Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБКА ЗАСТОСУНКУ ДЛЯ СИМУЛЯЦІЇ ТА ЗОБРАЖЕННЯ ЧАСТИНОК І МАТЕРІАЛІВ З ВИКОРИСТАННЯМ ГОТОВОЇ ПЛАТФОРМИ ВІЗУАЛІЗАЦІЇ
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-21-3
Корж А.В.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Яковлева О.В.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Коржу Андрію Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка застосунку для симуляції та зображення частинок і матеріалів з використанням готової платформи візуалізації

затверджена наказом університету від 19 травня 2025 року № 381Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 30 травня 2025 р.
3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, платформа .NET 7, мова C#; бібліоткеки MonoBehaviour; середовище розробки Visual Studio 2022.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз сучасних фізичних моделей та методів симуляції рідини, зокрема методу згладженої гідродинаміки частинок.
2. Дослідження можливостей графічних шейдерів для рендерингу рідини та огляд відповідних платформ і рушіїв.
3. Проєктування архітектури застосунку для симуляції, включаючи структуру основних модулів і обчислювальне ядро на GPU.
4. Реалізація симуляції рідини в Unity: створення сцени, розробка скриптів і інтеграція compute.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Схема архітектури застосунку, блок-схеми обчислювального процесу, структурна діаграма модулів, ілюстрації сцени симуляції, приклади рендерингу частинок, графіки продуктивності, лістинги коду та тестові зображення.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	07.04.2025	
2	Аналіз завдання, підбір літератури	08.04.25-10.04.25	
3	Аналіз літератури з досліджуваної проблеми	11.04.25-14.04.25	
4	Аналіз технічних засобів	15.04.25-20.04.25	
5	Розробка методів	21.04.25-27.04.25	
6	Програмна реалізація	28.04.25-11.05.25	
7	Оформлення пояснювальної записки	12.05.25-20.05.25	
8	Перевірка на нормоконтроль	21.05.25-01.06.25	
9	Перевірка на плагіат	21.05.25-01.06.25	
10	Рецензування	21.05.25-01.06.25	
11	Підготовка презентації та доповіді	21.05.25-18.06.25	
12	Занесення роботи в електронний архів	02.06.25-18.06.25	
13	Попередній захист кваліфікаційної роботи	02.06.25-18.06.25	

Дата видачі завдання 7 квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ _____ доц. Яковлева О.В._____

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 58 с., 23 рис., 45 джерело.

UNITY, C#, ШЕЙДЕРИ, ОБЧИСЛЕННЯ НА GPU, SPH, ПРОСТОРОВЕ ХЕШУВАННЯ, СИМУЛЯЦІЯ РІДИН, ОБ'ЄКТНО−ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ.

Об'єктом роботи є система симуляції частинок у середовищі рідини.

Метою роботи є розробка застосунку для моделювання та візуалізації поведінки рідини на основі методу згладженої гідродинаміки частинок з використанням рушія Unity як готової платформи рендерингу.

У процесі реалізації застосовано числові методи моделювання фізичних процесів, алгоритми просторового хешування для оптимізації пошуку сусідів, а також обчислення на GPU за допомогою compute shaders. Окрему увагу приділено оптимізації сортування частинок у просторі за допомогою алгоритмів GPU Count Sort і паралельного сканування (prefix sum). Для відображення результатів симуляції реалізовано шейдери візуалізації частинок у 2D-просторі.

У результаті роботи створено ефективну систему моделювання динаміки рідини в реальному часі з візуалізацією частинок, яка може бути використана для наукових, освітніх або демонстраційних цілей.

UNITY, C#, SHADERS, GPU COMPUTING, SPH, SPATIAL HASHING, FLUID SIMULATION, OBJECT-ORIENTED PROGRAMMING.

The object of this work is a particle-based fluid simulation system.

The goal of the work is to develop an application for simulating and visualizing fluid behavior using the Smoothed Particle Hydrodynamics method, implemented within the Unity engine as a ready-made rendering platform.

The project employs numerical methods for modeling physical processes, spatial hashing algorithms to accelerate neighbor searches, and GPU compute shaders to perform real-time computations. Special attention was given to optimizing particle sorting through GPU-based Count Sort and parallel prefix sum (scan) algorithms. The visualization of simulation results is achieved through custom 2D particle shaders.

As a result, an efficient real-time fluid simulation system was developed, capable of visualizing dynamic particle behavior and suitable for scientific, educational, or demonstrative purposes.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SPH − Smoothed Particle Hydrodynamics

GPU − Graphics Processing Unit

Шейдери − програми, що виконуються на GPU для обробки графіки

FPS − Frames Per Second

Buffer − область пам’яті, що використовується для зберігання даних, які GPU використовує під час рендерингу

Інстансінг − техніка рендерингу кількох копій одного об'єкта з меншими витратами ресурсів

HLSL − мова програмування для написання шейдерів, яка використовується в екосистемі Microsoft

URP − універсальний графічний рендеринг-пайплайн у Unity, оптимізований для широкого спектра пристроїв

HDRP − пайплайн високої якості графіки у Unity, орієнтований на потужні пристрої та фотореалізм

ВСТУП

Моделювання фізичних явищ у режимі реального часу є важливим напрямом сучасної комп'ютерної графіки та обчислювальної інженерії. Одним із найбільш цікавих і складних прикладів таких явищ є симуляція рідин, яка має широке застосування у відеоіграх, візуалізації, наукових дослідженнях та освіті.

Готові рушії, зокрема Unity, забезпечують зручне середовище для створення інтерактивних застосунків з використанням розширених можливостей рендерингу та обчислень. Завдяки підтримці C# та шейдерів, Unity дозволяє поєднувати об'єктно-орієнтоване програмування з ефективними GPU-обчисленнями.

Метод згладженої гідродинаміки частинок (Smoothed Particle Hydrodynamics, SPH) є одним з найпоширеніших методів симуляції рідин, який дозволяє моделювати рух та взаємодію частинок з урахуванням сил тиску та в'язкості. Однак, для забезпечення високої продуктивності та масштабованості необхідно застосовувати додаткові оптимізації, зокрема просторове хешування та сортування на GPU.

Актуальність цієї роботи полягає в розробці ефективного застосунку для симуляції та візуалізації поведінки рідини на основі частинок із використанням рушія Unity. Такий застосунок поєднує в собі сучасні фізичні моделі, обчислювальні шейдери та методи візуалізації, що дає змогу створити наочну та динамічну демонстрацію фізичних процесів.

1 СУЧАСНІ МЕТОДИ СИМУЛЯЦІЇ ТА РЕНДЕРИНГУ РІДИНИ

1.1 Фізичні моделі для симуляції рідини

Моделювання рідини – це складна фізико-математична задача, яка потребує врахування низки нелінійних процесів: тиску, в'язкості, сил поверхневого натягу та гравітації. Залежно від цілей симуляції, обираються різні фізичні моделі, що мають свої переваги та обмеження. На відміну від строго наукових симуляцій, де пріоритетом є максимальна фізична точність, у реальному часі основна увага приділяється продуктивності та візуальній правдоподібності.

Однією з найпоширеніших моделей у сучасних симуляціях є метод згладженої гідродинаміки частинок (Smoothed Particle Hydrodynamics, SPH). Він був розроблений у 1977 році для астрофізичних задач, однак згодом адаптований до візуалізації рідин у комп'ютерній графіці [1]. SPH моделює рідину як набір частинок, кожна з яких має масу, положення, швидкість та інші параметри. Основна ідея полягає в тому, що фізичні величини, такі як тиск чи густина, обчислюються шляхом згладженого усереднення значень сусідніх частинок за допомогою ядрової функції (рис. 1.1).

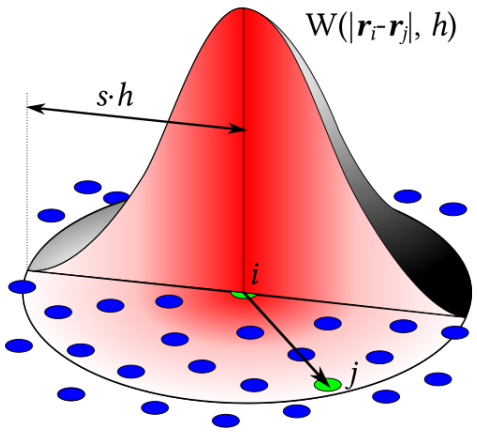

Рисунок 1.1 − Ілюстрація впливу згладжувального ядра на обчислення параметрів у SPH

SPH добре підходить для моделювання деформабельних середовищ, міжфазних границь та розбризкування. Водночас метод чутливий до параметрів симуляції: розміру часу кроку, радіусу впливу, коефіцієнтів тиску і в’язкості. Для уникнення числових нестабільностей часто застосовується штучна в’язкість або додаткове згладжування тиску. Ці механізми дозволяють уникати надмірного стиснення або хаотичного руху частинок, що можуть виникнути при некорректному балансі сил.

Альтернативою до SPH є метод фліп-частинок (FLIP) та його варіація PIC (Particle-in-Cell), які поєднують частинкові та сіткові підходи. Вони часто використовуються в кіноіндустрії для складніших симуляцій, однак потребують значно більше пам’яті та складніших обчислень. У реальному часі SPH залишається більш придатним, оскільки його реалізацію легше оптимізувати на GPU, а також масштабувати під різну кількість частинок без складних реконструкцій сітки (рис. 1.2).

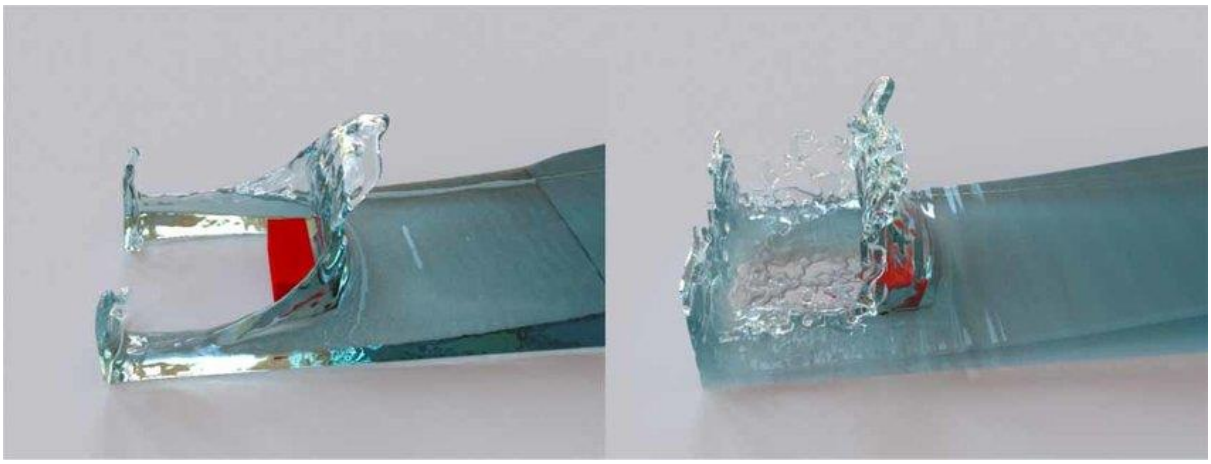

Рисунок 1.2 − Порівняння результатів симуляції: SPH vs FLIP

У двовимірному просторі (2D) фізична модель SPH стає ще ефективнішою, адже зменшується кількість сусідів і спрощуються обчислення. Це дозволяє запускати симуляції з тисячами частинок навіть на середньому рівні графічного обладнання. Проте важливо розуміти, що

спрощення розмірності тягне за собою і втрату деяких об’ємних ефектів, які природно виникають лише в тривимірному середовищі.

Для побудови фізично обґрунтованої, але швидкої симуляції SPH, зазвичай використовуються такі складові: рівняння стану для обчислення тиску, модель в’язкості, та згладжувальні ядра з компактною підтримкою, які забезпечують обмеження області впливу кожної частинки.

Таким чином, фізичні моделі, що застосовуються у симуляції рідини, є балансом між точністю опису процесів і швидкодією обчислень. У межах цього проекту використано метод SPH як базову модель, що забезпечує необхідну реалістичність при високій ефективності, особливо при реалізації на графічному процесорі.

1.2 Метод згладженої гідродинаміки частинок

Метод згладженої гідродинаміки частинок (Smoothed Particle Hydrodynamics, SPH) є чисельним методом, що широко використовується у комп’ютерній графіці, обчислювальній фізиці та інженерному моделюванні для опису динаміки континуумів, зокрема рідин та газів. Його ключовою особливістю є безсітковий підхід до розв’язання рівнянь гідродинаміки: замість використання фіксованої сітки простір дискретизується множиною частинок, кожна з яких переносить із собою фізичні властивості середовища, такі як маса, положення, швидкість, густина, тиск тощо. Такі частинки є не матеріальними об’єктами, а радше чисельними носіями інформації про стан рідини в певній локальній області простору.

В основі методу лежить апроксимація неперервних полів (густини, швидкості, тиску тощо) шляхом згладженого середнього значень сусідніх частинок у межах певного радіуса впливу h. Для цього використовується згладжувальна функція, або ядро (*kernel function*), яка визначає ступінь внеску кожної частинки в обчислення фізичних величин в околі іншої

частинки. Найчастіше використовуються ядра, які мають компактну підтримку, тобто їх значення є ненульовими лише в межах обмеженого радіуса. Це дозволяє зменшити обчислювальні витрати, зберігаючи при цьому фізичну коректність [2].

$$p_i = \Sigma_j m_j \cdot W(r_i - r_j, h), \tag{1.1}$$

де ρ_i – густина частинки *i*;

m_j – маса частинки *j*;

r_i, r_j – положення частинок *i* та *j*;

W – згладжувальна функція (ядро);

h – радіус впливу.

Таким чином, кожна частинка взаємодіє лише з обмеженою кількістю сусідів, що істотно знижує складність розрахунків (рис. 1.3). Після обчислення густини визначається тиск, зазвичай через рівняння стану, яке встановлює залежність тиску від густини [2]. Найпростішою формою такого рівняння є лінійне

$$p_i = k \cdot (\rho_i - \rho_0), \tag{1.2}$$

де p_i – тиск частинки *i*;

k – константа стисливості;

ρ_0 – початкова (референтна) густина.

Формули для сил, що діють на частинку, виводяться з рівнянь Нав'є–Стокса в лагранжевий формі. Зокрема, сила тиску на частинку *i*, спричинена частинкою *j*.

$$f_{pressure,ij} = m_j \cdot \left(\frac{p_i - p_j}{2p_j}\right) \cdot \nabla W(r_i - r_j, h). \tag{1.3}$$

Окрім тиску, на частинки діє також сила в’язкості, яка моделює внутрішнє тертя рідини

$$f_{viscosity,ij} = \mu \cdot m_j \cdot (\frac{(v_j - v_i)}{p_j}) \cdot \nabla^2 W(r_i - r_j, h), \quad (1.4)$$

де μ – коефіцієнт в’язкості.

