Wydział Matematyki i Nauk Informacyjnych
Politechnika Warszawska

Warsztaty badawcze 2

Symulacja Przepływu z Przeszkodami

Autorzy:
Michał Dybowski
Kacper Kurowski
Paweł Lefelbajn

Prowadzący:
mgr Przemysław Kosewski

Warszawa, 2021

Spis treści

1 Wstęp **2**

2 Octree **3**
2.1 Cel . . . 3
2.2 Opis działania . . . 3
2.3 Kompilacja pakietu *Octree* . . . 3
2.4 Zawartość pakietu *Octree* . . . 4
2.5 Testy czasu budowy drzewa i przeszukiwania . . . 5

3 Algorytm propagacji cząstek **7**
3.1 Opis ogólny . . . 7
3.2 Jedna iteracja . . . 8
3.2.1 Wykrywanie ewntualnego zderzenia . . . 8
3.2.2 Propagowanie cząstek . . . 9
3.2.3 Aktualizowanie prędkości cząstek . . . 10
3.2.4 Wyznaczenie nowych przewidywanych położeń . . . 10
3.3 Wizualizacja działania algorytmu . . . 10

4 Aplikacja okienkowa **12**
4.1 Sposób użytkowania . . . 12
4.1.1 Przygotowanie środowiska . . . 12
4.1.2 Obsługa aplikacji . . . 13
4.2 Opis modułu . . . 16

Wstęp

Poniższy raport opisuje działanie programu symulującego przepływ cząstek z przeszkodami. Składa się on z trzech głównych części:

1. Budowy drzewa *octree* mającego na celu przyspieszenie procesu wykrywania zderzeń (**Michał Dybowski**),
2. Algorytmu propagacji cząstek wraz z zachowaniem ich podczas zderzeń (**Kacper Kurowski**),
3. Aplikacji okienkowej, przy pomocy której użytkownik uruchamia program tworzący i wizualizujący symulację (**Paweł Lefelbajn**).

Całość projektu została zrealizowana przy pomocy języka *python3*; kod źródłowy dostępny jest na repozytorium na githubie: `https://github.com/Korigami/Warsztaty_Badawcze_2020-2021_Lagrangian`

Octree

2.1 Cel

Celem budowy drzewa *octree* jest znaczne skrócenie czasu znajdowania trójkątów znajdujących się dostatecznie blisko poszczególnych cząstek w celu rozważenia ewentualnego odbicia cząstki od powierzchni (trójkąta) obiektu.

2.2 Opis działania

Ogólny przebieg budowy drzewa *octree* oraz następnie przeszukiwanie drzewa cząstkami wygląda następująco:

Na początku interesujący nas obiekt należy obudować jedną kostką (sześcianem), która to będzie stanowiła *świat* dla naszych rozważań i jednocześnie korzeń drzewa *octree*. Każdy wierzchołek drzewa będzie przechowywał trójkąty, które przecinają się z kostką reprezentującą dany wierzchołek, przy czym narzucamy ograniczenie na maksymalną liczbę trójkątów, które może przechowywać jeden wierzchołek.

Mając już pierwszą, globalną kostkę do drzewa dodajemy kolejno trójkąty. Trójkąty dodawane są do korzenia do momentu gdy wierzchołek nie będzie pełen trójkątów - wówczas kostka ta ulega podziałowi na 8 mniejszych kostek równej objętości - kostki te są *dziećmi* dla kostki, którą dzieliliśmy oraz ogólnie rzecz biorąc wierzchołkami. Następnie trójkąty z korzenia przenoszone są na dzieci. Przy czym tak naprawdę bierzemy pod uwagę jedynie te dzieci spośród 8, których kostki przecinają się z co najmniej jednym trójkątem. Taki zabieg służy temu, aby drzewo nie posiadało zbędnych, pustych wierzchołków, które nie odgrywają żadnej roli z punktu widzenia symulacji. Wierzchołki, które nie posiadają dzieci będziemy nazywać *liśćmi*.

Po opróżnieniu korzenia z trójkątów, następuje ponowne dodanie rozważanego trójkąta do drzewa. Wówczas następuje procedura dodania trójkąta do tych dzieci, których kostki przecinają się z tym trójkątem. Dodawanie to wykonywane jest na tej samej zasadzie co dodawanie do korzenia. Procedura ta jest kontynuowana dopóty, dopóki nie dodamy wszystkich trójkątów z siatki.

Uwaga: Na drzewo naniesione jest ograniczenie w postaci maksymalnej wysokości, aby uniknąć „nieskończonego" podziału wierzchołków. W związku z tym liście znajdujące się na najwyższym poziomie drzewa być może będą zawierały więcej trójkątów, niż jest to określone poprzez ograniczenie na maksymalną liczbę trójkątów w wierzchołku.

2.3 Kompilacja pakietu *Octree*

Procedura opisana wyżej obudowana jest pakietem *Octree*. Aby skompilować pakiet na swoim urządzeniu, należy wykorzystać plik *setup.py* wpisując w wiersz poleceń/terminal następującą komendę:

python setup.py build_ext --inplace

Zbudowany zostanie wówczas pakiet *Octree* na podstawie pliku *Octree/octree.pyx*.

2.4 Zawartość pakietu *Octree*

Pakiet składa się trzech klas: **Cube**, **Node**, **Octree**.

