Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

МОДЕЛЮВАННЯ ФІЗИКИ РІДИНИ ЗА ДОПОМОГОЮ ПАРАЛЕЛЬНИХ ОБЧИСЛЕННЬ

Текстова частина до курсової роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи

Доктор фіз.-мат. наук,

професор

_____

(прізвище та ініціали)

_____

(підпис)

“____” _____ 2023 р.

Виконав студент _____

_____

(прізвище та ініціали)

“____” _____ 2023 р.

Київ 2023

Зміст

Анотація …… 4

Вступ …… 5

Основна частина …… 6

1 Фізика рідини …… 6

1.1 Базові фізичні властивості рідини …… 6

1.2 Методи дискретизації об’єму рідини …… 7

1.3 Гідродинаміка згладжених частинок …… 8

1.3.1 Обчислення густини …… 9

1.3.2 Обчислення тиску …… 10

1.3.3 Обчислення сил …… 11

1.3.4 Обчислення прискорення, швидкості і позиції ……13

2. Модель …… 14

2.1 Поняття радіусу частинки і сусідніх частинок …… 14

2.2 Структура даних …… 15

2.2.1 Специфікація …… 15

2.2.2 Імплементація …… 16

2.3 Алгоритм фізичних обчислень …… 20

2.3.1 Технологія Nvidia CUDA …… 20

2.3.1.1 Пам'ять відеокарти …… 20

2.3.1.2 Шаблон роботи з Nvidia CUDA …… 21

2.3.1.3 Специфіка роботи з пам’яттю в Nvidia CUDA 21

2.3.2 Етапи алгоритму фізичних обчислень 22

2.3.3 Оптимізації 27

2.3.3.1 Мінімізація кількості звернень до глобальної пам’яті 27

2.3.3.2 Мінімізація об’єму пам’яті 27

3 Візуалізація 29

3.1 Основні положення 29

3.2 Алгоритм трасування променів 29

3.3 Функція перетину 30

3.3.1 Імплементація на основі густини 30

3.3.2 Оптимізації 31

3.3.2.1 Уникнення обрахунків зайвих променів 31

3.3.2.2 Змінна довжина кроку 31

3.3.2.3 Бінарний пошук точки перетину 33

Висновки 34

Список використаних джерел 35

Анотація

Метою даної роботи є створення програмного забезпечення для моделювання динаміки рідини на основі методу SPH (Smoothed particle hydrodynamics) з використанням розподілених обчислень на GPU, а також розробки методу візуалізації рідини за допомогою технології трасування променів.

Вступ

Потужні обчислювальні машини з’явилися лише років 30-40 тому, а вже стали невід’ємною частиною безлічі галузей науки та виробництва. В більшості високотехнологічних сфер виробництва комп’ютери вже посідають провідне місце, особливо в задачах симуляції. Наприклад, випробування таких складних деталей як лопатка турбореактивного двигуна або крило літака набагато дешевше виконати на створеній в комп’ютері віртуальній моделі аніж на реальному прототипі. За допомогою комп’ютерних симуляцій можна підвищити міцність деталей, оптимізувати форму і звести до мінімуму витрати матеріалу. Це дозволяє знизити енергоспоживання і підвищити швидкість транспортних засобів, будувати хмарочоси, дамби і інші споруди, що будуть стійкими до землетрусів і поривів вітру, повеней і інших природних явищ. Крім галузі технічного виробництва симуляція активно використовується і в сфері розваг, зокрема в створенні мультфільмів та зйомці фільмів зі спецефектами. Такі сцени, як цунамі, що затоплює місто або падіння на землю космічного корабля прибульців зараз не обходяться без симуляції рідин та твердих тіл.

Моделювання рідини є дуже ресурсоємкою задачею, навіть за мірками сучасних обчислювальних потужностей. Саме тому необхідно мати інструмент, який би звів до мінімуму використання обчислювальних ресурсів.

Зазвичай для задач моделювання рідини використовується метод SPH (Smoothed particle hydrodynamics). Вперше цей метод був запропонований в 1977 році вченими Гінгольдом, Монаганом та Люсі [2] для астрофізичних обчислень. Пізніше, шляхом запозичення ідей з молекулярної динаміки, метод був адаптований для задач моделювання рідких речовин.

Основна частина

1 Фізика рідини

1.1 Базові фізичні властивості рідини

Для того, щоб побудувати модель рідини необхідно розуміти її базові фізичні властивості: інертність, нестисливість і здатність до легкої зміни форми.

Інертність - це властивість, що визначає рівень того, наскільки тіло легко змінює свою швидкість Інертність властива всім тілам, що мають масу, проте для моделювання газоподібних речовин з низькою густиною вона не грає важливу роль і іноді нею можна нехтувати. Рідина ж має досить велику густину, тому інертність має суттєвий вплив на поведінку рідини.

Здатність до легкої зміни форми властива рідинам оскільки для них відсутнє поняття пластичності. Проте деякі рідини мають високу в’язкість, що дозволяє їм спричиняти певний спротив деформації.

Нестисливість є основною відмінністю рідких речовин від газоподібних. Завдяки цій властивості густина рідини залишається постійною в будь-якій точці, навіть за умови великого тиску. Варто зазначити, що в реальному світі абсолютно нестисливих рідин не існує, тому має сенс говорити лише про умовну нестисливість рідких речовин порівняно з газоподібними.

