Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Аналіз методів моделювання реалістичної фізики рідин в Unity

Курсова робота
за спеціальністю „Комп'ютерні науки"

Керівник курсової роботи
Старший викладач
Картавий М.О.

_____
(підпис)

"____" _____ 2025 р.

Виконала
студентка 3 курсу
факультету інформатики
Дяченко В.Ю.

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
проф., д.ф.-м.н.
_____М. М. Глибовець
(підпис)
„____”_____2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу
студентці Дяченко Владиславі Юліївні факультету
інформатики 3-го курсу
ТЕМА: АНАЛІЗ МЕТОДІВ МОДЕЛЮВАННЯ РЕАЛІСТИЧНОЇ ФІЗИКИ РІДИН В UNITY

Зміст ТЧ до курсової роботи:
Індивідуальне завдання
Календарний план
Анотація
Вступ
Розділ 1. Відомості про фізику рідин і методи їх моделювання
Розділ 2. Аналіз методів моделювання рідин
Розділ 3. Практична реалізація моделювання рідин
Висновки
Список використаної літератури

Дата видачі „___” _____ 2025 р.
Керівник _____
(підпис)
Завдання отримала _____
(підпис)

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Отримання теми курсової роботи	11.10.2024	
2	Пошук тематичної літератури	30.10.2024	
3	Написання вступу та змісту роботи	20.11.2024	
4	Ознайомлення з методами моделювання фізики рідин	13.12.2024	
5	Вивчення можливостей Unity для моделювання рідин	11.01.2025	
6	Написання відповідних розділів теоретичного блоку	10.02.2025	
7	Реалізація симуляції у Unity	27.03.2025	
8	Аналіз результатів моделювання	13.04.2025	
9	Написання висновків та рекомендацій	18.04.2025	
10	Оформлення відповідно до вимог написання курсової роботи	22.04.2025	
11	Підготовка до захисту роботи	24.04.2025	
12	Узгодження роботи з науковим керівником та внесення коректив	30.04.2025	
13	Захист курсової роботи	15.05.2025	

ЗМІСТ

Використані терміни

VR - віртуальна реальність

GPU - графічний процесор

CPU - центральний процесор

URP - універсальний конвеєр рендерингу

HDRP - конвеєр рендерингу високої чіткості

C# - C Sharp - об'єктно-орієнтована мова програмування

2D/3D - дво- та тривимірні простори

D2Q9 - двовимірна модель з 9 швидкостями

D3Q19 - тривимірна модель з 19 швидкостями

VFG - граф візуальних ефектів

Asset Store - офіційний цифровий магазин Unity для придбання та завантаження готових активів, компонентів, інструментів та проектів

UV-координати - система двовимірних текстурних координат

FPS - кількість кадрів на секунду, міра швидкодії рендерингу та продуктивності програми

Unity Profiler - інструмент для аналізу та оптимізації продуктивності застосунків Unity, який відстежує використання ресурсів, такі як CPU, GPU, пам'ять та мережеві операції

Анотація до курсової роботи

Метою даної роботи є аналіз та порівняння методів моделювання фізики рідин в середовищі Unity для визначення найефективніших підходів у різних сценаріях.

У ході виконання роботи було розглянуто основні рівняння фізики рідин, проведено класифікацію методів моделювання та проаналізовано їх придатність для використання в Unity.

На основі проведеного аналізу відібрано три методи (SPH, FLIP, LBM) для практичної реалізації, досліджено технічні засоби та обмеження Unity при роботі з фізикою рідин.

Розроблено програмні реалізації для кожного з відібраних методів та проведено порівняльний аналіз результатів моделювання.

Вступ

Моделювання фізики рідин є важливим напрямком у галузі комп'ютерної графіки, який активно використовується в ігровій індустрії, кінематографі, віртуальній реальності та інженерних симуляціях.

До того, як індустрія комп'ютерної графіки почала розвиватися, симуляція рідин активно моделювалася математично ще в 1950-х і 60-х роках.

У 1980-х почали з’являтися перші алгоритми вже для комп’ютерної графіки, а у 1990-х відбувся прорив у графіці та зросла кількість технік та методів, а разом з тим і фільмів, де можна побачити їх масштабне використання. Першими такими фільмами були *Водний світ (1995), Титанік (1997), Володар перснів: Дві вежі (2002)*.

Сучасні методи моделювання рідин дозволяють використовувати складні та гібридні моделі для максимальної реалістичності, вони, також, стають інтерактивними у реальному часі, що має величезну перевагу у сферах відеоігор та VR.

Середовище розробки Unity, як один із найпопулярніших інструментів, забезпечує розробників широким спектром можливостей для створення різних фізичних симуляцій. Додаткове застосування бібліотек та спеціалізованого програмного забезпечення суттєво розширює потенціал цієї платформи, що дозволяє досягати ще більш реалістичних результатів.

Розділ 1

Відомості про фізику рідин і методи їх моделювання

1.1. Основні рівняння і закони фізики рідин

Фізика рідин вивчає рух і взаємодію рідин на основі численних рівнянь і законів.

Одним з основних є **рівняння Нав'є-Стокса** (формула 1.1), яке описує рух рідини, враховуючи такі важливі параметри, як: швидкість потоку, тиск, в'язкість і сили тяжіння. Це рівняння є основою для чисельних методів моделювання рідин, оскільки воно виражає закони збереження імпульсу, енергії та маси.

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\frac{1}{\rho}\nabla p + \nu \nabla^2 u + f \qquad (1.1)$$

де: u - швидкість потоку рідини; p – тиск; ν - кінематична в'язкість;

ρ - густина рідини; f - зовнішні сили (наприклад, гравітація).

Рівняння Ейлера (формула 1.2) є спрощеною версією рівняння Нав'є-Стокса, проте воно не враховує в'язкість рідини. Воно описує ідеальний потік рідини, що може використовуватися в умовах, де ефекти в'язкості не мають суттєвого впливу.

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\frac{1}{\rho}\nabla p + f \qquad (1.2)$$

Ще одним важливим рівнянням є **рівняння неперервності** (формула 1.3), що описує збереження маси рідини в потоці. Воно встановлює, що потік маси через будь-яку поверхню в замкнутій системі залишається сталим, що є ключовим принципом для моделювання руху рідин.

$$\nabla \cdot u = 0 \qquad (1.3)$$

де: u - швидкість потоку рідини

Для опису теплових властивостей рідин застосовують **рівняння енергії** (формула 1.4), яке відображає баланс між теплообмінними процесами та механічними властивостями рідини.

$$\frac{\partial e}{\partial t} + \nabla \cdot (eu) = k\nabla^2 e + Q \quad (1.4)$$

де: e - енергія на одиницю об'єму; u – швидкість; k - коефіцієнт теплопровідності; Q - джерело тепла.

Ці рівняння формують ґрунтовну теоретичну основу для чисельних моделей, що використовуються для моделювання рідин.

1.2. Класифікація методів моделювання рідин

Існує кілька основних підходів, які класифікуються як методи Лагранжа та Ейлера. Більшість методів Ейлера використовують сітку для опису рідинної області. Більшість методів Лангража, з іншого боку, використовують точки, що вільно переміщаються у просторі.

Отже, можна сказати, що методи поділяються в основному на дві групи: методи на основі сітки та на основі частинок.

1.2.1. Сіткові методи (Grid-Based Methods)

Сіткові методи базуються на розбитті простору на регулярну або нерегулярну сітку (гратку), в якій розраховуються фізичні величини, такі як швидкість, тиск та густина рідини. До сіткових методів належать:

1. **Метод скінченних різниць (Finite Difference Method, FDM)** - чисельний підхід для розв'язання рівнянь Нав'є-Стокса, який апроксимує похідні через кінцеві різниці.

Його не можна назвати дуже популярним методом, але він явно представляє інтерес у простих застосуваннях.

Переваги: простий у реалізації, найбільш ефективний зі звичайними прямокутними сітками.

Недоліки: обмежений для задач, пов'язаних із межами складної форми.

2. **Метод скінченних елементів (Finite Element Method, FEM)** - розбиває простір на невеликі зв'язані елементи, у яких функції стану наближено описуються простішими залежностями.

Досить активно використовується для дискретизації будь-яких диференціальних рівнянь частинних похідних.

Переваги: гнучкий у моделюванні складних форм, добре працює із задачами зі складними областями чи межами.
Недоліки: має високі обчислювальні витрати.

3. **Метод скінченних об'ємів (Finite Volume Method, FVM)** - розділяє весь домен на невеликі контрольні томи та обмінює потоки через сполучні грані.

Він є найвідомішим методом і використовується у кардинально різних напрямках.

Переваги: має точний розрахунок потоків, підходить для моделювання великих потоків рідини, взаємодіє з неструктурованими сітками та складними георметріями.
Недоліки: є менш ефективним для дрібних деталей та турбулентних потоків.

4. **Метод Болцмана на решітці (Lattice Boltzmann Method, LBM)** - є альтернативою традиційним сітковим методам. LBM моделює динаміку рідин, представляючи їх як набір частинок, що взаємодіють між собою за допомогою відрізків простору та часу.

Цей метод є ще дуже «молодим», однак він часто перевершує традиційні методи.

Переваги: має високу ефективність для моделювання складних потоків і вільних поверхонь; підходить для багатьох різних умов, включаючи складні геометрії та багатокомпонентні рідини, може повторно використовувати вже усталені ідеї сіткових методів.

Недоліки: має обмеження щодо розміру решітки та необхідність використання спеціалізованих обчислювальних ресурсів.

Отже, сіткові методи забезпечують високу чисельну точність, однак часто страждають від втрати маси і часто є повільнішими.

1.2.2. Частинкові методи (Mesh-Free, Particle-Based Methods)

Частинкові методи базуються на лагранжевому описі, де кожна частинка рухається відповідно до законів фізики, а взаємодії моделюються за допомогою згладжувальних функцій або чисельних диференціювань. До таких методів належать:

1. **Метод згладжених частинок (Smoothed Particle Hydrodynamics, SPH)** - описує рідину у вигляді набору частинок, кожна з яких рухається на основі своєї швидкості. Ці частинки взаємодіють між собою через функцію згладжування.

Цей метод дуже активно використовується та є першим і основним у димамічних та рухомих потоках.

Переваги: ефективний для моделювання складних рідинних потоків, добре підходить для симуляцій з вільною поверхнею, має паралельну обробку.

Недоліки: менш ефективний у задачах, які можна змоделювати за допомогою сітки, через дорогий процес пошуку сусідніх частинок, , важко сформулювати деякі моменти математично.