- **Cube** - klasa reprezentuje kostkę.
  Parametry wejściowe i jednocześnie atrybuty:
  - *origin* - środek kostki.
  - *r* - promień kostki, rozumiany jako odległość środka kostki od dowolnej ściany.
- **Node** - klasa reprezentuje wierzchołek drzewa.
  Parametry wejściowe i jednocześnie atrybuty:
  - *cube* - kostka reprezentująca wierzchołek.
  - *level* - wysokość wierzchołka na drzewie.

  Pozostałe główne atrybuty:
  - *triangles* - macierz trójkątów przypisana do wierzchołka.
  - *children* - lista dzieci wierzchołka.
- **Octree** - klasa reprezentuje drzewo.
  Parametry wejściowe:
  - *init_cube* - kostka reprezentująca świat.
  - *max_leaf_size* - maksymalna liczba trójkątów, które może przechowywać wierzchołek.
  - *max_height* - maksymalna wysokość drzewa.
  - *triangles_mesh* - siatka trójkątów.

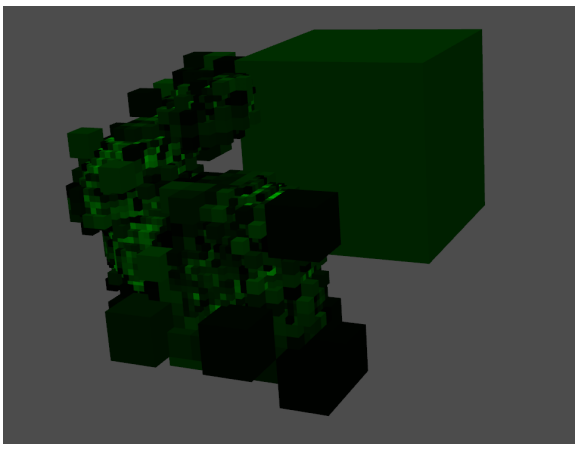
  Atrybuty:
  - *root* - korzeń drzewa, czyli startowy wierzchołek na bazie, którego budowane są pozostałe wierzchołki.
  - *height* - obecna wysokość drzewa.
  - *real_max_leaf_size* -„rzeczywista" maksymalna liczba trójkątów, które może przechowywać wierzchołek (wierzchołki na samej górze drzewa mogą mieć więcej trójkątów niż określa parametr *max_leaf_size* - patrz **Uwaga** w **Opisie działania**).
  - *triangles_mesh_v0*, *triangles_mesh_v1*, *triangles_mesh_v2*, *triangles_mesh_normals* - siatka trójkątów (rozbita na składowe), z której budowane jest drzewo.
  - *max_leaf_size*, *max_height* wspomniane wyżej.

Z praktycznego punktu widzenia do symulacji istotne są jedynie dwie metody klasy *Octree*, a mianowicie

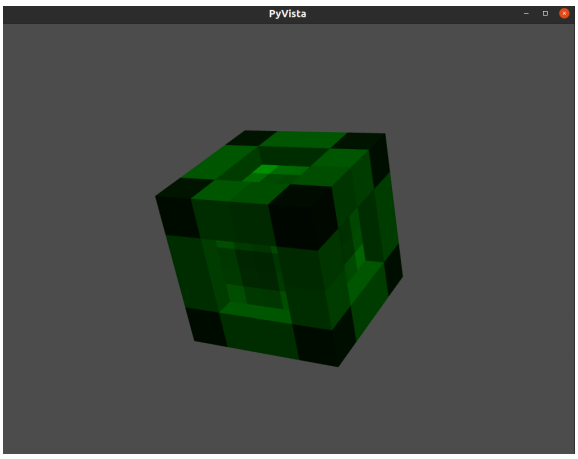
- *Octree.build(how_many_triangles)* - funkcja służy do budowy drzewa. Parametr *how_many_triangles* określa z ilu trójkątów ma zostać zbudowane drzewo - parametr służy raczej do potrzeb testowych. Brak podania parametru skutkuje zbudowaniem drzewa z całej siatki trójkątów.
- *Octree.search(particles)* - funkcja służy do przeszukania drzewa macierzą cząstek podaną w parametrze wejściowym. Cząstka począwszy od korzenia przechodzi przez kolejne wierzchołki, w których to kostkach się znajduje, aż do osiągnięcia liścia. Na wyjściu otrzymujemy macierz, której kolejne wiersze oznaczają kolejne cząsteczki, natomiast kolejne kolumny oznaczają przypisane do danej cząstki trójkąty z liści. Liczbę kolumn określa parametr *real_max_leaf_size*. Wartości -1 są wartościami technicznymi i należy je pominąć w interpretacji.