1.2 Методи дискретизації об’єму рідини

Математично рідину можна визначити як множину точок (поле), для кожної з яких визначена густина, швидкість руху, температура та деякі інші властивості. Мета симуляції – забезпечити визначення цієї множини на певному часовому проміжку з урахуванням вищеперерахованих властивостей рідини.

Щоб моделювання стало можливим необхідно провести дискретизацію об’єму рідини. Для цього існують два основні підходи: метод Ейлера та метод Лагранжа.

Метод Ейлера побудований на основі сітки, в комірках якої визначені властивості рідини. Для визначення властивостей рідини в певній точці простору використовується інтерполяція між значеннями сусідніх комірок. Основна перевага цього методу полягає у простоті реалізації і високій швидкості роботи, проте його недолік – це недостатня фізична коректність.

Метод Лагранжа побудований на основі частинок, до яких прив’язані властивості рідини. Частинки рухаються в просторі і взаємодіють між собою. Для визначення властивостей рідини в певній точці простору використовується інтерполяція між значеннями сусідніх частинок. Метод Лагранжа повільніший за метод Ейлера, проте забезпечує високу фізичну коректність і високу гнучкість, що дозволяє використовувати його не тільки для моделювання рідин а й газів і твердих тіл.

Рисунок 2. Дискретизація методом Ейлера

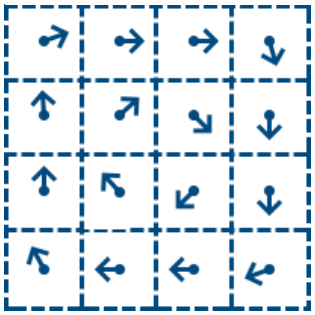
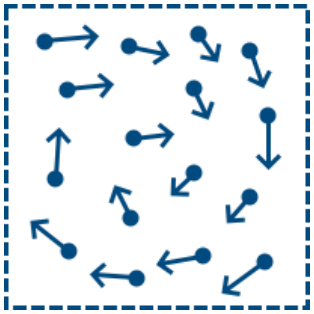


Рисунок 1. Дискретизація методом Лагранжа



1.3 Гідродинаміка згладжених частинок

SPH (Smoothed particle hydrodynamics) – метод моделювання рідини на основі згладжених частинок, що є одним з різновидів методу Лагранжа. В контексті цього методу частинки представляють певний об’єм рідини. Саме цей метод був використаний в дані роботі.

В SPH для обчислення стану рідини в певній точці об’єму використовується **вагова функція ядра**. Це монотонно спадаюча функція від одного аргументу, що визначена на проміжку [0, ∞) і набуває лише невід’ємних значень. Вибір конкретної реалізації функції може відрізнятися в залежності від поставленої задачі.

Алгоритм SPH складається 4 основних етапів:

1. Обчислення густини рідини в позиціях частинок.
2. Обчислення тиску рідини в позиціях частинок.
3. Обчислення сил, що діють на частинки.
4. Обчислення прискорення, швидкості і позиції частинки.

1.3.1 Обчислення густини

В першу чергу необхідно обчислити значення густини в позиціях кожної частинки, що відбувається за наступною формулою [1]:

$$\rho_i = \sum_j \frac{m_j}{\rho_j} \rho_j W_{ij}$$

ρ_i – густина рідини в позиції частинки i,

m_j – маса частинки j,

ρ_j – густина частинки j,

W_{ij} – вагова функція ядра від відстані між частинками i та j.

Сума обчислюється по всім частинкам, що приймають участь у симуляції.

Для обчислення густини використана наступна реалізація функції ядра [2]:

$$W(x) = (1 - x/h)^2$$

h – згладжуваний параметр.

1.3.2 Обчислення тиску

Тиск обчислюється за наступною формулою на основі значень густини, обрахованих на попередньому етапі [1]:

$$p_i = k(\rho_i - \rho_{rest})$$

ρ_i – густина рідини в позиції частинки і, обрахована на попередньому етапі,

ρ_{rest} – густина спокою рідини, яку представляє частинка,

k – константа нестисливості рідини.

Як вже було сказано, абсолютно нестисливих рідин не існує, тому має сенс говорити про **нестисливість як параметр**, що може мати різні значення в залежності від задачі. Для досягнення необхідного рівня нестисливості константу можна збільшувати, проте встановлення великих значень може викликати **дестабілізацію**.

1.3.3 Обчислення сил

На цьому етапі необхідно обчислити всі сили, що діють на частинку: силу тиску, силу тяжіння та силу в’язкості. Кожна сила є векторною величиною.

Сила тиску обчислюється за наступною формулою [1]:

$$F_i = \sum_j \frac{m_j}{\rho_j} \frac{p_{i+}p_j}{2} C_{pressure} W_{ij}$$

F_i – сила тиску на частинку і,

p_i – тиск в позиції частинки і,

p_j – тиск в позиції частинки j,

$C_{pressure}$ – константа тиску рідини,

W_{ij} – вагова функція ядра від відстані між частинками і та j.

Сума обчислюється по всім частинкам, що приймають участь у симуляції.

