МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из предмета

Програмирање и програмски језици

на тему

Рачунарске симулације на примеру симулације флуида

Ученик:

Даниил Грбић, 4д

Ментор:

Иван Танасијевић

Београд, мај 2020

Садржај

1. Увод

Симулација динамике флуида је једна од најпопуларнијих тема у области савремене компјутерске симулације, и користи се као при проучавању својстава разних флуида тако и при рендирању њихових анимација.

Нама је добро познато да се сви флуиди састоје од молекула, а познати су нам и закони по којима ти молекули интерагују. Тако долазимо до закључка да је један начин да се симулира флуид јесте заправо да симулирамо сваки молекул и сваку интеракцију тог молекула са осталим деловима флуида (другим молекулима).

Иако би овакав приступ дао најпрецизнији и најреалистичнији резултат, тај резултат не би смо добили у разумном времену – чак се и мале количине флуида састоје од огромне количине честица. Баш зато се често користе разне нумеричке апроксимације, поједностављења и претпоставке (најчешће основане на Навје-Стоксовим једначинама) које драстично смањују временску сложеност симулације, али при томе до неког степена задржавају њену поверљивост и реалистичност.

Циљ овог рада је да опишем један од приступа проблему симулације/анимације флуида у реалном времену и моју имплементацију тог приступа, са фокусом на техникама које ће ту симулацију учинити стабилном и брзом.

За израду симулације сам одлучио да користим програмски језик Python уз графичку библиотеку Pygame. Иницијално сам био забринут да ће мој избор имати осетљив негативан утицај на брзину симулације, али сам успео да тај утицај максимално смањим применом разних оптимизација које су ми оригинално и биле инспирација за избор ове теме.

2. Модел

Главни циљ овог рада није извођење једначина специфичних за симулацију флуида, већ креирање њене имплементације. Ипак, пре него што кренемо са тим, посветићемо мали део нашег времена на кратак преглед физичког модела који стоји иза једначина које ћемо користити у симулацији.

2.1 Умирена хидродинамика честица

Умирена хидродинамика честица (en. smoothed particle hydrodynamics, даље у тексту SPH) је интерполациона метода коју су 1977. независно развили Л. Б. Луси[1] и Гинголд и Монаган[2] у сврхе прављења астрофизичких симулација.

SPH омогућава рачунање интензитета поља полазећи само од позиције честице и информације о њеним суседима коришћењем симетричних умирујућих језгара. По дефиницији SPH, апроксимација A_S скаларне величине A на позицији $\boldsymbol{r}$ (векторске величине су у даљем тексту подебљане) се добија интерполацијом по тежинској суми вредности доприноса свих осталих честица.

$$A_S(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\boldsymbol{r} - \boldsymbol{r_j}, h) \tag{1}$$

где је m_j маса честице j, $\boldsymbol{r_j}$ њена позиција, ρ_j густина на тој позицији, а A_j вредност поља. Индекс j пролази кроз све честице.

Функција $W(\boldsymbol{r}, h)$ је такозвано умирујуће језгро где је h радијус – максимална раздаљина на којој честице делују једна на другу. Формално,

$$W(\boldsymbol{r}, h) = \begin{cases} f(\boldsymbol{r}), & 0 \le |\boldsymbol{r}| \le h \\ 0, & \text{у супротном} \end{cases} \tag{2}$$

где је $f(\boldsymbol{r})$ функција која зависи од избора језгра.

Заменом A у једначини (1) са ρ добијамо израз за процену густине ρ_i сваке честице:

$$\rho_S(\boldsymbol{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\boldsymbol{r} - \boldsymbol{r_j}, h) = \sum_j m_j W(\boldsymbol{r} - \boldsymbol{r_j}, h) \tag{3}$$

Код многих једначина флуида је потребно рачунати и изводе разних физичких величина. Код SPH рачунамо само изводе одговарајућих умирујућих језгара.

Градијент је вектор који има правац највећег пораста неке величине. Градијент величине A је

$$\nabla A_S(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\boldsymbol{r} - \boldsymbol{r_j}, h) \tag{4}$$

где ∇ (набла) означава векторски диференцијални оператор.

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right) \tag{5}$$

Лапласијан у ознаци ∇^2 представља дивергенцију градијента функције.

$$\nabla^2 A_S(\boldsymbol{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\boldsymbol{r} - \boldsymbol{r_j}, h) \tag{6}$$

$$\nabla^2 f = \left(\frac{\partial^2 f}{\partial x^2}, \frac{\partial^2 f}{\partial y^2}, \frac{\partial^2 f}{\partial z^2}\right) \tag{7}$$

SPH свакако није идеална метода и ствара веома значајне проблеме јер сама не може да гарантује симетрију сила, нити одржање импулса. Даље ћемо да причамо о моделу базираном на SPH и техникама које ћемо користити да решимо ове проблеме.