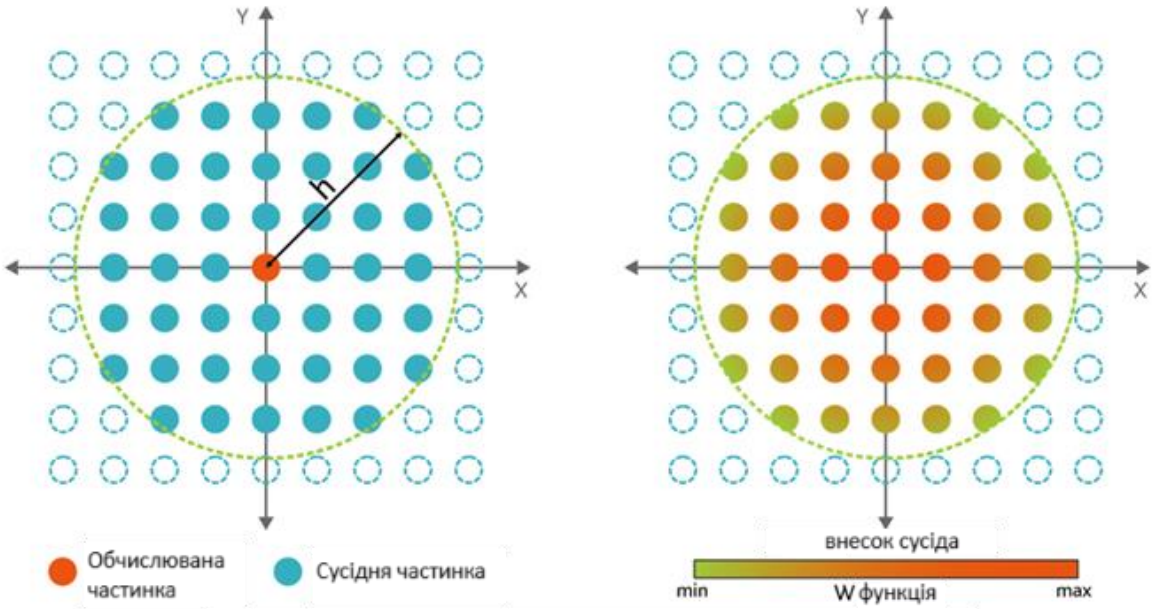

Рисунок 1.3 − Схема впливу сусідніх частинок у згладжувальному ядрі

Результуюче прискорення частинки визначається як сума всіх сил, що діють на неї, поділена на її густину, з урахуванням додаткових зовнішніх сил (рис. 1.4), таких як гравітація [2]

$$a_i = (\frac{1}{p_i}) \cdot \Sigma_j (f_{pressure,ij} + f_{viscosity,ij}) + g, \quad (1.5)$$

На основі прискорення виконується оновлення швидкості та положення частинок, зазвичай із використанням простого методу чисельної інтеграції, такого як метод Ейлера:

$$v_i(t + \Delta t) = v_i t + a_i \cdot \Delta t, \tag{1.6}$$

$$r_i(t + \Delta t) = r_i t + v_i(t + \Delta t) \cdot \Delta t, \tag{1.7}$$

де g – сила тяжіння;

Δt – крок часу.

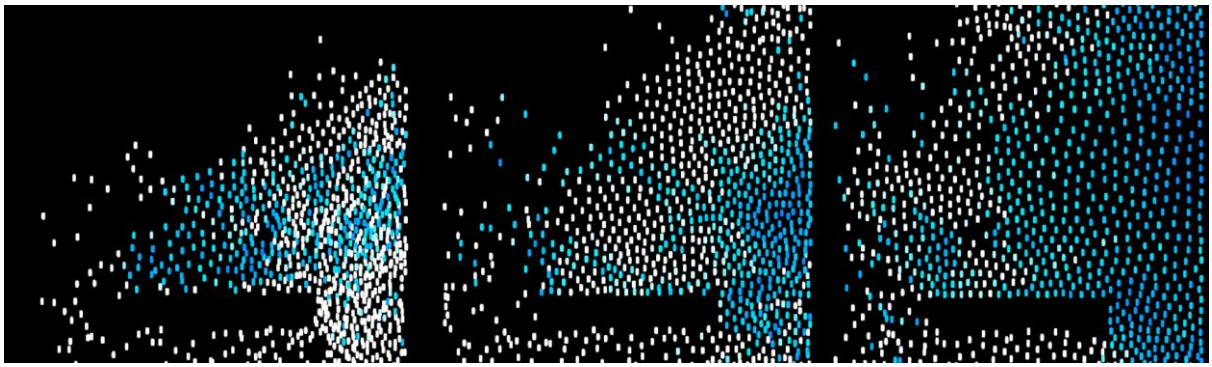

Рисунок 1.4 – Часова еволюція частинок у SPH-моделі

У наявних рішеннях, що базуються на методі згладжених частинок (SPH), обчислювальні ядра часто виносяться в окремі GPU-шейдери (рис. 1.5), що дозволяє ефективно виконувати розрахунки густини, тиску та сил взаємодії у паралельному режимі. Також поширеним є використання просторового хешування для пришвидшення пошуку сусідів, що сприяє зменшенню часової складності симуляції при великій кількості частинок.

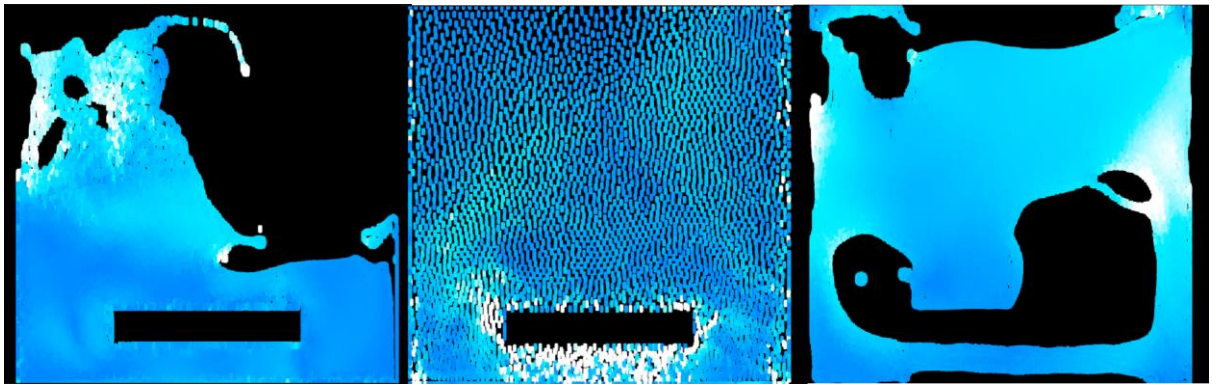

Рисунок 1.5 − Візуалізація результатів симуляції SPH з різними параметрами

Цикл обчислень повторюється для кожного кроку часу, утворюючи анімацію потоку рідини в реальному або наближеному до реального часі.

Метод SPH має низку важливих переваг. По-перше, він добре пристосований для складних геометрій, оскільки не вимагає явного визначення сітки. По-друге, він інтуїтивно зрозумілий та легко розширюється на багатофазні середовища, твердо-рідинні взаємодії, поверхневу напругу тощо. По-третє, SPH чудово масштабується в умовах паралельного обчислення, оскільки кожна частинка може оновлюватися незалежно, що робить метод придатним для реалізації на сучасних графічних процесорах.

Водночас метод має і свої виклики. Наприклад, складність пошуку сусідів у наївному вигляді має квадратичну залежність від кількості частинок. Для розв’язання цієї проблеми зазвичай застосовуються просторові структури даних, зокрема сітки, дерева або хеш-таблиці. Крім того, стабільність та точність симуляції значною мірою залежать від правильного вибору ядрових функцій (рис. 1.5), параметрів моделі та методу інтегрування.

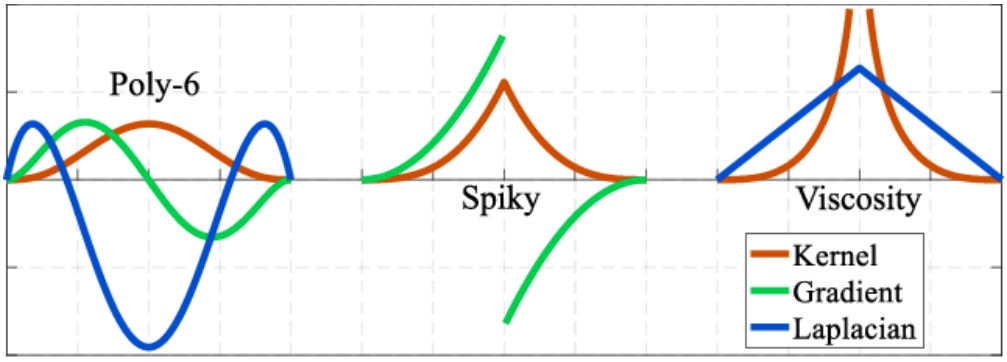

Рисунок 1.5 − Графік ядрових функцій Poly6, Spiky та Viscosity

Загалом, SPH є потужним інструментом для моделювання поведінки рідини, який поєднує фізичну достовірність, гнучкість та ефективність. Завдяки цим властивостям він набув широкого застосування в ігровій індустрії, віртуальній реальності, візуалізації фізичних процесів і навчальному моделюванні [3].

1.3 Фізичні моделі для симуляції рідини

У симуляціях на основі частинок, таких як SPH, продуктивність значною мірою залежить від ефективності пошуку сусідів. Оскільки сили взаємодії обчислюються лише для частинок, що знаходяться в межах певного радіусу, необхідно мати метод, який дозволяє швидко знаходити найближчих сусідів серед тисяч або десятків тисяч частинок. Перевірка кожної частинки з усіма іншими має часову складність $O(n^2)$, що є неприйнятним для реального часу. Для вирішення цієї проблеми використовуються методи просторової оптимізації – передусім рівномірна сітка (Uniform Grid) та просторове хешування (Spatial Hashing). Рівномірна сітка передбачає розбиття простору на регулярну сітку з фіксованим розміром комірки. Кожна частинка приписується до однієї з комірок на основі своїх координат. Для знаходження сусідів частинки потрібно перевірити лише ті частинки, що розміщені в поточній та суміжних комірках. Це суттєво зменшує кількість перевірок і дозволяє наблизитися до складності O(n).

Визначення індексу комірки у двовимірному просторі можна подати як

$$cellIndex_{\mathrm{x}} = \frac{x}{h} \vee, cellIndex_{\mathrm{y}} = \frac{y}{h} \vee, \tag{1.8}$$

де h – розмір комірки;

Просторове хешування є оптимізованим варіантом рівномірної сітки (рис. 1.6), який не потребує створення явної 2D-або 3D-таблиці. Замість цього використовується хеш-функція, яка для кожної частинки обчислює одновимірний індекс комірки

$$hash(x, y) = (x \cdot p_1 \oplus y \cdot p_2) mod N, \tag{1.9}$$

де p_1, p_2 – прості числа, що зменшують кількість колізій;

N – розмір хеш-таблиці;

⊕ – побітова операція XOR.

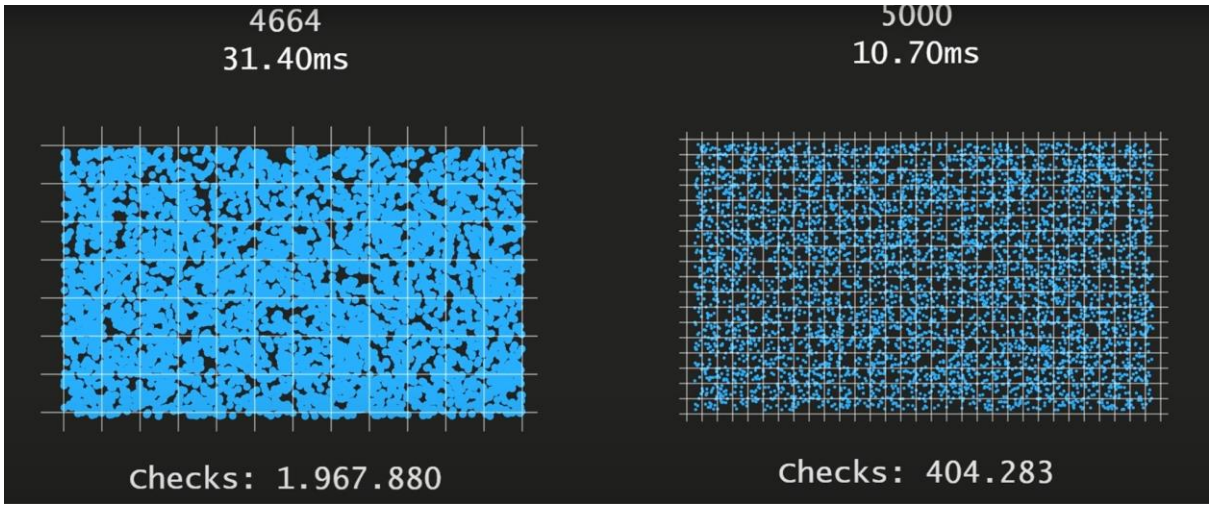

Рисунок 1.6 − Візуалізація розбиття простору на сітку та приклад хешування комірок

Однією з переваг просторового хешування є відсутність необхідності зберігати окремі масиви для кожної комірки: частинки розміщуються в одному спільному масиві, відсортованому за хеш-ключами. Для цього зазвичай використовуються GPU-алгоритми сортування, зокрема Count Sort, який ефективно виконує сортування частинок за ключами комірок на графічному процесорі [4].

Після сортування генерується так званий offset buffer, який містить початкові індекси частинок у кожній хешований комірці. Це дозволяє швидко знаходити всі частинки у заданій комірці, а отже – й ефективно здійснювати пошук сусідів для SPH-розрахунків.

Такі методи дозволяють зменшити обчислювальну складність симуляції до O(n) і забезпечити високу продуктивність навіть за великої кількості частинок. Просторове хешування розглядається як один із ключових підходів до оптимізації симуляцій частинок із можливістю масштабування та розширення фізичних моделей без суттєвої втрати швидкодії.

1.4 Графічні шейдери та їх застосування в рендерингу частинок

Графічні шейдери є важливою складовою сучасної комп’ютерної графіки, дозволяючи реалізовувати візуальні ефекти на GPU з високою продуктивністю. У контексті симуляції рідини шейдери відіграють ключову роль на етапі візуалізації – саме вони відповідають за відображення частинок, їхню форму, колір, прозорість та поведінку в просторі. Завдяки шейдерам можна в реальному часі створити привабливу, фізично обґрунтовану картинку, що відображає динаміку рідини.