Для обчислення сили тиску була використана наступна реалізація функції ядра [2]:

$$W(x) = (1 - x/h)$$

h – згладжуваний параметр.

Сума обчислюється по всім частинкам, що приймають участь у симуляції.

Сила в’язкості обчислюється за наступною формулою [1]:

$$F_i = \sum_j \frac{m_j}{\rho_j}(V_j - V_i)C_{visc}W_{ij}$$

V_i – швидкість частинки і,

V_j – швидкість частинки j,

C_{visc} – константа в’язкості рідини,

W_{ij} – вагова функція ядра від відстані між частинками і та j.

Сума обчислюється по всім частинкам, що приймають участь у симуляції.

Для обчислення сили в’язкості була використана наступна реалізація функції ядра [2]:

$$W(x) = (1 - x/h)$$

h – згладжуваний параметр.

В’язкість рідини – властивість, що викликає сповільнення відносного руху близько розташованих частинок і, як результат, спричиняє певний **спротив деформації**. Приклади речовин з високою в’язкістю: мед, смола. Приклади речовин з низькою в’язкістю: спирт, вода, олія.

Сила тяжіння обчислюється за формулою [1]:

$$F_i = \rho_i g$$

g – прискорення вільного падіння (для планети Земля становить 9.8 м/с²).

1.3.4 Обчислення прискорення, швидкості і позиції

Прискорення частинки обчислюється за формулою [1]:

$$a_i = f_i/\rho_i$$

f_i – рівнодійна сил, що діють на частинку.

Швидкість частинки обчислюється за формулою [1]:

$$V_{next} = V_{prev} + a\Delta t$$

V_{prev} – швидкість частинки на попередньому кроці симуляції,

a – прискорення частинки,

Δt – часовий проміжок поточного кроку симуляції.

Позиція частинки обчислюється за формулою [1]:

$$P_{next} = P_{prev} + V\Delta t$$

P_{prev} – позиція частинки на попередньому кроці симуляції,

V – швидкість частинки,

Δt – часовий проміжок поточного кроку симуляції.

2. Модель

2.1 Поняття радіусу частинки і сусідніх частинок

Взаємодія між частинками послаблюється зі збільшенням відстані між ними. На певній відстані взаємодія між частинки припиняється. Таку відстань будемо називати **радіусом** частинки. **Сусідами** деякої частинки будемо називати частинки, відстань до яких менша за радіус. Іншими словами, сусіди – це близько розташовані частинки, між якими виникає взаємодія.

Рисунок 3. Розташування частинок у просторі

Рисунок 4. Радіус частинки P і її сусідні частинки

2.2 Структура даних

2.2.1 Специфікація

В моделі кожна частинка являє собою об’єкт, що зберігає значення власної швидкості руху та позиції у просторі.

```
struct Particle {
    float x;
    float y;
    float z;
    float vx;
    float vy;
    float vz;
};
```

Структура даних для зберігання частинок повинна відповідати деяким критеріям:

1. Можливість швидкого визначення сусідів певної частинки без необхідності перебору всіх частинок, що приймають участь у симуляції.
2. Можливість паралельного доступу для операцій зчитування.

2.2.2 Імплементація

Найбільш вдалим вибором в якості структури даних для зберігання частинок виявилася **просторова сітка.**

Виділимо **прямокутну область** простору і розіб’ємо її на квадратні **комірки**, довжина сторони яких дорівнює радіусу частинки. Частинки можуть існувати лише в межах цієї області.

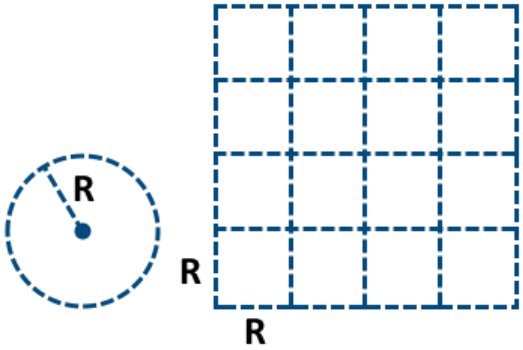


Рисунок 3. Відношення розміру комірок і радіусу частинки

Комірки задаються двома або трьома (в залежності від розмірності простору) невід’ємними **цілочисельними координатами**.

Частинка **належить** комірці, якщо її позиція розташована в межа цієї комірки. Кожна частинка належить строго одній комірці, при цьому кожній комірці може належати одразу багато частинок.

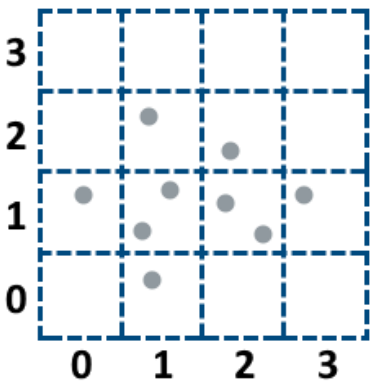


Рисунок 4. Нумерація комірок просторової сітки

Основна ідея полягає у тому, щоб зберігати частинки у відповідних їх розташуванню комірках. В такому випадку потенційні сусідні частинки будуть розташовані в сусідніх комірках, а для визначення сусідів певної частинки буде достатньо перебрати частинки в 27 сусідніх комірках.

