

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.92

«До захисту допущено»

Завідувач кафедри

_____ Євгенія СУЛЕМА

«__» _____ 2025 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою

**«Інженерія програмного забезпечення мультимедійних та
інформаційно-пошукових систем»**

зі спеціальності 121 Інженерія програмного забезпечення

**на тему: «Спосіб та програмне забезпечення для симуляції поведінки
рідин в реальному часі»**

Виконала:

Студентка II курсу, групи КП-41мн
Седухіна Аліна Дмитрівна

Керівник:

Доцент кафедри ПЗКС, к.ф.-м.н., доц.
Нещадим Олександр Михайлович

Консультант із нормоконтролю:

Доцент кафедри ПЗКС, к.т.н., доц.
Онай Микола Володимирович

Рецензент:

Доцент кафедри СПСКС, к.т.н.
Боярінова Юлія Євгенівна

Засвідчую, що у цій магістерській
дисертації немає запозичень із
праць інших авторів без
відповідних посилань.

Студентка _____

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Євгенія СУЛЕМА

«___» _____ 2024 р.

ЗАВДАННЯ

на магістерську дисертацію студентці

Седухіній Аліні Дмитрівні

1. Тема дисертації: «Спосіб та програмне забезпечення для симуляції поведінки рідин в реальному часі», науковий керівник дисертації Нещадим Олександр Михайлович, к.ф.-м.н., доц., доцент кафедри ПЗКС, затверджені наказом № 4836-С по університету від «б» листопада 2025 р.
2. Термін подання студенткою роботи: «19» грудня 2025 р.
3. Об'єкт дослідження: процес генерації даних для симуляції рідин.
4. Предмет дослідження: методи та способи генерації даних для симуляції рідин.
5. Перелік завдань, які потрібно розробити:
 - провести аналіз існуючих методів та способів симуляції рідин,
 - розробити спосіб симуляції рідин в реальному часі,
 - розробити програмне забезпечення симуляції рідини в реальному часі використанням розробленого способу,
 - провести тестування програмного забезпечення,
 - провести аналіз отриманих результатів,
 - побудувати бізнес-модель.
6. Орієнтовний перелік графічного(ілюстративного) матеріалу:
 - блок-схема запропонованого способу (плакат),
 - діаграма архітектури програмного забезпечення (плакат),
 - приклад роботи розробленого програмного забезпечення (плакат),
 - вибір швидкостей сітки на які впливає частинка (плакат),
 - клітина сітки для розробленого способу (плакат),
 - бізнес-модель стартапу (плакат).

7. Орієнтовний перелік публікацій:

- тези доповіді “Спосіб та програмне забезпечення для симуляції рідин в реальному часі”.

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	к.т.н., доцент Онай М. В.		

9. Дата видачі завдання: «5» жовтня 2024 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Ознайомлення з предметною галуззю	01.11.2024	
2	Визначення структури дипломної роботи; вивчення літератури, пошук додаткової літератури	24.12.2024	
3	Проведення порівняльного аналізу існуючих методів симуляції рідин	20.02.2025	
4	Підготовка матеріалів першого розділу роботи	10.03.2025	
5	Розробка способу симуляції рідин в реальному часі	03.05.2025	
6	Підготовка матеріалів другого розділу роботи	25.06.2025	
7	Підготовка матеріалів третього розділу роботи	25.08.2025	
8	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження; підготовка матеріалів доповіді на конференції ПМК-2025	24.10.2025	
9	Завершення роботи над основною частиною дипломної роботи, підготовка ілюстративного матеріалу	22.11.2025	
10	Оформлення текстової та графічної частини дипломної роботи	12.12.2025	

Студентка _____

Аліна СЕДУХІНА

Науковий керівник _____

Олександр НЕЩАДИМ

РЕФЕРАТ

Актуальність теми. На сьогоднішній день можливості обладнання стрімко ростуть з кожним роком, разом з цим відкриваючи все нові можливості для створення швидших та реалістичніших симуляцій. Зокрема це стосується симуляції рідин, в області якої щорічно виходить десятки робіт створюючи нові методи, покращуючи існуючі та вивчаючи взаємодію з іншим об'єктами симуляції, наприклад, тканинами. Ці методи використовують в широкому спектрі індустрій, починаючи з розваг, кіно, ігри, закінчуючи симуляціями для наукових робіт, наприклад, у області фізики.

Покращення методів є актуальною задачею та має практичне застосування.