Przykładowy wydruk zbudowanego drzewa octree można znaleźć w pliku *Octree/wydruki/bunny_octree.txt*, natomiast przykładową macierz otrzymaną za pomocą metody *search()* można znaleźć w pliku *Octree/wydruki/bunny_triangles_matrix.txt*

Ponadto tak zbudowane drzewo prezentuje się graficznie (im ciemniejszy kolor, tym więcej trójkątów zawiera dana kostka):



Rysunek 2.1: Drzewo zbudowane na podstawie królika

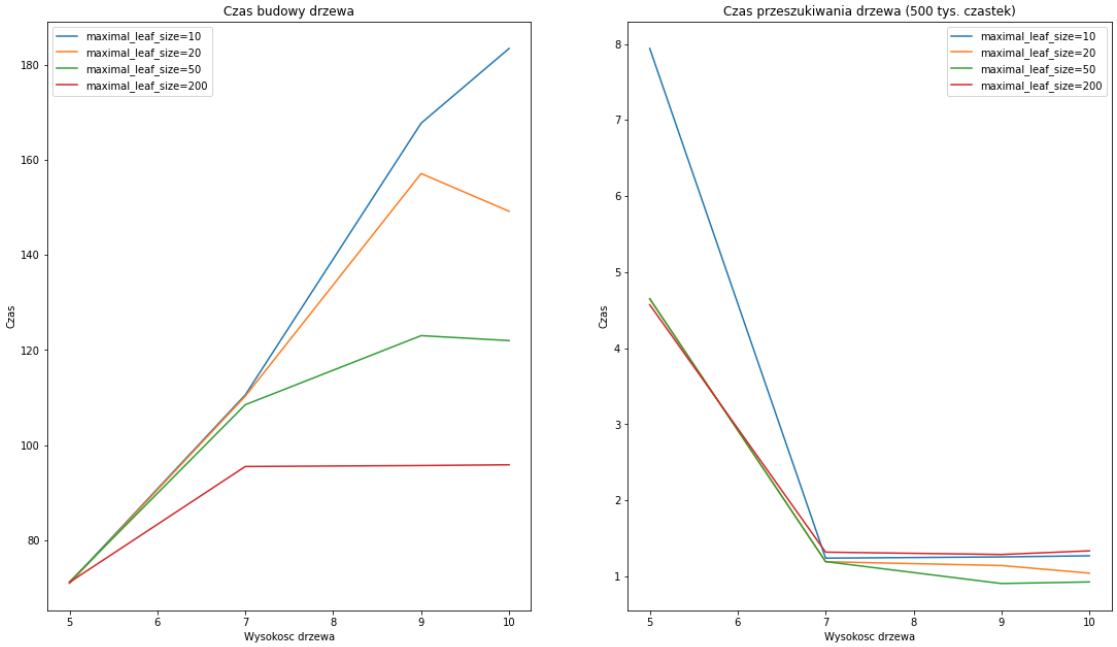


Rysunek 2.2: Drzewo zbudowane na podstawie kostki Mengera

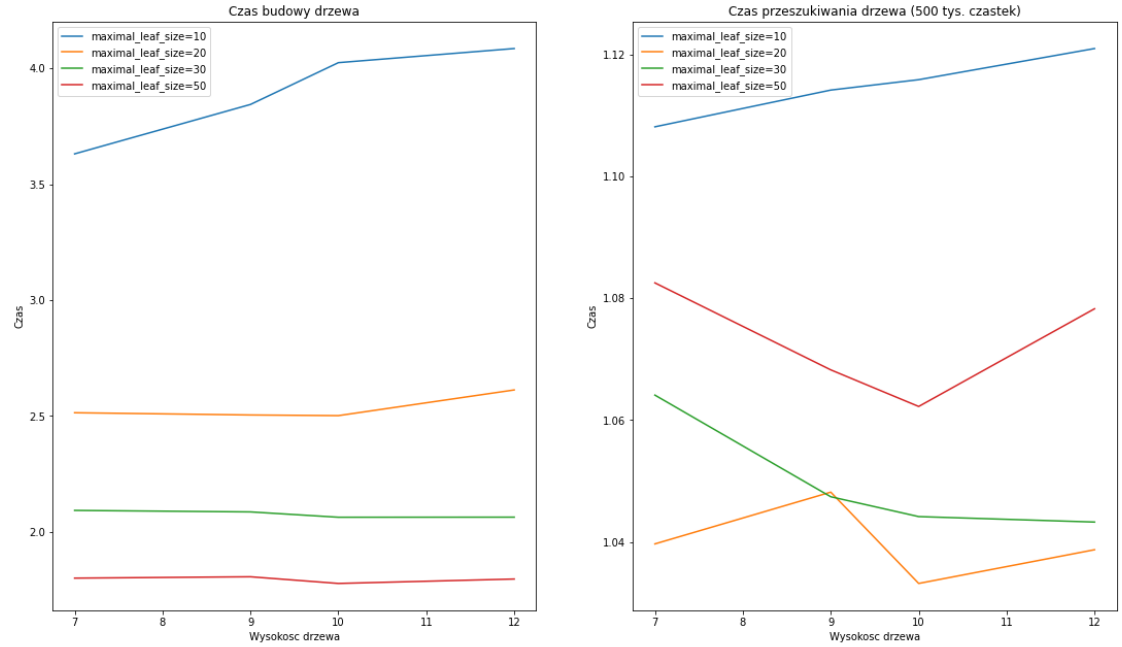
Powyższe ilustracje zostały wykonane za pomocą dwóch bliźniaczych skryptów *Octree/bunny_mesh_graphic.py* oraz *Octree/menger_mesh_graphic.py*

2.5 Testy czasu budowy drzewa i przeszukiwania

Ponieważ celem drzewa *octree* jest optymalizacja czasu wykonywania symulacji, przeprowadzone zostały testy czasu budowy drzewa vs. czasu przeszukiwania drzewa cząstkami przy różnych parametrach *max_leaf_size* oraz *max_height*. W ramach testów napisane zostały skrypty *Octree/bunny_time_test.py* oraz *Octree/menger_time_test.py*, których rezultatem są następujące wykresy:



Rysunek 2.3: Testy czasów dla królika



Rysunek 2.4: Testy czasów dla kostki Mengera

Testy wskazują, że optymalnymi parametrami (*max_leaf_size*, *max_height*) dla królika są $(50, 9)$ natomiast dla kostki Mengera $(20, 10)$ - należy brać głównie pod uwagę czas przeszukiwania, gdyż drzewo buduje się jedynie raz, natomiast przeszukiwanie następuje w każdej iteracji czasowej. Pomimo wybranych optymalnych parametrów do symulacji zastosowane zostały inne parametry, ze względu na czas wykonywania części dotyczącej propagacji cząstek.