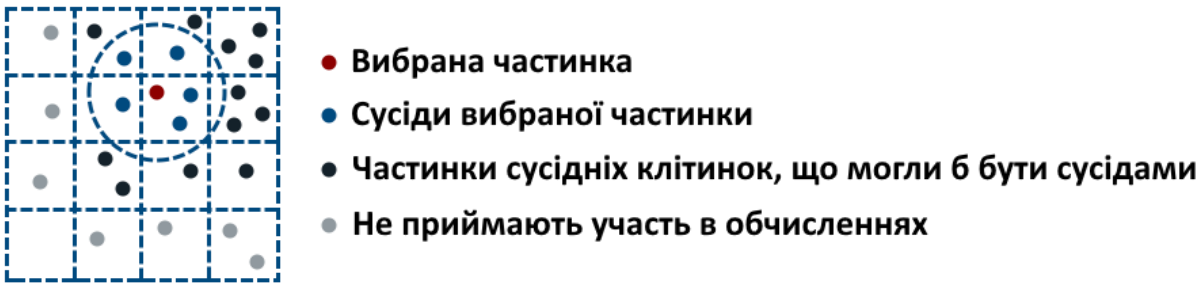


Рисунок 5. Схема знаходження сусідів певної частинки

Визначимо константи `CELLS_PER_GRID_X`, `CELLS_PER_GRID_Y`, `CELLS_PER_GRID_Z`, що відповідають розміру сітки в кількості комірок по трьом осям. Також визначимо константу `BUFFER_SIZE`, яка відповідає максимальній кількості частинок, що може знаходитися в одній комірці.

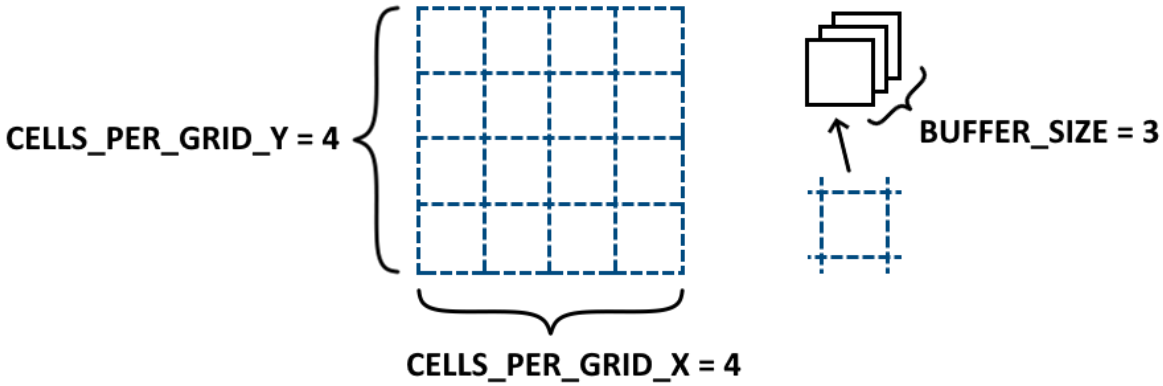


Рисунок 6. Відповідність констант параметрам просторової сітки

Визначимо структуру даних для зберігання частинок як чотиривимірний масив частинок і тривимірних масив, що зберігає фактичну кількість частинок у відповідній комірці:

```
Particle data[CELLS_PER_GRID_X]
             [CELLS_PER_GRID_Y]
             [CELLS_PER_GRID_Z]
             [BUFFER_SIZE];

int size[CELLS_PER_GRID_X]
        [CELLS_PER_GRID_Y]
        [CELLS_PER_GRID_Z];
```

Координати комірки, якій відповідає певна частинка в тривимірному просторі обчислюються за наступним алгоритмом:

```
int cell_x = floor(particle.x / CELL_SIZE);
int cell_y = floor(particle.y / CELL_SIZE);
int cell_z = floor(particle.z / CELL_SIZE);
```

Для перебору частинок всередині клітинки використовується наступний алгоритм:

```
int count = size[cell_x][cell_y][cell_z];
for (int i = 0; i < count; i++) {
    Particle p = particles[cell_x][cell_y][cell_z][i];

    // Код обробки частинки
}
```

Для перебору 27 сусідніх комірок для тривимірного простору (9 клітинок для двовимірного) використовується наступний алгоритм:

```
for (int c_x = cell_x - 1; c_x <= cell_x + 1; c_x++) {
    for (int c_y = cell_y - 1; c_y <= cell_y + 1; c_y++) {
        for (int c_z = cell_z - 1; c_z <= cell_z + 1; c_z++) {
            int count = size[c_x][c_y];

            for (int i = 0; i < count; i++) {
                Particle p = particles[c_x][c_y][c_z][i];

                // Код обробки частинки
            }
        }
    }
}
```

2.3 Алгоритм фізичних обчислень

2.3.1 Технологія Nvidia CUDA

Для забезпечення високої продуктивності було вирішено використовувати технологію Nvidia CUDA, яка дозволяє використовувати **GPU** для обчислень, **не пов’язаних з графікою**. Основним поняттям в CUDA є **ядро**. Це програма, написана на мові програмування CUDA C що виконується одночасно на багатьох ядрах мультипроцесорів відеокарти. Мова програмування CUDA C дуже схожа на мову C, проте має деякі відмінності при роботі з пам’яттю.