2. **Метод (Moving Particle Simulation, MPS)** - ділить область рідини на частинки, яким надаються властивості швидкості, маси і тиску. При оновленні властивостей розв’язується рівняння для кожної з частинок.

Переваги: гнучкий у моделюванні рідинних потоків з вільними поверхнями, має високу точність при детальних симуляціях.
Недоліки: потребує великої кількості частинок для досягнення високої точності, має високі обчислювальні витрати для великих систем та точного фіксування поведінки потоку.

Отже, моделювання на основі частинок зазвичай набагато швидші, але вони виглядають значно гірше, ніж сіткові.

1.2.3. Гібридні методи

Сучасні симуляції рідин часто використовують поєднання сіткових та частинкових методів, що дозволяє отримати більш реалістичну поведінку рідин.

Одним із найпопулярніших таких методів є:

1. **Метод неявних частинок (Fluid-Implicit Particle, FLIP)** - використовує частинки для моделювання рідини та інтегрує їх рух у сіткові точки для опису змін потоків. Цей метод поєднує техніки частинок і сітки, що знижує енергетичні втрати при великих деформаціях, таких як хвилі та бризки.

Переваги: має високу точність і хорошу обробку складних деформацій і змін форм об’єктів.

Недоліки: складний у реалізації для певних задач, потребує оптимізації для зменшення обчислювальних витрат.

Як висновок, можна сказати, що сіткові методи дають високу точність, маючи обмеження у гнучкості, на відміну він частинкових методів, що чудово моделюють вільні поверхні та складні форми, але поступаються у точності ефективності своїх обчислень.

1.3. Огляд сучасних алгоритмів моделювання рідин

Сучасні методи моделювання рідин постійно вдосконалюються, зосереджуючись на підвищенні точності, продуктивності та ефективності обчислень. Основні напрями розвитку включають:

1. **Гібридні методи** - поєднують сіткові та частинкові підходи для отримання більш реалістичних результатів. Наприклад, метод FLIP (Fluid-Implicit Particle), який вже був описаний раніше
2. **GPU-оптимізовані алгоритми** - використовують графічні процесори, що значно прискорює розрахунки, дозволяючи виконувати симуляції рідин у реальному часі. Завдяки масовій паралельності SPH (Smoothed Particle Hydrodynamics) та MPS (Moving Particle Simulation) ефективно реалізується в цьому випадку.
3. **Глибоке навчання та нейромережі** - застосовуються для генерації швидкостей рідин у симуляціях. Вони мають високу точність, здайснюють реальне часове оновлення, інтерполяцію та масштабування даних. Також, стискають дані, що робить симуляції набагато ефективнішими. Наприклад, NeuralSPH та DeepFluids.
4. **Розподілені обчислення** - використовуються для моделювання великих і детальних потоків рідин. Паралельні алгоритми дозволяють розподіляти симуляцію між кількома процесорами або кластерами,

завдяки чому забезпечують швидку та ефективну обробку великих обсягів даних. Наприклад, OpenFOAM та Houdini FX.

5. **Алгоритми адаптивного уточнення сітки (Adaptive Mesh Refinement, AMR)** - застосовуються для підвищення точності в зонах складних течій, наприклад, турбулентних потоків або взаємодії рідин із твердими тілами. Вони динамічно змінюють роздільну здатність сітки залежно від особливостей потоку, оптимізуючи витрати ресурсів.

У контексті інтерактивного моделювання рідин особливе значення для Unity мають GPU-оптимізовані алгоритми, оскільки вони досить ефективно використовують паралельну архітектуру графічних процесорів. Завдяки різноманітним шейдерам та обчислювальним буферам ці алгоритми успішно реалізуються в Unity.

Розділ 2

Аналіз методів моделювання рідин

2.1. Аналіз та відбір методів для використання в Unity в реальному часі

Для об’єктивного порівняння методів моделювання рідин я склала таблицю (табл. 1) з ключовими критеріями.

Критерій	FDM	FEM	FVM	SPH	MPS	FLIP	LBM
Швидкодія в реальному часі	Низька	Дуже низька	Низька	Висока	Середня	Середня	Висока
Складність імплементації	Середня	Дуже висока	Висока	Середня	Висока	Висока	Середня
Масштабованість на GPU	Низька	Низька	Середня	Хороша	Середня	Середня	Відмінна
Обчислювальні вимоги	Високі	Дуже високі	Високі	Середні	Високі	Досить високі	Середні
Якість візуалу	Середня	Дуже висока	Висока	Хороша	Хороша	Висока	Хороша
Застосовність у реальному часі	Можлива	Обмежена	Обмежена	Хороша	Середня	Середня	Хороша
Стійкість до змін топології	Проблемна	Проблемна	Проблемна	Відмінна	Відмінна	Хороша	Середня

Табл. 2.1

Причиною вибору саме таких критеріїв стало те, що вони охоплюють як технічні аспекти, так і якісні характеристики. Швидкодія в реальному часі є ключовим фактором для інтерактивних застосунків. Складність імплементації визначає витрати ресурсів на розробку та інтеграцію методу. Масштабованість на GPU стає все важливішою з огляду на тенденцію до

використання графічних процесорів для паралельних обчислень у сучасних іграх. Обчислювальні вимоги безпосередньо впливають на продуктивність гри. Якість візуалу є важливим аспектом для створення переконливих візуальних ефектів. Застосовність у реальному часі відображає, наскільки метод придатний для динамічних інтерактивних сцен. Стійкість до змін топології є особливо важливою для моделювання динамічних рідин.

Така система оцінювання дозволяє комплексно проаналізувати методи та визначити найбільш перспективні для практичної реалізації в Unity.

Отже, за результатами аналізу виявлено, що методи SPH, FLIP та LBM демонструють найкраще співвідношення швидкості, реалістичності та вимог до реалізації саме в Unity.

Метод MPS, хоча і забезпечує високу реалістичність, проте має підвищені вимоги до обчислювальних ресурсів порівняно з SPH.

А от сіткові методи FDM, FEM та FVM, попри високу точність моделювання, видають значно нижчу продуктивність, що робить їх менш придатними для інтерактивних застосувань.

2.2. Технічні засоби моделювання рідин для вибраних методів

2.2.1. Екосистема Unity та базові інструменти для моделювання рідин

Unity, насправді, надає велике різноманіття інструментів для моделювання фізики рідин, хоча більшість важчих, ніж прості, симуляцій вже потребують сторонніх рішень або власних винаходів.

Основними вбудованими інструментами є:

1. Particle System – це базова система частинок, яка дозволяє створювати спрощені симуляції рідин. Ця система підтримує налаштування емітерів, колізії з об’єктами сцени та гравітацію, але при цьому має

обмеження в моделюванні взаємодії між частинками, що є найважливішим в реалістичній поведінці рідин.

2. Visual Effect Graph (VFG) – це спеціалізований інструмент, який забезпечує нодовий інтерфейс для роботи з великою кількістю частинок. VFG використовує GPU-прискорення, що значно підвищує продуктивність при створенні візуальних ефектів рідини, хоча й не забезпечує повноцінної фізичної симуляції їх поведінки.
3. Шейдери - в Unity використовуються різні типи шейдерів для моделювання рідин:
    - Surface Shaders - для реалістичного відображення оптичних властивостей рідин
    - Compute Shaders - для GPU-прискорених обчислень при застосуванні фізичних моделей
    - Geometry Shaders - для динамічної генерації геометрії поверхні рідини
4. Render Pipelines - сучасні системи рендерингу, такі як: URP (Universal Render Pipeline, універсальний конвеєр рендерингу) та HDRP (High Definition Render Pipeline, конвеєр рендерингу високої чіткості), надають розширені можливості для реалістичного відображення відбиття, заломлення, підповерхневого розсіювання.

Також, Unity дає можливість використовувати сторонні рішення.

Asset Store пропонує готові плагіни на основі SPH методу, наприклад, Fluvio, інструменти Flow для створення водних поверхонь, Obi Fluid для підтримки багатопоточності.

Unity дозволяє імпортувати професійні симуляції рідин завдяки Houdini Engine, а також переносити симуляції з інших програм, наприклад, Blender.

Нерідко розробники використовують власні реалізації та скрипти, написані на мові програмування C#. А коли ці реалізації поєднуються з вбудованими

інструментами, то виходить хороша комбінація для результата, збалансованого за реалістичністю, продуктивністю та складністю.

2.2.2. Спеціалізовані інструменти для реалізації різних методів моделювання рідин

На основі аналізу доступних джерел та доступних практичних рішень можна виділити ключові технічні засоби для реалізації трьох обраних методів моделювання рідин у середовищі Unity.

1. Для SPH методу:

- CPU-реалізації, що базуються на стандартних компонентах Unity і оптимізуються через просторове розбиття для ефективного пошуку сусідніх частинок
- GPU-прискорені рішення використовують обчислювальні шейдери для паралельної обробки великої кількості частинок, що суттєво підвищує продуктивність;
- Спеціалізовані плагіни як Fluvio та Obi Fluid надають готові рішення з оптимізованими алгоритмами та візуальними компонентами.

2. Для FLIP методу:

- Гібридні структури даних для зберігання як сітки (для обчислення тиску), так і частинок (для перенесення властивостей)
- Обчислювальні шейдери для ефективної реалізації операцій проектування швидкостей на сітку та оновлення частинок;
- Спеціалізовані системи візуалізації для створення поверхні рідини на основі розподілу частинок;
- Ключові параметри налаштування: розмір комірки сітки (cell size) та кількість частинок на комірку (particles per cell);

- Комерційні рішення, такі як Fluidity та RealFlow, надають оптимізовані реалізації з різними налаштуваннями.

3. Для LBM методу:

- 3D текстури та рендер-текстури для зберігання функцій розподілу та полів швидкості;
- Багатопрохідні обчислювальні шейдери для реалізації етапів колізії та поширення;
- Системи візуалізації векторних полів для відображення потоків та взаємодії з перешкодами;
- Типи решіток: у Unity реалізаціях найчастіше використовуються моделі D2Q9 для двовимірних та D3Q19 для тривимірних симуляцій (це впливає на точність);
- Доступні інструменти включають FlowX та адаптації відкритих бібліотек, таких як Palabos.

2.2.3. Технічні обмеження двигуна Unity

При використанні Unity для моделювання фізики рідин важливо враховувати технічні обмеження рушія, які впливають на вибір методу та якість реалізації. Ось, що я б виділила:

1. Обмеження продуктивності. Unity має суттєві обмеження продуктивності при роботі зі складними сценами та великою кількістю об'єктів. Це особливо критично для моделювання рідин, де часто потрібна взаємодія багатьох елементів. Порівняно зі спеціалізованими програмами для моделювання, Unity показує нижчу продуктивність при високодеталізованих сценах.
2. Обмеження системи рендерингу. Unity має обмеження у створенні фотореалістичних ефектів, які важливі для правдоподібного

відтворення рідин. Стандартні інструменти рендерингу не завжди забезпечують точне відображення оптичних властивостей води та інших рідин, таких як відбиття, заломлення та прозорість.