2.2 Моделирање флуида уз помоћ честица

Навје-Стоксове једначине описују кретање течности и изводе се из другог Њутновог закона примењеног на мале запремине флуида. Једначина одржања импулса гласи:

$$\rho\left(\frac{D\boldsymbol{v}}{Dt}\right) = -\nabla p + \rho \boldsymbol{g} + \eta \nabla^2 \boldsymbol{v} \tag{8}$$

Са десне стране ове једначине се налазе три сабирка који одговарају силама које делују на честице: притиску, гравитацији и вискозности. Њихов збир $f = -\nabla p + \rho g + \eta \nabla^2 v$ одређује промену импулса $\rho\left(\frac{Dv}{Dt}\right)$, где је $\frac{Dv}{Dt} = \frac{\partial v}{\partial t} + v \cdot \nabla v$ такозвани материјални извод који представља убрзање инфинитезималне честице флуида. Убрзање такве честице i у SPH је једнако

$$\boldsymbol{a_i} = \frac{d\boldsymbol{v_i}}{dt} = \frac{\boldsymbol{f_i}}{\rho_i} \tag{9}$$

2.3 Притисак

Применом једначине (1) на компоненту $-\nabla p$ добијамо израз за промену импулса честице услед деловања притиска:

$$\boldsymbol{f_i^{\text{притисак}}} = -\nabla p(\boldsymbol{r_i}) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{10}$$

Ова сила није симетрична – две честице делују једна на другу у различитој мери, супротно другом Њутновом закону. На нашу срећу, овај проблем има једно веома једноставно решење: p_i и p_j можемо да заменимо са аритметичком средином $\frac{p_i+p_j}{2}$.

$$\boldsymbol{f_i^{\text{притисак}}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{11}$$

Пошто су у нашем систему честице описане са три вредности – масом, позицијом и брзином – прва ствар коју морамо да урадимо да би одредили силу притиска јесте да израчунамо притисак у околини сваке од честица. То можемо да урадимо у два корака. Прво, израз (3) нам даје густину у околини сваке честице. Затим притисак рачунамо користећи се једначином идеалног гаса:

$$p = \frac{nRT}{V} = \frac{\frac{m}{M}RT}{V} = \left(\frac{m}{M}\right)\left(\frac{RT}{V}\right) = \rho k \tag{12}$$

где је k гасна константа зависна од температуре.

У симулацији ћемо користити модификовану верзију једначине (10) коју је свом раду предложио Десбрун[3].

$$p = (\rho - \rho_0)k \tag{13}$$

где је ρ_0 густина мировања.

2.4 Вискозност

Као што смо урадили и за притисак, применом једначине (1) на компоненту $\eta\nabla^2 v$ добијамо израз за промену импулса честице услед силе међусобног трења (вискозности):

$$\boldsymbol{f_i^{\text{вискозност}}} = \eta\nabla^2 v(\boldsymbol{r_i}) = \eta \sum_j m_j \frac{v_j}{\rho_j} \nabla^2 W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{14}$$

Пошто ова вредност зависи од разлике брзина честица, једначину (14) симетризујемо тако што v_i заменимо са разликом $v_j - v_i$:

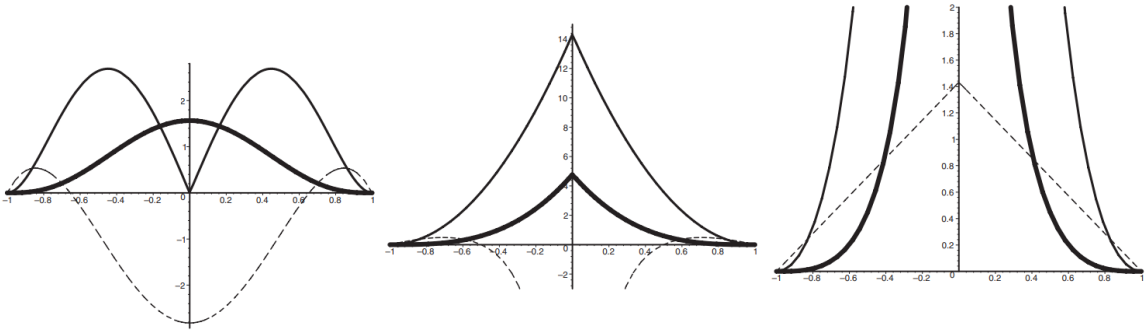
$$\boldsymbol{f}_i^{\text{вискозност}} = \eta \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{15}$$

2.5 Спољашње силе

Спољашње силе су у нашем случају силе које настају услед деловања гравитације или кретања целокупног система. Њих представљамо једним збирним вектором и тако настала сила делује директно на сваку честицу без употребе SPH.

2.6 Умирујућа језгра, језгарне функције

Умирујуће језгро (en. smoothing kernel) је тежинска функција у ознаци W која скалира интензитете међусобних интеракција честица. Језгарне функције одређују интензитет интеракције две посматране честице у зависности од њиховог међусобног растојања.



Слика 1: Милерова језгра: poly6, spiky, viscosity (с лева на десно). Дебље линије представљају сама језгра, танке – интензитете градијената, а шрафиране – интензитете Лапласијана.

Од избора језгара зависи стабилност, прецизност и брзина SPH методе. М. Милер[4] је предложио три језгра за моделирање кретања флуида, уз то да су те изабране функције нормализоване и парне, то јест задовољавају следећа својства:

$$\int W(\boldsymbol{r}, h) d\boldsymbol{r} = 1 \tag{16}$$

$$W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h) \tag{17}$$

Као основно језгро Милер је предложио следећу функцију:

$$W_{poly6}(\boldsymbol{r},h)=\frac{315}{64\pi h^9}\begin{cases}(h^2-r^2)^3, & 0\le r\le h\\ 0, & \text{у супротном}\end{cases} \tag{18}$$

Постоје два случаја у којима *poly6* не даје задовољавајући резултат. Заиста, ако га користимо за израчунавање сила притиска, честице теже да се под великом притиску сакупљају у густе кластере (групе). Не помаже ни то да се смањењем растојања смањује и градијент, чиме нестаје сила одбијања, као што се види на левом графику на слици 1.

