Projekt zaliczeniowy z Programowania 2019/2020

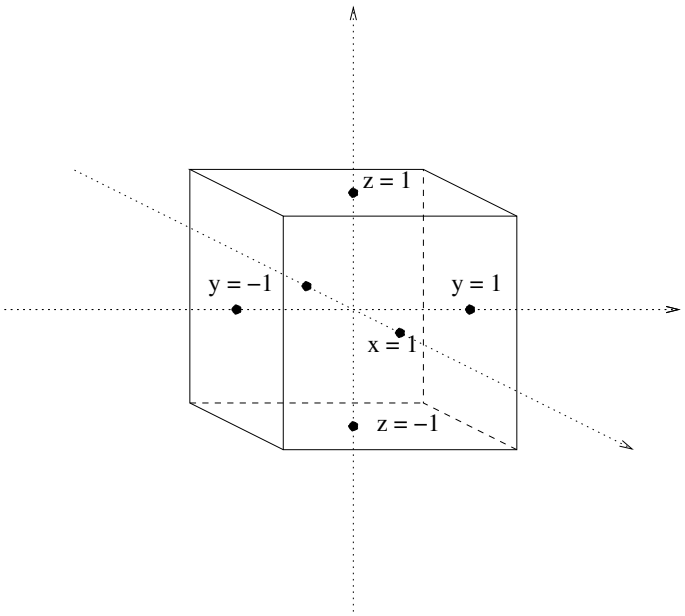
Termin nadsyłania rozwiązań - 10.06.2020

Program, który nie kompiluje się na komputerach pracownia.okwf.fuw.edu.pl (dostęp z komputera tempac.okwf.fuw.edu.pl) **nie podlega ocenie!**

Program powinien zostać napisany samodzielnie!

Prosimy o nadsyłanie rozwiązań do koordynatora przedmiotu w postaci archiwum o nazwie *Imie_Nazwisko.zip* lub *Imie_Nazwisko.tar.gz* lub *Imie_Nazwisko.rar*

Symulacja zderzeń kul w pojemniku



Rysunek 1: Pojemnik z zaznaczonym położeniem układu współrzędnych i wymiarami boków

Zagadnienie: W sześciennym pojemniku (długości boków jak na rysunku 1) znajduje sie n kul. Napisz program, który wykonuje symulację ruchu kul w pojemniku. Każda kula może mieć inną masę i promień, a kule w czasie ruchu mogą zderzać się ze sobą oraz ze ściankami naczynia. Zakładamy, że pomiędzy zderzeniami kule poruszają się ruchem jednostajnym, prostoliniowym.
Przy pomocy tej symulacji należy stworzyć animację pokazującą ruch kul w czasie.

Głównym elementem programu jest pętla po kolejnych chwilach czasu, w której wykonywane są następujące działania:

- wykonanie przemieszczenia każdej z kul zgodnie ze wzorem:

$$\vec{r_i}(t + \Delta t) = \vec{r_i}(t) + \vec{v_i}\Delta t$$

- sprawdzanie czy zaszło zderzenie z którąś ze ścianek naczynia (przykładowo warunek na zderzenie ze ścianką $x = 1$: $x + R \geqslant 1$, a ze ścianką $x = -1$: $x - R \leqslant -1$; ogólnie warunek kontaktu ze ścianką $\vec{r_i} + R_i\hat{n} = \pm 1$, $\hat{n}$ to jednostkowy wektor normalny, skierowany na zewnątrz danej ścianki)

- po zderzeniu z każdą ścianką zmiana prędkości kuli zgodnie z formułą:

$$\vec{v_i'} = \vec{v_i} - 2(\vec{v_i} \cdot \hat{n})(-\hat{n})$$

  (składowa prędkości prostopadła do ścianki zmienia po zderzeniu ze ścianką kierunek na przeciwny)