Algorytm propagacji cząstek

3.1 Opis ogólny

Celem algorytmu jest znalezienie wszystkich położeń cząstek poprzez uprzednio zadaną liczbę iteracji. Jest on w całości zawarty w pliku *PropagateParticles/propagate_particles.py*.

Dane wejściowe:

- Związane ze światem:
  - Wektor przyspieszenia grawitacyjnego g,
  - Wartość współczynnika tarcia powierza C_D,
  - Prędkość wiatru w,
  - Gęstość powietrza ρ
- Związane z *octree*, wymienione w poprzedniej części raportu
- Związane z cząstkami:
  - Położenia początkowe x_0^p,
  - Prędkości początkowe v_0^p,
  - Masy cząstek m^p,
  - Pola przekrojów poprzecznych cząstek S^p,
- Związane z czasem:
  - Liczba kroków czasowych n_t,
  - Długość kroku czasowego Δt.

Dane wyjściowe:

- Trójwymiarowa tablica położeń cząstek x_t^p,
- Plik typu *.pkl* przechowujący rzeczoną tablicę.

Z danych początkowych wyznaczamy przewidywane położenia cząstek $\hat{x}_{\Delta t}^p$ w chwili Δt przy założeniu, że w przedziale czasowym $[0, \Delta t]$ będą się one poruszały ruchem prostoliniowym jednostajnym z prędkością v_0^p. Mamy zatem $\hat{x}_{\Delta t}^p := x_0^p + \Delta t v_0^p$. Algorytm iteruje opisaną w sekcji 3.2 procedurę n_t razy.

3.2 Jedna iteracja

Jedna iteracja algorytmu składa się z czterech głównych części:

1. Wykrywanie ewentualnego zderzenia,
2. Propagowanie cząstek (z ewentualnymi zderzeniami) przez dany krok czasowy Δt,
3. Aktualizowanie prędkości cząstek,
4. Wyznaczenie nowych przewidywanych położeń.

Poniżej opisujemy każdą z nich.

3.2.1 Wykrywanie ewntualnego zderzenia

Pierwszy etap algorytmu działa przy następujących założeniach:

- W danym odcinku czasowym $[t, t+\Delta t]$ ruch cząstki jest ruchem prostym, o ile nie dojdzie do zderzenia,
- Krok czasowy Δt jest wystarczająco mały, by poprzednie założenie nie powodowało problemów,
- Zderzenia są idealnie sprężyste,
- Dla każdej cząstki p przechowujemy informację o jej pozycji początkowej x_t^p w chwili t, prędkości chwilowej v^p oraz o jej przewidywanej pozycji $\hat{x}_{t+\Delta t}^p$ w chwili $t+\Delta t$ przy założeniu, że cząstka w czasie $[t, t+\Delta t]$ porusza się ruchem jednostajnym, prostoliniowym, z prędkością v^p. Innymi słowy, $\hat{x}_1^p = x_0^p + \Delta t v^p$.

Niech I_t^p będzie zbiorem indeksów trójkątów, które znajdują się w tym samym fragmencie świata w chwili t, co cząstka p. Podobnie, niech $I_{t+\Delta t}^p$ będzie zbiorem indeksów trójkątów, które znajdują się w tym samym fragmencie świata w chwili $t+\Delta t$, co cząstka p, przy założeniu, że poruszała się ruchem jednostajnym prostoliniowym. Oznaczmy $I_{t,\text{tot}}^p \coloneqq I_t^p \cup I_{t+\Delta t}^p$.

Oznaczmy:

$$T[I_{t,\text{tot}}^p] \coloneqq \left\{ \triangle_i \in T \colon i \in I_{t,\text{tot}}^p \right\}$$

Zakładamy zatem, że krok czasowy Δt jest wystarczająco mały, by cząstka p nie mogła przejść przez więcej niż jeden kawałek świata w czasie $[t, t+\Delta t]$.

Dla każdego trójkąta $\triangle_i \in T[I_{t,\text{tot}}^p]$ wyznaczamy teoretyczny czas zderzenia $ttaui^p$ cząstki p z jego powierzchnią w następujący sposób.

1. Wyznaczamy początkową wysokość względną h_t^p cząstki p nad $\triangle_i$ przy pomocy wzoru

   $$h_t^p \coloneqq (x_t^p - v_{i,1}) \cdot n_i,$$

   gdzie $v_{i,1}$ jest położeniem pierwszego wierzchołka $\triangle_i$, zaś n_i unormowanym wektorem normalnym do powierzchni trójkąta $\triangle_i$.