Об'єктом дослідження є процес генерації даних для симуляції поведінки рідин.

Предмет дослідження є методи та способи генерації даних для симуляції поведінки рідин.

Мета дослідження: підвищення швидкодії процесу симуляції поведінки рідин.

Наукова новизна: удосконалено метод симуляції рідин, характерною рисою якого є застосування адаптивного порогу до шару корекції густини, що дозволило прискорити симуляцію в середньому на 20.84%.

Практична значущість. Запропонований підхід може бути імплементований розробниками як у вже існуючих програмних рішеннях, так і під час створення нових застосунків. Крім того, програмну реалізацію методу виконано у вигляді плагіна для Unreal Engine, що забезпечує можливість його використання широким колом користувачів: розробниками ігор, архітекторами, фахівцями зі створення візуальних ефектів та іншими спеціалістами

Апробація роботи: частково основні положення і результати роботи представлені та обговорювались на XVIII науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2025 та опубліковані у збірнику тез доповідей.

Структура та обсяг роботи: магістерська дисертація складається з вступу, п'яти розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, пояснено актуальність розглянутої проблеми. Сформовано мету і задачі дослідження. Показано наукову новизну та практичну цінність роботи.

У першому розділі проведено аналіз існуючих рішень. В рамках якого було описано існуючі методи симуляції рідин та розділено їх на групи. Для кожної з груп було виділено переваги та недоліки методів, що до неї входять. На основі зібраних даних було обрано групу методів з якою було проведено подальшу роботу.

У другому розділі детально проаналізовано алгоритм методу обраного для покращення. На основі його слабких сторін було запропоновано вдосконалення методу. Також, запропонований спосіб було описано по етапах.

У третьому розділі було описано програмну реалізацію запропонованого способу. Було проаналізовано, обрано та аргументовано вибір інструментів програмної реалізації способу. Крім того, було описано архітектуру створюваного програмного забезпечення та описано його окремі модулі.

У четвертому розділі було проведено тестування створеного програмного забезпечення. Результати було проаналізовано та зроблено висновки.

У п'ятому розділі було побудовано модель стартапу, продуктом якого є розроблене програмне забезпечення. В рамках побудови стартапу було проведено аналіз цільової аудиторії, виділено конкурентні переваги та

унікальну ціннісну пропозицію. На основі всієї інформації створено бізнес модель.

У висновках проаналізовано отримані результати.

Робота виконана на 73 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 34 найменувань. У роботі наведено 17 рисунків, 9 таблиць.

Ключові слова: інженерія програмного забезпечення, симуляція рідин, обчислювальна гідродинаміка, комп'ютерна графіка.

ABSTRACT

Theme urgency. Today, the capabilities of equipment are growing rapidly every year, opening up new opportunities for creating faster and more realistic simulations. This is especially true for fluid simulation, where dozens of papers are published every year, creating new methods, improving existing ones, and studying interactions with other simulation objects, such as fabrics. These methods are used in a wide range of industries, from entertainment, cinema and games to simulations for scientific work, for example in the field of physics.

The object of the research is the process of generating data for simulating fluid behaviour.

The subject of the research is the methods and means of generating data for simulating fluid behaviour.

The purpose of the research is to increase the speed of the fluid behaviour simulation process.

Scientific novelty. The fluid simulation method has been improved by adding an adaptive threshold to the density correction layer, which has accelerated simulation by 20.84% and solved the problem of excessive local fluid density in real time.

Practical value. The proposed approach can be implemented by developers both in existing software solutions and when creating new applications. In addition, the software implementation of the method is performed as a plugin for Unreal Engine, which provides the opportunity for its use by a wide range of users: game developers, architects, visual effects specialists, and other professionals.

Work approbation. The main provisions and results of the work were presented and discussed at the XVIII scientific conference of master's and doctoral students 'Applied Mathematics and Computing' PMK-2025 and published in a collection of abstracts.

Structure and scope of the work. The master's thesis consists of an introduction, five chapters, conclusions and appendices.

The introduction provides a general description of the work and explains the relevance of the problem under consideration. The aim and objectives of the study are formulated. The scientific novelty and practical value of the work are demonstrated.

The first section analyses existing solutions. It describes existing methods of fluid simulation and divides them into groups. For each group, the advantages and disadvantages of the methods included in it are highlighted. Based on the collected data, a group of methods was selected for further work.

The second section provides a detailed analysis of the algorithm of the method selected for improvement. Based on its weaknesses, improvements to the method are proposed. Also, the proposed method was described in details. Each stage described step by step.