Завдяки використанню шейдерів можливо в реальному часі створювати привабливу, фізично обґрунтовану візуалізацію, що точно відображає динаміку рідини. Для досягнення ефекту безперервного середовища, попри дискретну природу моделювання, зазвичай застосовуються текстури, альфа-змішування, м’які краєві переходи та градієнтне забарвлення частинок відповідно до їх параметрів − таких як густина або швидкість [5].

Ці візуальні прийоми дозволяють покращити сприйняття симуляції та зробити її більш наочною для користувача (рис. 1.7).

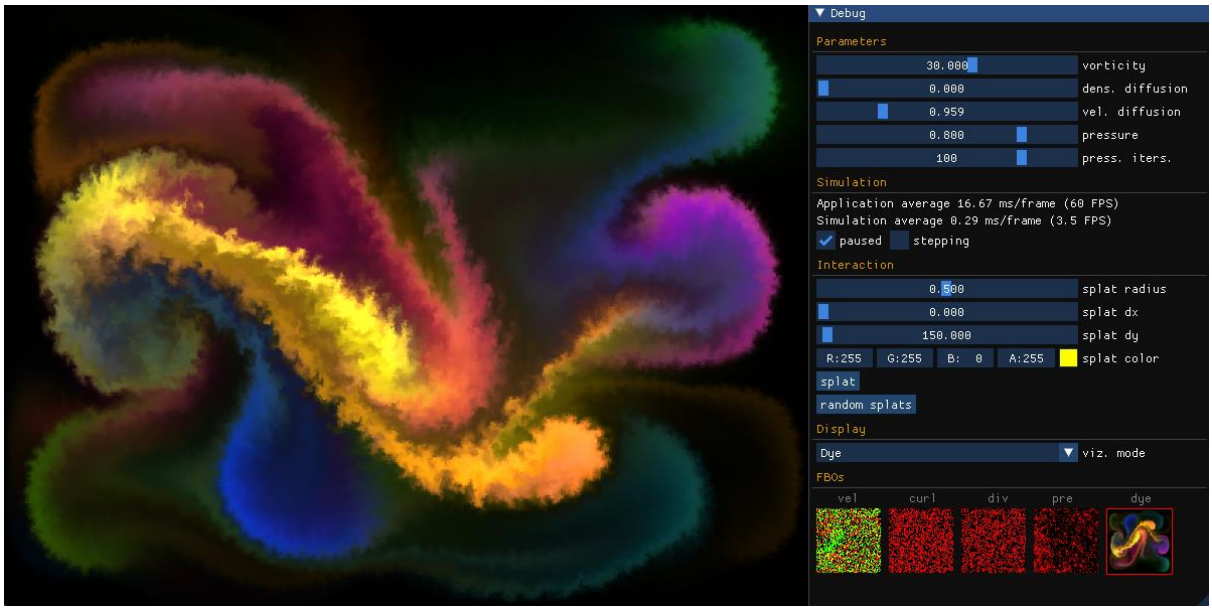

Рисунок 1.7 − Приклад візуалізації частинок рідини з використанням custom-шейдера

Основні етапи рендерингу частинок у шейдері включають:

– побудову геометрії частинки – зазвичай це точка, яка у віртуальному просторі розтягується до круга або еліпса;

– обчислення кольору залежно від фізичних властивостей – густини, тиску або швидкості (передаються через буфери або як текстури);

– альфа-змішування (alpha blending) для створення ефекту прозорості та візуального об’єднання частинок;

– застосування додаткових ефектів – таких як глоу, edge-blur або кольорове кодування;

– отримання позиції частинки з GPU-буфера.

Для 2D-візуалізацій часто використовується графічний pipeline типу Vertex/Fragment, який добре підходить для рендерингу великої кількості об’єктів у реальному часі. У подібних симуляціях частинки зазвичай відображаються як billboard-об’єкти − тобто такі, що завжди орієнтовані в бік камери, навіть у 2D-сценах. Їхній розмір може масштабуватися динамічно, наприклад, залежно від значень у буфері, що дозволяє візуалізувати фізичні характеристики частинок, такі як тиск або щільність (рис. 1.8).

Для передачі даних з симуляції до графічного шейдера застосовуються структуровані GPU-буфери (StructuredBuffer), які містять позиції частинок, а також додаткові атрибути, такі як тиск або маса. Ці буфери передаються шейдеру за допомогою рендереру частинок, який відповідає за підключення ресурсів, оновлення матеріалів та виклик команд рендерингу [5].

У випадку великої кількості частинок (тисячі й більше) для зменшення навантаження на графічний процесор доцільно використовувати інстансінг (GPU instancing) − технологію, яка дозволяє рендерити велику кількість однакових об’єктів за одну GPU-команду. Це особливо ефективно для простих геометричних форм, таких як точки або краплі рідини.

Завдяки цим методам графічні шейдери не лише створюють привабливу візуальну картину, а й дають змогу в реальному часі відображати динамічні характеристики фізичної моделі. Вони виступають завершальним етапом обчислювального конвеєра, трансформуючи чисельні розрахунки у візуально зрозумілу сцену.

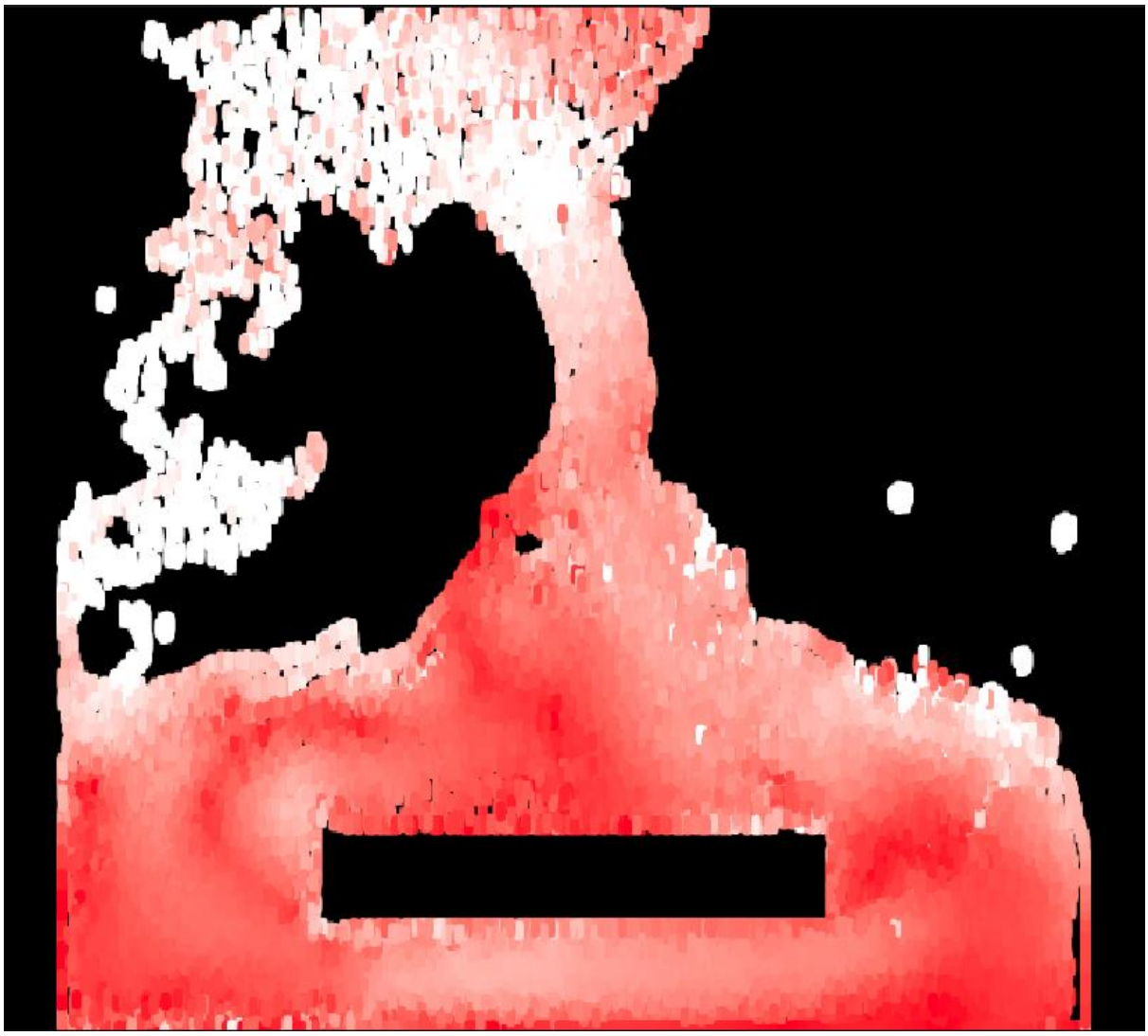

Рисунок 1.8 − Шейдерне розфарбування частинок за тиском (від красного до білого)

1.5 Огляд популярних платформ для реалізації симуляцій

Реалізація фізичних симуляцій вимагає поєднання високопродуктивних обчислень та гнучких засобів візуалізації. Сучасні програмні платформи, такі як Unity, Unreal Engine, Godot та OpenGL, надають розробникам різні підходи та інструменти для створення інтерактивних симуляцій, включаючи симуляцію рідин, частинок та інших фізичних явищ. У цьому розділі розглянуто ключові особливості кожної з платформ.

Unity – один із найпопулярніших ігрових рушіїв, орієнтований на кросплатформну розробку. Його головною перевагою є зручність використання, підтримка C#, а також потужна система обробки графіки та фізики. Unity підтримує обчислення на GPU через Compute Shaders (HLSL), що робить його придатним для складних симуляцій у режимі реального часу. Існує також підтримка URP та HDRP для складної візуалізації. У межах проєкту реалізація SPH-симуляції була здійснена саме в Unity, з активним використанням compute shaders, структурованих буферів і кастомних графічних шейдерів.

Unreal Engine – потужний рушій з високоякісною графікою та вбудованою фізикою (PhysX/Chaos). Він базується на C++ та Blueprints, що дозволяє створювати як низькорівневі, так і візуально запрограмовані симуляції. Unreal має нативну підтримку Niagara – системи частинок нового покоління, яка дозволяє створювати складні симуляції з GPU-обробкою та інтеграцією з іншими компонентами рушія. Платформа підходить для високоякісної графіки та великих проектів, однак має вищий поріг входу у порівнянні з Unity. Також Unreal Engine та Unity мають схожий принцип побудови сцени (рис. 1.9).

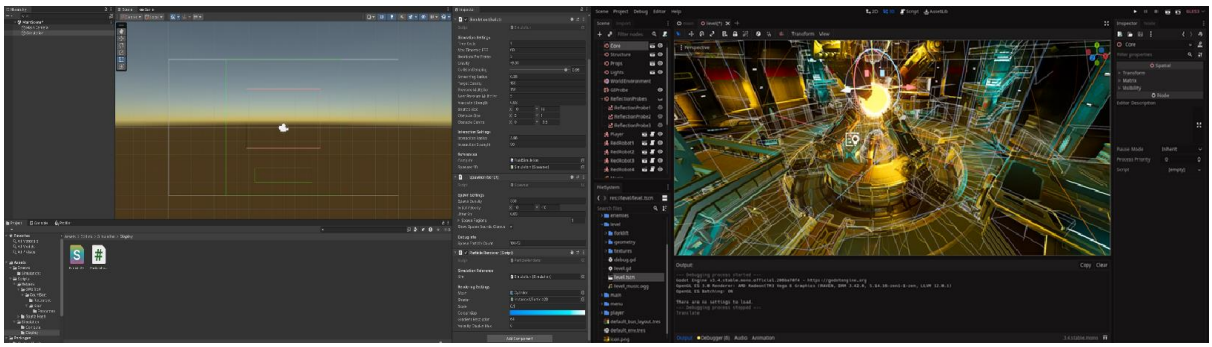

Рисунок 1.9 − Порівняння інтерфейсів Unity та Godot

Godot Engine – незалежний рушій з відкритим вихідним кодом, що активно розвивається та має все ширшу спільноту. Godot підтримує GDScript, C#, C++, та має вбудовану фізику, проте його можливості GPU-обчислень обмежені в порівнянні з Unity або Unreal. У Godot 4.x з'явилася підтримка Compute Shaders (використовуючи Vulkan), що відкриває нові можливості для реалізації частинкових симуляцій. Проте на момент написання даної роботи ця функціональність все ще вважається експериментальною і має обмеження щодо продуктивності та інструментарію налагодження.

OpenGL – низькорівнева графічна бібліотека, що надає повний контроль над процесом рендерингу та обчисленнями на GPU. Вона дозволяє реалізовувати compute shaders, VBO, FBO та інші механізми, що дають змогу створити кастомну симуляцію з нуля. Основна перевага OpenGL – це повна гнучкість і незалежність від рушіїв, однак вона вимагає глибокого знання графічного пайплайну (рис. 1.10) та обробки буферів вручну. Вибір OpenGL доцільний у дослідницьких або академічних проєктах, де потрібно експериментувати з нетиповими алгоритмами або досягати максимальної оптимізації.

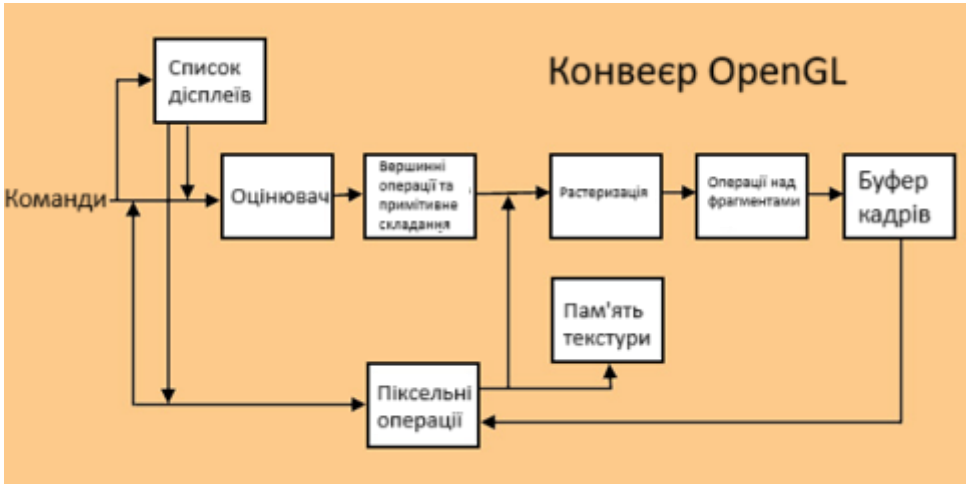

Рисунок 1.10 − Архітектура обчислювального пайплайну в OpenGL

Таким чином, вибір платформи для симуляції залежить від пріоритетів проєкту: продуктивності, складності реалізації, графічної якості, рівня доступу до GPU та гнучкості коду. Однією з популярних платформ, що поєднує доступність, потужні інструменти GPU-програмування, підтримку compute shaders, а також має активну спільноту з великою кількістю прикладів і документації, є Unity.