2. Wyznaczamy końcową wysokość względną $\hat{h}_{t+\Delta t}^p$ cząstki p nad $\triangle_i$ przy pomocy wzoru

   $$\hat{h}_{t+\Delta t}^p \coloneqq (\hat{x}_{t+\Delta t}^p - v_{i,1}) \cdot n_i.$$

3. Przy założeniu, że cząstka p porusza się ruchem prostoliniowym jednostajnym z prędkością v_t^p, zderzenie z płaszczyzną indukowaną przez trójkąt $\triangle_i$ zajdzie po czasie

   $$\tau_i^p \coloneqq -\frac{h_t^p}{\hat{h}_{t+\Delta t}^p - h_t^p} \Delta t$$

   w punkcie:

   $$\widetilde{x}_{\tau_i^p}^p \coloneqq x_t^p + \tau_i^p v_t^p.$$

4. Wyrażamy punkt (hipotetycznego) zderzenia $\widetilde{x}^p_\tau$ we współrzędnych barycentrycznych względem trójkąta $\triangle_i$. Innymi słowy, szukamy $\widetilde{y}^p_\tau$, które rozwiązuje poniższe równanie:

$$[v_{i,1}, v_{i,2} - v_{i,1}, v_{i,3} - v_{i,1}]\widetilde{y}^p_{\tau^p_i} = \widetilde{x}^p_{\tau^p_i},$$

gdzie $v_{i,2}$ i $v_{i,3}$ są współrzędnymi, kolejno, drugiego i trzeciego wierzchołka trójkąta $\triangle_i$.

Aby nie rozwiązywać w każdej iteracji powyższego równania liniowego, dla każdego trójkąta $\triangle_i$ przechowujemy informację o macierzy $B_i \coloneqq [v_{i,1}, v_{i,2} - v_{i,1}, v_{i,3} - v_{i,1}]^{-1}$, dzięki czemu znalezienie $\widetilde{y}^p_{\tau^p_i}$ odbywa się poprzez proste przy pomocy wzoru:

$$\widetilde{y}^p_{\tau^p_i} = B_i \widetilde{x}^p_{\tau^p_i},$$

5. Zderzenie z trójkątem $\triangle_i$ może zajść (od strony zewnętrznej do powierzchni $\triangle_i$ (, wyznaczonej na podstawie n_i), przy następujących założeniach:

   - $h^p_t > 0$, czyli cząstka p w chwili t była nad trójkątem $\triangle_i$,
   - $\hat{h}^p_{t+\Delta t} < 0$, czyli przewidujemy, że cząstka po czasie Δt znajdzie się pod trójkątem $\triangle i$,
   - $\widetilde{y}^p_{\tau^p_i}[2] \geq 0$, $\widetilde{y}^p_{\tau^p_i}[3] \geq 0$ oraz $\widetilde{y}^p_{\tau^p_i}[2] + \widetilde{y}^p_{\tau^p_i}[3] \leq 1$, czyli zderzenie z płaszczyzną indukowaną przez $\triangle_i$ zachodzi w istocie wewnątrz trójkąta $\triangle_i$, gdzie $\widetilde{y}^p_{\tau^p_i}[2]$ i $\widetilde{y}^p_{\tau^p_i}[3]$ są, kolejno, drugą i trzecią współrzędną miejsca zderzenia z płaszczyzną indukowaną przez $\triangle_i$ wyrażonymi we współrzędnych barycentrycznych.

   Jeżeli spełnione są wszystkie powyższe warunki, to wiemy, że odcinek łączący punkty x^p_t i $\hat{x}^p_{t+\Delta t}$ przecina się z trójkątem $\triangle_i$ z odpowiedniej strony, a więc, pod warunkiem, że nie przeciął się wcześniej (tj. bliżej x^p_t) z odpowiedniej strony z innym trójkątem, to wówczas zderzenie nastąpi z trójkątem $\triangle_i$.

   Jeżeli którykolwiek z powyższych warunków nie jest spełniony, to wiemy, że zderzenie z trójkątem $\triangle_i$ zajść nie powinno, co pozwala nam na ustawienie $\tau^p_i \coloneqq +\infty$.

Stosując powyżej opisany algorytm otrzymujemy listę czasów τ^p_i do teoretycznego zderzenia cząstki p z trójkątem $\triangle_i$. Wybieramy z tych czasów najmniejszy z nich:

$$\tau^p \coloneqq \min\left\{\tau^p_i \colon i \in I^p_{t,\mathrm{tot}}\right\}.$$

Jeżeli $\tau^p \neq +\infty$, to wiemy, że istnieje trójkąt $\triangle_i$, z którym dojdzie do zderzenia. Zapamiętujemy indeks i^p_t tego trójkąta.

3.2.2 Propagowanie cząstek

Następnym krokiem jest propagowanie cząstek od czasu t do czasu $t + \Delta t$. Proces ten może zachodzić na dwa sposoby, zależnie od tego, czy cząstka p zderzyła się z jakimś trójkątem (, a więc, $\tau^p \neq +\infty$), czy nie.

- W pierwszym przypadku, gdy zaszło zderzenie (, a więc, $\tau^p \neq +\infty$), propagujemy cząstkę przez czas τ^p ruchem prostoliniowym jednostajnym z prędkością v^p_t aż do osiągnięcia zderzenia z trójkątem i^p_t.