The third section describes the software implementation of the proposed method. The choice of tools for the software implementation of the method was analysed, selected and justified. In addition, the architecture of the software being developed and its individual modules were described.

In the fourth section, the created software was tested. The results were analysed and conclusions were drawn.

In the fifth chapter, a startup model was built, the product of which is the developed software. As part of the startup construction, an analysis of the target audience was carried out, competitive advantages and a unique value proposition were identified. Based on all the information, a business model was created.

The conclusions analyse the results obtained.

The work is presented on 73 pages, contains 3 appendices and references to a list of 34 sources used. The work contains 17 figures and 9 tables.

Keywords: software engineering, fluid simulation, CFD, computer graphics.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ	4
ВСТУП	5
1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ	7
1.1. Загальні положення та аналіз предметної області.....	7
1.2. Огляд існуючих рішень	10
1.3. Порівняння існуючих рішень.....	19
1.4. Висновки до розділу	22
2. РОЗРОБЛЕННЯ СПОСОБУ ДЛЯ СИМУЛЯЦІЇ ПОВЕДІНКИ РІДИН..	24
2.1. Постановка задачі.....	24
2.2. Опис запропонованих модифікацій	26
2.3. Детальний опис запропонованого способу	27
2.4. Висновки до розділу	37
3. ПРОЄКТУВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СИМУЛЯЦІЇ ПОВЕДІНКИ РІДИН.....	39
3.1. Вимоги до програмної реалізації.....	39
3.2. Інструменти реалізації програмного забезпечення (Стек)	40
3.3. Паралельні обчислення.....	41
3.4. Архітектура розроблюваної системи	42
3.5. Висновки до розділу	47
4. АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ РОЗРОБЛЕНОГО СПОСОБУ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	49
4.1. Верифікація роботи програмного забезпечення	49
4.2. Порівняння роботи за різних порогів.....	50
4.3. Висновки до розділу	51
5. ПОБУДОВА БІЗНЕС МОДЕЛІ	52
5.1. Зацікавлені сторони	52
5.2. Унікальна ціннісна пропозиція рішення та конкурентні переваги...	55
5.3. Доходи та витрати	57

5.4. Бізнес-модель.....	59
5.5. Висновки до розділу	63
ВИСНОВКИ	65
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	67
ДОДАТКИ	72

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

VFX – visual effects – візуальні ефекти.

CPU – central processing unit – також званий головним процесором або просто процесором, є основним процесором у певному комп'ютері.

GPU – graphics processing unit – окремий пристрій персонального комп'ютера або ігрової приставки, виконує графічний рендеринг.

ВСТУП

Вже щонайменше півстоліття активно триває розвиток симуляції реальних об'єктів у віртуальному середовищі. Цей напрям перебуває на перетині галузей прикладної математики, обчислювальної фізики та комп'ютерної графіки. Однією з найбільш актуальних та складних задач у цій галузі є симуляція рідин. Але повний розрахунок поведінки потоків рідини в загальному випадку вимагає розв'язання системи рівнянь Нав'є-Стокса [1], що є однією з відкритих проблем тисячоріччя, а отже станом на сьогодні не має єдиного розв'язку та навіть не доведено його існування. Тому якісні апроксимації є надзвичайно важливими та наразі залишаються єдиним практичним засобом симуляції рідин.

Технології цієї галузі знаходять застосування у багатьох різноманітних сферах. Зокрема, індустріях кіно та анімації, ігровій індустрії, та фізичні симуляції для науки та практичних задач інженерії. Особливо актуальною проблема стала з поширенням використання VFX в індустрії розваг, зокрема в кіно та відеоіграх. Великі студії, такі як Disney, навіть почали запускати свої дослідження по темі. Наприклад, одні із найсучасніших методів APIC (Affine Particle-In-Cell Method) та IPIC (Impulse Particle-In-Cell method) було розроблено при співпраці з ними. А приклад реалізації методу APIC та його застосування в реальних проєктах студії можливо побачити у серії мультфільмів Моана, де вода присутня протягом всього мультфільму, а тому і потребує симуляції практично в кожному кадрі.

Метою даної роботи є підвищення ефективності симуляції рідин в реальному часі шляхом поєднання існуючих сучасних методів.