Десбрун предлаже да користимо такозвано spiky језгро (*spiky* означава бодљасто, а функција је назив добила због свог специфичног облика).

$$W_{spiky}(\boldsymbol{r},h)=\frac{15}{\pi h^6}\begin{cases}(h-r)^3, & 0\le r\le h\\ 0, & \text{у супротном}\end{cases} \tag{19}$$

$$\nabla W_{spiky}(\boldsymbol{r},h)=-\frac{45}{\pi h^6}\begin{cases}(h-r)^2, & 0\le r\le h\\ 0, & \text{у супротном}\end{cases} \tag{20}$$

Spiky језгро гарантује одговарајуће одбојне силе независно од удаљености честица. Поред тога, на растојању h од центра *spiky* језгра први и други извод су једнаки 0.

Вискозна сила делује услед трења између честица. Зато она у сваком тренутку смањује кинетичку енергију честица, при томе одајући топлоту. То својство вискозних сила није задовољено ако користимо стандардно (*poly6*) језгро. Наиме, како се смањује растојање између честица, Лапласов оператор постаје негативан те брзине честица расту. То крши законе физике и нарушава стабилност симулације (честице под великом брзином излећу из флуида и долази до неконтролисаног случајног кретања). Зато је Милер у сврхе рачунања вискозних сила направио ново језгро чији је Лапласов оператор дат једначином

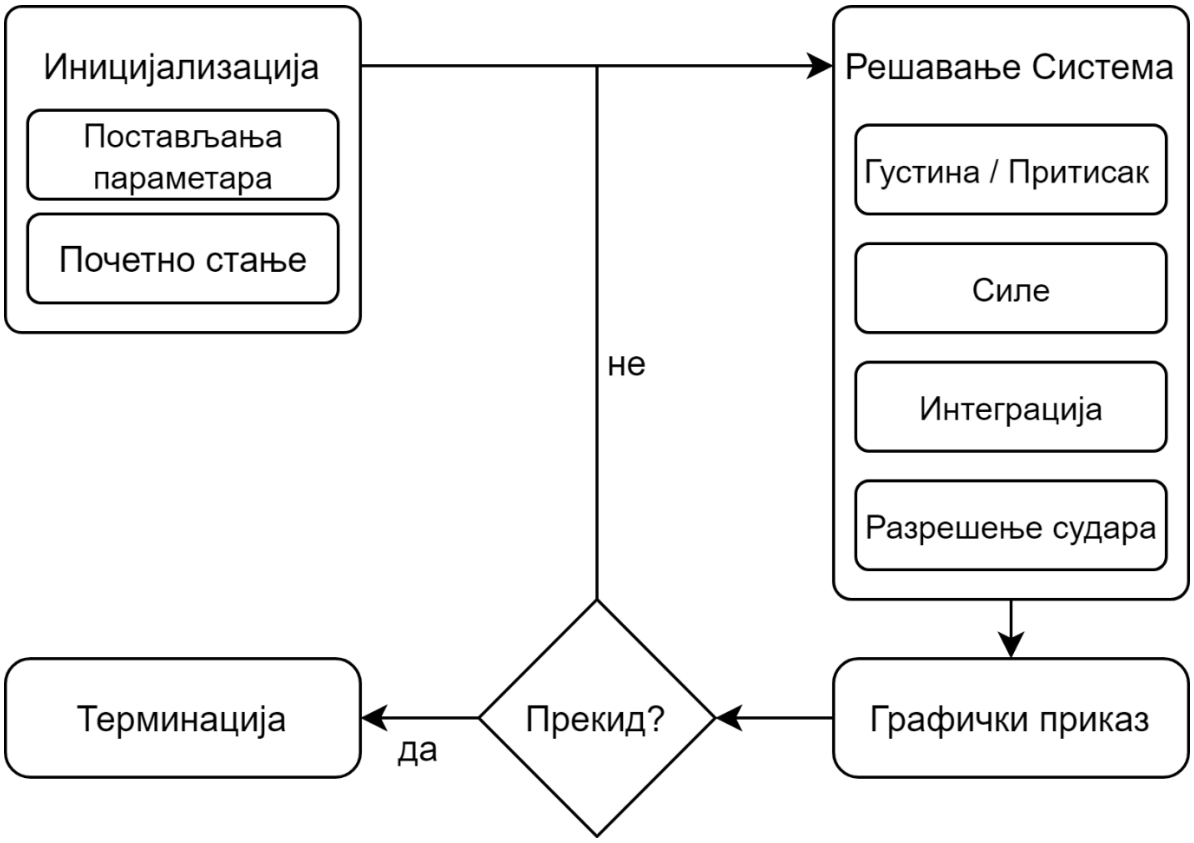
$$\nabla^2 W_{viscosity}(\boldsymbol{r},h)=\frac{45}{\pi h^6}\begin{cases}(h-r), & 0\le r\le h\\ 0, & \text{у супротном}\end{cases} \tag{21}$$

Лапласов оператор $W_{viscosity}$ је истог знака за свако r, што није случај код друга два језгра.

3. Ток Симулације

Симулација ће се састојати од више целина, а главне су иницијализација, систем за решавање и функције за графички приказ резултата. При томе, свака од ових целина се састоји од више делова, о којима ће више бити речено у наредним поглављима.