- sprawdzanie czy zaszło zderzenie kul ze sobą (zderzenie kul i i j zachodzi wtedy, gdy odległość między nimi jest mniejsza lub równa od sumy ich promieni):

$$|\vec{r_i} - \vec{r_j}| \leqslant R_i + R_j$$

- modyfikacja prędkości kul po zderzeniu:

$$\vec{v_i'} = \vec{v_i} + \vec{q}/m_i$$

$$\vec{v_j'} = \vec{v_j} - \vec{q}/m_j$$

  gdzie

$$\vec{q} = q\hat{r}_{ij}$$

  $\hat{r}_{ij}$ jest jednostkowym wektorem o kierunku łączącym środki dwóch kul:

$$\hat{r}_{ij} \equiv \frac{(\vec{r_i} - \vec{r_j})}{|\vec{r_i} - \vec{r_j}|}$$

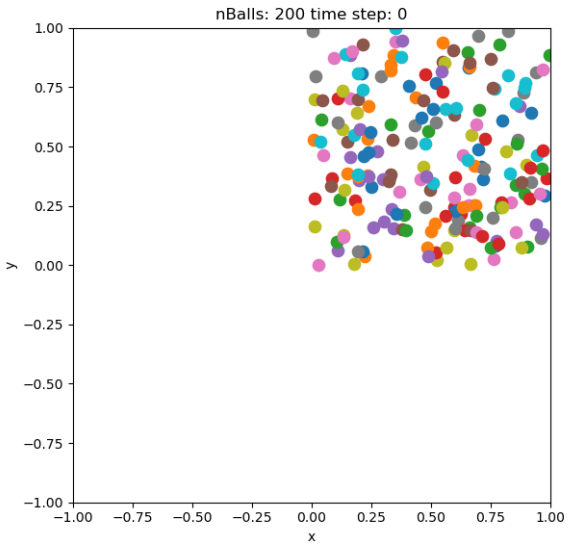
$$\vec{q} = -2\frac{m_i m_j}{m_i + m_j}[(\vec{v_i} - \vec{v_j}) \cdot \hat{r}_{ij}]\hat{r}_{ij}$$

Organizacja kodu: Cały program musi być umieszczony w katalogu Imie_Nazwisko. W katalogu musi być plik tekstowy o nazwie `README`, z zapisanym pełnym poleceniem do kompilacji i ewentualnymi wyjaśnieniami dotyczącymi uruchamiania i działania programu. Projekt musi być podzielony na klasy, a kod każdej klasy podzielony na deklarację w pliku nagłówkowym **Klasa.h**, oraz implementację w pliku **Klasa.cpp**, **Klasa.cc** lub **Klasa.C**. W pełnej wersji działanie programu musi być sterowane przez parametry wczytywane z linii poleceń:

- `-nSteps` - liczba kroków czasowych w symulacji
- `-dt` - długość kroku czasowego
- `-nBalls` - liczba kul w pojemniku
- `-input` *typ* - wybór sposobu wprowadzania parametrów kul. Możliwe wartości parametru *typ*:
  1. `random` - składowe położenia i prędkości kul są losowane z rozkładu płaskiego z zakresu [0,1] ($x_0, y_0, z_0, v_{x0}, v_{y0}, v_{z0} \in [0,1]$); wszystkie kule mają masę $m = 1$ i promień $r = 0.025$
  2. `test` - konfiguracja służąca do testowania poprawności działania programu. W pudle są umieszczane dwie kule o masach $m = 1$, promieniach $r = 0.025$ oraz składowych położenia i prędkości: $\vec{r}_1 = (0,0,0)$, $\vec{v}_1 = (1,0,0)$, $\vec{r}_2 = (0.5,0,0)$, $\vec{v}_2 = (0,0,0)$,
  3. `file` *nazwa pliku* – dane dotyczące kul wczytywane są z pliku
- `-output` *nazwa pliku* – nazwa pliku wyjściowego
- `-doFrames` - przyjmuje wartości `true` lub `false` i kontroluje zapisywanie klatek i tworzenie końcowej animacji.

Przykładowe uruchomienie programu:

```
./mySimulation -nSteps 1000 -dt 0.02 -input random -nBalls 200
```



Rysunek 2: Położenie kul w chwili początkowej. Parametry symulacji: $-input\ random - nBalls$ 200.