2.3.1.1 Пам'ять відеокарти

Відеокарта має декілька різних видів пам’яті. Для обчислень було використано лише три з них: регістрову, спільну та глобальну.

Регістрова пам'ять - це найбільш швидкий вид пам’яті, що доступний кожному потоку в розмірі до 2 кілобайт. Зазвичай використовується для зберігання локальних змінних потоку.

Спільна пам'ять повільніша за регістрову, проте вона єдина для всіх потоків одного блоку, що дозволяє цим потокам взаємодіяти між собою. Розмір спільної пам’яті складає 16 кілобайт для одного мультипроцесора, на якому може бути запущено до 1024 потоків.

Глобальна пам'ять - це найповільніший та найбільший вид пам’яті, що використовується для збереження вхідних та вихідних даних потоків, великих структур даних, текстур та інших об’єктів. Ця пам'ять спільна для всіх потоків відеокарти і її розмір може складати більше 2 ТБ.

2.3.1.2 Шаблон роботи з Nvidia CUDA

Основний шаблон для роботи з технологією CUDA складається з наступних етапів.

1. Виділення глобальної пам’яті на відеокарті.
2. Копіювання вхідних даних потоків з оперативної пам’яті в глобальну.
3. Запуск потоків на відеокарті.
4. Копіювання вихідних даних потоків з глобальної пам’яті в оперативну.

2.3.1.3 Специфіка роботи з пам’яттю в Nvidia CUDA

При роботі з пам’яттю в CUDA потрібно слідувати певним правилам та рекомендаціям, щоб використання відеокарти було найбільш ефективним.

Особливість будови пам’яті не дозволяє багатьом потокам **одночасно зчитувати** дані з однієї комірки. При спробі одночасного зчитування відбувається затримка роботи потоків, що має назву **конфлікт банок**.

Запис вихідних даних окремими потоками потрібно виконувати в різні комірки. При спробі **одночасного запису** даних в одну комірку значення в ній не буде визначено, оскільки специфікація CUDA не гарантує атомарність операцій запису.

Зчитування даних з глобальної пам’яті **повільніше** ніж зчитування зі спільної. Якщо в одному блоці потоки багато разів зчитуються дані з одного місця в глобальній пам’яті, то має сенс перенести дані з цього місця в спільну пам'ять відповідного блоку.

2.3.2 Етапи алгоритму фізичних обчислень

Враховуючи вищеперераховані правила і рекомендації визначимо порядок запуску потоків.

Перший етап: обчислення густини. Під час першого етапу запускаються потоки, для обчислення густини в позиції кожної частинки. Кількість запущених потоків відповідає максимальній кількості частинок, тобто `CELLS_PER_GRID_X*CELLS_PER_GRID_Y*CELLS_PER_GRID_Z*BUFFER_SIZE`. Таким чином кожен потік відповідає за обчислення густини для однієї частинки. Кожен потік зчитує значення позиції поточної і сусідніх частинок і записує в тимчасове сховище значення густини в позиції поточної частинки. Легко зрозуміти, що кількість запущених потоків більша за фактичну кількість частинок в симуляції, тому деякі запущені потоки виявляються зайвими. Такі зайві потоки одразу припиняються і не використовують багато обчислювальних ресурсів відеокарти.

```
__global__ void kernel1(
    Particle global_input_particles_data[CELLS_PER_GRID_X]
                                        [CELLS_PER_GRID_Y]
                                        [CELLS_PER_GRID_Z]
                                        [BUFFER_SIZE],

    int global_input_particles_counts[CELLS_PER_GRID_X]
                                     [CELLS_PER_GRID_Y]
                                     [CELLS_PER_GRID_Z],

    float global_output_particles_desnity[CELLS_PER_GRID_X]
                                         [CELLS_PER_GRID_Y]
                                         [CELLS_PER_GRID_Z]
                                         [BUFFER_SIZE]) {
```

```
    // Обчислення координат комірки на основі значень
    // об'єктів blockIdx та threadIdx
    const int cell_x = ...
    const int cell_y = ...
    const int cell_z = ...
    const int index = ...


    // Зчитування об'єкту частинки з глобальної пам'яті
    Particle current = global_input_particles_data[cell_x]
                                                  [cell_y]
                                                  [cell_z]
                                                  [index];

    // Обчислення густини
    float density = ...

    // Запис значення густини в глобальну пам'ять
    global_output_particles_desnity[cell_x]
                                   [cell_y]
                                   [cell_z]
                                   [index] = density;
}
```