  Następnie symulujemy zderzenie idealnie sprężyste, otrzymując nową prędkość:

  $$\widetilde{v}^p_t \coloneqq v^p_t + 2(v^p_t \cdot n_{i^p_t})n_{i^p_t}.$$

  Finalnie, propagujemy cząstkę z nową prędkością ruchem jednostajnym prostoliniowym aż do końca rozważanego czasu.

- W drugim przypadku propagujemy cząstkę przez cały krok czasowy Δt z prędkością v^p_t ruchem jednostajnym prostoliniowym.

3.2.3 Aktualizowanie prędkości cząstek

Ostatnim krokiem każdej iteracji jest aktualizowanie prędkości cząstek. Rozważamy działanie dwóch sił:

- Siły grawitacyjnej $F_g = m^p g$, gdzie wartość wektora g oraz masy m^p cząstki p jest zadana od początku,
- Siły oporu powietrza $F_w = -C_D S^p \rho \frac{\|w^p\|}{2} w^p$, gdzie C_D, S^p, ρ, w^p to kolejno: współczynnik oporu, pole przekroju cząstki p, gęstość powietrza, w^p prędkość względna cząstki p względem wiatru.

Uzyskujemy z nich w chwili t przyspieszenie $a_t^p \coloneqq \frac{1}{m^p}(F_g + F_w)$ i na jego podstawie obliczamy nowe prędkości:

$$v_{t+\Delta t}^p \coloneqq v_t^p + \Delta t a_t^p.$$

3.2.4 Wyznaczenie nowych przewidywanych położeń

Finalnym etapem iteracji jest wyznaczenie nowych przewidywanych położeń cząstek $\hat{x}_{t+2\Delta t}^p$ zadanych wzorem:

$$\hat{x}_{t+2\Delta t}^p \coloneqq x_{t+\Delta t}^p + \Delta t v_{t+\Delta t}^p.$$

3.3 Wizualizacja działania algorytmu

Głównymi częściami algorytmu, które możemy zwizualizować są:

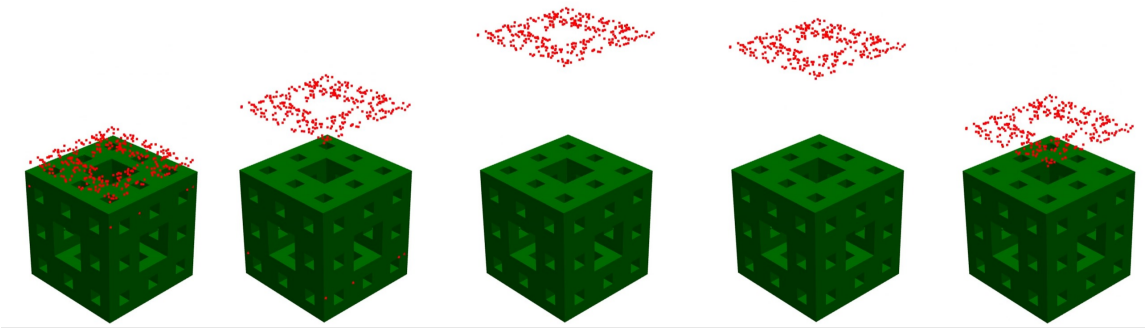
- Odbijanie się cząstek od powierzchni,
- Aktualizowanie prędkości cząstek

Poniższe zrzuty ekranu pochodzą z jednej z wizualizacji działania algorytmu w sytuacji, gdy przeszkodą jest królik.

Rysunek 3.1: Proces odbijania się cząstek; lewe zdjęcie: przed odbiciem, środkowe: niedługo po, prawe: na dłuższy czas po

Jak widać, zderzenia rejestrują się i kierunek odbicia jest uzależniony od wektora normalnego do powierzchni trójkątów, którymi przybliżamy powierzchnię królika. Z tego też powodu na moment po odbiciu możemy zauważyć powierzchnie poszczególnych trójkątów, które przyczyniają się do odrzucenia cząstek w sposób nieciągły.

Przejdźmy następnie do wizualizacji aktualizacji prędkości cząstek:



Rysunek 3.2: Proces aktualizacji prędkości cząstek: Od lewej do prawej ewolucja w czasie; wpierw cząstki kierują się do góry, potem spadają. Niewielkie czerwone cząstki na prawym boku kostki na dwóch pierwszych zdjęciach świadczą o obecności wiatru — jego niewielka obecność sprawiła, że niektóre z cząstek przesunęły się poza obrys kostki i zaczęły spadać.

Jak widać, cząstki po odbiciu opadają — stwierdza to występowanie siły grawitacyjnej, która sprawia, że cząstki niedługo po odbiciu od powierzchni królika do góry, zaczynają opadać. Siła oporu powietrza jest nieznaczna — jej wpływ jest niewielki, co wynika zarówno z niewielkiej prędkości wiatru, współczynnika tarcia oraz gęstości powietrza jak i tego, iż pola przekroju cząstek zostały przyjęte jako niewielkie.

Aplikacja okienkowa

4.1 Sposób użytkowania

4.1.1 Przygotowanie środowiska

Linux

W celu uruchomienia aplikacji należy zainstalować odpowiednie moduły, znajdujące się w pliku *python-requirements.txt*.

W tym celu można wykorzystać skrypt *makevenv.sh*, który utworzy wirtualne środowisko i zainstaluje wszystkie wymagane moduły.Aby uruchomić skrypt należy mieć zainstalowany instalator paczek pip oraz virtualenv. W tym celu można wykorzystać skrypt install_venv_ubuntu.sh.

