Particle Hydrodynamics P. Krištof1, B. Beneš1, J. Křivánek2, O. Šťava1 Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

[4] "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection." S. Gottschalk, M. Lin, and D. Manocha. ACM SIGGRAPH. 1996.

[5] Smoothed particle hydrodynamics: Instrumentation and Methods for Astrophysics Peter J.Cossins Cornel University Library 2010

[6] Constraint Fluids Graphics Kenneth Bodin, Claude Lacoursi`ere, Martin Servin IEEE Transactions on Visualization and Computer 2012

[7] Particle-based Liquids Adrien Treuille

[8] Particle-Based Fluid Simulation for Interactive Applications. Eurographics/SIGGRAPH MATTHIAS MÜLLER,DAVID CHARYPAR AND MARKUS GROSS. Symposium on Computer Animation 2003

[9] Smoothed Particle Hydrodynamics (SPH) Krister Wiklund Visual Interactive Simulation D 5p 2007

[10] Particle-Based Fluid Simulation for Interactive Applications Matthias Müller, David Charypar and Markus Gross Department of Computer Science, Federal Institute of Technology Zürich (ETHZ), Switzerland

[11] Real time fluid simulations using hard core approximation Olov Br¨andstr¨om Ume°a University Department of Physics 2007

[12] Position Based Fluids Miles Macklin and Matthias M¨uller NVIDIA

[13] The SPH equations David Bindel Applications of Parallel Computers 2011

[14] Game Development: Collision Detection Using the Separating Axis Theorem Kah Shiu Chong
http://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169

[15] Dyn4j: SAT (Separating Axis Theorem) William Bittle 2010
http://www.dyn4j.org/2010/01/sat/

[16] Wikipedia: Smoothed-particle hydrodynamics
https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics

[17] University of Leicester :The Formation of Stars and Brown Dwarfs and the Truncation of Protoplanetary Discs in a Star Cluster Matthew R. Bate, Ian A. Bonnell, and Volker Bromm
http://www.ukaff.ac.uk/starcluster/

[18] Cse.ohio-state: SPH WATER SIMULATION http://web.cse.ohio-state.edu/~whmin/courses/cse788-2011-fall/results/Yi_Liu/SPH%20Water%20Simulation.htm