Ради прегледности сам пре писања програма одлучио да направим и поједностављен дијаграм операција које би извршавала симулација, у редоследу у којем се извршавају (слика 2).



Слика 2: Дијаграм тока симулације.

4. Имплементација

Главни део програма је *run.py* модул који повезује све остале делове програма и у коме се дефинишу сви параметри симулације, као и њено почетно стање.

4.1 Укључивање потребних модула

Најпре, као и у сваком програму, укључујемо остале модуле и спољашње зависности. Модул *graphics.py* садржи све функције везане за графички приказ и управљање симулацијом; модули *force.py*, *pressure.py* и *integrate.py* садрже функције потребне за решавање постављених једначина; а модул *grid.py* садржи класу Grid која омогућава брзо проналажење суседних честица (више о томе у секцији 4.10). Поред тога, укључујем и спољашње библиотеке *NumPy* (за брзу обраду података) и *time* (за процену брзине симулације).

```
from graphics import *
from force import compute_force
from pressure import compute_density_pressure
from integrate import do_step
from grid import Grid

# спољашње библиотеке
import numpy as np
from time import time
```

4.2 Иницијализација параметара

Следећи корак у симулацији је иницијализација параметара.

Крећем од основних параметара симулације, а то су број честица *N*, димензије прозора *width* и *height* , и границе симулације (*boundary_x* и *boundary_y*).

```
N = 1000  # број честица
width, height = 350, 350 # димензије PyGame прозора
boundary_x,boundary_y = [40,width-40],[40,height-40] # границе симулације
```

Након тога следи иницијализација параметара који описују физичка својства честица, као и оних који описују изглед честица у симулацији.

```
# параметри честица
mass = 1  # маса честице
k = 40  # Десбрунов коефицијент ( p=k(ρ-ρ0) )
rest_density = 1  # густина мировања ρ0
h = 5  # радијус језгара
visc = 0.15  # коефицијент вискозности
f_external = np.array([0, -0.1])  # вектор спољашње силе
delta = 0.03  # временски корак
damping = -0.5  # више у подеоку 4.9

# изглед симулације (позадина; боја и димензије честица)
background_color, border_color= (255,255,255), (200,200,200)
particle_radius, particle_width = 1, 1
particle_color = (0,102,255)
```

Овакав избор вредности параметара наравно није једини, а скоро сигурно није ни идеалан. Те вредности сам бирао тако да добијем најреалистичнији резултат, односно д добијем флуид са својствима најсличнијим својствима воде; и то тек након писања остатка програма.

4.3 Иницијализација потребних структура

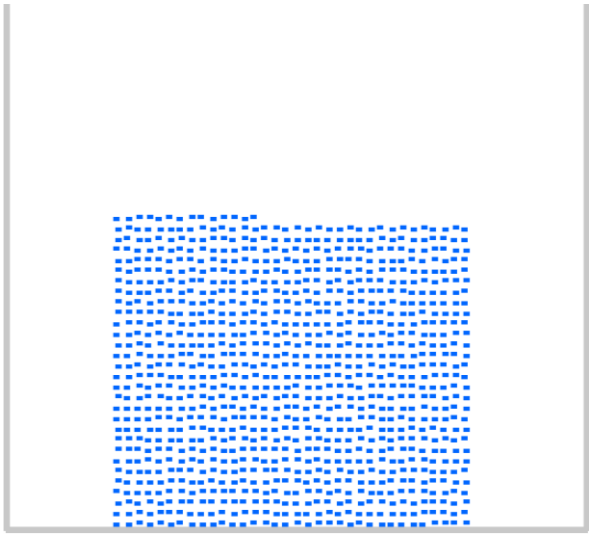
Последња група промењљивих које иницијализујемо су *NumPy* структуре за чување тренутног стања симулације и просторни хеш – *Grid*.

```
# NumPy низови
coordinates = np.zeros((N, 2)) # позиције
vel = np.zeros((N, 2))  # брзине
force = np.zeros((N, 2))  # резултујуће силе
density = np.zeros(N)  # густине у околини честица
pressure = np.zeros(N) # притисак на честице

# просторни хеш
hashmap = Grid(width, height, h)
```

4.4 Иницијализација почетног стања

Након тога што смо декларисали све потребне промењљиве, можемо приступити иницијализацији почетног стања симулације. Изабрао сам сценарио који лепо приказује могућности програма – такозвани *dam-break* експеримент: запремина флуида је са једне или више страна ограничена зидовима бране и та брана се проломи (посматрамо кретање флуида након пролома). Честице постављам на целобројну решетку са малим пертурбацијама које су независне величине са просеком 0 и девијацијом 0.25.

```
# иницијализација dam-break експеримента
def setup_sim():
    i = 0
    for y in range(boundary_y[0], boundary_y[1], h):
        for x in range(boundary_x[0] + 80, boundary_x[1]-80, h):
            if i < N:
                coordinates[i] = x + np.random.normal(0, 0.25),
                                 y + np.random.normal(0, 0.25)
                hashmap.move(i, coordinates[i])
            i += 1
```



Слика 3: *dam-break* експеримент

4.5 Главна петља програма и функција *next_frame()*

Последња ствар коју морамо да иницијализујемо је *Pygame* прозор у коме ћемо и приказивати симулацију. Након тога долази петља која чини костур целог програма: из ње се позива функција *next_frame()* и она омогућава интеракцију корисника са програмом. Ова функција обједињује позив осталих функција које врше израчунавања (део дијаграма тока симулације приказан на слици 4), а њена сврха је искључиво побољшање читљивости кода.