3. Платформні обмеження. Unity має значні відмінності у продуктивності та можливостях на різних платформах. Наприклад, рішення, які добре працюють на настільних комп'ютерах, можуть мати серйозні обмеження на мобільних пристроях або у веб-версіях
4. Проблеми з фізичним рушієм. Стандартний фізичний рушій Unity (PhysX) має обмеження при роботі з великою кількістю об'єктів, що взаємодіють. Це стає проблемою при частинкових методах моделювання рідин, де кількість взаємодій може бути дуже високою.
5. Обмеження імпорту та інтеграції. Досить часто виникають технічні складнощі при імпорті складних моделей з інших спеціалізованих програм. Це обмежує можливості використання готових симуляцій рідин, створених у спеціалізованому програмному забезпеченні.

Розуміння обмежень та проблем, що можуть виникнути, є критично важливим для подальшої розробки. Незважаючи на виявлені обмеження, Unity все ж залишається потужним інструментом із широкими можливостями для реалізації різноманітних методів моделювання рідин, тому всі висновки будуть безпосередньо застосовані у практичній частині.

Розділ 3

Практична реалізація моделювання рідин у Unity

3.1. Підготовка середовища Unity для роботи з фізикою рідин

На рисунку 1 зображено головний інтерфейс Unity у стандартній конфігурації після створення нового проєкту (в цьому випадку 3D-проєкту).

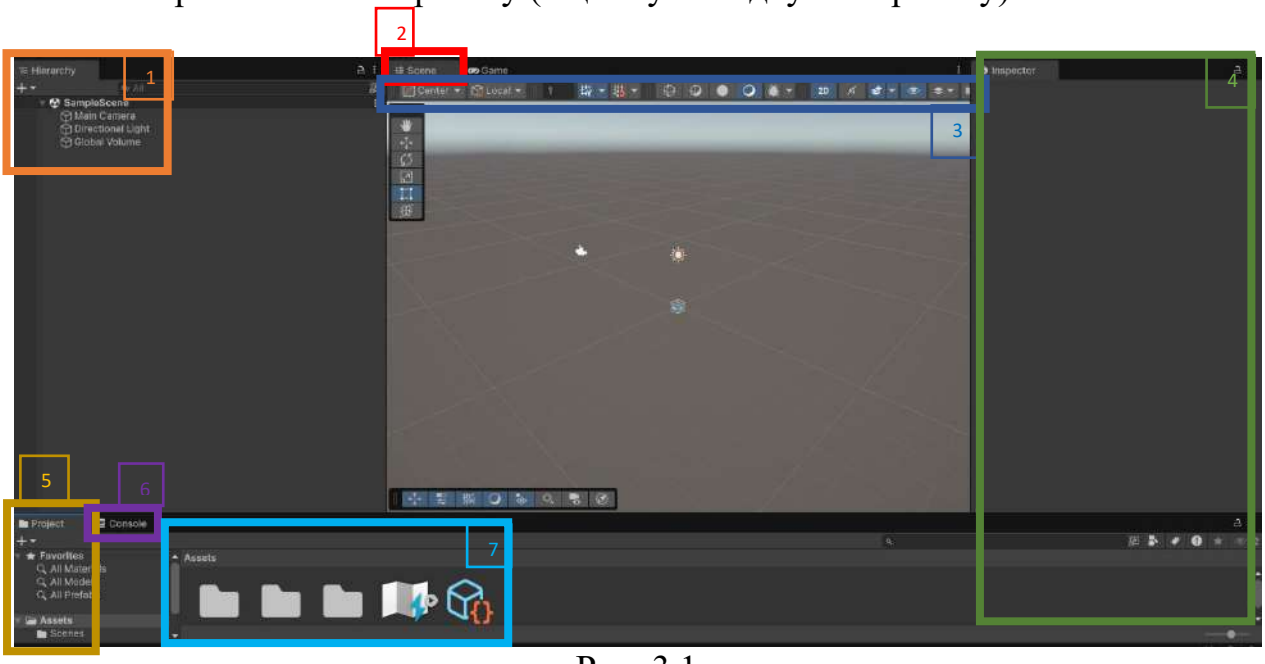

Рис. 3.1

Основні елементи середовища:

1) Hierarchy. Тут відображаються всі об’єкти, які є на сцені. У цьому випадку:

- Main Camera - головна камера, яка визначає, що побачить гравець;
- Directional Light - джерело світла, що імітує сонячне освітлення;
- Global Volume - об’єкт для глобальних постобробок (може використовуватись, зокрема, для ефектів води, таких як заломлення).

2) Scene View. Це основне вікно, де можна бачити, переміщати та змінювати об’єкти в просторі. На сцені зараз видно іконки камери, джерела світла та інші елементи.

3) Toolbar. Тут розміщені інструменти переміщення, обертання та масштабування об’єктів, а також кнопки перемикання між режимами (2D/3D, Gizmos, освітлення тощо).

4) Inspector. Ця панель використовується для перегляду та редагування властивостей вибраного об’єкта на сцені. Зараз вона порожня, бо жоден об’єкт не обрано.

5) Project. У цьому розділі зберігаються всі файли та ресурси проєкту (скрипти, моделі, матеріали, сцени тощо).

6) Console. Тут з’являються повідомлення про помилки, попередження або вивід із скриптів.

7) Assets. Це папки, де зберігаються всі імпортовані або створені файли для сцени.

На цьому етапі середовище вже готове для додавання фізичних компонентів і скриптів, які будуть відповідати за моделювання рідини.

3.2. Реалізація вибраних методів моделювання

3.2.1. Реалізація методу SPH

Для забезпечення контролю над симуляцією було створено спеціальний клас ***Config***, який містить усі необхідні константи та параметри.

```
public class Config : MonoBehaviour
{
    public static int N = 20;
    public static float SIM_W = 0.5f;
    public static float BOTTOM = -2f;
    public static float DAM = -0.3f;
    public static int DAM_BREAK = 200;
    public static float DT = 20f;
    public static float WALL_POS = 0.08f;
    public static float G = 0.02f * 0.25f;
    public static float SPACING = 0.08f
    public static float K = SPACING / 1000.0f
    public static float K_NEAR = K * 10f;
    public static float REST_DENSITY = 3.0f;
```

```
    public static float R = SPACING * 1.25f;
    public static float SIGMA = 0.2f;
    public static float MAX_VEL = 0.25f;
    public static float WALL_DAMP = 0.2f;
    public static float VEL_DAMP = 0.5f;
}
```

Принципи визначення параметрів симуляції:

1. Просторові параметри визначають фізичні межі симуляції:

- N = 20 - кількість частинок
- SIM_W = 0.5f - ширина простору симуляції
- BOTTOM = -2f та DAM = -0.3f - нижня межа та позиція віртуальної дамби

2. Часові параметри впливають на стабільність симуляції:

- DT = 20f - часовий крок (більший - швидша симуляція, менший - вища точність)

3. Фізичні параметри визначають властивості рідини:

- G = 0.02f * 0.25f - прискорення вільного падіння
- SPACING = 0.08f - початкова відстань між частинками
- K та K_NEAR - коефіцієнти для розрахунку сил тиску
- REST_DENSITY = 3.0f - густина рідини
- R = SPACING * 1.25f - радіус взаємодії частинок
- SIGMA = 0.2f - коефіцієнт в'язкості

4. Параметри стабілізації забезпечують надійність:

- MAX_VEL = 0.25f - обмеження максимальної швидкості
- WALL_DAMP = 0.2f та WALL_POS = 0.08f - взаємодія зі стінами
- VEL_DAMP = 0.5f - затухання швидкості для уникнення нестабільності

Підбір цих параметрів забезпечує баланс між реалістичністю поведінки рідини та обчислювальною ефективністю.

Наступним і, не менш важливим, класом є ***Particle***. Він є центральним елементом симуляції і відповідає за моделювання поведінки окремих частинок рідини.

```
public class Particle : MonoBehaviour
{
    public vector2 pos;
    public vector2 previous_pos;
```

```
    public vector2 visual_pos;
    public float rho = 0.0f;
    public float rho_near = 0.0f;
    public float press = 0.0f;
    public float press_near = 0.0f;
    public list neighbours = new list();
    public vector2 vel = vector2.zero;
    public vector2 force = new vector2(0f, -G);
    public float velocity = 0.0f;

    public int grid_x;
    public int grid_y;

    void Start()
    {
        pos = transform.position;
        previous_pos = pos;
        visual_pos = pos;
    }

    public void UpdateState()
    {
        previous_pos = pos;
```

розрахунок швидкості за формулою Ейлера [3.1]:

```
        vel += force * Time.deltaTime * DT;
```

розрахунок позиції за формулою Ейлера[3.2]:

```
        pos += vel * Time.deltaTime * DT;
        visual_pos = pos;
        transform.position = visual_pos;
        force = new vector2(0, -G);
```

розрахунок перерахунку швидкості за формулою [3.3]:

```
        vel = (pos - previous_pos) / Time.deltaTime / DT;
        velocity = vel.magnitude;
        if (velocity > MAX_VEL)
        {
            vel = vel.normalized * MAX_VEL;
        }

        rho = 0.0f;
        rho_near = 0.0f;

        neighbours = new list();

        if (pos.y < BOTTOM)
        {
            if (name != "Base_Particle")
            {
                Destroy(gameObject);
            }
        }
    }
```

```
    public void CalculatePressure()
    {
```

розрахунок тиску за формулою [3.4]:

```
        press = K * (rho - REST_DENSITY);
        press_near = K_NEAR * rho_near;
}

    void OnCollisionStay2D(Collision2D collision)
    {
        vector2 normal = collision.contacts[0].normal;
        float vel_normal = Vector2.Dot(vel, normal);
        if (vel_normal > 0)
        {
            return;
        }
        vector2 vel_tangent = vel - normal * vel_normal;
```

розрахунок відбиття за формулою [3.5]:

```
        vel = vel_tangent - normal * vel_normal * WALL_DAMP;
        pos = collision.contacts[0].point + normal * WALL_POS;
    }
}
```

Метод **Start** встановлює початкові позиції. **UpdateState** оновлює фізичні властивості частинки на кожному кроці симуляції. **CalculatePressure** обчислює тиск на основі густини. **OnCollisionStay2D** забезпечує реалістичну взаємодію частинок зі стінами.