1.6 Постановка задачі

Симуляція фізичних процесів, зокрема поведінки рідини, у режимі реального часу потребує поєднання точних чисельних методів, ефективних алгоритмів оптимізації та високопродуктивної графіки. З огляду на активний розвиток візуальних технологій, потреба у наочному моделюванні частинкових середовищ зростає не лише у сфері комп’ютерних ігор, а й у наукових, освітніх та дослідницьких застосуваннях.

У межах роботи розглядається розробка ефективного інструменту для моделювання та візуалізації рідини в реальному часі, що базується на методі згладженої гідродинаміки частинок (SPH). Реалізація виконується у

середовищі Unity з використанням обчислювальних можливостей графічного процесора (GPU).

Основна увага приділяється алгоритмам і технологіям, які дозволяють забезпечити реалістичну поведінку та зображення частинок у двовимірному просторі..

Для досягнення поставленої мети необхідно реалізувати такі основні завдання:

– побудова фізичної моделі на основі SPH, адаптованої для двовимірного простору;

– реалізація просторової оптимізації за допомогою хешування (spatial hashing) для ефективного пошуку сусідів;

– використання compute shaders для виконання основних обчислень на GPU (розрахунок густини, тиску, сил, оновлення положення частинок);

– реалізація GPU-сортування частинок у комірках за допомогою алгоритмів Count Sort та prefix sum (scan);

– створення кастомного графічного шейдера для візуалізації рідини на основі атрибутів частинок;

– забезпечення гнучкої архітектури проєкту з можливістю масштабування та подальшого розширення.

У межах цієї роботи увагу зосереджено на досягненні балансу між фізичною достовірністю симуляції та її продуктивністю на сучасному споживчому обладнанні. Результати мають бути придатними для використання в інтерактивних застосунках, демонстраціях або освітньому контексті.

2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ ДЛЯ СИМУЛЯЦІЇ ТА ВІЗУАЛІЗАЦІЇ РІДИНИ

2.1 Вибір середовища розробки та рендерингового рушія

Успішна реалізація симуляції рідини в режимі реального часу потребує вибору відповідного середовища розробки, яке забезпечує не лише зручну архітектуру, а й доступ до низькорівневих ресурсів графічного процесора. Основними критеріями вибору платформи для даного проєкту стали: підтримка обчислень на GPU, гнучкість у налаштуванні пайплайнів, наявність засобів для роботи з шейдерами, широке ком'юніті та наявність документації.

На основі порівняльного аналізу сучасних платформ було обрано Unity як основний рендеринговий рушій і середовище розробки. Unity поєднує в собі підтримку об'єктно-орієнтованого програмування на C#, вбудовані засоби 2D та 3D-графіки, доступ до обчислювальних шейдерів (compute shaders), а також має розвинену інфраструктуру для побудови масштабованих інтерактивних додатків.

Однією з ключових переваг Unity є можливість безпосередньої інтеграції GPU-обчислень у вигляді .compute-файлів, що дає змогу реалізовувати складні алгоритми (наприклад, SPH або сортування частинок) на низькому рівні продуктивності. Всі основні компоненти симуляції (фізична модель, сортування, хешування, відображення) реалізовані в Unity за допомогою скриптів C# у поєднанні з шейдерами.

Розгортання проєкту в Unity дозволить також скористатися перевагами модульної структури: симуляційна частина, рендеринг і допоміжні утиліти були чітко розділені на просторові модулі (папки Simulation, Helpers, Display), що полегшує розширення системи та подальше обслуговування.

Крім того, Unity має розвинену систему профілювання та візуалізації продуктивності (рис. 2.1), що дозволяє відстежувати час виконання шейдерів, навантаження на GPU та ефективність передачі даних між CPU і GPU – це особливо критично для симуляцій з великою кількістю частинок.

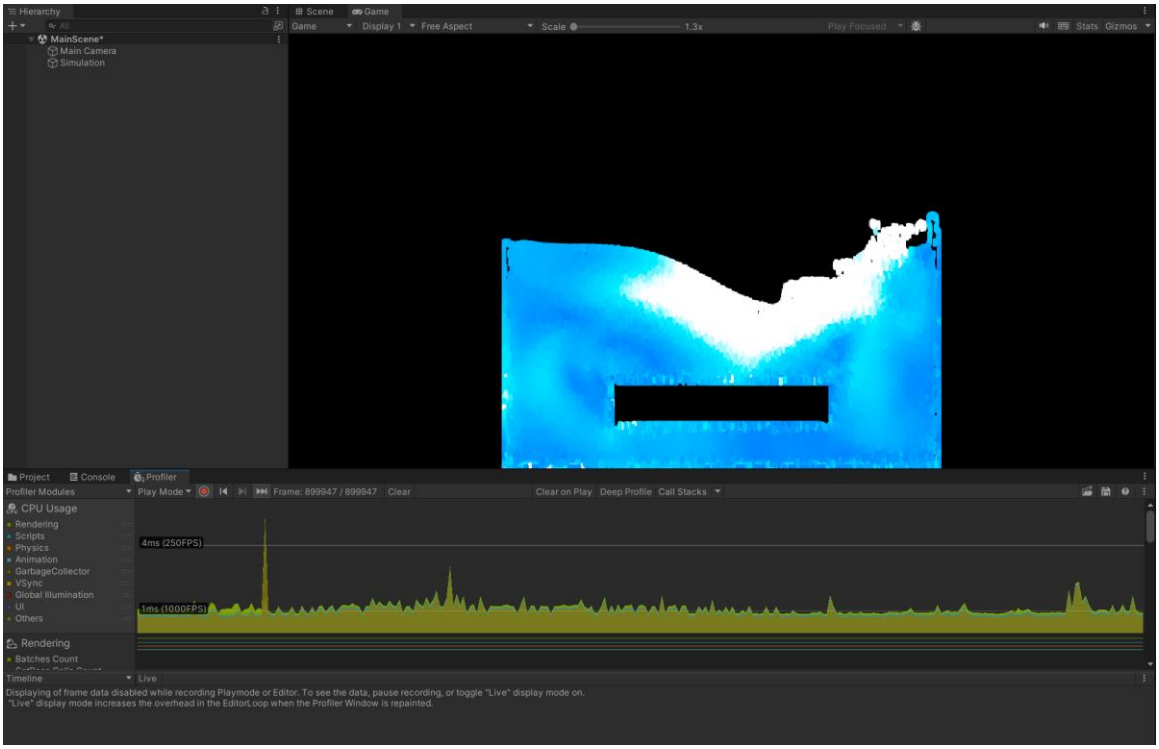

Рисунок 2.1 − Інтрефейс Unity при профілюванні

Таким чином, вибір Unity як основного інструменту для реалізації даного проєкту забезпечує оптимальне поєднання зручності розробки, обчислювальної потужності та візуальної гнучкості. Його використання дозволяє повністю реалізувати цілі проєкту та забезпечити стабільну, масштабовану симуляцію в реальному часі.

2.2 Архітектура проєкту: основні модулі та структура

Архітектура проєкту побудована таким чином, щоб забезпечити чітке розділення відповідальностей між обчислювальними, візуалізаційними та

допоміжними компонентами. Такий підхід дозволяє не лише спростити розробку, а й забезпечити можливість подальшого масштабування, тестування та повторного використання окремих частин системи (рис.2.2). Проєкт реалізовано в середовищі Unity з використанням мови програмування C# та шейдерів.

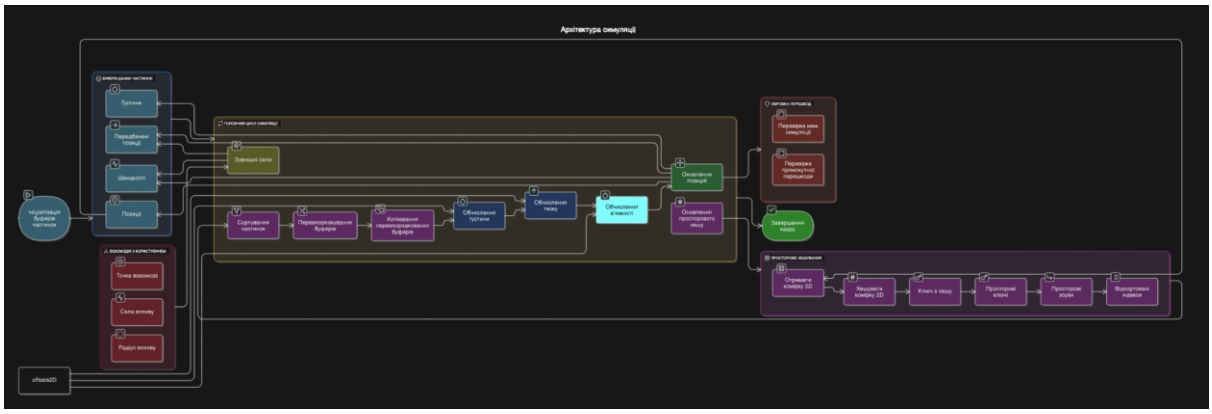

Рисунок 2.2 − Схема архітектури симуляції: компоненти симуляції, хешування, сортування

Фізична симуляція рідини на основі методу SPH винесена в окрему структуру каталогів та скриптів, логічно поділених за функціональністю. Основні модулі можна класифікувати наступним чином:

Модуль Simulation. Цей модуль містить основну логіку симуляції, включаючи оновлення станів частинок, керування життєвим циклом симуляції, запуск та синхронізацію обчислень на GPU.

До складу входять такі ключові компоненти:

– Simulation.cs керує всім процесом симуляції: ініціалізація буферів, виклики обчислювальних шейдерів, оновлення положень частинок;

– Spawner.cs відповідає за генерацію нових частинок та початкові умови симуляції;

– папка Compute містить ядра симуляції:

– FluidSimulation.compute є основними ядрами симуляції SPH: сили, щільність, в’язкість, хешування, зіткнення, оновлення;

– FluidMaths2D.hlsl містить згладжувальні ядра Poly6 і Spiky та їх похідні для обчислення сил у SPH-симуляціях;

– SpatialHash.hlsl є функціями для просторового хешування частинок: обчислення сіткових комірок, генерація хешів і ключів для швидкого пошуку сусідів.

Модуль Helpers. Допоміжні утиліти та алгоритми низького рівня. Його компоненти реалізують GPU-сортування, побудову хеш-таблиці та підготовку offset-буферів для пошуку сусідів:

– ComputeHelper.cs є зручним інтерфейсом для роботи з compute shaders: запуск, передача параметрів, зчитування буферів;

– GPUCountSort.cs, Scan.cs є реалізацією сортування та префіксної суми на GPU;

SpatialHash.cs, SpatialOffsetCalculator.cs для обчислення хеш-ключів і зсувів для прискореного пошуку.

Модуль Display. Відповідає за візуалізацію частинок у сцені.

Зокрема:

– ParticleRenderer.cs отримує буфери частинок та передає їх графічному шейдеру;

– Particle2D.shader є кастомним шейдером для виводу частинок з ефектами прозорості та градієнтів.

Папка Scenes. Містить сцену MainScene.unity, у якій розташовані всі об’єкти, необхідні для запуску симуляції: камери, рендерери, контролери часу, інтерфейс користувача тощо.

Загалом архітектура реалізована таким чином, що кожен модуль ізолює власну відповідальність, а взаємодія між ними здійснюється через чітко визначені інтерфейси (рис. 2.3). Комунікація між CPU та GPU організована через ComputeBuffers, а логіка оновлення та візуалізації виконується в рамках циклу Update або LateUpdate Unity.

Такий підхід дозволяє досягти високої продуктивності, зберігаючи при цьому модульність і зручність розробки. Усі ключові етапи симуляції – від

створення частинок до їх відображення на екрані – реалізовані у вигляді окремих, повторно використовуваних компонентів.

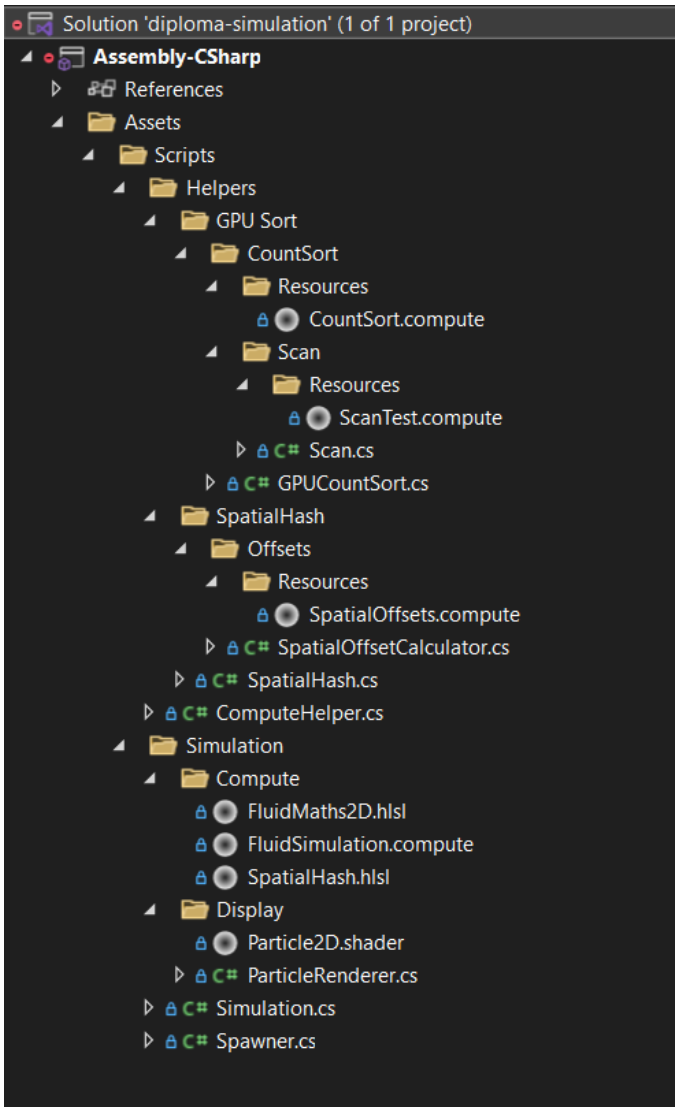

Рисунок 2.3 − Структура папок проєкту в Unity

2.3 Проєктування обчислювального ядра симуляції на GPU

Для досягнення високої продуктивності при моделюванні динаміки рідини використовується обчислення на графічному процесорі за допомогою compute shaders. У рамках цього проєкту обчислювальне ядро реалізовано у вигляді окремого .compute-файлу – FluidSimulation.compute, що виконує всі основні розрахунки симуляції SPH: оновлення густини, тиску, сил та переміщення частинок.