Aby wykonać wyżej opisane czynności należy otworzyć terminal w katalogu projektu i wpisać

sudo bash install_venv_ubuntu.sh

sudo bash makevenv

Aby aktywować wirtualne środowisko należy wpisać w terminalu

source /venv/bin/activate

Teraz możemy uruchomić aplikację wpisując w terminal

python3 run.py

Aby dezaktywować wirtualne środowisko należy wpisać

deactivate

Windows

W przypadku windowsa również potrzeba zainstalować odpowiednie pakiety. W tym celu również można wykorzystac wirtualne środowiska. Żeby móc z nich skorzystać można zainstalowac je np za pomocą skryptu

install_venv_windows.bat.

Następnie aby utworzyć virtualenva, należy otworzyć terminal i będąc w katalogu projektu wpisać

python3 -m venv venv

venv/Scripts/activate.bat

python3 -m pip install -r python-requirements.txt

Teraz możemy uruchomić aplikację wpisując w terminal

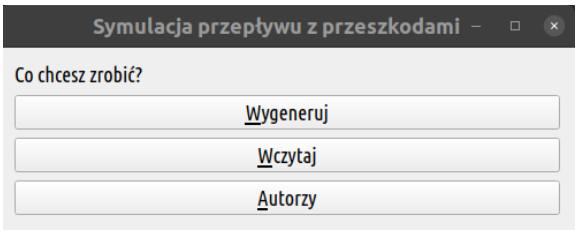
python3 run.py

Aby dezaktywować wirtualne środowisko należy wpisać

deactivate

4.1.2 Obsługa aplikacji

Po uruchomieniu aplikacji pojawi się następujące okno



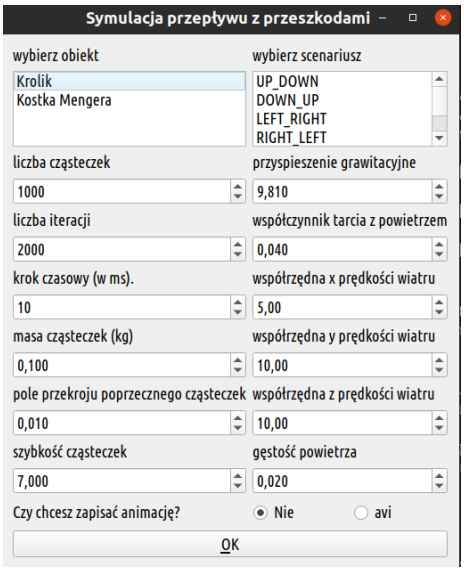
Rysunek 4.1: Okno startowe aplikacji

Mamy następujące możliwości

- **Wygeneruj** - możliwość wygenerowania nowej symulacji
- **Wczytaj** - możliwość wczytania pliku w rozszerzeniu .pkl z już obliczonymi trajektoriami cząsteczek
- **Autorzy** - wyświetlenie listy autorów projektu

Wygenerowanie nowej symulacji

Po wybraniu przycisku Wygeneruj pojawi nam się następujące okno



Rysunek 4.2: Okno do wygenerowania nowej symulacji

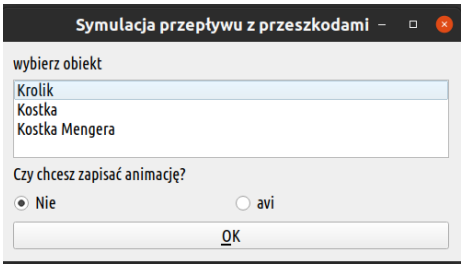
Możemy tutaj wykonać następujące czynności:

- wybrać obiekt spośród dostępnych
- wybrać kierunek spośród dostępnych
  - *UP_DOWN* - cząsteczki lecą z góry w dół
  - *DOWN_UP*- cząsteczki lecą z dołu do góry
  - *LEFT_RIGHT*- cząsteczki lecą lewej strony na prawą
  - *RIGHT_LEFT*- cząsteczki lecą z prawej strony w lewą
  - *FRONT_BACK*- cząsteczki lecą od przodu do tyłu
  - *BACK_FRONT*- cząsteczki lecą od tyłu do przodu
- podać liczbę cząsteczek od 1 do 1000000
- podać liczbę kroków czasowych, które chcemy wykonać (od 1 do 2000)
- podać długość kroku czasowego (w milisekundach)
- podać masę cząsteczek w kg
- podać pole przekroju poprzecznego cząsteczek
- podać szybkość początkową cząsteczek
- podać przyspieszenie grawitacyjne
- podać współczynnik tarcia cząsteczek z powietrzem
- podać prędkości wiatru wzdłuż współrzędnych x,y,z
- podać gęstość powietrza/płynu
- wskazać czy chcemy zapisać symulację jako film w formacie .avi.

Po wybraniu odpowiednich opcji wybieramy przycisk OK

Wczytanie trajektorii cząsteczek z pliku

Po wybraniu przycisku Wczytaj pojawi nam się następujące okno



Musimy wybrać obiekt, dla którego zostały wygenerowane trajektorie cząsteczek, które chcemy wczytać. Możemy również wybrać opcję zapisu symulacji w postaci filmu w formacie .avi. Po wybraniu przycisku OK zostaniemy poproszeni o wybranie pliku w rozszerzeniu .pkl.