Цей клас реалізує основні принципи методу, де кожна частинка зберігає свій стан (позицію, швидкість, густину, тиск) та взаємодіє з іншими частинками через сили тиску та в'язкості.

Особливістю реалізації є система просторового розбиття (grid_x, grid_y), яка оптимізує пошук сусідніх частинок, що є критичним для продуктивності методу SPH.

Реалізовані математичні формули у коді:

[3.1] $v(t + \Delta t) = v(t) + a(t) \cdot \Delta t$, де $a(t) = \frac{F}{m}$ при $m = 1$

[3.2] $r(t + \Delta t) = r(t) + v(t) \cdot \Delta t$

[3.3] $v = \frac{\Delta r}{\Delta t}$

[3.4] $p = k(\rho - p_0)$

$p_{near} = K_{near} p_{near}$

[3.5] $v' = v_t - v_n \cdot n \cdot d$, де v_t- тангенціальна складова швидкості, v_n - нормальна складова, n - нормаль, d - коефіцієнт затухання.

Для створення потоку частинок рідини було реалізовано клас ***Shower***, який відповідає за генерацію нових частинок із заданою швидкістю та періодичністю.

```
public class Shower : MonoBehaviour
{
    public GameObject Simulation;
    public GameObject Base_Particle;
    public Vector2 init_speed = new Vector2(1.0f, 0.0f);
    public float spawn_rate = 1f;
    private float time;

    void Start()
    {
        Simulation = GameObject.Find("Simulation");
        Base_Particle = GameObject.Find("Base_Particle");
    }

    void Update()
    {
        if (Simulation.transform.childCount < 1000)
        {
            time += Time.deltaTime;
            if (time < 1.0f / spawn_rate)
            {
                return;
            }
            GameObject new_particle = Instantiate(Base_Particle,
transform.position, Quaternion.identity);

            new_particle.GetComponent<Particle>().pos =
transform.position;
            new_particle.GetComponent<Particle>().previous_pos =
transform.position;
            new_particle.GetComponent<Particle>().visual_pos =
transform.position;
            new_particle.GetComponent<Particle>().vel =
init_speed;

            new_particle.transform.parent =
Simulation.transform;
            time = 0.0f;
        }
    }
}
```

Цей клас має 2 основні методи: метод **Start** знаходить необхідні об'єкти в сцені, а **Update** створює нові частинки рідини з заданою періодичністю.

В результаті клас ***Shower*** забезпечує постійне надходження нових частинок у симуляцію, імітуючи джерело рідини. Параметри init_speed та spawn_rate

дозволяють контролювати початкову швидкість та інтенсивність потоку, що дає можливість моделювати різні типи джерел рідини.

Для запобігання надмірному навантаженню на систему реалізовано обмеження максимальної кількості частинок (1000), що забезпечує стабільну продуктивність симуляції.

Для створення динамічних перешкод, що взаємодіють з рідиною, реалізовано клас ***Wall***.

```
public class Wall : MonoBehaviour
{
    void Update()
    {
        if (Input.GetMouseButton(0))
        {
            Vector3 mousePos =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
            mousePos.z = 0;
            if (GetComponent<Collider2D>() ==
Physics2D.OverlapPoint(mousePos))
            {
                transform.position = mousePos;
            }
        }
    }
```

Цей клас відіграє важливу роль для інтерактивного тестування поведінки рідини при взаємодії з перешкодами. Він дозволяє користувачу переміщувати стіни в реальному часі під час роботи симуляції.

І на останок, центральний компонент реалізації методу SPH - клас ***Simulation***, який керує взаємодіями між частинками та загальним станом симуляції рідини.

```
public class Simulation : MonoBehaviour
{
    public list particles = new list();
    public GameObject Base_Particle;

    public int grid_size_x = 60;
    public int grid_size_y = 30;
    public list[,] grid;
    public float x_min = 1.8f, x_max = 6.4f;
    public float y_min = -1.4f, y_max = 0.61f;

    void Start()
    {
        Base_Particle = GameObject.Find("Base_Particle");
```

```
        grid = new list[grid_size_x, grid_size_y];
        for (int i = 0; i < grid_size_x; i++)
        {
            for (int j = 0; j < grid_size_y; j++)
            {
                grid[i, j] = new list();
            }
        }

        private float density, density_near, dist, distance,
normal_distance, relative_distance;
        private float total_pressure, velocity_difference, time;
        private vector2 pressure_force, particule_to_neighbor,
pressure_vector, normal_p_to_n, viscosity_force;
    }


public void calculate_density(list particles)
{
 foreach (Particle p in particles)
 {
            density = 0.0f;
            density_near = 0.0f;
  for (int i = p.grid_x - 1; i <= p.grid_x + 1; i++)
   {
      for (int j = p.grid_y - 1; j <= p.grid_y + 1; j++)
       {
         if (i >= 0 && i < grid_size_x && j >= 0 && j <
grid_size_y)
                    {
                    foreach (Particle n in grid[i, j])
                    {
                        dist = Vector2.Distance(p.pos, n.pos);

                            if (dist < R)
                            {
```

Розрахунок густини за формулою [3.6]:

```
                                normal_distance = 1 - dist / R;
                                p.rho += normal_distance *
normal_distance;
                                p.rho_near += normal_distance *
normal_distance * normal_distance;
                                n.rho += normal_distance *
normal_distance;
                                n.rho_near += normal_distance *
normal_distance * normal_distance;

                                p.neighbours.Add(n);
                            }
                        }
                    }
                }
```

```
            }
            p.rho += density;
            p.rho_near += density_near;
        }
    }
```

Метод **Start** ініціалізує просторову сітку, яка використовується для оптимізації пошуку сусідніх частинок.
Метод **Calculate_density** реалізовує обчислення густини для кожної частинки на основі розподілу сусідніх частинок.

```
    public void create_pressure(list particles)
    {
        foreach (Particle p in particles)
        {
            pressure_force = vector2.zero;

            foreach (Particle n in p.neighbours)
            {
                particule_to_neighbor = n.pos - p.pos;
                distance = Vector2.Distance(p.pos, n.pos);
```

Розрахунок сили тиску за формулою [3.7]:

```
                normal_distance = 1 - distance / R;
                total_pressure = (p.press + n.press) *
normal_distance * normal_distance +
                                 (p.press_near + n.press_near) *
normal_distance * normal_distance * normal_distance;
                pressure_vector = total_pressure *
particule_to_neighbor.normalized;

                n.force += pressure_vector;
                pressure_force += pressure_vector;
            }
            p.force -= pressure_force;
        }
    }

    public void calculate_viscosity(list particles)
    {
        foreach (Particle p in particles)
        {
            foreach (Particle n in p.neighbours)
            {
                particule_to_neighbor = n.pos - p.pos;
                distance = Vector2.Distance(p.pos, n.pos);
                normal_p_to_n =
particule_to_neighbor.normalized;
                relative_distance = distance / R;
                velocity_difference = Vector2.Dot(p.vel - n.vel,
normal_p_to_n);
```

Застосування сили в'язкості за формулою [3.8]:

```
                if (velocity_difference > 0)
                {
                    viscosity_force = (1 - relative_distance) *
velocity_difference * SIGMA * normal_p_to_n;
                    p.vel -= viscosity_force * 0.5f;
                    n.vel += viscosity_force * 0.5f;
                }
            }
        }
    }
```

Метод **Create_pressure** обчислює сили відштовхування між частинками на основі їх тиску.

Метод **Calculate_viscosity** моделює в'язкість рідини, застосовуючи силу, що вирівнює швидкості частинок.

```
    void Update()
    {
        particles.Clear();
        foreach (Transform child in transform)
        {
            particles.Add(child.GetComponent<Particle>());
        }

        for (int i = 0; i < grid_size_x; i++)
        {
            for (int j = 0; j < grid_size_y; j++)
            {
                grid[i, j].Clear();
            }
        }

        foreach (Particle p in particles)
        {
            p.grid_x = (int)((p.pos.x - x_min) / (x_max - x_min)
* grid_size_x);
            p.grid_y = (int)((p.pos.y - y_min) / (y_max - y_min)
* grid_size_y);


            if (p.grid_x >= 0 && p.grid_x < grid_size_x &&
p.grid_y >= 0 && p.grid_y < grid_size_y)
            {
                grid[p.grid_x, p.grid_y].Add(p);
            }
        }
        foreach (Particle p in particles) { p.UpdateState(); }
        calculate_density(particles);
        foreach (Particle p in particles) {
p.CalculatePressure(); }
```

```
            create_pressure(particles);
            calculate_viscosity(particles);
        }
    }
```

Метод Update є головним циклом симуляції, який координує всі інші методи. Він виконує такі кроки:

- Оновлення списку частинок з ієрархії об'єктів
- Очищення та оновлення просторової сітки
- Оновлення стану кожної частинки
- Розрахунок густини та тиску
- Застосування сил тиску та в'язкості

Реалізовані математичні формули у коді:

[3.6] $\rho_i = \Sigma_j W(r_{ij}, h)$

[3.7] $F^{pressure} = -\Sigma_j (P_i + P_j) \nabla W(r_{ij}, h)$

[3.8] $F^{viscosity} = \mu \Sigma_j (v_j - v_i) \nabla^2 w(1 r_{ij}, h)$

Отже, метод SPH було реалізовано власноруч через написання коду без використання готових рішень чи плагінів. Це дозволило повністю контролювати всі аспекти симуляції та краще зрозуміти фізичні процеси, що лежать в основі методу. Такий підхід повністю виправданий, оскільки для SPH часто практикується саме написання власного коду завдяки відносній простоті алгоритму та можливості тонкого налаштування параметрів.

На рисунку 3.2 можна побачити фінальний результат.