Другий етап: обчислення швидкості та позиції частинок. Під час цього етапу запускаються потоки, що обчислюють оновлену швидкість та позицію частинок на поточному кроку симуляції. Кількість запущених потоків цього етапу також відповідає максимальній кількості частинок в симуляції. Кожен потік зчитує позицію та швидкість поточної та сусідніх частинок а також густину в позиціях цих частинок і записує в проміжне сховище оновлену позицію та швидкість частинок.

```
__global__ void kernel2(
    Particle global_input_particles_data[CELLS_PER_GRID_X]
                                        [CELLS_PER_GRID_Y]
                                        [CELLS_PER_GRID_Z]
                                        [BUFFER_SIZE],

    float global_input_particles_density[CELLS_PER_GRID_X]
                                        [CELLS_PER_GRID_Y]
                                        [CELLS_PER_GRID_Z]
                                        [BUFFER_SIZE],

    int global_input_particles_counts[CELLS_PER_GRID_X]
                                     [CELLS_PER_GRID_Y]
                                     [CELLS_PER_GRID_Z],

    Particle global_output_particles_data[CELLS_PER_GRID_X]
                                         [CELLS_PER_GRID_Y]
                                         [CELLS_PER_GRID_Z]
                                         [BUFFER_SIZE]) {


    // Обчислення координат комірки на основі значень
    // об'єктів blockIdx та threadIdx
    const int cell_x = ...
    const int cell_y = ...
    const int cell_z = ...
    const int index = ...

    // Зчитування об'єкту частинки з глобальної пам'яті
    Particle current = global_input_particles_data[cell_x]
                                                  [cell_y]
                                                  [cell_z]
                                                  [index];

    // Зчитування значення густини з глобальної пам'яті
    float density = global_input_particles_density[cell_x]
                                                  [cell_y]
                                                  [cell_z]
                                                  [index];

    // Обчислення нової позиції частинки current


    // Запис об'єкту частинки в глобальну пам'ять
    global_output_particles_desnity[cell_x]
                                   [cell_y]
                                   [cell_z]
                                   [index] = current;
}
```

Третій етап: перерозподіл. На попередньому етапі було змінено позицію частинок, проте їх фізичне розташування в просторовій сітці не змінилося. На цьому етапі відповідність частинок коміркам в просторовій сітці не відповідає дійсності. Щоб розташувати частинки в правильні комірки необхідно виконати перерозподіл. На цьому етапі запускаються потоки для кожної клітинки просторової сітки. Ці потоки зчитують всю інформацію про частинки в поточній комірці та 26-ми сусідніх і записує в результат лише ті частинки, що фактично відповідають поточній комірці. Таким чином створюється нова структура даних, що містить вже правильно розташовані частинки.

```
__global__ void kernel3(
    Particle global_input_particles_data[CELLS_PER_GRID_X]
                                        [CELLS_PER_GRID_Y]
                                        [CELLS_PER_GRID_Z]
                                        [BUFFER_SIZE],

    int global_input_particles_counts[CELLS_PER_GRID_X]
                                     [CELLS_PER_GRID_Y]
                                     [CELLS_PER_GRID_Z],

    Particle global_output_particles_data[CELLS_PER_GRID_X]
                                         [CELLS_PER_GRID_Y]
                                         [CELLS_PER_GRID_Z]
                                         [GLOBAL_BUFFER_SIZE],

    int global_output_particles_counts[CELLS_PER_GRID_X]
                                      [CELLS_PER_GRID_Y]
                                      [CELLS_PER_GRID_Z]) {


    // Обчислення координат комірки на основі значень
    // об'єктів blockIdx та threadIdx
    const int cell_x = ...
    const int cell_y = ...
    const int cell_z = ...

    // Створення змінної, яка відповідає за кількість
    int count = 0;
```

```
    // Цикл по частинкам в сусідніх комірках
    for (Particle current : ...) {
        //Знаходження фактичних координат комірки
        const int actual_cell_x = ((int)current.x);
        const int actual_cell_y = ((int)current.y);
        const int actual_cell_z = ((int)current.z);

        //Перевірка на еквівалентність фактичних
        //і поточних координат комірки
        if (actual_cell_x != cell_a_x ||
            actual_cell_y != cell_a_y ||
            actual_cell_z != cell_a_z) continue;

        //Запис об'єкту частинки у поточну комірку
        global_output_particles_data[cell_x]
                                    [cell_y]
                                    [cell_z]
                                    [count] = current;
         //Інкремент поточної кількості частинок
        count++;
    }

    // Запис кількості частинок у поточну комірку
    global_output_particles_counts[cell_x]
                                  [cell_y]
                                  [cell_z] = count;
}
```

2.3.3 Оптимізації

2.3.3.1 Мінімізація кількості звернень до глобальної пам’яті

Потік, що оброблює частинку звертається до частинок, що знаходяться в 27 комірках (1 центральна і 26 оточуючих сусідніх). Таким чином до однієї частинки буде відбуватися багато звернень з потоків, що оброблюють частинки в сусідніх клітинках. Щоб мінімізувати кількість звернень до глобальної пам’яті потоки були об’єднані в блоки, кожен з яких відповідає за обробку частинок в певній прямокутній ділянці просторової сітки. Дані частинок з цієї прямокутної ділянки копіюються в спільну пам'ять відповідного блоку і використовуються потоками цього блоку. Ця оптимізація не дала суттєвого прискорення на відеокарті ноутбука (лише 5-10%), адже час фізичних обчислень виявилися набагато довшим за час звернення до глобальної пам’яті на ній. Проте ефективність цієї оптимізації на інших більш потужних пристроях може бути досить суттєвою, оскільки кількість мультипроцесорів відеокарти можна досить легко збільшити на відміну від пропускної здатності глобальної пам’яті. Також цю оптимізацію можна адаптувати для обчислення на суперкомп’ютерах.