Слика 4: Дијаграм који описује структуру функције *next_frame()*

```
paused = False
while True:
    event = process_event()
    if event == 'PAUSE':
        paused = not paused
        print("Paused:", paused)
    elif event == 'QUIT':
        print("Process terminated")
        break
    if paused:
        continue
    screen_clear(screen, background_color)
    next_frame() # му се одвија рачун
    border_draw(screen, boundary_x, boundary_y,
                width, height, border_color)
    for x, y in coordinates:
        screen_draw(screen, particle_color,
                    int(x), int(350-y),
                    particle_radius, particle_width)
    screen_refresh(screen)


# један 'фрејм' или корак
def next_frame():
    global pressure, density, force
    pressure, density = compute_density_pressure(coordinates, pressure,
             density, h, k, rest_density, hashmap)
    force = compute_force(coordinates, vel, pressure, density, h, mass,
             visc, f_external, force, hashmap)
    do_step(coordinates, vel, force, density, delta, boundary_x,
             boundary_y, damping, hashmap)
```

4.6 Рачунање густине и притиска

Функција *compute_density_pressure()* је задужена за рачунање притиска. Подсетимо се једначина (9), (11) и (16).

$$\boldsymbol{f_i}^{\text{притисак}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{11}$$

$$p = (\rho - \rho_0)k \tag{13}$$

$$W_{poly6}(\boldsymbol{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3, & 0 \le r \le h \\ 0, & \text{у супротном} \end{cases} \tag{18}$$

Приметимо да су горе издвојени плавом бојом делови *poly6* језгра исти без обзира на то које две честице интерагују. Зато у сврхе оптимизације уводимо константе *h_2* и *kernel_poly6* како би ове делове језгра рачунали само једном (множење и дељење су процесорски захтевне операције).

```
from math import pi

def compute_density_pressure(xy,pressure,density,h,k,rest_density,grid):
    """
    рачунамо делове језгра
    """
    h_2 = h ** 2
    kernel_poly6 = 315 / (64 * pi * (h ** 9))

    for i in range(xy.shape[0]):
        density[i] = 0
        for j in grid.neighbours(xy[i]):
            r = xy[j] - xy[i]
            r_2 = r.dot(r)

            if r_2 <= h_2:
                density[i] += kernel_poly6 * (h_2 - r_2) ** 3

        pressure[i] = k * (density[i] - rest_density)

    return pressure, density
```

4.7 Рачунање резултујућих сила

Исто као и код рачунања притиска, делове *spiky* и *viscosity* језгара можемо израчунати једном непосредно пре него што кренемо са рачунањем сила чиме значајно смањујемо број редундантних израчунавања.

```
import numpy as np
from math import sqrt, pi

def compute_force(xy,vel,press,density,h,mass,visc,f_external,force,grid):
    """
    рачунамо силе притиска и вискозне силе
    """
    h_6 = h ** 6
    kernel_spiky = mass * -45 / (pi * h_6)
    kernel_visc = visc * mass * 45 / (pi * h_6)

    for i in range(xy.shape[0]):
        f_pres = np.array([0, 0], dtype=float)
        f_visc = np.array([0, 0], dtype=float)

        for j in grid.neighbours(xy[i]):
            if i == j:
                continue

            r = xy[j] - xy[i]
            n = sqrt(r.dot(r))

            if 0 < n < h:
                f_pres += -r / n * (press[i] + press[j]) /
                          (2 * density[j]) * kernel_spiky * ((h - n) ** 2)
                f_visc += visc * mass * (vel[j] - vel[i]) /
                          density[j] * kernel_visc * (h - n)

        force[i] = f_pres + f_visc + f_external

    return force
```

4.8 Ојлерова интеграција

У тренутку када су нам познате резултујуће силе на сваку честицу приступамо рачунању промена брзина, односно рачунању нових позиција по једначини линераног кретања.

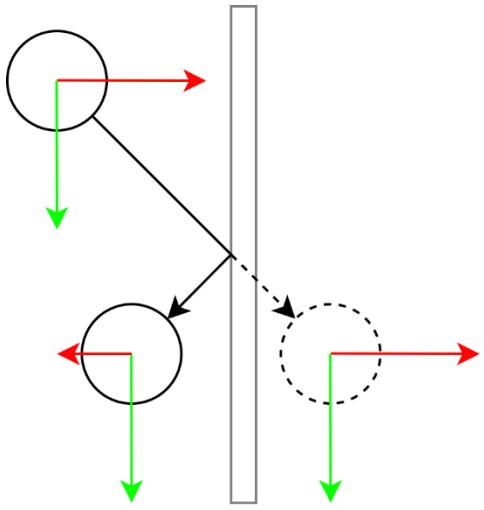
```
from boundary import apply_bound

def do_step(coordinates,vel,force,density,dt,xbound,ybound,damp,grid):
    """
    интеграција
    """
    for i in range(coordinates.shape[0]):
        vel[i] += dt * force[i] / density[i]
        coordinates[i] += dt * vel[i]
        apply_bound(coordinates, vel, xbound, ybound, damp, i)
        grid.move(i, coordinates[i])
```

До проблема долази када се након померања честица нађе изван предела симулације. У таквом случају честицу враћамо унутар граница и смањујемо њену кинетичку брзину (део енергије је дала зиду током судара). Ово ради функција *apply_bound()*.