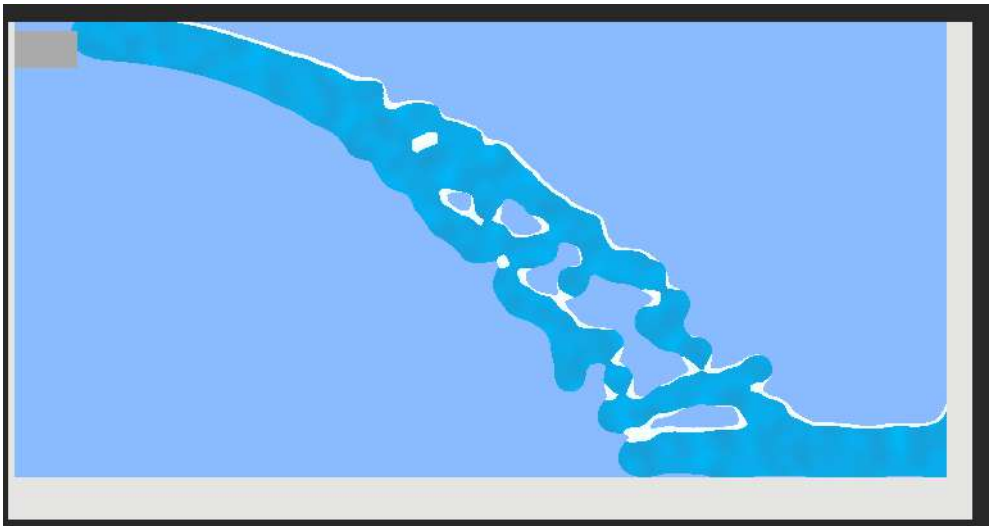

Рис. 3.2

Використання методу SPH для моделювання води з крану є обґрунтованим вибором, оскільки цей сценарій ідеально демонструє сильні сторони частинкового підходу: природнє формування вільної поверхні рідини, динамічне утворення крапель та бризок, а також реалістичну взаємодію з перешкодами. При цьому не потребує надмірної кількості частинок, що дозволяє досягти продуктивності в реальному часі навіть без значних оптимізацій.

3.2.2. Реалізація методу FLIP

На відміну від попереднього методу, який було реалізовано власноруч, для методу FLIP було використано набір спеціалізованих компонентів:

1. Обчислювальні шейдери (Compute Shaders) - спеціалізовані програми, що виконуються на GPU для масивних паралельних обчислень:

- ParticleAdvectionCS.compute - відповідає за рух частинок у потоці рідини
- ParticleInitCS.compute - ініціалізує частинки з початковими параметрами
- ParticleToGridCS.compute - переносить фізичні властивості з частинок на вузли сітки
- PressureProjectionCS.compute - обчислює поле тиску на сітці та забезпечує нестисливість рідини
- RenderingCS.compute - візуалізує результати симуляції

2. Допоміжні структури даних:

- Система сортування частинок (GridSortHelperCS.compute), яка оптимізує пошук сусідніх частинок
- Граничні умови (BoundaryCondition.hlsl), що визначають поведінку рідини при зіткненні з перешкодами
- Хеш-функції (Hash.hlsl) для ефективного просторового розподілу частинок

3. Оптимізаційні компоненти:

- Паралельні алгоритми сортування (RadixSort.compute)

- Операції префіксного сканування (PrefixScanCS.compute) для ефективної обробки великих масивів даних
- Утиліти для роботи з буферами GPU (GPUUtil.cs)

4. Візуалізаційні елементи:

- Шейдери для рендерингу поверхні рідини (SimulationArea.shadergraph)
- Градієнтні матеріали для реалістичного відображення рідини (CircularGradient.shadergraph)

А також клас ***FLIPSimulation,*** який є найголовнішим компонентом реалізації і відповідає за координацію всіх етапів симуляції рідини. Він реалізує інтерфейс IDisposable для коректного звільнення ресурсів графічного процесора.

```
public class FLIPSimulation : MonoBehaviour, IDisposable
{
    private struct Particle
    {
        public uint ID;
        public float3 Position;
        public float3 Velocity;
    }
    [SerializeField] private float _flipness = 0.99f;
    [SerializeField] private Vector3 _gravity = Vector3.down *
9.8f;
    [SerializeField] private float _viscosity = 0f;

    private GPUDoubleBuffer<Particle> _particleBuffer;
    private GPUBuffer<float4> _particleRenderingBuffer;

    private GPUComputeShader _particleInitCs;
    private GPUComputeShader _particleToGridCs;

    private GridSortHelper<Particle> _gridSortHelper;

    private void DispatchParticleToGrid()
{
    _gridSortHelper.Sort(_particleBuffer, _gridParticleIDBuffer,
GridMin, GridMax, GridSize, GridSpacing);

    var cs = _particleToGridCs;
    var k = cs.FindKernel("ParticleToGrid");
    k.SetBuffer("_ParticleBufferRead", _particleBuffer.Read);
    k.SetBuffer("_GridVelocityBufferWrite",
_gridVelocityBuffer);
    k.Dispatch(NumGrids);
}

private void DispatchGridToParticle()
```

```
{
    var cs = _gridToParticleCs;
    var k = cs.FindKernel("GridToParticle");

    cs.SetFloat("_Flipness", math.saturate(_flipness));

    k.SetBuffer("_ParticleBufferRW", _particleBuffer.Read);
    k.SetBuffer("_GridVelocityBufferRead", _gridVelocityBuffer);
    k.Dispatch(NumParticles);
}
```

Метод **DispatchParticleToGrid** переносить інформацію про швидкість з частинок на сітку.

Метод **DispatchGridToParticle** переносить оновлені швидкості з сітки назад на частинки.

```
private void DispatchExternalForce(bool isFirstIteration)

{

    var cs = _externalForceCs;

    var k = cs.FindKernel("AddExternalForce");

    cs.SetVector("_Gravity", _gravity);

    if (isFirstIteration)

    {

        var mouseRay =
Camera.main.ScreenPointToRay(Input.mousePosition);

        cs.SetVector("_MouseForceParameter", new
float4(_mouseForce, _mouseForceRange, 0, 0));

    }

    k.SetBuffer("_GridVelocityBufferRW", _gridVelocityBuffer);

    k.Dispatch(NumGrids);

}

private void DispatchPressureProjection()

{

    var cs = _pressureProjectionCs;

    var k = cs.FindKernel("CalcDivergence");

    k.SetBuffer("_GridVelocityBufferRead", _gridVelocityBuffer);

    k.SetBuffer("_GridDivergenceBufferWrite",
_gridDivergenceBuffer);
```

```
    k.Dispatch(NumGrids);

    k = cs.FindKernel("Project");

    for (uint i = 0; i < _pressureProjectionJacobiIteration;
i++)

    {

        k.SetBuffer("_GridPressureBufferWrite",
_gridPressureBuffer.Write);

        k.Dispatch(NumGrids);

        _gridPressureBuffer.Swap();

    }

}
```

Метод **DispatchExternalForce** додає зовнішні сили до поля швидкості, включаючи гравітацію та взаємодію з користувачем.

Метод **DispatchPressureProjection** забезпечує нестисливість рідини через проекцію поля тиску.

```
  private void Update()
    {
        for (int i = 0; i < _frameSimulationIteration; i++)
        {
            DispatchParticleToGrid();
            DispatchExternalForce(i == 0);
            DispatchDiffusion();
            DispatchPressureProjection();
            DispatchGridToParticle();
            DispatchAdvection();
            if (_activeDensityProjection)
DispatchDensityProjection();
        }
        RenderParticles();
    }
```

Метод **Update** організовує послідовність операцій, які забезпечують гібридну симуляцію: спочатку властивості частинок переносяться на сітку, там виконуються ефективні розрахунки фізичних взаємодій (зовнішні сили, в'язкість, тиск), потім оновлені дані повертаються до частинок і відбувається їх переміщення.

Параметр _frameSimulationIteration дозволяє виконувати кілька циклів симуляції за один кадр для підвищення точності при збереженні візуальної плавності.

Всі математичні розрахунки в FLIP реалізовані в обчислювальних шейдерах, на які код лише посилається через виклики.

Отже, для реалізації методу FLIP було використано спеціалізований фреймворк з готовими шейдерами та структурами даних, що є типовим підходом для цього методу. Така стратегія цілком виправдана, оскільки FLIP є значно складнішим алгоритмом, що вимагає ефективної роботи з GPU та оптимізованих структур даних для швидкого обміну інформацією між сіткою та частинками.

На рисунку 3.3 можна побачити фінальний результат.

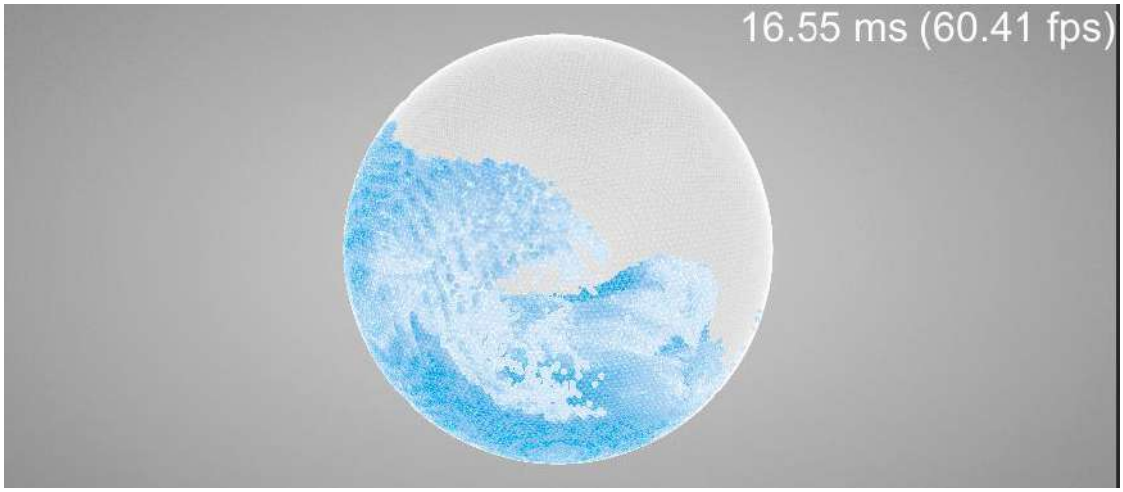

Рис. 3.3

Використання методу FLIP для моделювання коливання води в сферичному контейнері є обґрунтованим вибором, оскільки цей сценарій ідеально демонструє сильні сторони гібридного підходу: точне дотримання граничних умов, високу стабільність при моделюванні замкнутих об'ємів та збереження дрібних деталей турбулентних потоків.

3.2.3. Реалізація методу LBM

Для методу LBM був використаний шейдер "Lattice Boltzmann Methods/Flow Viz", що відповідає за відображення результатів симуляції LBM. Він фокусується саме на візуальному представленні потоків рідини:

- Перетворює дані про швидкість потоку у кольорове представлення
- Виділяє зони турбулентності та завихрення
- Відображає розподіл тиску через градієнти кольорів
- Підтримує різні режими візуалізації граничних умов та перешкод

Цей шейдер працює напряму з рендер-текстурами, які містять результати обчислень методу LBM, забезпечуючи ефективне візуальне відображення складної гідродинамічної поведінки.