Обчислення поділено на окремі kernel−функції, кожна з яких відповідає за певний етап симуляції. Такий поділ дозволяє ефективно керувати ресурсами GPU, уникати надмірної конкуренції за пам’ять і забезпечувати правильну послідовність виконання.

Перший крок – обчислення густини кожної частинки на основі її оточення. Густина визначається формулою:

$$p_i = \Sigma_j m_j \cdot W(r_i - r_j, h), \tag{2.1}$$

де W − згладжувальна функція (Spiky або Poly6);

h – радіус впливу.

Отримані значення густини передаються у функцію обчислення тиску:

$$p_i = k \cdot (p_i - p_o), \tag{2.2}$$

Цей етап реалізовано в kernel-функції DensityPressureKernel, яка паралельно обробляє всі частинки, використовуючи Spatial Hashing для обмеження кола перевірок лише найближчими комірками [1].

Далі обчислюється сумарна сила, що діє на кожну частинку. Основні компоненти:

$$f_{ij} = k \cdot (p_i - p_o), \tag{2.2}$$

– сила тиску:

$$f_{ij}^{pressure} = -m_j \cdot \frac{2}{p_j} + \frac{p_i + p_j}{2} \cdot \nabla W(r_i - r_j, h), \tag{2.3}$$

– сила в’язкості:

$$f_{ij}^{viscosity} = \mu \cdot \frac{m_j}{p_j} \cdot (v_j \cdot v_i) \cdot \nabla^2 W(r_i - r_j, h), \quad (2.4)$$

Розрахунок реалізовано в ForcesKernel, який застосовує сили до буфера швидкостей.

Завершальний етап симуляції – оновлення стану кожної частинки згідно з рівняннями руху

$$v_i(t + \Delta t) = v_i(t) + a_i \cdot \Delta t, \quad (2.5)$$

$$r_i(t + \Delta t) = r_i(t) + v_i(t + \Delta t) \cdot \Delta t, \quad (2.6)$$

Оновлення виконується у IntegrationKernel, що враховує гравітацію, сили та межі екрану – для відбиття частинок від країв сцени [6].

Для зберігання інформації про частинки використовуються StructuredBuffers – масиви: позиції, швидкості, тиску, густини, маси, проміжних значень.

Кожна kernel-функція працює у паралельному потоці над окремою частинкою (*1 thread = 1 particle*), що дозволяє повністю використати переваги паралельної архітектури GPU [8].

Запуск compute shaders здійснюється через метод Dispatch, який викликається з CPU-скрипту Simulation.cs, попередньо задаючи параметри, буфери та розмір thread-груп.

З метою досягнення стабільної продуктивності реалізовано:

- обмеження зони пошуку за допомогою spatial hashing;
- обчислення у float2 (2D) замість float3;
- повторне використання буферів між kernel-функціями;
- уникнення атомарних операцій та блокуючих синхронізацій.

Таким чином, обчислювальне ядро симуляції реалізовано як повністю GPU-орієнтоване рішення, що дозволяє обробляти тисячі частинок з високою частотою оновлення в реальному часі.

2.4 Створення системи просторового хешування

Одним з найважливіших аспектів оптимізації симуляцій на основі частинок є ефективний пошук сусідів. Оскільки метод SPH передбачає обчислення фізичних величин на основі взаємодії з частинками в межах певного радіусу, необхідно обмежити перевірку лише тими елементами, що справді потенційно можуть впливати. У цьому проекті для цього реалізовано систему просторового хешування (Spatial Hashing) – техніку, що дозволяє швидко знаходити сусідні частинки [8], розподіляючи їх у комірки рівномірної сітки.

Суть методу полягає у розбитті простору на дискретні комірки фіксованого розміру (зазвичай трохи більшого за радіус впливу частинки h). Кожна частинка приписується до відповідної комірки, і всі обчислення проводяться лише з тими частинками, що знаходяться в поточній або сусідніх комірках.

У двовимірному просторі положення частинки $r_i = (x_i, y_i)$ перетворюється в дискретні координати комірки [7]

$$c_x = \lfloor hxi \rfloor, c_y = \lfloor hyi \rfloor, \tag{2.7}$$

Далі, за допомогою хеш-функції ці координати перетворюються у одновимірний індекс [7]

$$hash(c_x, c_y) = ((c_x \cdot p_1) \oplus (c_y \cdot p_2)) mod N, \tag{2.8}$$

де p_1 і p_2 – великі прості числа;

N – кількість комірок;

$\oplus$ – побітова XOR-операція.

Це дозволяє зберігати дані у лінійних масивах без створення двовимірної таблиці.

Дана процедура здійснюється за наступною схемою:

Крок 1. Обчислення хеш-ключів. У шейдері SpatialOffsets.compute кожна частинка паралельно обробляється з метою обчислення хеш-ключа за її позицією.

Крок 2. Сортування частинок. Після створення хеш-ключів частинки сортуються за ключем за допомогою GPUCountSort.cs. Це дозволяє згрупувати всі частинки однієї комірки разом.

Крок 3. Формування offset-буфера. Наступний крок – створення offset-таблиці (start indices), яка зберігає початковий індекс у відсортованому масиві для кожного унікального хеш-ключа. Для цього реалізовано префіксну суму (scan), що дозволяє швидко визначити межі кожної групи частинок.

Крок 4. Пошук сусідів. Під час обчислення густини, тиску та сил, для кожної частинки виконується перевірка не всіх nnn частинок, а лише тих, що знаходяться в 9-ти комірках: поточній і восьми суміжних [4]. Це зменшує кількість перевірок із O(n2) до майже O(n), що критично важливо при великих об’ємах симуляції.

У структурі проєкту реалізація просторового хешування охоплює кілька ключових компонентів:

– SpatialHash.cs – керування логікою хешування на CPU, виділення буферів, ініціалізація shader-параметрів;

– SpatialOffsetCalculator.cs – модуль для обчислення offset-таблиці;

– SpatialOffsets.compute – GPU-шейдер, який реалізує обчислення хешів, зсувів і зберігання даних у буфери.

Планується розробка ефективної системи просторової адресації частинок на основі хешування, яка дозволить масштабувати симуляцію до десятків тисяч елементів без істотної втрати продуктивності (рис.2.4).

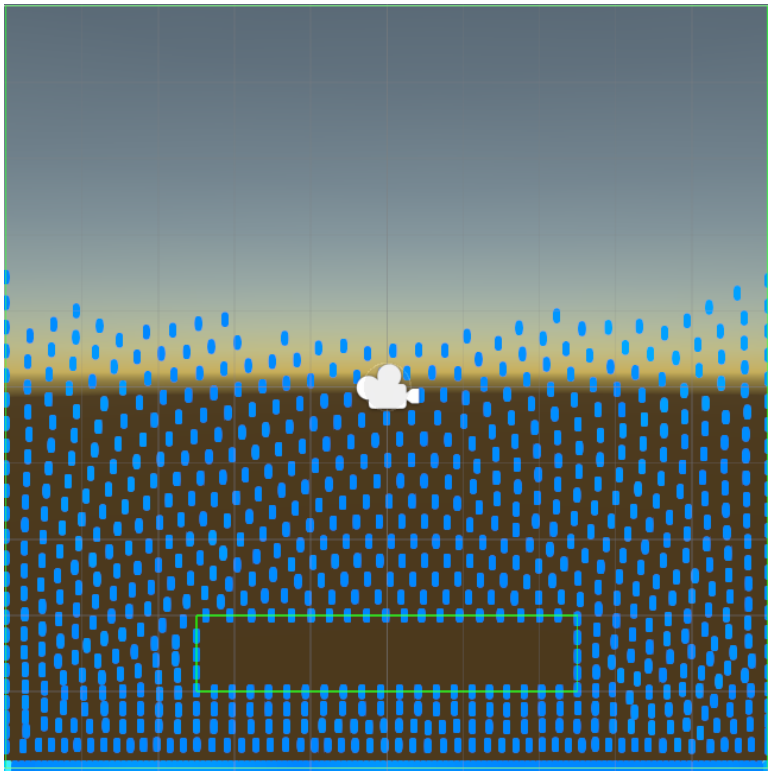

Рисунок 2.4 − Візуалізація розподілу частинок у просторовій сітці

2.5 Інтеграція рендерингу частинок за допомогою шейдерів

Візуалізація частинок у симуляції рідини є ключовим компонентом для досягнення переконливої та інформативної графіки. З метою забезпечення ефективного відображення великої кількості частинок із врахуванням їхніх фізичних параметрів − положення, швидкості та густини − у проєкті передбачається використання графічних шейдерів у поєднанні з GPU-базованим підходом до інстансингу.

Планується створення спеціалізованого шейдера Particle2D.shader, який використовуватиме параметри, передані з CPU-сторони, зокрема буфери позицій (Positions2D), швидкостей (Velocities) та густин (DensityData). Ці

буфери оновлюватимуться кожен кадр і передаватимуться до шейдера через матеріал, що створюється у відповідному компоненті ParticleRenderer [9].

Компонент ParticleRenderer відповідає за підготовку матеріалів, текстур та буферів інстансингу. Для відображення частинок використовується метод Graphics.DrawMeshInstancedIndirect, який дозволяє рендерити велику кількість об'єктів за допомогою одного GPU-виклику. Це критично важливо для забезпечення високої продуктивності при симуляціях із тисячами частинок.

Кольорова карта (Gradient) використовується для візуалізації швидкості кожної частинки, що дозволяє користувачу інтуїтивно оцінити динаміку рідини. Вона перетворюється у текстуру за допомогою статичної функції TextureFromGradient, що генерує 1D-текстуру кольорів і передає її в шейдер [11].

У шейдері обчислюється остаточний колір частинки, який базується на її нормалізованій швидкості відносно максимального значення, заданого параметром velocityMax. Це дозволяє адаптувати візуалізацію до різних масштабів руху у симуляції. Також враховується масштаб частинок, який контролюється параметром scale (рис. 2.5).

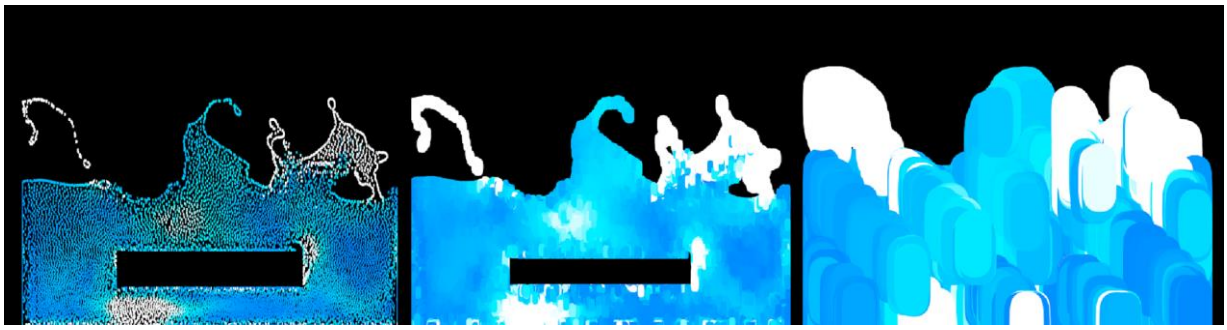

Рисунок 2.5 − Візуалізація частинок із різним значенням параметра scale

У результаті реалізована система рендерингу частинок дозволяє гнучко та ефективно візуалізувати як структурні, так і динамічні властивості рідинної симуляції (рис. 2.6), зберігаючи при цьому високу продуктивність на сучасних графічних процесорах.

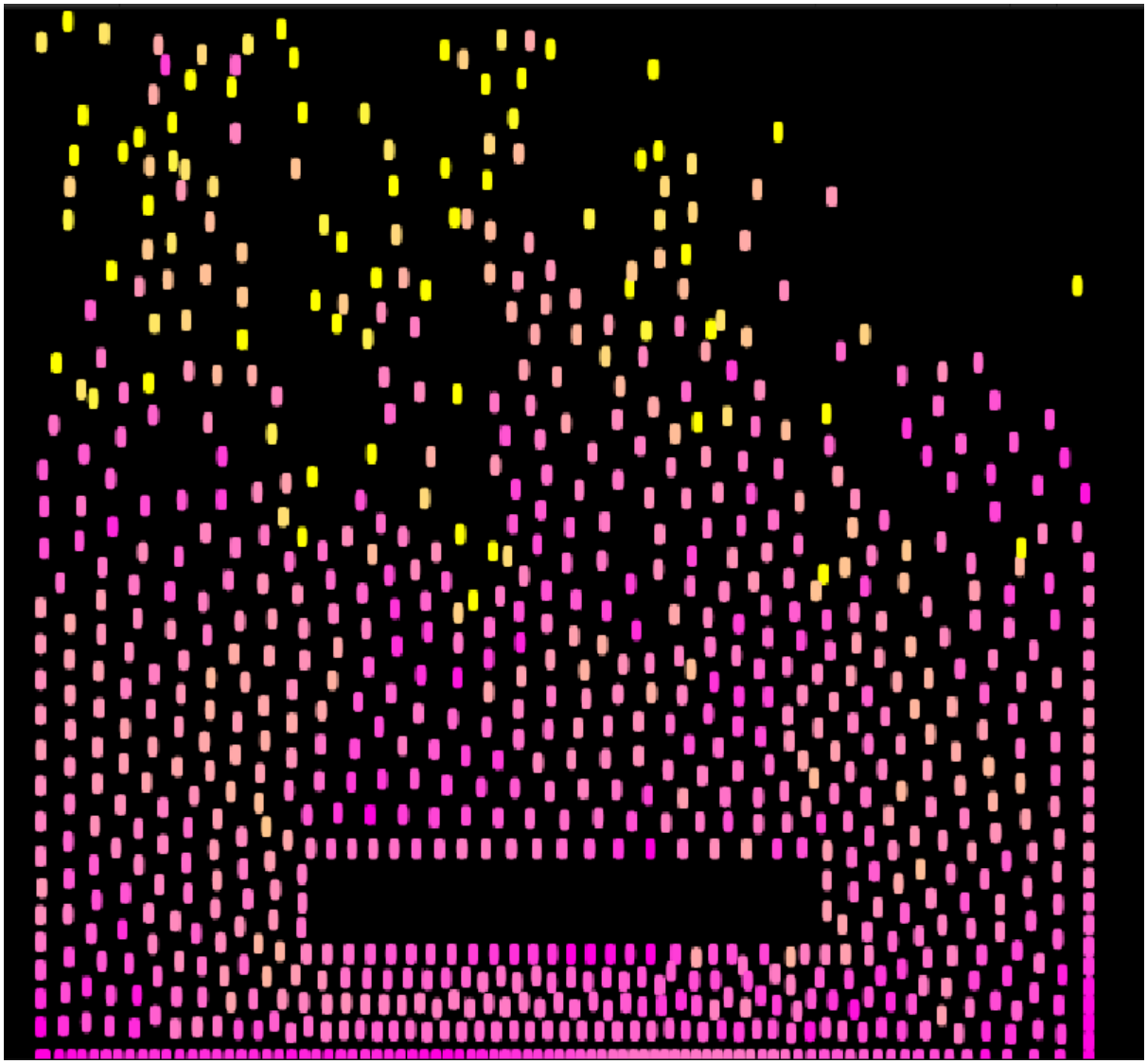

Рисунок 2.6 − Візуалізація частинок із кольоровим градієнтом швидкості (від пурпурного до жовтого)

2.6 Відлагодження та візуалізація результатів

У процесі розробки симуляції рідини важливою складовою є не лише реалізація фізичної моделі, а й побудова інструментів для візуального аналізу й відлагодження. Враховуючи складність обчислень, що виконуються на GPU, класичні методи налагодження через дебагери виявляються

недостатніми, тому велика увага приділялася саме візуальній інспекції результатів.