В результаті дослідження, що було проведено в рамках виконання даної магістерської дисертації, було проведено аналіз різних методів та підходів до симуляції рідин, проаналізовано та порівняно їх. На основі проведеного аналізу запропоновано спосіб симуляції рідин, який

відрізняється від існуючих застосувань адаптивного критерію застосування шару корекції густини, що дозволяє підвищити швидкість симуляції.

1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ

Цей розділ магістерської дисертації присвячено аналізу предметної області, опису актуальності проблеми симуляції рідин, поясненню поняття симуляції рідин та її різновидів.

В рамках опису обраного різновиду симуляції описано сфери її використання, а також наведено програмне забезпечення, що використовується у різних сферах для симуляції рідин на основі фізики.

При проведенні аналізу існуючих рішень методи було розбито на декілька основних категорій, кожна з яких описано, виділяючи характерні для неї риси. Також для кожної категорії було наведено декілька методів представників, принцип роботи яких також було описано. Далі на основі зібраної інформації спочатку створено порівняння груп методів за виділеними властивостями, після чого дані зведено до таблиці переваг та недоліків. Переваги й недоліки проаналізовано та обрано групу методів з якою було найбільш доречно та перспективно працювати в рамках даної роботи.

1.1. Загальні положення та аналіз предметної області

Перед початком роботи необхідно уточнити, який саме тип симуляції рідин мається на увазі. Загальне поняття “симуляція рідин у віртуальному просторі” охоплює великий набір методів та способів, орієнтованих на суттєво різні аспекти моделювання. Серед них можна відокремити декілька видів:

- симуляцію тільки поверхні рідини,
- симуляцію середовища, тобто візуальний ефект знаходження у середовищі рідин та можливе підлаштування при знаходженні об’єкту всередині об’єму, наприклад зміна ефекту гравітації навколо,

- симуляція рідини на основі фізики, коли рідина моделюється у вигляді окремої сутності, набору її потоків.

У межах даної роботи увагу приділяється саме симуляції рідин на основі фізики та їх потоків, тобто методів, які з деякою точністю апроксимують рівняння Нав'є-Стокса [1], наведені у наступних формулах:

$$\frac{\partial \vec{v}}{\partial t} = -(\vec{v} \cdot \nabla) \vec{v} + \nu \Delta \vec{v} - \frac{1}{\rho} \nabla p + \vec{f}, \quad (1)$$

$$\nabla \cdot \vec{v} = 0. \quad (2)$$

Приклади використання подібних симуляцій можливо знайти у великій кількості галузей. Наприклад в індустріях кіно та анімації фізично достовірні моделі використовуються практично усюди. У фільмах Аватар 2, Моана, Пірати Карибського моря. Відеоіграх в Red Dead Redemption 2, Uncharted 4 та серії Just Cause. При чому, якщо одні використовують відоме та широко поширене програмне забезпечення, таке як Houdini, то інші обирають інший підхід та створюють цілі досліджувальні центри та студії для покращення якості анімації. Так велика кількість сучасних гібридних методів була створена у лабораторії Disney Research Studios, які створювали їх для анімації не тільки мультфільмів студії Disney, прикладом вже згадана вище Моана, а й для таких гігантів, як Pixar, Marvel, abc, Lucasfilm та інших.

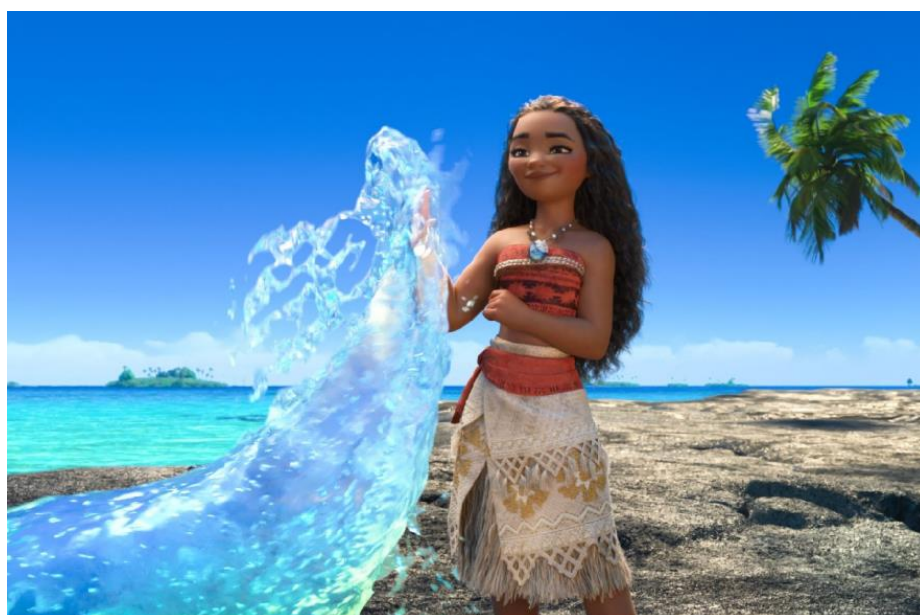


Рис. 1. Кадр симуляції рідини з мультфільму Моана [2]