Для ефективної реалізації методу LBM були використані такі системні технології:

1. Система задач Unity (Job System) для паралельного виконання обчислень:

- ComputeEquilibriumDistributionJob - розрахунок рівноважних розподілів
- CollideJob - моделювання зіткнень (релаксації до рівноваги)
- StreamJob - поширення розподілів між вузлами решітки

2. Технологія рендер-текстур для ефективного зберігання та візуалізації даних симуляції:

- UpdateFlowTexture - візуалізація поля швидкостей
- UpdateHeightTexture - візуалізація розподілу висоти рідини
- UpdateMaskTexture - візуалізація розташування твердих перешкод
-

Також було створено клас ***LbmDebugViz***, який забезпечує інтерактивну візуалізацію та взаємодію з симуляцією.

```
public class LbmDebugViz : MonoBehaviour
{
    [SerializeField] private RawImage _flowRawImage;
    [SerializeField] private RawImage _maskRawImage;
    [SerializeField] private LbmSimulator _lbmSimulator;

    private EventSystem _eventSystem;
    private GraphicRaycaster _graphicRaycaster;
}
private void OnEnable()
{
    _eventSystem = EventSystem.current;
    _graphicRaycaster = FindObjectOfType<GraphicRaycaster>();

    _lbmSimulator.FlowTextureUpdated -=
LbmSimulator_FlowTextureUpdated;
    _lbmSimulator.FlowTextureUpdated +=
LbmSimulator_FlowTextureUpdated;

    _lbmSimulator.MaskTextureUpdated -=
LbmSimulator_MaskTextureUpdated;
    _lbmSimulator.MaskTextureUpdated +=
LbmSimulator_MaskTextureUpdated;
}

private void OnDisable()
{
```

```
    _lbmSimulator.FlowTextureUpdated -=
LbmSimulator_FlowTextureUpdated;
    _lbmSimulator.MaskTextureUpdated -=
LbmSimulator_MaskTextureUpdated;
}
```

Метод **OnEnable** ініціалізує системи обробки введення та підписується на події оновлення текстур від симулятора.

Метод **OnDisable** скасовує підписки на події, щоб уникнути витоків пам'яті при відключенні компонента.

```
private void LbmSimulator_FlowTextureUpdated(object sender,
Texture2D e)
{
    _flowRawImage.texture = e;
    _flowRawImage.rectTransform.sizeDelta = new Vector2(e.width,
e.height);
    _flowRawImage.GetComponent<AspectRatioFitter>().aspectRatio
= e.width / (float)e.height;
}

private void LbmSimulator_MaskTextureUpdated(object sender,
Texture2D e)
{
    _maskRawImage.texture = e;
    _maskRawImage.rectTransform.sizeDelta = new Vector2(e.width,
e.height);
    _maskRawImage.GetComponent<AspectRatioFitter>().aspectRatio
= e.width / (float)e.height;
}
```

Метод **LbmSimulator_FlowTextureUpdated** обробляє подію оновлення текстури потоку рідини, встановлюючи нову текстуру та налаштовуючи розмір зображення.

Метод **LbmSimulator_MaskTextureUpdated** обробляє подію оновлення текстури маски перешкод, оновлюючи відповідне зображення з правильними пропорціями.

```
private void Update()
    {
        if (Input.GetKey(KeyCode.Mouse0))
        {
            var pointerEventData =
                new PointerEventData(_eventSystem)
                {
                    position = Input.mousePosition
                };

            var results = new List<RaycastResult>();
            _graphicRaycaster.Raycast(pointerEventData,
results);
```

```
            foreach (var result in results)
            {
                if (result.gameObject ==
_flowRawImage.gameObject)
                {
                    RectTransformUtility.ScreenPointToLocalPoint
InRectangle(_flowRawImage.rectTransform, Input.mousePosition,
null, out var localMousePositionXY);
                    var localPositionXY = new
Vector2(_flowRawImage.rectTransform.rect.x,
_flowRawImage.rectTransform.rect.y);
                    var uv = (localMousePositionXY -
localPositionXY) / _flowRawImage.rectTransform.rect.size;
                    _lbmSimulator.AddSolidNodeCluster(uv);
                    break;
                }
            }
        }
    }
```

Метод **Update** перевіряє натискання лівої кнопки миші, визначає, чи натискання відбулося на зображенні потоку, і якщо так, перетворює координати миші в координати симуляції та додає перешкоду в цьому місці.

Центральним компонентом реалізації методу LBM є клас ***LbmSimulator***, який відповідає за повний цикл моделювання рідини на двовимірній решітці.

public class LbmSimulator : MonoBehaviour

```
{
    private const float GravitationalForce = 9.8f;
    [SerializeField] private float _simulationStepTime = 0.016f;
    [SerializeField] private float _latticeSpacingInMeters =
0.05f;
    [SerializeField] private int _latticeWidth = 65;
    [SerializeField] private int _latticeHeight = 193;
    [SerializeField] private float _relaxationTime = 0.51f;

    private NativeArray<float2> _linkDirection;
    private NativeArray<sbyte> _linkOffsetX;
    private NativeArray<sbyte> _linkOffsetY;

    private NativeArray<byte> _solid;
    private NativeArray<float2> _velocity;
    private NativeArray<float> _height;
    private NativeArray<float> _lastDistribution;
}
private static void InitializeLinkData(
    out NativeArray<float2> linkDirection,
    out NativeArray<sbyte> linkOffsetX,
```

```
    out NativeArray<sbyte> linkOffsetY)
{
    linkDirection = new NativeArray<float2>(8,
Allocator.Persistent);
        var _linkOffsetX = new sbyte[] { 1,  1,  0, -1, -1, -1,
0,  1 };
    var _linkOffsetY = new sbyte[] { 0,  1,  1,  1,  0, -1, -1,
-1 };

    for (var linkIdx = 0; linkIdx < 8; linkIdx++)
    {
        var angle = PiOverFour * linkIdx;
        linkDirection[linkIdx] = math.normalize(new
float2(math.cos(angle), math.sin(angle)));
        if (linkIdx % 2 == 1)
        {
            linkDirection[linkIdx] *= math.SQRT2;
        }
        linkOffsetX[linkIdx] = _linkOffsetX[linkIdx];
        linkOffsetY[linkIdx] = _linkOffsetY[linkIdx];
    }
}
```

Метод **InitializeLinkData** ініціалізує масиви для дискретних напрямків моделі D2Q9, обчислює вектори цих напрямків з використанням тригонометричних функцій, нормалізує їх для забезпечення правильних фізичних властивостей, і зберігає відповідні зміщення для подальшого використання в алгоритмі поширення.

Модель D2Q9 (2-вимірна з 9 швидкостями) є стандартною моделлю для двовимірних LBM симуляцій, де кожен вузол решітки має 8 напрямків руху до сусідніх вузлів плюс центральний напрямок (стан спокою).

```
public void AddSolidNodeCluster(float2 uv)
        {
            var colRowIdx =
math.int2(math.round(math.saturate(uv) * new
float2(_latticeWidth - 1, _latticeHeight - 1)));
            var rowIdx = colRowIdx.y;
            var colIdx = colRowIdx.x;

            AddSolidNode(math.int2(colIdx - 0, rowIdx - 1));
            AddSolidNode(math.int2(colIdx - 1, rowIdx + 0));
            AddSolidNode(math.int2(colIdx - 0, rowIdx + 0));
            AddSolidNode(math.int2(colIdx + 1, rowIdx + 0));
            AddSolidNode(math.int2(colIdx - 0, rowIdx + 1));
        }
```

Метод **AddSolidNodeCluster** перетворює UV-координати (нормалізовані координати між 0 та 1) у реальні індекси решітки, а потім додає перешкоду у

формі хреста з п'яти твердих вузлів. Це дозволяє користувачеві інтерактивно додавати перешкоди в потік рідини під час симуляції, щоб спостерігати за зміною характеру потоку в реальному часі.

```
private void Update()
        {
            ComputePerFrameTerms();
            var inverseESq = 1.0f / (_e * _e);
            var smagorinskyConstantSq = _smagorinskyConstant *
_smagorinskyConstant;
            var relaxationTimeSq = _relaxationTime *
_relaxationTime;

            if (_lastJobHandles != null)
            {
                _lastJobHandles.Value.Item1.Complete();
                _lastJobHandles.Value.Item2.Complete();
                _lastJobHandles.Value.Item3.Complete();
                _lastJobHandles = null;

                NativeArray<float>.Copy(_height, _heightResult);
                NativeArray<float2>.Copy(_velocity,
_velocityResult);
                NativeArray<byte>.Copy(_solidResult, _solid);

                SimulationStepCompleted?.Invoke(this,
EventArgs.Empty);
                UpdateTextures(_maxHeight, MaxSpeed);
                UpdateMarkers();
                DumpStats();
            }
```

Метод **Update** послідовно виконує всі етапи алгоритму LBM:

- Оновлення параметрів симуляції
- Релаксація до рівноваги (колізія) - моделює локальні зіткнення частинок
- Поширення (streaming) - переміщення функцій розподілу між вузлами решітки
- Обчислення макроскопічних величин (висота, швидкість)
- Застосування граничних умов (вхідні та вихідні потоки)
- Розрахунок нових рівноважних розподілів

Всі математичні розрахунки в LBM виконуються з використанням стандартних математичних функцій Unity.

Отже, для реалізації методу LBM було використано системні компоненти Unity та спеціалізовані структури даних, що дозволило ефективно

використовувати паралельні обчислення. Такий підхід виправданий, оскільки метод Болцмана на решітці вимагає регулярних операцій з великими масивами даних, але при цьому має високий ступінь локальності обчислень, що робить його ідеальним для паралельної обробки на багатоядерних процесорах. Використання системи задач Unity (Job System) та оптимізований доступ до пам'яті через нативні масиви забезпечили високу продуктивність симуляції в реальному часі.
Результат можна побачити на рисунку 3.4.