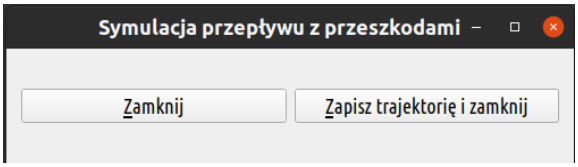
Animacja

Wówczas pojawi nam się okno programu PyVista z wybranym przez nas obiektem.

Za pomocą myszy możemy ustawić obiekt w taki sposób jak chcemy. Następnie wciskamy klawisz 'q' w celu rozpoczęcia animacji.

Jeśli wybraliśmy opcję zapisania animacji, pojawi nam się okno w którym możemy wybrać lokalizację oraz nazwę zapisywanego pliku. Domyślna lokalizacja do zapisywania to katalog MovieSample. **W przypadku nagrywania animacji nie można zmieniać wielkości okna animacji!**

Po zakończeniu animacji pojawi się okno, w którym będzie można wybrać dwie opcje



- Zamknij - zakończ działanie aplikacji
- Zapisz trajektorie i zamknij - zapisz trajektorie cząsteczek do pliku w formacie .pkl. Domyślna ścieżka do zapisu to katalog PickleSample)

4.2 Opis modułu

Do wykonania graficznego GUI zostały wykorzystane następujące biblioteki:

- **PyQt5**
- **pyvista**

Kod interfejsu graficznego znajduje się w katalogu *Interfejs* w skrypcie *aplikacja_okienkowa.py*

Każde okienko aplikacji jest osobną klasą. W skrypcie występują następujące klasy:

1. **BaseWindow** - podstawowa klasa, z której dziedziczą pozostałe.
   Metody:
   - **___init___** - konstruktor
   - **center** - metoda pozwalająca wyświetlać okno aplikacji w centrum aktywnie używanego ekranu
2. **StartWindow** - klasa odpowiadająca za okno startowe.
   Metody:
   - **___init___** - konstruktor
   - **interfejs(self)** - metoda tworząca interfejs okna
   - **switch_vizualize_window** - funkcja przekierowująca do okna z wyborem danych do wygenerowania symulacji
   - **switch_ChooseLoadObject_window** - funkcja przekierowująca do okna z wyborem gotowego pliku z wyliczonymi trajektoriami do wczytania
   - **switch_authors_window** - funkcja przekierowująca do okna z autorami
3. **AuthorsWindow** - klasa odpowiadająca za okno z autorami projektu.
   Metody:
   - **___init___** - konstruktor
   - **interfejs** - metoda tworząca interfejs okna
   - **switch_start_window** - funkcja przekierowująca z powrotem do okna startowego.
4. **ChooseLoadObjectWindow** - klasa odpowiadająca za okno do ładowania trajektorii z pliku.
   Metody:
   - **___init___** - konstruktor
   - **interfejs** - metoda tworząca interfejs okna
   - **Load** - metoda ładująca plik w formacie .pkl z trajektoriami cząsteczek do *numpy.ndarray*
   - **choose_object** - metoda odpowiedzialna za wybór obiektu, na którym wskazana symulacja była przeprowadzana
   - **btnstate** - funkcja pomocnicza do **choose_if_save**
   - **choose_if_save** - metoda odpowiedzialna za umożliwienie wyboru czy chcemy zapisać daną symulację w formacie .avi
   - **switch_result_window** - funkcja przekierowująca do okna z symulacją
5. **VizualizeWindow** - klasa odpowiadająca za okno do wpisania danych do wygenerowania symulacji.
   Metody:

- **___init___** - konstruktor
- **interfejs** - metoda tworząca interfejs okna
- **choose_object** - metoda odpowiedzialna za wybór objektu, na którym wskazana symulacja była przeprowadzana
- **choose_direction** - metoda odpowiedzialna za wybór kierunku w którym poruszają się cząsteczki
- **btnstate** - funkcja pomocnicza do **choose_if_save**
- **choose_if_save** - metoda odpowiedzialna za umożliwienie wyboru czy chcemy zapisać daną symulację w formacie .avi
- **set_QSpinBox** - metoda pomocnicza umożliwiająca wpisywanie przez użytkownika danych całkowitoliczbowych
- **set_QDoubleSpinBox** - metoda pomocnicza umożliwiająca wpisywanie przez użytkownika danych zmiennoprzecinkowych
- **Calculations** - metoda wywołująca algorytm tworzący trajektorię na podstawie danych wprowadzonych przez użytkownika.
- **switch_result_window** - funkcja przekierowująca do okna z symulacją. Podczas przekierowania uruchamiana jest funkcja **Calculations**.

6. **ResultWindow** - klasa odpowiadająca za okno z symulacją.
   Metody:

   - **___init___** - konstruktor
   - **interfejs** - metoda tworząca interfejs okna
   - **single_animation** - metoda odpowiedzialna za wykonanie animacji w jednym kroku czasowym
   - **animate** - metoda odpowiadająca za całkowitą animację. Wywoływana jest w niej metoda **single_animation**
   - **save** - metoda pozwalająca zapisać wygenerowaną trajektorię do pliku w formacie .pkl
   - **close_window** - funkcja zamykająca aplikację
   - **save_and_close_window** - funkcja wywołująca metodę **save** a następnie zamykająca aplikację