2.3.3.2 Мінімізація об’єму пам’яті

Частинки в тривимірному просторі займають в пам'яті 24 байти (по 3 поля типу `float` на позицію та швидкість). Значення типу `float` (4 байти), визначене в певних межах можна закодувати в значення типу `unsigned short` (2 байти) з певною втратою точності. Таким чином можна знизити використання пам’яті частинкою в два рази.

Визначимо алгоритм кодування значень `float` з нижньою межею `MIN` та верхньою межею `MAX` в значення `unsigned short`:

Нормалізація: зведемо проміжок (`MIN`, `MAX`) до проміжку (0, 1). Для цього віднімаємо `MIN` від початкового значення і ділимо на (`MAX−MIN`).

Масштабування: помножимо отримане значення на максимальне значення типу `unsigned short` (65535 або 0xFFFF) і отримуємо значення в проміжку (0, 65535).

Зведення: зведемо отримане значення до типу `unsigned short` попередньо додавши 0.5 для правильного округлення.

Декодування значень відбувається аналогічно в оберненому порядку. Ця оптимізація дозволяє розмістити в спільній пам’яті вдвічі більше частинок. Також це дозволяє прискорити копіювання даних з глобальної пам’яті.

3 Візуалізація

3.1 Основні положення

Для візуалізації рідини в тривимірному просторі була вибрана **технологія трасування променів**. Цей вибір зумовлений відносною простотою реалізації, високої якості отриманих зображень і високою швидкістю роботи в контексті поставленої задачі.

Основними концепціями в технології є поняття камери, променю і об’єкту сцени.

Камера – об’єкт у просторі з позиції якого відбувається візуалізація. Камера має своє положення і орієнтацію в просторі. **Промінь** починає рух від **джерела світла** і взаємодіє з **об’єктами на сцені**, що може спричинити його **заломлення**, **віддзеркалення** або **поглинання**. Промінь має **інтенсивність**, яка може змінитися при його проходженні через простір і при контакті з об’єктами сцени.

3.2 Алгоритм трасування променів

Трасування променю відбувається з кінця його руху (позиція камери) до початку (джерело світла). Метою трасування є дізнатися **звідки** цей промінь прийшов і яку **інтенсивність** він має в кінці.

Трасування променів запускається для кожного пікселя екрану. Під час трасування обчислюється інтенсивність променю, в залежності від якої буде визначено **яскравість** зафарбованого пікселю. Для того, щоб отримувати кольорові зображення використовується інтенсивність окремо для кожного світлового компоненту (червоного, зеленого та синього).

Приблизний алгоритм трасування променю виглядає наступним чином:

Крок перший: визначення позиції та напрямку променю в залежності від розташування камери та номеру відповідного пікселя.

Крок другий: пошук найближчого перетину з об’єктами на сцені, визначення координат точки перетину з об’єктом, нормалі поверхні в точці перетину і кольору цієї поверхні.

Крок третій: обробка віддзеркалення променю поверхнею об’єкту-перешкоди, враховуючи зміну напрямку руху (кут падіння дорівнює куту віддзеркалення) і поглинання світла цією поверхнею.

Після завершення третього кроку виконується повернення до другого кроку. Якщо на другому кроці не було знайдено перетину з об’єктами, то виконання припиняється, я джерело світла визначається як небо.

3.3 Функція перетину

Технологія трасування променів може бути використана для візуалізації будь-якої фігури, для якої визначена **функція перетину**. Функція перетину за позицією та напрямком променю визначає координати точки перетину цього променю з фігурою та нормаль поверхні фігури в цій точці перетину.

3.3.1 Імплементація на основі густини

Для візуалізації гладкої поверхні рідини використовується функція перетину на основі значень густини.

Вибирається деяке **мінімальне** значення густини. Якщо значення густини в певній точці простору **більше за мінімальне**, то ця точка **належить** фігурі.

Функцію перетину променю з такою фігурою можна визначити **числовим методом**. Основна ідея полягає у тому, щоб послідовно, з певним кроком перевіряти точки, через які проходить промінь, поки належність точки фігурі не буде виявлено.

3.3.2 Оптимізації

Досить легко зрозуміти, що в такому методі точність визначення точки перетину з фігурою залежить від довжини кроку. Зменшення довжини кроку підвищує точність і, як результат, якість отриманого зображення. Проте це також підвищує час виконання. Певні оптимізації дозволяють не тільки мінімізувати цей час але й досягнути максимальної якості отриманого зображення.

3.3.2.1 Уникнення обрахунків зайвих променів

Оскільки вся рідина знаходиться в обмеженій паралелепіпедом області простору, немає сенсу обраховувати промені, що не перетинають цю область. Пікселі, що відповідають таким променям одразу можуть бути зафарбовані в колір неба.

Якщо ж промінь перетинає виділену область простору, пошук перетину з фігурою в частинах, що не належать цій області також можна опустити. Для цього потрібно визначити функцію перетину променю з паралелепіпедом.

3.3.2.2 Змінна довжина кроку

На кожній **ітерації** виконується зміщення позиції променю на **довжину кроку**. Використовуючи вже розроблені структури даних для зберігання частинок

можна швидко визначити частинки, що знаходяться в сусідніх комірках відносно тих, де на поточній ітерації знаходиться позиція променю і де вона знаходитиметься на наступній.