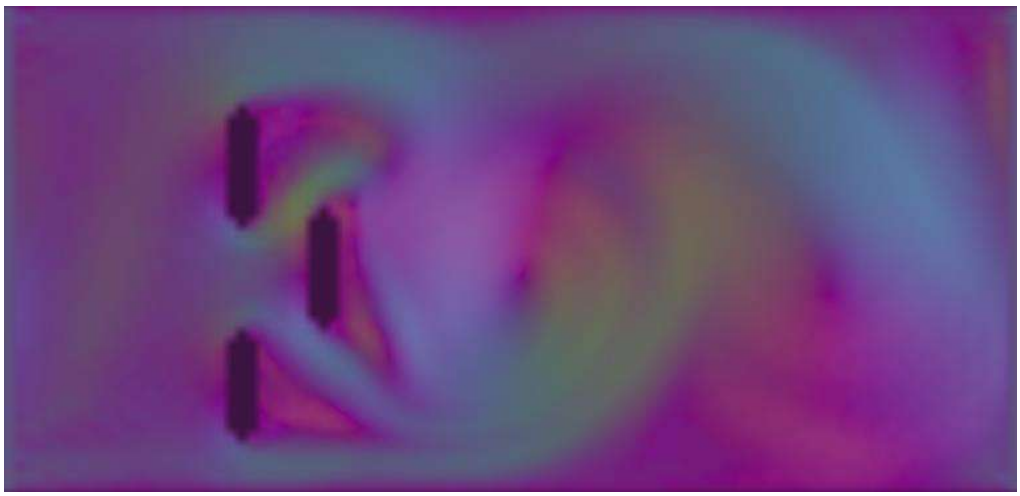

Рис. 3.4

Використання методу LBM для моделювання потоку рідини навколо перешкоди є обґрунтованим вибором, оскільки цей сценарій ідеально демонструє сильні сторони підходу Болцмана на решітці: моделювання обтікання об'єктів складної форми та природне формування турбулентних зон за перешкодами.

3.3. Аналіз результатів моделювання

На основі результатів практичної реалізації можна провести порівняльний аналіз методів за ключовими характеристиками, які є визначальними для розробки та очевидні при первинному оцінюванні, представлений в таблиці 3.1.

Характеристика	SPH	FLIP	LBM
Базовий підхід	Частинковий	Гібридний	Решітковий
Основні обчислювальні компоненти	CPU (C# скритпи)	GPU (обчислювальні шейдери)	CPU (системні компоненти)
Математична реалізація	Явні формули в коді	Інкапсуляція з шейдерів	Розподілена між системними компонентами
Підхід до візуалізації	Безпосереднє відображення частинок	Спеціалізовані шейдери	Рендер-текстури
Змодельований сценарій	Потік води з крану	Коливання води у сфері	Обтікання перешкоди потоком
Сильні сторони	Проста і зрозуміла реалізація, пряме керування параметрами частинок	Ефективне використання GPU, менш помітна «зернистість»	Ефективність системи компонентів

Табл. 3.1

Важливою складовою порівняльного аналізу є також оцінка продуктивності методів при різній кількості елементів симуляції.

Для вимірювання продуктивності було використано Unity Profiler, який дозволяє відстежувати такі метрики як частота кадрів (FPS), час обробки кадру, використання центрального (CPU) та графічного (GPU) процесорів, а також обсяг використаної пам'яті.

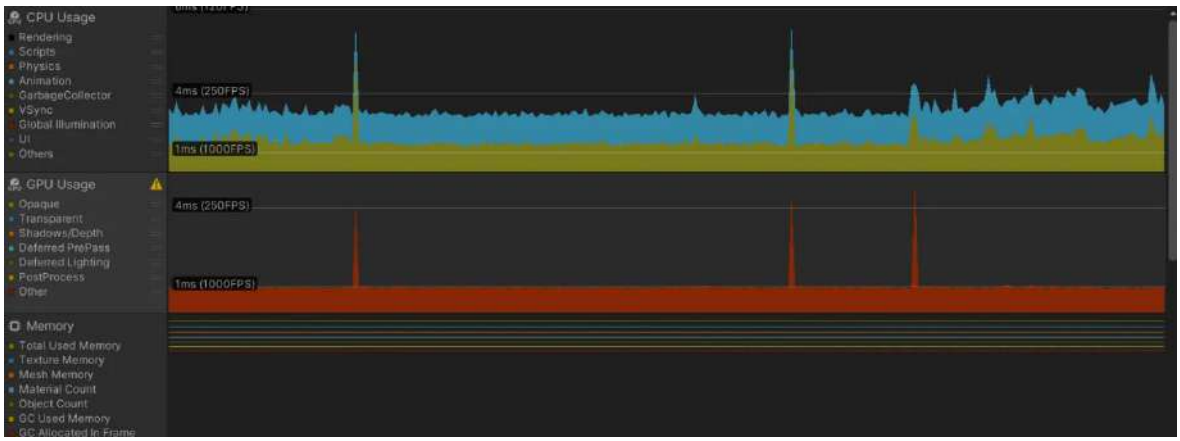

Рис. 3.5

На основі даних із Profiler (рис. 3.5) для методу FLIP можна зробити наступні висновки:

Використання CPU (CPU Usage):

- Графік показує стабільну продуктивність з частотою кадрів близько 250-1000 FPS;
- Спостерігаються окремі піки навантаження на CPU, але загальний рівень використання процесора залишається низьким
- Синя область графіка показує, що більшість операцій скриптів виконуються дуже ефективно, оскільки основні обчислення перенесені на GPU
- Стабільність роботи CPU вказує на добре оптимізовану архітектуру коду

Використання GPU (GPU Usage):

- Помаранчева область показує значне використання GPU
- Видно чіткі високі піки, які відповідають моментам інтенсивних обчислень на шейдерах;
- Загальний рівень використання GPU значно вищий за CPU, що підтверджує, що підтверджує ефективне перенесення обчислень на графічний процесор

Пам'ять (Memory):

- Графік пам'яті демонструє стабільне використання ресурсів без суттєвих коливань
- Відсутність помітного зростання використання пам'яті з часом свідчить про ефективне управління ресурсами та відсутність витоків пам'яті

Ці дані підтверджують, що метод FLIP забезпечує найвищу продуктивність серед досліджуваних методів завдяки ефективному використанню GPU через обчислювальні шейдери.

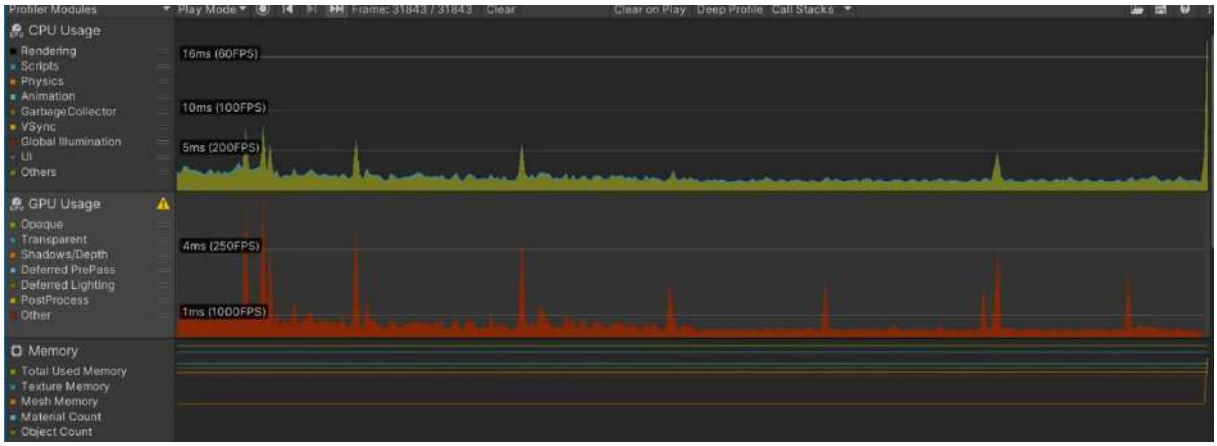

Рис. 3.6

На основі даних із Profiler (рис. 3.6) для методу SPH можна зробити наступні висновки:

Використання CPU (CPU Usage):

- Графік показує стабільну частоту кадрів близько 60 FPS, що відповідає стандартній частоті оновлення дисплея
- Помітні регулярні піки навантаження (сині піки), що свідчить про циклічні інтенсивні обчислення
- Значна частина обчислень відбувається на CPU, що підтверджує високе навантаження на центральний процесор при використанні C# скриптів
- Жовто-синя заповнена область графіка вказує на рівномірний розподіл навантаження між системними процесами та скриптами фізики

Використання GPU (GPU Usage):

- Червона область показує помірне використання GPU з піками де-інде
- Піки навантаження на GPU відповідають моментам рендерингу кадрів з високою кількістю частинок

Пам'ять (Memory):

- Графік пам'яті показує стабільне використання ресурсів з повільним зростанням, що відповідає поступовому додаванню нових частинок у симуляцію
- Відносно низький рівень використання пам'яті, оскільки SPH зберігає лише необхідні дані для кожної частинки
- Невеликі коливання у нижній частині графіка пам'яті свідчать про ефективну роботу збирача сміття (garbage collector)

Ці дані підтверджують, що метод SPH, хоча і забезпечує стабільну роботу при помірній кількості частинок, має обмеження в продуктивності через високе навантаження на CPU. Це добре узгоджується з очікуваною поведінкою алгоритму, заснованого на C# скриптах без використання шейдерної оптимізації.

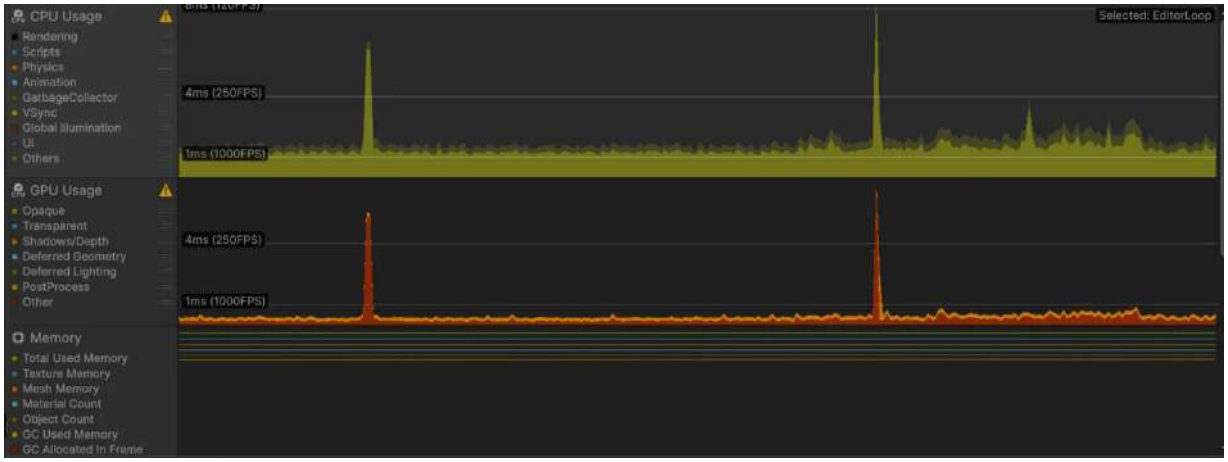

Рис. 3.7

На основі даних із Profiler (рис. 3.7) для методу LBM можна зробити наступні висновки:

Використання CPU (CPU Usage):

- Графік демонструє високу продуктивність з частотою кадрів у діапазоні 250-1000 FPS
- Характерною особливістю є наявність окремих виразних піків, що відповідають моментам перебудови решітки або застосування граничних умов
- Між піками спостерігається стабільна робота з низьким рівнем навантаження на CPU
- Жовта область графіка вказує на рівномірний розподіл навантаження, з більшим акцентом на системні компоненти, а не на скрипти

Використання GPU (GPU Usage):

- Помаранчева область показує помірне використання GPU
- Більшість часу графічний процесор працює з низьким навантаженням, оскільки він використовується переважно для візуалізації результатів
- Піки активності GPU відповідають моментам оновлення текстур для візуалізації потоків та взаємодії з перешкодами

Пам'ять (Memory):

- Демонструє найнижчий рівень використання ресурсів серед усіх трьох методів
- Стабільні горизонтальні лінії без помітних коливань вказують на ефективне управління пам'яттю

Ці дані підтверджують, що метод LBM забезпечує збалансовану продуктивність з ефективним використанням обох процесорів. Особливо важливою перевагою є найнижче споживання пам'яті, що робить LBM найбільш економічним методом з точки зору використання ресурсів.