В інженерії фізично достовірні симуляції взагалі є обов'язковими, так як використовуються для моделювання критично важливих систем, а від якості симуляції можуть залежати проекти вартістю у мільйони. Симуляція рідин використовується для аналізу вентиляційних систем, систем водопостачання, теплообміну, поведінки води у трубопроводах. Прикладами програмного забезпечення для цього є ANSYS Fluent, OpenFOAM, та FLOW-3D. Зокрема, останній активно використовується для аналізу розливів нафти в природне середовище та вивчення динаміки рідин під час морських перевезень.

У наукових дослідженнях симуляція рідин застосовується для моделювання великої кількості процесів. В галузі астрофізики прикладами є моделювання зіткнень зірок, формування галактик, симуляція руху плазми в цілому. Серед найбільш відомих прикладів програмного забезпечення для наукових астрофізичних симуляцій існують GADGET та ChaNGa. Також симуляції рідини використовують в галузі моделювання процесів магнітної гідродинаміки, в тому числі для моделювання процесу термоядерного синтезу, прикладом програмного забезпечення для такого випадку є NIMROD.

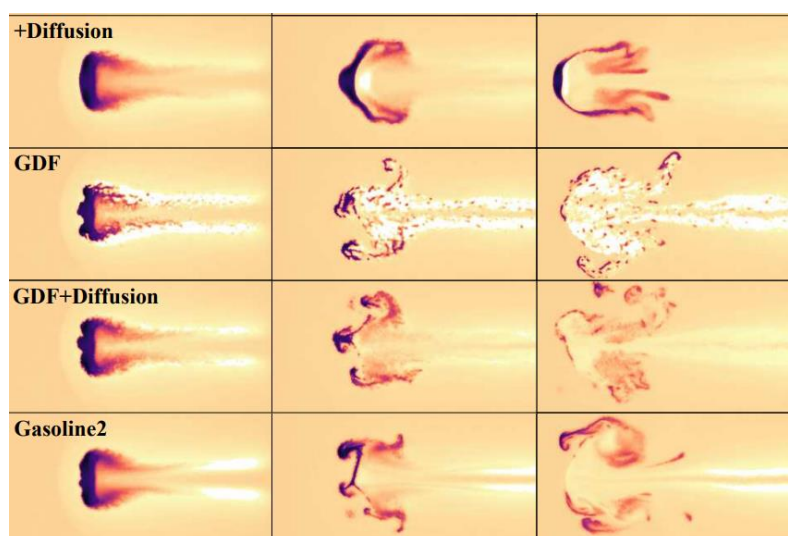


Рис. 2 Варіанти методів симуляції, що використовуються в астрофізиці [3]

Ці приклади наочно демонструють наскільки симуляція рідин є потужним та дуже важливим інструментом з практичної точки зору. Водночас, даний перелік застосувань не є повним, важливість і кількість практичних застосувань методів симуляції рідин складно переоцінити, що лише підкреслює актуальність подальших досліджень у цій сфері.

1.2. Огляд існуючих рішень

Існує велика кількість конкретних способів та методів симуляції рідин, які відрізняються ідеями, підходами та способами апроксимації руху рідини. Але незважаючи на існуючу варіацію способів для симуляції рідин, у загальному випадку існує можливість їх згрупувати. Таким чином кожен спосіб чи метод підпаде під одну з чотирьох чітко описаних категорії. При цьому, три такі категорії можливо вважати класичними та тісно пов'язаними між собою [4].

Названі три категорії, хоча й використовують різні підходи, але здебільшого використовують традиційні математичні методи для роботи. Тоді як четверта категорія працює на основі штучного інтелекту, сформувалась порівняно нещодавно [5], коли у світі виник тренд на розвиток та використання моделей на основі штучного інтелекту. І на відміну від інших ця категорія все ще активно розвивається і для знаходиться на етапі експериментів. Крім того, вона часто поєднується з традиційними методами для досягнення кращого результату.