4.9 Разрешење судара

Кретање сваке честице је одређено хоризонталном и вертикалном компонентом брзине. Пошто су границе симулације паралелне са координатним осама, када дође до судара честице са зидом, једна од компоненти брзина је нормална а друга је паралелна том зиду.

Паралелна компонента остаје иста, а нормалну множимо константом *damp*, таквом да важи *-1<damp<0*. На овај начин честица се одбија од зидова и у процесу губи део кинетичке енергије.



Слика 5: Одбијање честице од зида

```
def apply_bound(xy, vel, xbound, ybound, damp, i):

    vel0 = vel[i]
    if xy[i, 0] < xbound[0]:
        vel[i, 0] *= damp
        xy[i, 0] = xbound[0]
    elif xy[i, 0] > xbound[1]:
        vel[i, 0] *= damp
        xy[i, 0] = xbound[1]

    if xy[i, 1] < ybound[0]:
        vel[i, 1] *= damp
        xy[i, 1] = ybound[0]
    elif xy[i, 1] > ybound[1]:
        vel[i, 1] *= damp
        xy[i, 1] = ybound[1]
```

4.10 Просторни хеш *Grid*

Као што је већ пре било поменуто, интеракције удаљених честица се могу занемарити, те нам је потребан ефикасан начин да пронађемо суседе било које изабране честице.

Ово можемо постићи коришћењем просторног хеша – матрице подељене на квадрате странице h у којима се налазе честице – такве да можемо максимално брзо приступити свим честицама у неком пољу, као и ефикасно померати честице из једног поља у друго.

Класе *Grid* и *Node* сам написао тако да пружа баш ову функционалност. Свака честица је инстанца класе *Node* и чува се у сету *Grid.nodes*. То омогућава приступ било којој честици у логаритамској сложености. Такође, свака инстанца класе *Node* садржи референце на прошлу и следећу честице.

Grid.cells је матрица у којој свако поље држи референцу на неку од честица у том пољу, па тако можемо да добијемо и списак свих осталих честица у том пољу у линеарној сложености (инстанце класе *Node* чине повезану листу).

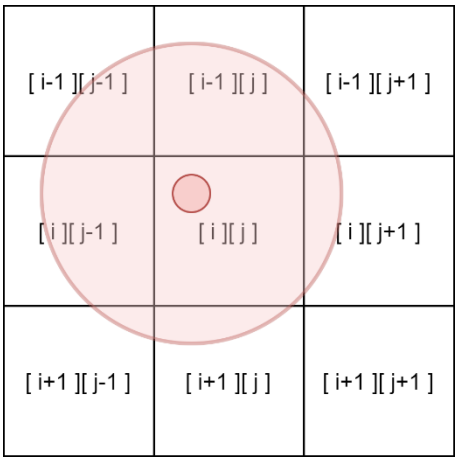
```
import math
import itertools
class Grid:
    def __init__(self, width, height, cellsize):
        self.width = width
        self.height = height
        self.cellsize = cellsize
        self.cells = [[Node(-1)
                       for _ in range(math.ceil(height / cellsize))]
                       for _ in range(math.ceil(width / cellsize))]
        self.xcells = len(self.cells)
        self.ycells = len(self.cells[0])
        self.nodes = {}  # сет инстанца класе Node

    def coords_to_index(self, coords):
        return int(coords[0]//self.cellsize),int(coords[1]//self.cellsize)

    def neighbours(self, coords):
        location = self.coords_to_index(coords)
        for dx, dy in itertools.product(range(-1, 2), range(-1, 2)):
            node = self.cells[location[0] + dx][location[1] + dy].next
            while node is not None:
                yield node.id
                node = node.next

    def move(self, id, new_coords):
        node = self.nodes.get(id)
        if node is None:
            node = Node(id)
            self.nodes[id] = node
        location = self.coords_to_index(new_coords)
        node.attach(self.cells[location[0]][location[1]])
```

Како таква структура омогућује максимално ефикасан приступ свим честицама у *h*-околини посматране честице? Димензије поља у матрици су нам познате, те ако знамо координате честице можемо директно да одредимо у ком пољу се он налази.

```
i,j = int(coords[0]//self.cellsize),int(coords[1]//self.cellsize)
```

Пошто су поља истих димензија као и радијус језгра (*h*), знамо да се све суседне честице налазе баш у 8 суседних поља (као и у сопственом пољу шестице), а приступамо им тако што прођемо одговарајуће повезане листе.

Ова оптимизација убрзава симулацију број пута једнак количнику укупног броја честица са 9, а за изабране параметре тај количник је 324!