Також, ще одним важливим аспектом порівняння методів моделювання рідин є їх масштабованість, тобто здатність зберігати прийнятну продуктивність при збільшенні кількості елементів симуляції.

Для дослідження цього аспекту було проведено серію тестів, де поступово збільшувалась кількість частинок або вузлів решітки для кожного методу і вимірювалась частота кадрів (FPS).

Результати цих тестів представлено на рисунку 3.8.

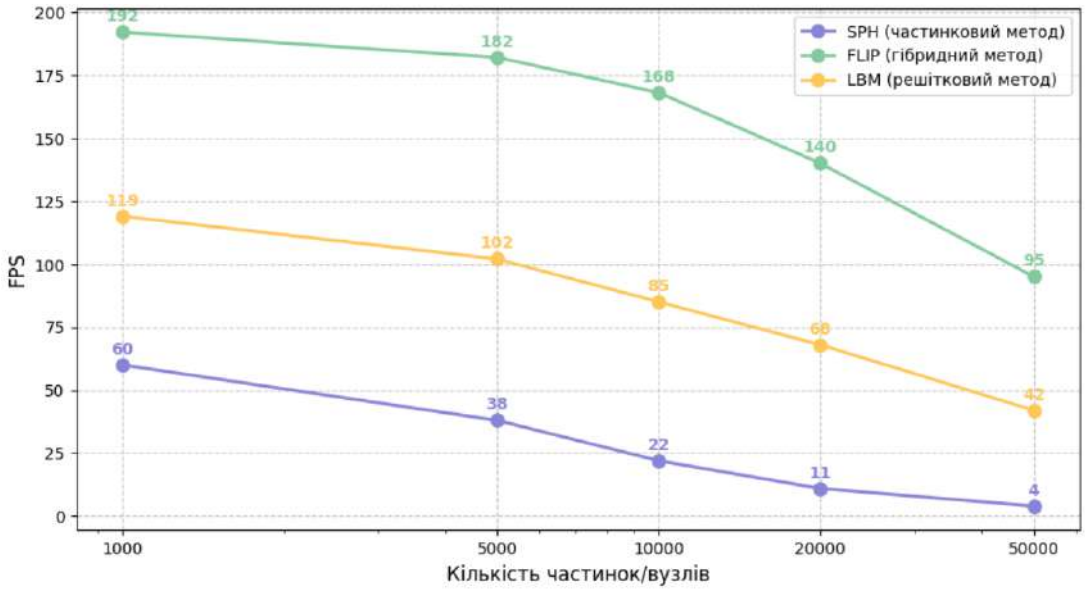

Рис. 3.8

Отримані дані демонструють суттєву різницю в масштабованості досліджуваних методів, як видно з графіка на рисунку 3.8:

- Метод FLIP показує найвищу продуктивність на всіх рівнях деталізації та найкращу масштабованість, бо при збільшенні кількості частинок з 1000 до 50000 FPS знижується з 192 до 95, зберігаючи прийнятну швидкість
- Метод SPH демонструє найбільш різке падіння продуктивності: з 60 FPS при 1000 частинках до критично низьких 4 FPS при 50000 частинках, що підтверджує його обмежену масштабованість

- Метод LBM займає проміжне положення з лінійним зниженням продуктивності: з 119 FPS при 1000 вузлах до 42 FPS при 50000 вузлах решітки

Проведене практичне дослідження підтвердило більшість теоретичних висновків, зроблених у розділі 2.

Таким чином, при виборі методу моделювання рідин для конкретного проєкту в Unity необхідно враховувати не лише візуальні вимоги, але й цільову платформу та доступні обчислювальні ресурси. Після аналізу усіх аспектів можна сказати, що FLIP забезпечує найкращу продуктивність при високій деталізації, проте вимагає сучасних GPU; LBM є хорошим компромісом для проєктів, де важлива реалістична поведінка потоків; а SPH підходить для проєктів, де не потрібна висока деталізація або є обмеження на використання шейдерів.

Висновки

У ході виконання курсової роботи було проведено дослідження методів моделювання реалістичної фізики рідин у середовищі Unity. На основі теоретичного аналізу для практичної реалізації було обрано три методи: SPH, FLIP та LBM, які представляють різні підходи до моделювання рідин.

Також була проведена практична реалізація цих методів у Unity, яка підтвердила їх ефективність у різних сценаріях застосування.

Порівняльний аналіз реалізованих методів показав, що кожен з них має власні технічні особливості та підходи до обчислень, що відповідають їх математичній природі. Цей аналіз підтверджує теоретичні висновки, зроблені в першій частині роботи.

Перспективними напрямками подальших досліджень є розробка гібридних підходів, що поєднують переваги різних методів, оптимізація частинкових методів за допомогою GPU-прискорення та інтеграція з технологіями машинного навчання для прогнозування поведінки рідин.

Використані джерела:

1. Mike Seymour. The Science of Fluid Sims. 2017.

https://www.fxguide.com/fxfeatured/the-science-of-fluid-sims/

2. Ann Steffora Mutschler. The Era Of Fluid Simulations In Hollywood. 2022. https://semiengineering.com/the-era-of-fluid-simulations-in-hollywood/

3. Shahriyar Shahrabi. Gentle Introduction to Fluid Simulation for Programmers and Technical Artists. 2022.

https://shahriyarshahrabi.medium.com/gentle-introduction-to-fluid-simulation-for-programmers-and-technical-artists-7c0045c40bac

4. Wikipedia. Computational fluid dynamics. https://en.wikipedia.org/wiki/Computational_fluid_dynamics

5. Wikipedia. Fluid animation.

https://en.wikipedia.org/wiki/Fluid_animation

6. Mike Ash. Fluid Simulation for Dummies. 2007. https://mikeash.com/pyblog/fluid-simulation-for-dummies.html

7. DIVE CAE. CFD Methods.

https://www.divecae.com/resources/cfd-methods

8. Matthias Müller. Fast and Stable Fluid-Solid Coupling for Incompressible SPH. https://cg.informatik.uni-freiburg.de/intern/seminar/gridFluids_fluid-EulerParticle.pdf

9. Y. Zhu, R. Bridson. FLIP: A Low-Dissipation Particle-in-Cell Method for Fluid Flow. 2005.

https://www.researchgate.net/publication/222452290_FLIP_A_Low-Dissipation_Particle-in-Cell_Method_for_Fluid_Flow

10. Junfeng Pan, Jiankai Li, Dan Xu. Geometric Deep Learning for Fluid Simulation: A Survey. 2024.

https://www.mdpi.com/2673-3951/5/1/15

11. Particleworks Europe. SPH & MPS Methods.

https://particleworks-europe.com/SPH-MPS.php

12. Robert Bridson. Fluid Simulation for Computer Graphics. 2015. https://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf

13. Jan Bender, Matthias Müller, Miles Macklin. SPH Techniques for the Physics Based Simulation of Fluids and Solids. 2019.

https://animation.rwth-aachen.de/media/papers/64/2019-EG-SPH_Tutorial.pdf

14. Unity Technologies. Particle Systems.
https://docs.unity3d.com/Manual/ParticleSystems

15. Chen Jingjing, Ma JunLiang, Lu Chen. Recent advances in fluid simulation using machine learning. 2023.

https://www.sciopen.com/article/10.1007/s41095-023-0368-y

16. Dan Koschier, Jan Bender, Barbara Solenthaler. Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids. 2019.

https://sph-tutorial.physics-simulation.org/pdf/SPH_Tutorial.pdf

17. Daily.dev. Unity Fluid Simulation Tutorial: CPU and GPU Methods.
https://daily.dev/blog/unity-fluid-simulation-tutorial-cpu-and-gpu-methods

18. Kui Wu, Nghia Truong, Cem Yuksel, Rama Hoetzlein. GVDB-FLIP: Fast, Accurate Particle-Based Fluid Simulation on a Sparse GPU Voxel Grid. 2018.
https://people.csail.mit.edu/kuiwu/GVDB_FLIP/gvdb_flip.pdf

19. Keijo Mattila. Implementation Techniques for the Lattice Boltzmann Method. 2010.

https://www.researchgate.net/profile/Keijo-Mattila/publication/268042210_Implementation_Techniques_for_the_Lattice_Boltzmann_Method/links/560244c808ae42bbd541f8dd/Implementation-Techniques-for-the-Lattice-Boltzmann-Method.pdf

20. Atit Vyas, Rahul Shagwat, Manikant Singh. A systematic review on forecasting river water quality using machine learning approaches. 2025.
https://link.springer.com/article/10.1007/s13201-025-02400-w

21. Guido Moretti, Alberto Pini, Riccardo Bianchi Janetti, Marco Mulas, Attilio Toscano. A novel approach for flood simulation adopting the lattice boltzmann method. 2017.

https://isprs-annals.copernicus.org/articles/IV-4-W4/161/2017/isprs-annals-IV-4-W4-161-2017.pdf

22. Vagon. Common Unity Problems and How to Solve Them.

https://vagon.io/blog/common-unity-problems-and-how-to-solve-them

23. Alexandre Sajus. Unity Fluid Simulation. 2023.
https://github.com/AlexandreSajus/Unity-Fluid-Simulation?tab=readme-ov-file

24. Abe Combe. FLIP-Fluid-for-Unity. 2023.

https://github.com/abecombe/FLIP-Fluid-for-Unity

25. Andrew Kircher. LBM-SWE-Unity. 2021.

https://docs.unity3d.com/Manual/ProfilerMemory.html