Одним із ключових інструментів є візуалізація буферів частинок у реальному часі, яку планується реалізувати через ParticleRenderer. Кольорова індикація швидкості частинок за допомогою градієнтної текстури дозволяла легко ідентифікувати аномальні значення – наприклад, занадто високі швидкості або статичні скупчення частинок. Це дає змогу оперативно виявляти проблеми в обчисленнях сили тиску, в’язкості чи гравітації [12].

Для забезпечення коректної роботи просторової структури симуляції передбачається використання візуалізаційних інструментів Unity, зокрема параметрів Gizmos. Під час тестування планується реалізувати виведення інформації про просторове хешування, включаючи графічне представлення комірок та індикацію кількості частинок у кожній з них. Це дозволить оцінити правильність просторового поділу сцени та ефективність пошуку сусідів у контексті алгоритму SPH.

Оскільки симуляція базується на обчислювальних шейдерах (ComputeShader), для перевірки їхньої коректності буде застосовано зчитування проміжних буферів, зокрема буферів густини та сили тиску. Отримані дані планується аналізувати за допомогою статистичних характеристик − таких як середні значення, мінімальні та максимальні відхилення − для виявлення аномалій або помилок у розрахунках.

Для тестування поведінки симуляції в умовах нестандартних ситуацій передбачається використання спеціального інструменту Spawner, який дозволяє вручну додавати частинки до сцени. Це забезпечить можливість моделювання сценаріїв з коливаннями тиску, зіткненням потоків або витіканням частинок за межі області симуляції.

Таким чином, поєднання візуальних засобів налагодження, аналізу буферів та ручного тестування стане основою для верифікації коректності та стабільності симуляції за різних умов (рис. 2.7).

Рисунок 2.7 − Приклад візуалізації аномальної густини частинок у просторі

3 РЕАЛІЗАЦІЯ СИМУЛЯЦІЇ РІДИНИ В UNITY

3.1 Створення сцени симуляції

Сцену симуляції було створено вручну в середовищі Unity, без використання стандартних шаблонів. Основу сцени становить об'єкт Simulation, до якого прив’язані ключові компоненти керування симуляцією:

– Simulation.cs для реалізації основної логіки фізичних обчислень на GPU;

– Spawner.cs відповідає за створення та ініціалізацію частинок;

– ParticleRenderer.cs забезпечує візуалізацію частинок із використанням шейдера Particle2D.shader.

Область симуляції визначається в Simulation.cs за допомогою параметрів, що задаються через Serialized Fields. Це дає змогу задавати розміри сцени, кількість частинок, параметри фізичної взаємодії (масу, в’язкість, радіус згладжування тощо) без потреби змінювати код – достатньо налаштувати відповідні поля у вікні Inspector (рис.3.1). Такий підхід спрощує експериментування з параметрами симуляції та прискорює налагодження.

Для зручної візуалізації меж області симуляції та налагодження обробки зіткнень використовувалися Gizmos. Вони дозволяли в редакторі Unity побачити граничні лінії області, що дало змогу оцінити коректність роботи граничних умов та відштовхування частинок від стін (рис.3.2).

У сцені також наявна MainCamera, яка зафіксована під кутом згори та забезпечує стабільне двовимірне відображення всієї області симуляції. Вся сцена зосереджена в одному файлі – MainScene.unity (рис.3.3), що міститься у директорії Scenes/Simulations/.

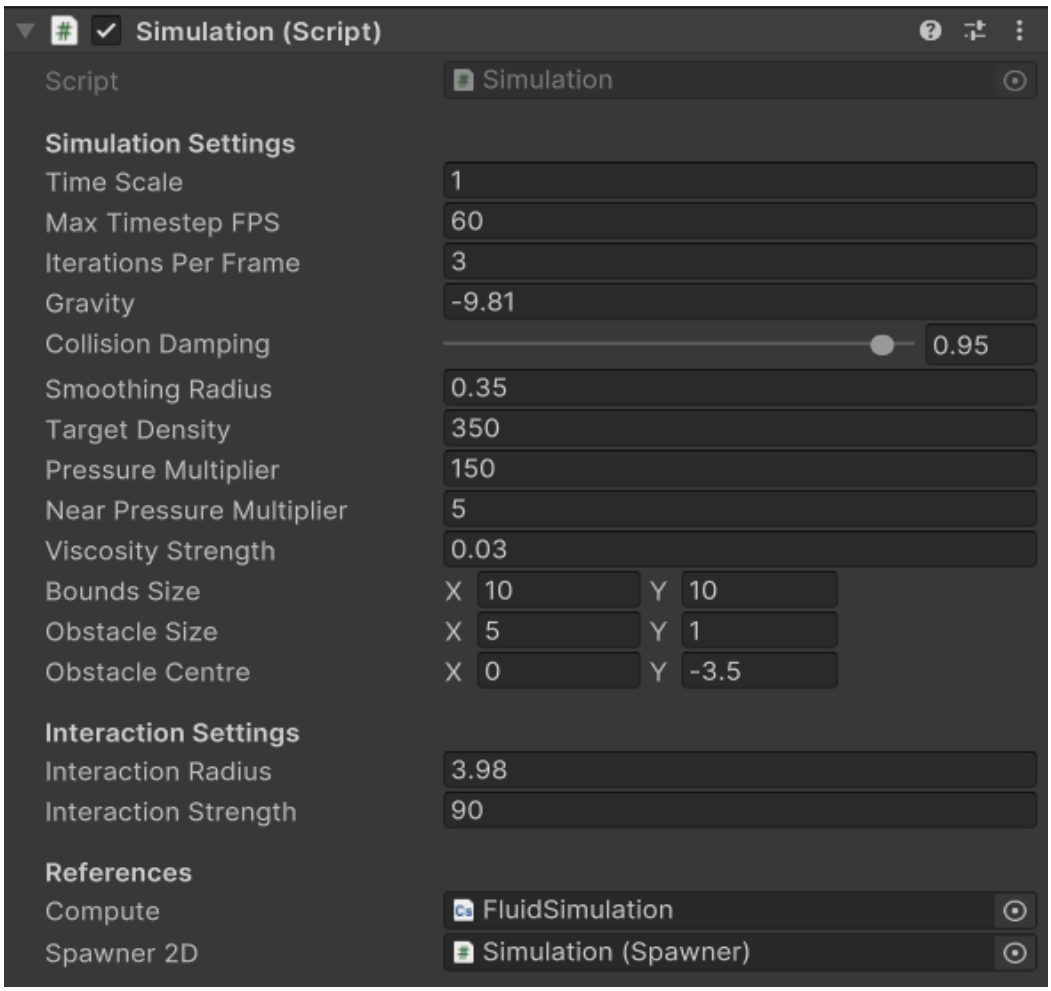

Рисунок 3.1 – Приклад налаштувань компонентів Simulation через Serialized Fields у Unity

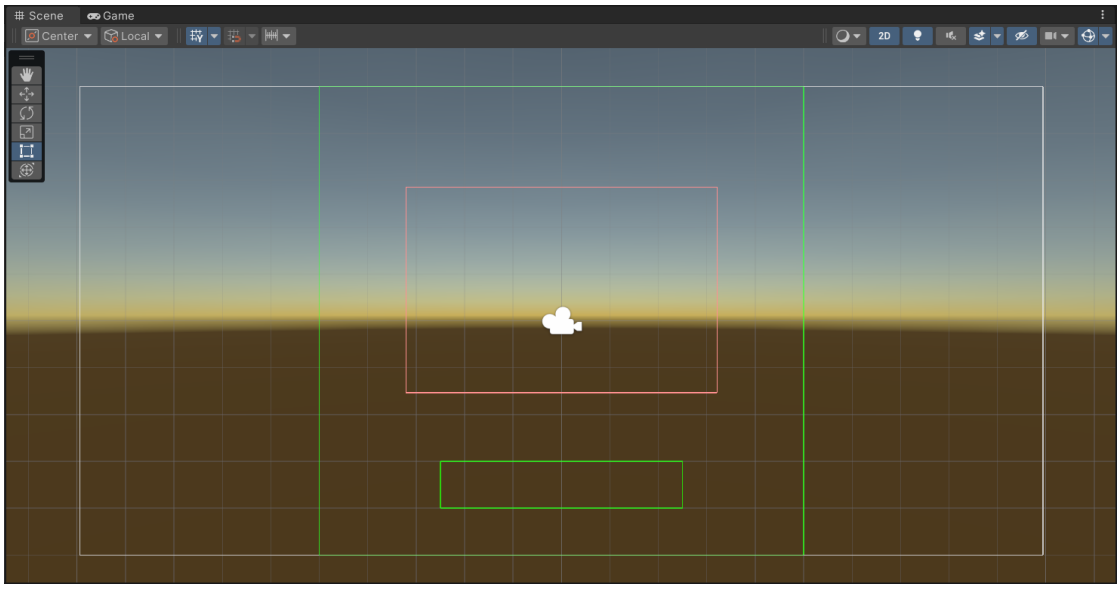

Рисунок 3.2 - Приклад Gizmos у головній сцені

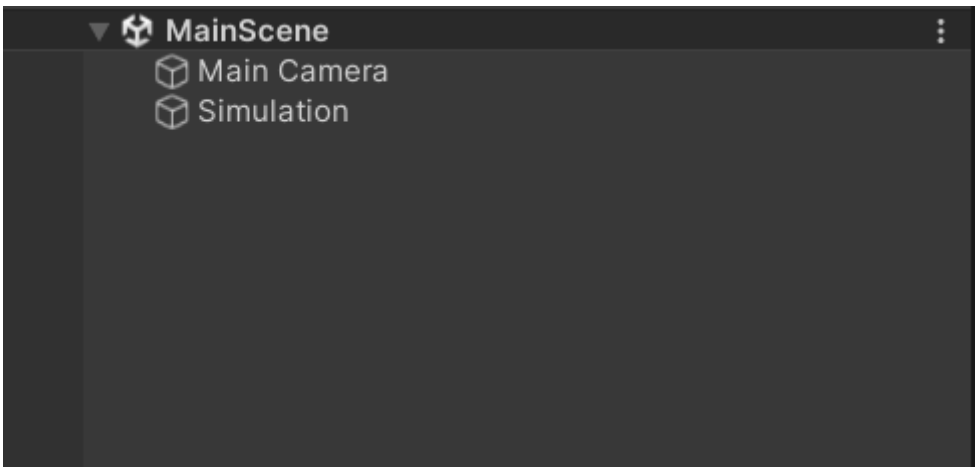

Рисунок 3.3 − Загальний вигляд сцени симуляції в Unity

Генерація частинок виконується автоматично або вручну через клас Spawner, що дозволяє змінювати початкові умови, зокрема конфігурацію розміщення, інтенсивність або форму потоку. Це особливо корисно для перевірки стабільності симуляції в умовах різних сценаріїв – від статичних скупчень до зіткнення потоків.

Лістинг 3.1 Метод створення положень та швидкостей частинок з випадковим зміщенням:

```
public ParticleSpawnData GetSpawnData()
{
    var rng = new Unity.Mathematics.Random(42);
    List<float2> allPositions = new();
    List<float2> allVelocities = new();
    List<int> allRegionIndices = new();

    for (int regionIndex = 0; regionIndex < spawnRegions.Length; regionIndex++)
    {
        var region = spawnRegions[regionIndex];
        float2[] basePoints = GeneratePointsInRegion(region);
```

```
        foreach (var point in basePoints)
        {
            float angle = (float)rng.NextDouble() * math.PI * 2f;
            float2 dir = new float2(math.cos(angle), math.sin(angle));
            float2 jitter = dir * jitterStr * ((float)rng.NextDouble() - 0.5f);

            allPositions.Add(point + jitter);
            allVelocities.Add(initialVelocity);
            allRegionIndices.Add(regionIndex);
        }
    }

    return new ParticleSpawnData
    {
        positions = allPositions.ToArray(),
        velocities = allVelocities.ToArray(),
        spawnIndices = allRegionIndices.ToArray(),
    };
}
```

Інтерфейс користувача в сцені відсутній: усі параметри задаються безпосередньо у властивостях компонентів через інспектор (рис.3.4), що відповідає фокусуванню проєкту на симуляційну логіку та GPU-обчислення.