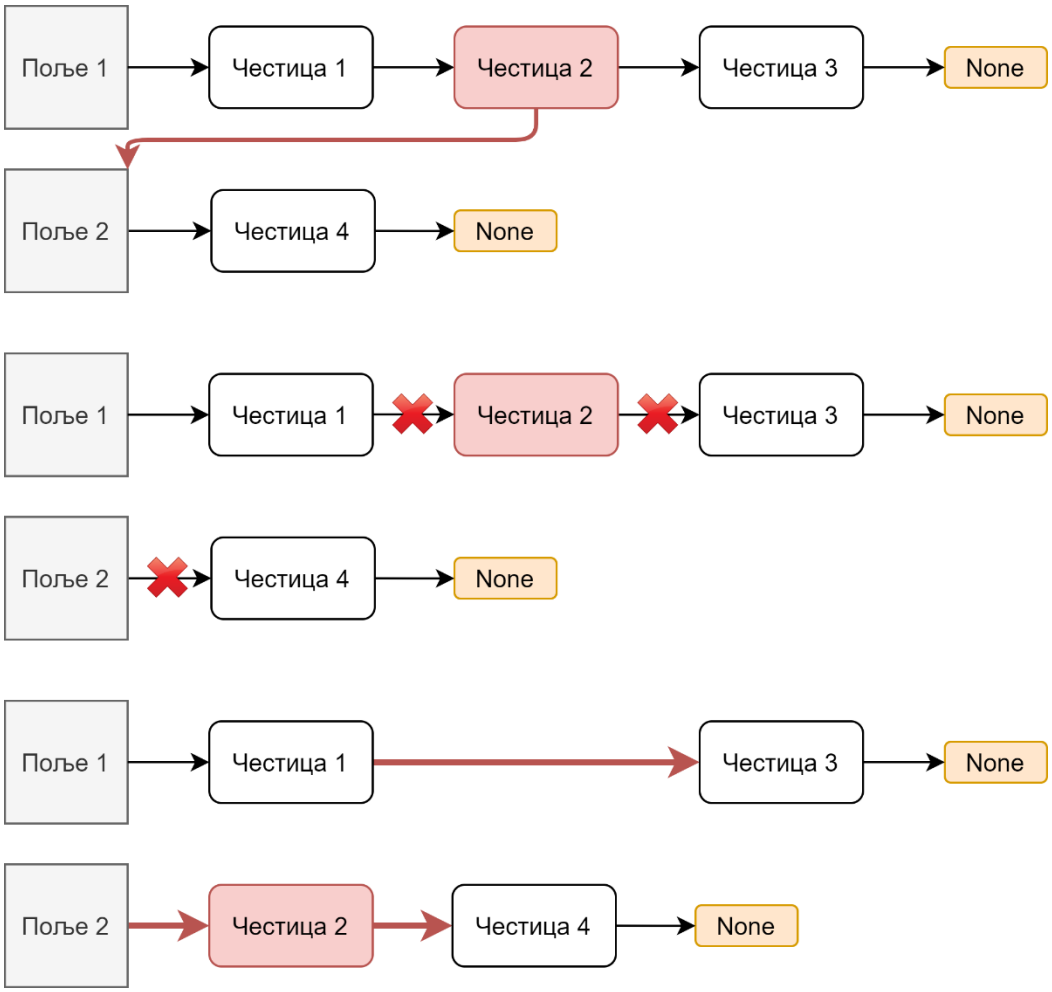


Слика 6: Суседи честице се налазе у суседних 8 поља

Остаје само да решимо проблем ефикасног пребацивања честица из једног поља у друго. Раније описана имплементација повезаних листи омогућава баш то. Објаснићемо како такво пребацивање на конкретном примеру (слика 7).

Свака честица у симулацији има додељени идентификациони број – *id*. Нека је *id* посматрана честице 2, а налази се у пољу број 1. У повезаној листи њени суседи су честице са идентификационим бројевима 1 и 3. Након интеграције и рачунања новог положаја, та честица напушта поље 1 и прелази у поље 2.

У првом кораку раскидамо везе између посматране честице и њених суседа, и између прве честице повезане на поље 2 и тог поља (честице 4). Након тога стварамо нове везе између њених суседа и између ње и поља 2, као и ње и честице 4.



Слика 7: Процес померања честице из једног поља у друго

Наравно постоје и неки гранични случајеви, на пример честица коју померамо је прва или последња у повезаној листи. Због начина на смо који имплементирали ову функционалност за програм не постоји разлика између суседних честица, односно поља матрице.

Једина ствар о којој морамо да водимо рачуна јесте да на крају сваке повезане листе у референцу на следећи члан ставимо *None*, јер ћемо тако знати када и где се завршава свака листа.

```
class Node:
    def __init__(self, id):
        self.id = id
        self.prev = None
        self.next = None

    def attach(self, start):
        if self.prev is not None:
            self.prev.next = self.next
            self.prev = None
        if self.next is not None:
            self.next.prev = self.prev
            self.next = None

        self.next = start.next
        if self.next:
            self.next.prev = self

        self.prev = start
        start.next = self
```