На початку довжина кроку дорівнює радіусу частинки. Якщо при спробі визначити сусідні частинки за наведеним вище принципом таких не було знайдено, це означає, що на всьому проміжку між поточною і наступною позицією променю значення густини рівне нулю. Таким чином при відсутності сусідніх частинок довжина кроку може дорівнювати радіусу частинки.

Якщо ж сусідні частинки були виявлені, то довжину кроку можна зробити меншою, оскільки в такому випадку існує ймовірність перетину променю з фігурою. Це можна зробити шляхом розбиття кроку, довжиною в радіус частинки, на декілька частин (наприклад десять). Назвемо цю дію **дробленням кроку**.

Цей метод можна вдосконалити, користуючись тим фактом, що густина на більшій за радіус частинки відстані завжди дорівнює нулю. Таким чином, якщо промінь не перетинає сферу з радіусом частинки центом у позиції цієї частинки, то вона не впливає на густину рідини на шляху променю і її розгляд можна опустити. Власне оптимізація полягає у тому, щоб не розглядати в якості сусідів такі частинки і тоді дроблення кроку буде виконуватися рідше.

Можна досягнути ще більшого прискорення за рахунок **кешування** сусідніх частинок у спільну пам'ять. Таким чином при дробленні кроку можна буде мінімізувати кількість звернень до глобальної пам’яті, адже всі сусідні частинки будуть знаходитися у спільній.

3.3.2.3 Бінарний пошук точки перетину

Ще одна оптимізація дозволяє не тільки зменшити рівень дроблення кроку, а й підвищити якість отриманого зображення. Ідея полягає у тому, щоб при виявленні належності точки фігурі виконати **бінарний пошук** точки перетину. Точка перетину шукається, коли виявлено ситуацію, де позиція променю на поточній ітерації ще не належить фігурі, а позиція на наступній ітерації вже належить їй. Зрозуміло, що точка перетину з фігурою знаходиться десь по середині. Точна координата цієї точки і визначається шляхом бінарного пошуку.

```
// Ініціалізація позицій променю на
// попередній та наступній ітерації
Vector outside = ...
Vector inside = ...
Vector current = mul(add(outside, inside), 0.5f);

// Бінарний пошук точки перетину
for (int i = 0; i < ITERATIONS; i++) {
    if (isInside(current)) {
        inside = current;
    } else {
        outside = current;
    }
    current = mul(add(outside, inside), 0.5f);
}

// Повернення точки перетину
return current;
```

Можна було б припустити, що використання бінарного пошуку дозволило б взагалі відмовитися від дроблення кроку, проте це не так. При великих значеннях кроку можна пропустити перетин з виступаючими частинами фігури.

Висновки

У цій роботі було проведено моделювання динаміки рідини і її візуалізація з використанням GPU.

Створено програмний пакет, проведено експерименти та отримано відеозображення. Програмний пакет для GPU було створено за допомогою технології Nvidia CUDA. Розробка спиралася на особливості архітектури відеокарти та принципи її роботи, що дозволило досягти високої продуктивності.

У ході роботи вибиралися спеціальні технічні рішення: використання структури даних, яка є ефективною для швидкого пошуку сусідніх елементів у просторі, мінімізація складних обчислень при моделюванні фізичного процесу, упаковка чисел для економії пам'яті, використання спільної пам'яті для кешування.

Було розроблено та програмно реалізовано ефективний спосіб візуалізації поверхні рідини, який використовує обчислення значення густини рідини на шляху трасуючого променю для пошуку поверхні рідини, а крім того - обчислення градієнта густини в точці на поверхні для визначення нормалі до цієї поверхні. Стандартний спосіб відомий під назвою алгоритму крокуючих кубів вимагає більшої кількості операцій для даної задачі. Використовувалися і інші оптимізації: змінна довжина кроку, бінарний пошук точки перетину та інші. Вони дозволили суттєво підвищити швидкість візуалізації.

Дане дослідження та отриманий пакет програм може бути використаний надалі для створення нового покоління програмних комплексів для моделювання взаємодії рідини та твердого тіла. Для цього повинен буде використовуватися суперкомп'ютер, процесори якого обладнані додатково одним або кількома GPU.

Список використаних джерел

[1] Particle-based fluids: https://www.cs.cmu.edu/~scoros/cs15467-s16/lectures/11-fluids2.pdf

[2] Particle-based viscoelastic fluid simulation: http://www.ligum.umontreal.ca/Clavet-2005-PVFS/pvfs.pdf

[3] CUDA: Робота с пам’яттю. Частина I: https://habr.com/ru/articles/55461/

[4] CUDA: Робота с пам’яттю. Частина II: https://habr.com/ru/articles/56514/

[5] CUDA: Як працює GPU: https://habr.com/ru/articles/54707/

[6] Трасування променів: https://habr.com/ru/companies/pixonic/articles/546328/

[7] Створення симуляції методом SPH: https://philip-mocz.medium.com/create-your-own-smoothed-particle-hydrodynamics-simulation-with-python-76e1cec505f1

[8] Реалізація методу SPH в двовимірному просторі: https://lucasschuermann.com/writing/implementing-sph-in-2d