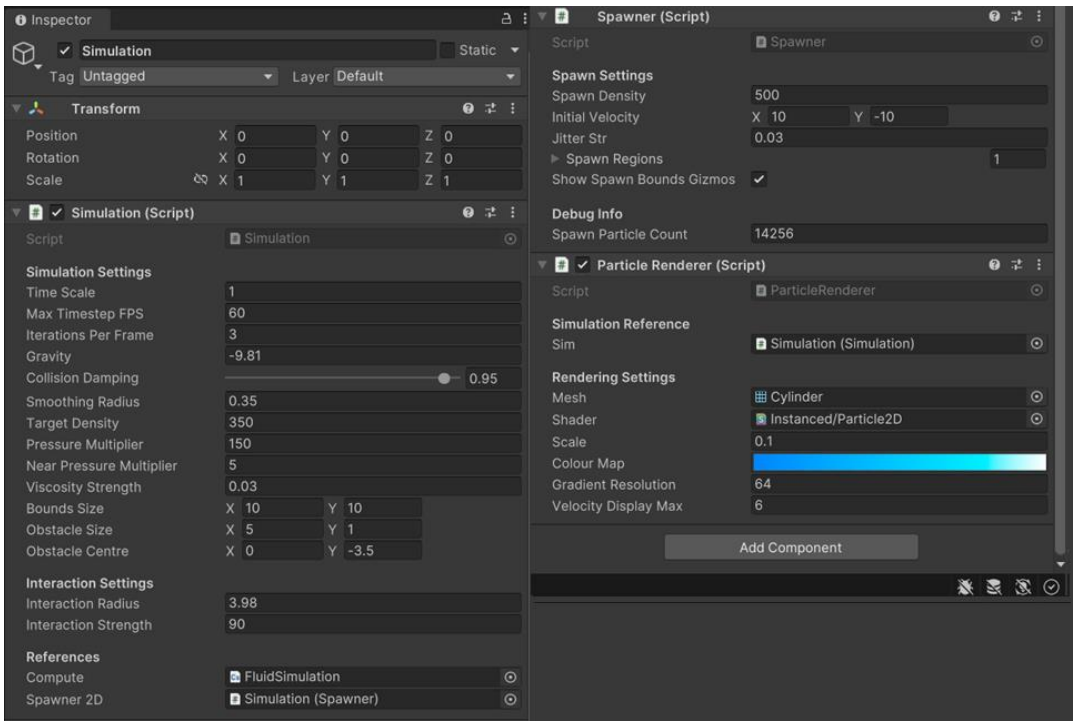

Рисунок 3.4 − Структура об’єкта Simulation у вікні інспектора з підключеними скриптами

3.2 Розробка обчислювальних скриптів та compute shaders

Одним із ключових етапів розробки цього проєкту стала побудова системи симуляції рідини, яка виконується на графічному процесорі за допомогою обчислювальних шейдерів. На відміну від традиційного процесорного виконання, GPU дозволяє паралельно обробляти тисячі частинок у режимі реального часу, що особливо важливо для створення правдоподібної поведінки рідини з високою деталізацією.

Серцем цієї системи є compute shader під назвою FluidSimulation.compute, у якому поетапно реалізовано фізичну модель руху частинок. Кожна частинка – це невелика незалежна одиниця, яка взаємодіє зі своїми сусідами згідно з принципами методу згладженої гідродинаміки частинок (SPH). Цей метод дозволяє обчислювати густину рідини, тиск, силу в’язкості та інші властивості, що виникають унаслідок локальної взаємодії частинок.

На початку кожного кроку симуляції обчислюється вплив зовнішніх сил. У нашій реалізації враховується гравітація, а також можливість втручання користувача – наприклад, вплив курсора миші, який імітує локальний тиск на рідину. Далі система прогнозує майбутні положення частинок на основі їхньої поточної швидкості, що дає змогу оцінити, як зміниться конфігурація системи найближчим часом.

Лістинг 3.2 Функція для налаштування параметрів обчислення:

```
void UpdateSettings(float deltaTime)
{
    compute.SetFloat("deltaTime", deltaTime);
    compute.SetFloat("gravity", gravity);
    compute.SetFloat("collisionDamping", collisionDamping);
    compute.SetFloat("smoothingRadius", smoothingRadius);
    compute.SetFloat("targetDensity", targetDensity);
    compute.SetFloat("pressureMultiplier", pressureMultiplier);
    compute.SetFloat("nearPressureMultiplier", nearPressureMultiplier);
    compute.SetFloat("viscosityStrength", viscosityStrength);
    compute.SetVector("boundsSize", boundsSize);
    compute.SetVector("obstacleSize", obstacleSize);
    compute.SetVector("obstacleCentre", obstacleCentre);

    compute.SetFloat("Poly6ScalingFactor", 4f / (Mathf.PI *
Mathf.Pow(smoothingRadius, 8)));
    compute.SetFloat("SpikyPow3ScalingFactor", 10f / (Mathf.PI *
Mathf.Pow(smoothingRadius, 5)));
    compute.SetFloat("SpikyPow2ScalingFactor", 6f / (Mathf.PI *
Mathf.Pow(smoothingRadius, 4)));
    compute.SetFloat("SpikyPow3DerivativeScalingFactor", 30f /
(Mathf.Pow(smoothingRadius, 5) * Mathf.PI));
```

```
        compute.SetFloat("SpikyPow2DerivativeScalingFactor", 12f /
(Mathf.Pow(smoothingRadius, 4) * Mathf.PI));

        Vector2 mousePos =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
        bool isPull = Input.GetMouseButton(0);
        bool isPush = Input.GetMouseButton(1);

        float currStrength = (isPush || isPull) ? (isPush ? -interactionStrength :
interactionStrength) : 0f;

        compute.SetVector("interactionInputPoint", mousePos);
        compute.SetFloat("interactionInputStrength", currStrength);
        compute.SetFloat("interactionInputRadius", interactionRadius);
    }
```

Щоб визначити, які частинки знаходяться поруч одна з одною, використовується просторове хешування – метод, який ділить простір на клітини та розподіляє частинки за цими клітинами. Це дає змогу обмежити коло перевірок під час взаємодії лише до тих частинок, які реально можуть вплинути одна на одну, значно підвищуючи швидкість обчислень. Після побудови просторової сітки та сортування частинок розпочинається розрахунок густини та близької густини для кожної з них. Ці значення визначаються на основі відстані до сусідів та використовуються для обчислення тиску.

Сили тиску моделюються так, ніби кожна частинка прагне підтримувати певну «ідеальну» густину. Якщо вона більша за норму, то утворюється тиск, який намагається відштовхнути частинку від інших.

Аналогічно розраховується і сила в’язкості, яка пом’якшує різкі зміни швидкості між частинками, імітуючи внутрішній опір рідини.

Після завершення всіх фізичних обчислень кожна частинка оновлює своє положення у просторі відповідно до нової швидкості. Якщо частинка стикається з межами сцени або перешкодою (у цьому випадку – прямокутним об’єктом), її рух змінюється з урахуванням сили відскоку та демпфування, щоб уникнути наскрізного проходження крізь тверді об'єкти.

Паралельно з фізичними обчисленнями працює шейдер Particle2D.shader, який відповідає за візуалізацію частинок на екрані. Кожна частинка зображується як маленький спрайт – квадрат, положення якого відповідає обчисленим координатам. Візуальний ефект досягається завдяки тому, що крайові області частинки є напівпрозорими, а колір залежить від її швидкості. Це створює ефект динаміки та напрямку руху: чим швидше частинка рухається, тим яскравіший і насиченіший її колір.

Лістинг 3.3 Функція віршинного шейдера для відображення частинок за швидкістю:

```
v2f vert (appdata_full v, uint instanceID : SV_InstanceID)
{
    float speed = length(Velocities[instanceID]);
    float speedT = saturate(speed / velocityMax);
    float colT = speedT;

    float3 centreWorld = float3(Positions2D[instanceID], 0);
    float3 worldVertPos = centreWorld + mul(unity_ObjectToWorld,
v.vertex * scale);
    float3 objectVertPos = mul(unity_WorldToObject,
float4(worldVertPos.xyz, 1));

    v2f o;
```

```
        o.uv = v.texcoord;
        o.pos = UnityObjectToClipPos(objectVertPos);
        o.colour = ColourMap.SampleLevel(linear_clamp_sampler,
float2(colT, 0.5), 0);

        return o;
}
```

Для досягнення плавного переходу прозорості від центру частинки до її країв використовується фрагментний шейдер, який обчислює значення альфа-каналу залежно від відстані пікселя до центру. У фрагментній функції спочатку визначається зміщення текстурної координати відносно центру частинки, після чого обчислюється квадрат відстані. Прозорість розраховується за допомогою згладженої функції переходу, яка забезпечує м’яке затухання на межах частинки. Завдяки цьому частинка сприймається як м’яке світле коло, а не як жорсткий квадрат.

Кольорове значення, що передається з віршинного шейдера, використовується як основний колір кожного пікселя. Його насиченість залежить від швидкості частинки, що підсилює ефект руху. Поєднання кольору та прозорості створює візуальну легкість, що дає змогу відображати значну кількість частинок без перевантаження зображення. Такий підхід дозволяє досягти якісного візуального ефекту навіть при високій щільності симуляції.

Лістинг 3.4 Фрагментний шейдер для м’якого згасання частинок:

```
float4 frag (v2f i) : SV_Target
{
        float2 centreOffset = (i.uv.xy - 0.5) * 2;
        float sqrDst = dot(centreOffset, centreOffset);
        float delta = fwidth(sqrt(sqrDst));
```

```
        float alpha = 1 - smoothstep(1 - delta, 1 + delta, sqrDst);

        float3 colour = i.colour;
        return float4(colour, alpha);
}
```

У підсумку обчислювальні шейдери і візуалізуючі засоби працюють разом, формуючи складну, але ефективну систему, яка симулює та відображає рідину в реальному часі. Всі обчислення відбуваються безпосередньо на GPU, що дозволяє досягти високої продуктивності навіть при великій кількості частинок.

3.3 GPU-сортування та оптимізація обчислень

Для ефективної симуляції рідини, яка складається з тисяч частинок, надзвичайно важливо швидко знаходити сусідів кожної з них у просторі. Щоб пришвидшити цей процес, було реалізовано спеціальне GPU-сортування на основі методу підрахункового сортування (counting sort), адаптоване для виконання на графічному процесорі. У нашому випадку це дозволяє відсортувати частинки за їх просторовими ключами, що генеруються при побудові просторової сітки (spatial hash), і, відповідно, згрупувати частинки, які фізично перебувають поряд одна з одною.

Сам алгоритм сортування працює в кілька етапів, кожен із яких реалізований як окремий GPU-шейдер. У перший момент ініціалізуються буфери даних: ключі (тобто просторові індекси), елементи (ідентифікатори частинок), а також буфери для збереження результатів сортування та підрахунків. Кожен ключ – це ціле число, яке вказує на ту область сітки, в якій знаходиться частинка.

На початку запускається шейдер ClearCounts, який очищає буфер підрахунків і ініціалізує буфер індексів. Далі CalculateCounts підраховує,

скільки частинок відповідають кожному ключу – тобто скільки з них потрапляють до кожної клітини сітки. Це виконується з використанням атомарної операції InterlockedAdd, яка дозволяє уникнути конфліктів при одночасному записі кількох потоків у ту саму комірку пам’яті.

Однак одного лише підрахунку недостатньо, щоб визначити конкретні позиції частинок у фінальному відсортованому списку. Для цього використовується алгоритм префіксної суми (prefix sum або scan). Він дозволяє для кожного ключа визначити початковий індекс, з якого мають записуватись елементи в результаті сортування. Наприклад, якщо в ключа 3 – п’ять елементів, а до цього ключ 2 мав чотири, то елементи з ключем 3 починаються з індексу 4. Префіксна сума виконується за допомогою окремого шейдера ScanTest.compute, який реалізує блокову версію scan-алгоритму з розбиттям на групи та поетапним комбінуванням результатів.

Після обчислення префіксної суми запускається шейдер ScatterOutput, який розміщує елементи в новому буфері в порядку зростання ключів, використовуючи інформацію з підрахунків. Він також виконує ще один атомарний запис у Counts, щоб отримати позицію запису для кожного елементу. Таким чином, частинки потрапляють у відсортований масив відповідно до своїх позицій у просторі.

Фінальним кроком є шейдер CopyBack, який просто копіює відсортовані елементи назад у початкові буфери. Після цього всі операції завершено, і дані готові до наступного етапу симуляції – наприклад, обчислення взаємодії частинок у межах однієї клітини простору.

Важливою особливістю цієї реалізації є те, що всі етапи виконуються виключно на GPU, що дозволяє обробляти величезну кількість частинок паралельно. Кожен потік відповідає за одну частинку, а завдяки розбиттю на блоки та використанню shared memory (групової пам’яті), досягається висока ефективність навіть для складних обчислювальних етапів, як-от prefix sum.

Клас GPUCountSort у C# відповідає за збирання та налаштування всіх необхідних буферів і шейдерів. Він також інкапсулює логіку запуску

обчислень, дбає про повторне використання буферів і коректне очищення пам’яті після завершення симуляції. Цей підхід дозволяє легко інтегрувати сортування в загальну архітектуру симуляції та масштабувати її для більших обсягів даних без втрати продуктивності.

3.4 Результати тестування та аналіз продуктивності

Після реалізації основних алгоритмів симуляції та GPU-сортування було проведено низку тестів для оцінки продуктивності системи. Основною метою тестування було визначення масштабованості симуляції залежно від кількості частинок, а також перевірка ефективності GPU-оптимізацій у порівнянні з CPU-реалізацією (для контрольних замірів).

Симуляція запускалась у середовищі Unity з використанням ComputeShader-процесів, де вся основна обробка (взаємодії частинок, сортування, оновлення позицій тощо) виконувалась на GPU. Для проведення замірів використовувались вбудовані інструменти Unity Profiler, а також вимірювання часу обчислень на кожному з етапів за допомогою System.Diagnostics.Stopwatch.

Було протестовано симуляції з різною кількістю частинок – від 513 до 98800. При малих об’ємах частинок GPU-навантаження було незначним, і симуляція виконувалась із запасом продуктивності. Проте із зростанням кількості частинок система почала демонструвати високий рівень паралелізації, де час оновлення позицій і обчислення взаємодій зростав майже лінійно лише до певної межі, після чого стабілізувався завдяки ефективному сортуванню та просторовому групуванню.

У рамках тестування була зафіксована частота кадрів (FPS) при різній кількості частинок у симуляції. Як видно з наведених результатів (рис.3.5), симуляція демонструє стабільну продуктивність при обробці до приблизно 25 тисяч частинок, після чого спостерігається поступове зниження FPS, що є

наслідком зростання навантаження на графічний процесор. Проте навіть при 98 тисячах частинок симуляція продовжує працювати в реальному часі з частотою понад 50 кадрів за секунду.

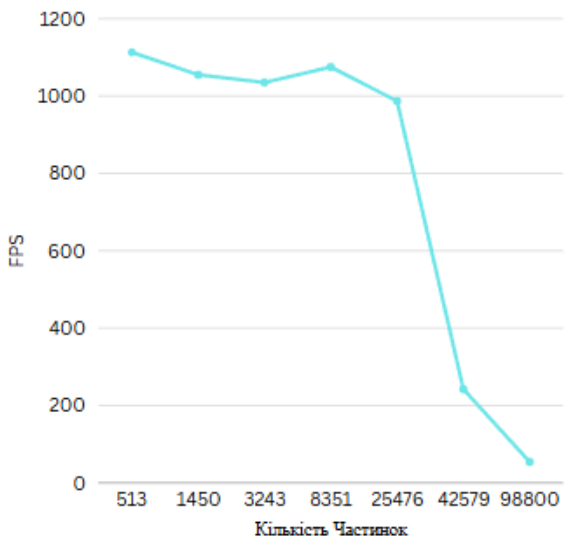

Рисунок 3.5 − Залежність FPS від кількості частинок у симуляції

Аналізуючи отримані значення, можна помітити, що в межах до ~30 тисяч частинок продуктивність залишається практично незмінною, завдяки використанню GPU-сортування та просторової хеш-структури для швидкого пошуку сусідів. Після цього настає переломний момент, де час виконання обчислень, зокрема сортування та розрахунків тиску, починає лінійно зростати. Проте завдяки ефективному розподілу навантаження між потоками GPU система залишається достатньо швидкою навіть при великих об’ємах даних.