Efektem działania programu muszą być rysunki przedstawiające położenie kul w rzucie na płaszczyznę XY, w poszczególnych chwilach czasu. Rysunki muszą być zapisane w plikach w formacie `png` o nazwach **frame_ABCD.png**, np. frame_0001.png. Rysunek powinien wyglądać tak jak Rys. 2.

W każdej wersji oceniana będzie czytelność kodu źródłowego – między innymi jego odpowiednie sformatowanie (wcięcia).

Wymagania (na 50%):

W pudle umieszczone są dwie kule o masach $m = 1$, promieniach $r = 0.025$ oraz składowych położenia i prędkości: $\vec{r}_1 = (0, 0, 0)$, $\vec{v}_1 = (1, 0, 0)$, $\vec{r}_2 = (0.5, 0, 0)$, $\vec{v}_2 = (0, 5, 0)$. Powyższe wartości wpisane są na stałe do programu. Kule odbijają się tylko od ścian.
Program musi zawierać następujące klasy:

- `Ball` - klasa reprezentująca pojedynczą kulę. Wśród prywatnych danych klasy powinny być: masa kuli, promień, współrzędne środka, składowe prędkości.
  Klasa powinna mieć co najmniej jeden konstruktor, metody typu `set_` do ustawiania wartości danych składowych (masę, promień, położenie, prędkość) i metody typu `get_` do ich odczytywania. Powinna tez mieć metodę `print` do wypisywania aktualnej pozycji kuli.

- `Universe` - klasa opisująca pudło zawierające kule. Wśród prywatnych danych składowych powinna co najmniej zawierać pojemnik `vector<Ball>`, zmienną reprezentujacą krok czasowy `dt` oraz prywatne metody:

  – `moveBalls` – wykonującą przemieszczenie wszystkich kul w danym kroku czasowym;
  – `detectCollisionsWithWalls` – sprawdzającą dla wszystkich kul po kolei czy zaszło zderzenie z którąś ze ścianek naczynia;
  – `bounceFromWall` – modyfikujacą prędkość kuli po zderzeniu ze ścianką;

  Klasa powinna też zawierać co najmniej jeden konstruktor. Ponadto w klasie powinny być metody:

  – `evolution` – przyjmująca jako argument liczbę iteracji i wykonująca pętlę po iteracjach;
  – `round` – wykonująca wszystkie działania w ramach jednej iteracji: `moveBalls`, `detectCollisionsWithWalls` itp.

- `Plotter` - klasa wykonująca rysunki przy użyciu biblioteki `matplotlib`. Powinna mieć co najmniej następujące metody publiczne:
  - `plotBalls2D` – wykonującą rysunek położeń kul dla danego kroku
  - `makeAnimation` – łączacą wszystkie rysunki w animację (plik w formacie gif) poprzez wywołanie komendy systemowej `convert`:

```
convert -delay 0.01 -loop 0 frame*.png animation.gif
```

Animacja powinna wyglądać jak przykład umieszczony pod adresem: https://tinyurl.com/uq8q9n4.

Funkcja `main()` powinna być bardzo krótka (powinien być w niej powoływany do życia obiekt klasy `Universe` i uruchamiane funkcje na rzecz tego obiektu).

Wymagania na 75%:

W pudle umieszczana jest dowolna liczba kul. Kule mogą oddziaływać ze ścianami i ze sobą. Wektor z kulami może być wypełniany kulami o losowych początkowych wartościach położenia i prędkości lub (wersja testowa) w pudle umieszczone są dwie kule o masach $m = 1$, promieniach $r = 0.025$ oraz składowych położenia i prędkości: $\vec{r}_1 = (0, 0, 0)$, $\vec{v}_1 = (1, 0, 0)$, $\vec{r}_2 = (0.5, 0, 0)$, $\vec{v}_2 = (0, 0, 0)$. Program powinien pytać czy losujemy kule, czy uruchamiamy wersję testową.
Do wyżej opisanych wymagań dotyczących budowy klas dochodzą następujące:

- W klasie `Ball` wśród prywatnych danych klasy powinna być dodatkowo zmienna typu logicznego `hasCollided`, reprezentująca aktualny status: czy w danej iteracji kula już się zderzyła z inną kulą czy nie.
- W klasie `Universe` dodatkowe metody:
  - `detectCollisionswithBalls` – sprawdzająca po kolei czy zaszło zderzenie kolejnych kul ze sobą;
  - `bounceFromBall` – modyfikujaca prędkości kul po zderzeniu;
  - `resetCollideFlag` – po wykonaniu działań w jednym kroku czasowym, ustawiająca zmienne logiczne `hasCollided` na `false`;
  - `randomInput` – losująca początkowe położenia i prędkości kul z rozkładu płaskiego z zakresu [0,1] i wypełniająca wektor kul. Wszystkie kule mają masę $m = 1$ i promień $r = 0.025$;
  - `testInput` - umieszczajaca w wektorze kul tylko dwie kule o nastepujących parametrach: masach $m = 1$, promieniach $r = 0.025$ oraz położeniach i prędkościach: $\vec{r}_1 = (0, 0, 0)$, $\vec{v}_1 = (1, 0, 0)$, $\vec{r}_2 = (0.5, 0, 0)$, $\vec{v}_2 = (0, 0, 0)$.

Wymagania na 100%:

W pudle umieszczana jest dowolna liczba kul. Kule mogą oddziaływać ze ścianami i ze sobą. Do opisanych powyżej metod wypełniania wektora z kulami dochodzi możliwość wczytywania pozycji i początkowych prędkości kul z pliku. Działanie programu jest sterowane wyłącznie przez parametry wczytywane z linii poleceń. Dane z ostatniej iteracji mogą być zapisywane w pliku.

Do wyżej opisanych wymagań dotyczących budowy klas dochodzą następujące:

- W klasie `Ball` powinien być przeładowany (przeciążony) operator << do wypisywania aktualnej pozycji kuli.
- W klasie `Universe` dodatkowe metody:

– `readFromInputFile` – przyjmująca nazwę pliku z danymi wejściowymi i wczytująca dane do wektora kul. Metoda powinna zwracać status wczytywania np. w postaci zmiennej logicznej typu `bool`. W przypadku problemów z plikiem powinna wypisać informację na ekranie i zaproponować inną metodę nadania kulom początkowych wartości (wartości losowe, albo testowe jak opisane wyżej).
Każda linia pliku wejściowego musi zawierać następujące informacje o jednej kuli: masę, promień, współrzędne położenia i prędkości w chwili początkowej.
Plik z parametrami testowymi wygląda tak:

```
1.0 0.025 0.0 0.0 0.0 1.0 0.0 0.0
1.0 0.025 0.5 0.0 0.0 0.0 0.0 0.0
```

– `writeOutputFile` – przyjmująca nazwę pliku do zapisu i zapisująca dane z ostatniej iteracji w formacie opisanym dla pliku z danymi wejściowymi;

Wymagania na punkty powyżej 100%:

Dodatkowe punkty można uzyskać poprzez napisanie bezbłędnego programu spełniającego wszystkie powyższe kryteria i rozszerzającego powyższą specyfikację, poprzez np.:

- stworzenie klasy `Vector3D` reprezentującej trójwymiarowy wektor, umożliwiającej operacje matematyczne na wektorach i wypisywanie zawartości wektora na ekran za pomocą odpowiednich przeładowanych (przeciążonych) operatorów. Klasa powinna mieć także metodę pozwalającą na liczenie długości wektora. Klasa powinna dziedziczyć z klasy `vector<double>` i jedynie rozszerzać jej interfejs. W definicji klasy `Vector3D` nie powinno być żadnych pól danych.
- `złożoność obliczeniowa` - napisanie dodatkowego programu, który tworzy rysunek pozwalający na oszacowanie złożoności obliczeniowej symulacji jako funkcji liczby kul. Rysunek pokazuje czas działania symulacji w funkcji liczby kul. Plik README powinien zawierać analizę rysunku i informację o szacowanej złożoności obliczeniowej: $\mathcal{O}(???)$, gdzie zamiast ??? jest podana szacowana złożoność.
- `dowolne inne rozszerzenie` - pomysł musi być skonsultowany z osobą prowadzącą ćwiczenia.