4.11 Графика и управљање

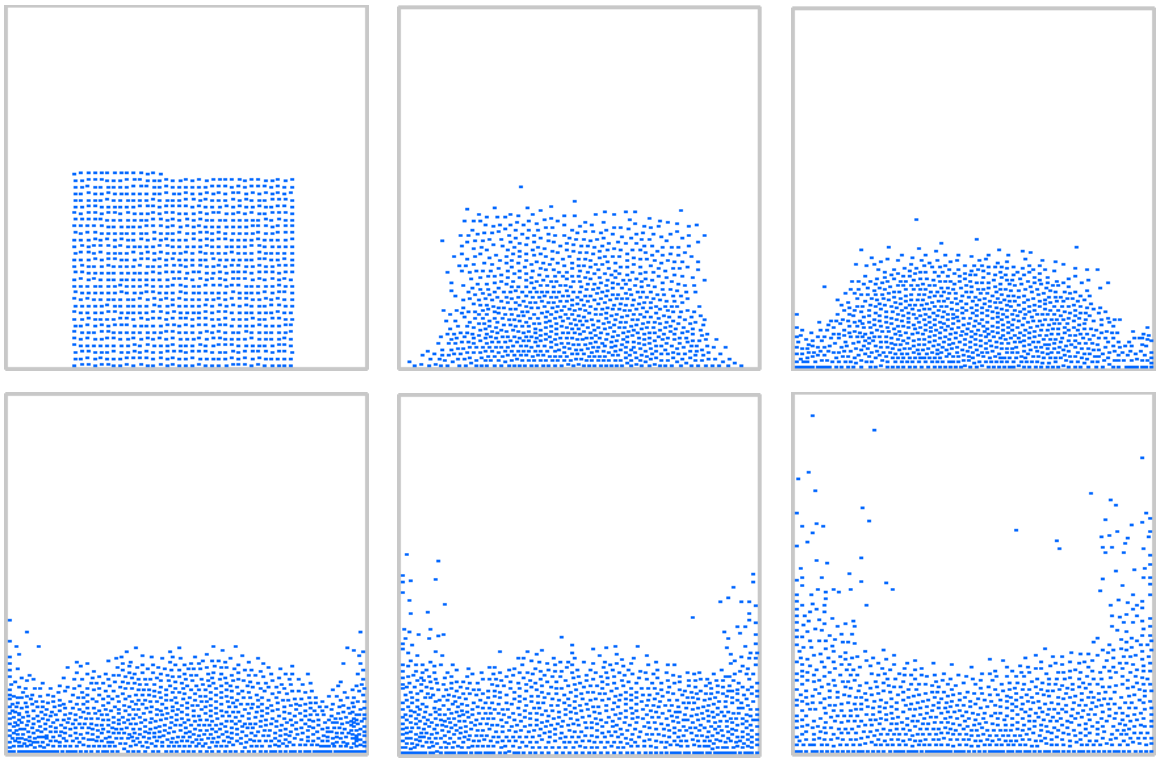
```
# иницијализација PyGame екрана са задатим параметрима
def screen_init(width, height, background_color) -> pygame.display:
    screen = pygame.display.set_mode((width,height))
    pygame.display.set_caption('SPH Fluid Sim')
    screen.fill(background_color)
    return screen

# ф-ја која чисти екран
def screen_clear(screen, background_color):
    screen.fill(background_color)

# ф-ја која "рефрешује" екран, односно исцртава најновију слику из бафера
def screen_refresh(screen):
    pygame.display.flip()

# ф-ја која смештава честицу са задатим координатама у display бафер
# чести се прикаже на екрану тек након позивања ф-је screen_refresh()
def screen_draw(screen, color, x, y, radius, width):
    pygame.draw.circle(screen, color, (x,y), radius, width)

# ф-ја која обрађује корисничке команде (event handler)
def process_event() -> str:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.display.quit()
            return 'QUIT'
        elif event.type == pygame.KEYDOWN and event.key == pygame.K_p:
            return 'PAUSE'
    return None
```

5. Резултати

Покретањем симулације видимо да добијени флуид заиста подсећа на праве флуиде, а на слици 8 су приказани 6 стања за временским разликама од око 15 секунди (времена извршавања).

Додавањем исписа времена на почетку *next_frame()* функције можемо проценити брзину симулације: она варира од око 2.6 до 3.3 корака или фрејма по секунди (на енглеском *frames per second*).

Повећањем временског корака *dt* можемо да убрзамо симулација на рачун мање тачности, али не у смислу повећаног броја корака у секунди већ у смислу привидне брзине симулације (то је мање битно код научних симулација, али је потребно кад се ради о анимацијама у реалном времену).



Слика 8: Резултати рада програма

6. Закључак

Оваква имплементација је далека од прецизне – физички модел је веома поједностављен, а неке битне силе (површински напон) уопште нису присутне – а није најбоље оптимизована. Без обзира на то, овај програм представља добру основу за даље развијање и приказује више техника које се користе не само у симулацијама, већ и у другим областима рачунарства.

Методе описане у овом раду се могу примењивати на различите флуиде и сличне симулације се широко користе при проучавању својстава неких флуида, па чак и при конструкцији бродова. Штовише, дељење физичких (а понекад и социјалних) система на честице даје велике могућности. При симулацији флуида честице се слободно крећу и реагују. Ако честице повежемо еластичним опругама, можемо да симулирамо еластична тела. SPH метода се чак може користити и у сложеним социјалним симулацијама (симулацијама понашања група људи)!

У скорој будућности планирам да наставим да побољшавам овај програм: и то да имплементирам површински напон, да га још више оптимизујем, да улепшам приказ и да додам више опција за корисничку контролу у реалном времену. Оптимизација би подразумевала паралелизацију процеса (истовремено можемо да обрађујемо удаљене честице чије је међусобно деловање занемарљиво) и извршавање захтевнијих израчунавања на графичкој картици. Можда ћу чак преписати читав код у *C++* и прећи у трећу димензију, а постоје и варијанте да се симулација преради и за друге врсте флуида: вискозне флуиде и гасове.

7. Референце

[1] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82:1013–1024, 1977.

[2] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375– 398, 1977.

[3] M. Desbrun and M. P. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. Proceedings of EG Workshop on Animation and Simulation, Springer-Verlag, 1996.

[4] M. Müller, D. Charypar and M. Gross. Particle-Based Fluid Simulation for Interactive Applications. Department of Computer Science, Federal Institute of Technology Zürich (ETHZ), Switzerland, 2003

[5] L. Schuermann and C. Martin. Implementing SPH in 2D, https://bigtheta.io/2017/07/08/implementing-sph-in-2d.html, 2017