Також було проаналізовано час виконання окремих етапів симуляції: сортування (GPU Count Sort), пошук сусідів, обчислення сил тиску та в’язкості. Було виявлено, що найбільшу частку часу займає сортування при високій кількості частинок. Саме тому впровадження оптимізованої версії паралельного підрахункового сортування з prefix sum суттєво вплинуло на загальну продуктивність.

Для наочного представлення результатів та відстеження змін у поведінці симуляції були реалізовані елементи керування через серіалізовані поля Unity. Це дозволяло змінювати фізичні параметри симуляції – такі як маса частинки, радіус впливу, коефіцієнти в’язкості та стиснення – безпосередньо в редакторі, що значно полегшило процес налагодження.

Окрім цього, для візуалізації меж області симуляції використовувалися Gizmos. Це дозволило в режимі редактора чітко бачити граничні лінії, в межах яких відбувається симуляція, а також перевірити, чи правильно працює відштовхування частинок від меж. Такий підхід дозволив оперативно виявляти похибки у фізичній моделі та точніше налаштувати параметри відскоку (рис. 3.6).

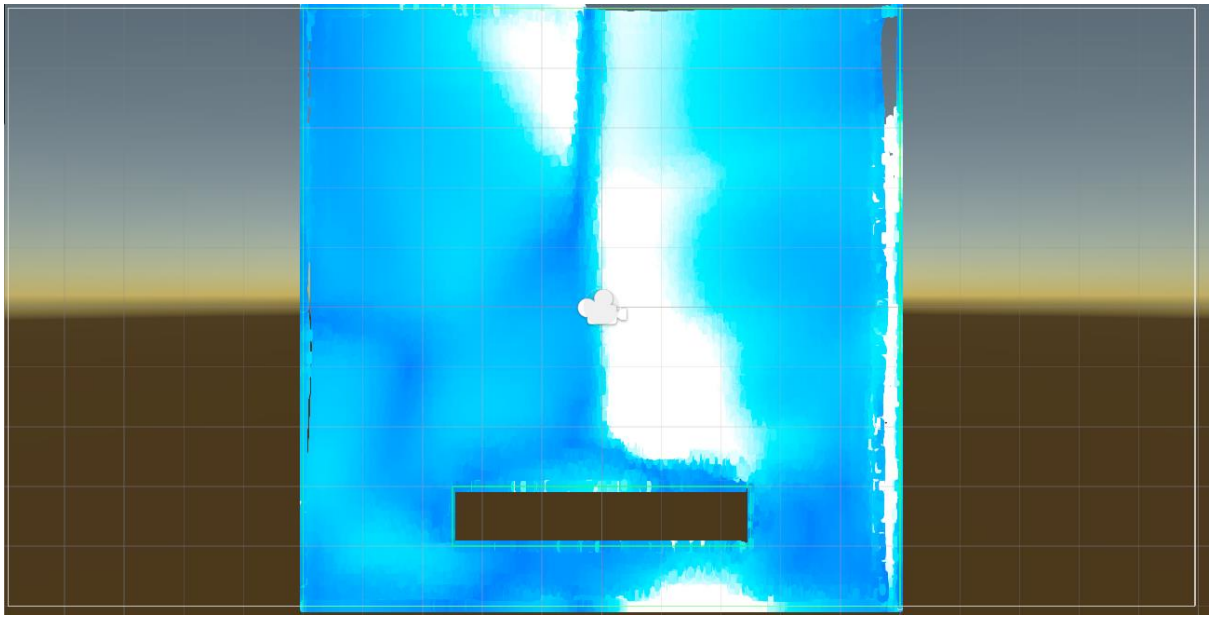

Рисунок 3.6 − Візуалізація області симуляції за допомогою Gizmos у редакторі Unity

Узагальнюючи результати, можна зробити висновок, що реалізована GPU-система демонструє хорошу масштабованість та здатна ефективно обробляти десятки тисяч частинок у реальному часі. Подальші оптимізації можуть бути спрямовані на зменшення кількості глобальної пам’яті, що використовується, та ще кращу локальну кешизацію даних у GPU-групах.

ВИСНОВКИ

Розробка застосунку для симуляції та візуалізації рідин на основі Unity стала практичною реалізацією сучасних підходів до моделювання фізичних процесів у цифровому середовищі. Проєкт поєднав у собі актуальні наукові методи, такі як метод згладжених частинок - Smoothed Particle Hydrodynamics, з можливостями GPU-прискорення та візуалізації у реальному часі, що дало змогу створити ефективну і масштабовану систему моделювання.

Однією з головних цілей було досягнення максимальної продуктивності при високій точності фізичних розрахунків. Для цього було реалізовано ефективну просторову сітку на базі хешування, що дозволило значно прискорити пошук сусідніх частинок − критично важливу операцію для SPH-алгоритму. Реалізація цієї частини на GPU у вигляді Compute Shader'ів забезпечила високу швидкодію симуляції навіть за умов великої кількості частинок.

Важливу роль у досягненні цілей проєкту відіграв вибір платформи − Unity. Цей рушій надав готову екосистему для швидкого створення візуального інтерфейсу, організації сцени, налагодження, а також гнучкі засоби візуалізації через Shader Graph або власні шейдери на HLSL. Це значно скоротило час розробки та дозволило сфокусуватися на оптимізації симуляції, а не на створенні низькорівневого рендеринга.

Отримані під час тестування результати продемонстрували, що реалізована система є масштабованою: при кількості частинок до 25 тисяч система зберігає стабільно високу частоту кадрів , а при збільшенні кількості до майже 100 тисяч продуктивність знижується, проте залишається в межах реального часу. Така поведінка є очікуваною і свідчить про правильну побудову архітектури обчислень.

Також було реалізовано систему зручного налагодження через Serialized Fields, Gizmos, а також модульну структуру коду, яка дозволяє

легко змінювати параметри симуляції, візуалізації та логіки. Це робить застосунок гнучким для подальших експериментів як в освітніх, так і наукових цілях.

Результати симуляції демонструють фізично достовірну поведінку рідини, включаючи такі ефекти, як тиск, в’язкість, взаємодія з межами області, відштовхування частинок від перешкод. Це дозволяє використовувати застосунок не лише як демонстраційний інструмент, а й як базу для побудови складніших фізичних моделей − наприклад, моделювання багатофазних потоків, взаємодії рідини з твердими тілами, чи візуалізації природних явищ.

Таким чином, реалізований застосунок є вагомим прикладом успішного поєднання комп’ютерної графіки, чисельних методів та сучасних можливостей ігрового рушія для моделювання фізичних процесів. Він підтверджує, що Unity може бути використаний не лише для створення ігор, але й для реалізації високопродуктивних наукових та освітніх інструментів. Отриманий досвід може стати основою для майбутніх досліджень у галузі цифрових симуляцій, геймдеву та інтерактивної фізики.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Dong, X., Hao, G., & Yu, R. (2022). Two−dimensional smoothed particle hydrodynamics (SPH) simulation of multiphase melting flows and associated interface behavior. Engineering Applications of Computational Fluid Mechanics, 16(1), 588−629.

2. Almasi, F., Shadloo, M. S., Hadjadj, A., Ozbulut, M., Tofighi, N., & Yildiz, M. (2021). Numerical simulations of multi−phase electro−hydrodynamics flows using a simple incompressible smoothed particle hydrodynamics method. Computers & Mathematics with Applications, 81, 772−785.

3. Wolmerud, M. (2015). Real−Time Fluid Simulation and Visualization.

4. Hoetzlein, R. C. (2014). Fast fixed−radius nearest neighbors: interactive million−particle fluids. In GPU Technology Conference (Vol. 18, p. 2).

5. Goswami, P., Schlegel, P., Solenthaler, B., & Pajarola, R. (2010). Interactive SPH simulation and rendering on the GPU.

6. Tang, J., Zheng, Y., Yang, C., Wang, W., & Luo, Y. (2020). Parallelized implementation of the finite particle method for explicit dynamics in GPU. Computer Modeling in Engineering & Sciences, 122(1), 5−31.

7. Monaghan, J. J. (2005). Smoothed particle hydrodynamics. Reports on progress in physics, 68(8), 1703.

8. Amrouche, S., Kiehn, M., Golling, T., & Salzburger, A. (2021). Hashing and metric learning for charged particle tracking. arXiv preprint arXiv:2101.06428.

9. TRY, J., & KHAN, Y. Fluid Simulation and Screen Space Fluid Rendering In VR.

10. Гороховатський, В. О., Передрій, О. О., Творошенко, І. С., & Марков, Т. Є. (2023). Матриця відстаней для множини компонентів структурного опису як інструмент для створення класифікатора зображень.

11. Akenine−Möller, T., Haines, E., & Hoffman, N. (2008). Real−time rendering 3rd edn. AK Peters, Natick.

12. Pozorski, J., & Olejnik, M. (2024). Smoothed particle hydrodynamics modelling of multiphase flows: an overview. Acta Mechanica, 235(4), 1685−1714.

13. Guan, X. S., Sun, P. N., Lyu, H. G., Liu, N. N., Peng, Y. X., Huang, X. T., & Xu, Y. (2022). Research progress of SPH simulations for complex multiphase flows in ocean engineering. Energies, 15(23), 9000.

14. Liu, J., Zhang, T., & Sun, S. (2022). Study of the imbibition phenomenon in porous media by the smoothed particle hydrodynamic (SPH) method. Entropy, 24(9), 1212.

15. Le Touzé, D., & Colagrossi, A. (2025). Smoothed Particle Hydrodynamics for free−surface and multiphase flows: a review. Reports on Progress in Physics.

16. Saffar, M. (2023). An Investigation of How Lighting and Rendering Technology Affects Filmmaking Relative to Arnold's Transition to a GPU−Based Path−Tracer.

17. Thanh, B. T. (2021). Boundary Treatment for Particle−based Fluid Simulation with Solid Wall and Open Boundary Conditions.

18. de Oliveira, F. M. C. GPU implementation of a fluid dynamics interactive simulator based on the Lattice Boltzmann method.

19. Suchde, P. (2024). Particle−based adaptive coupling of 3D and 2D fluid flow models. Computer Methods in Applied Mechanics and Engineering, 429, 117199.

20. Schröder, A., & Schanz, D. (2023). 3D Lagrangian particle tracking in fluid mechanics. Annual Review of Fluid Mechanics, 55(1), 511−540.

21. Ihmsen, M., Cornelis, J., Solenthaler, B., Horvath, C., & Teschner, M. (2013). Implicit incompressible SPH. IEEE transactions on visualization and computer graphics, 20(3), 426−435.

22. Macklin, M., & Müller, M. (2013). Position based fluids. ACM Transactions on Graphics (TOG), 32(4), 1−12.

23. Solenthaler, B., & Pajarola, R. (2009). Predictive−corrective incompressible SPH. In ACM SIGGRApH 2009 papers (pp. 1−6).

24. Zhang, F., Hu, L., Wu, J., & Shen, X. (2011, December). A SPH−based method for interactive fluids simulation on the multi−GPU. In Proceedings of the 10th international conference on virtual reality continuum and its applications in industry (pp. 423−426).

25. Akinci, N., Ihmsen, M., Akinci, G., Solenthaler, B., & Teschner, M. (2012). Versatile rigid−fluid coupling for incompressible SPH. ACM Transactions on Graphics (TOG), 31(4), 1−8.

26. Becker, M., & Teschner, M. (2007, August). Weakly compressible SPH for free surface flows. In Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation (pp. 209−217).

27. Kelager, M. (2006). Lagrangian fluid dynamics using smoothed particle hydrodynamics. University of Copenhagen: Department of Computer Science, 2.

28. Rudenko, D., Serhiienko, O., Zeleniy, O., & Lyashenko, V. (2022). Model for Predictive Analysis of International Trade Based on the Dynamics of Stock Indices (Example of Data from the USA, Canada and UK).

29. Zeleniy, O., Rudenko, D., Lyubchenko, V., & Lyashenko, V. (2022). Image Processing as an Analysis Tool in Medical Research.

30. Bodyanskiy, Y. V., Shafronenko, A., Rudenko, D., Plubiekhin, A., & Frolov, D. (2024). Online Image Segmentation using Credibilistic Fuzzy Clustering. In COLINS (1) (pp. 1−10).

31. Tvoroshenko, I. S., & Bielinskyi, Y. (2021). On the features of methods of processing and recognition of handwritten text.

32. Tvoroshenko, I., & Kharchenko, A. (2021). Some aspects of modern development for sign language recognition systems.

33. Iryna, T., & Maksym, K. (2021). Research results of functional, white box and smoke testing methods for mobile applications. Trends in science and practice of today, 5, 418.

34. Tvoroshenko, I., & Gorokhovatskyi, V. (2022). The Application of Hybrid Intelligence Systems for Dynamic Data Analysis.

35. Гороховатський, В. О., & Творошенко, І. С. (2022). Аналіз багатовимірних даних за описом у формі множини компонент.

36. Ibrahim, D. Y., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2022). Cluster representation of the structural description of images for effective classification.

37. Ibrahim Daradkeh, Y., Gorokhovatskyi, V., Tvoroshenko, I., & Al−Dhaifallah, M. (2022). Classification of Images Based on a System of Hierarchical Features. Computers, Materials & Continua, 72(1).

38. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2022). Tools for fast metric data search in structural methods for image classification. IEEE Access, 10, 124738−124746.

39. Pomazan, V., Tvoroshenko, I., & Gorokhovatskyi, V. (2023). Development of an application for recognizing emotions using convolutional neural networks.

40. Tvoroshenko, I., Gorokhovatskyi, V., Kobylin, O., & Tvoroshenko, A. (2023). Application of deep learning methods for recognizing and classifying culinary dishes in images.

41. Pomazan, V., Tvoroshenko, I., & Gorokhovatskyi, V. (2023). Handwritten character recognition models based on convolutional neural networks.

42. Tvoroshenko, I., Pomazan, V., Gorokhovatskyi, V., & Kobylin, O. (2023). Application of video data classification models using convolutional neural networks.

43. Gorokhovatskyi, V., Tvoroshenko, I., Kobylin, O., & Vlasenko, N. (2023). Search for visual objects by request in the form of a cluster representation for the structural image description. Advances in Electrical and Electronic Engineering, 21(1), 19.

44. Gorokhovatskyi, V., Tvoroshenko, I., & Olena, Y. (2024). Transforming image descriptions as a set of descriptors to construct classification features.

45. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., Hudáková, M., & Gorokhovatskyi, O. (2024). Application a committee of Kohonen neural networks to training of image classifier based on description of descriptors set. IEEE Access.