Отже, можливо сказати, що всі відомі рішення відносяться до однієї з представлених груп методів:

- методи Лагранжа, або методи на основі частинок,
- методи Ейлера, або методи на основі сітки,
- гібридні методи,
- методи з використанням штучного інтелекту.

1.2.1. Методи на основі частинок

Також відомі як методи Лагранжа, через використання Лагранжевого поняття до опису руху потоків. Згідно з концепцією якого, для моделювання поведінки потоку рідини потрібно слідкувати за рухом окремих частинок, які утворюють цей потік у часі та просторі. Кожна частинка містить повний набір інформації про себе, наприклад такі характеристики як маса, тиск, швидкість, тощо [6, 7].

Перевагами методів цього типу є їх відносна простота та якість роботи на складних формах. Також на відміну від більшості інших методів вони не обмежені сіткою, що з одного боку надає більшу гнучкість, але з іншого з ними досить складно досягти стабільності. Крім того, вони потребують великої кількості частинок для досягнення реалістичного результату, що призводить до більшої кількості обчислювальних ресурсів затрачених на роботу такого методу [8].

В сучасному світі використовуються для візуальних ефектів в реальному часі, взаємодія рідин з твердими об'єктами, моделювання сумішей біологічних рідин [6].

Далі буде представлено декілька прикладів методів цієї групи. Прояснено принципи їх роботи, недоліки та переваги. А також на рисунку 3 було наведено приклад симуляції частинками.

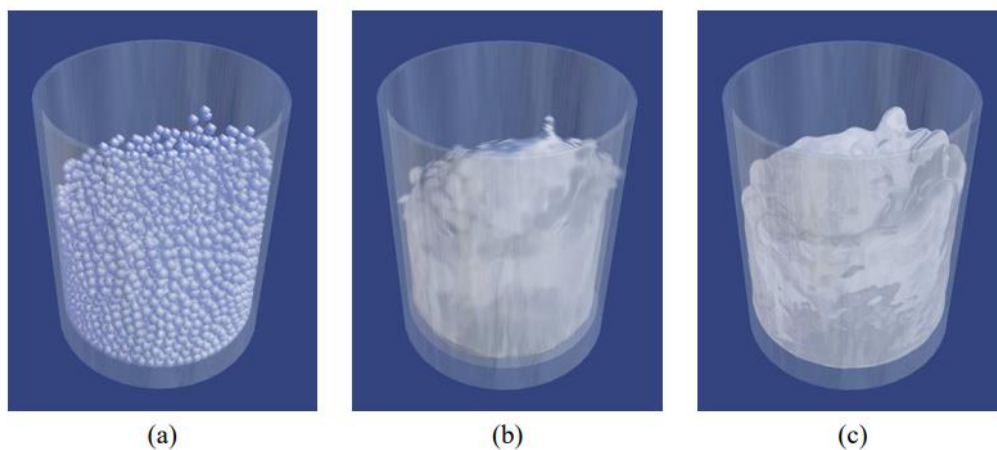


Рис. 3. Приклад симуляції частинками [7]

Гідродинаміка згладжених частинок

Гідродинаміка згладжених частинок, більш відома як SPH (Smooth Particle Hydrodynamics) – це метод, в якому частинки взаємодіють між собою. Кожна частинка впливає на своїх сусідів у заданому радіусі розмиття згладжуючи значення використовуючи функцію ядра для вирахування ваги внеску. Такий підхід дозволяє частинкам діяти більш злагоджено формуючи суцільний потік рідини, що в свою чергу забезпечує більш реалістичну поведінку [9].

Positional Based Fluids

Концепція цього методу об'єднує гідродинаміку згладжених частинок та один з відомих підходів симуляції – динаміка на основі розташування, основна ідея якого в першу чергу працювати не з силами, як було прийнято традиційно, а напряду редагувати положення часток враховуючи обмеження.

Ітерація цього методу виглядає наступним чином. Спочатку виконується демпінг швидкості та обчислюється прискорення частинки, для цього вираховується яким чином зовнішні сили впливали на неї протягом кроку. Потім обраховуються нові позиції частинок згідно швидкості та часу. І після цих обрахунків вираховуються динамічні обмеження і разом із статичними застосовуються до отриманих значень швидкості та позицій [10].

Такий підхід за рахунок обмежень надає більшу стабільність за гідродинаміку згладжених частинок.

1.2.2. Методи на основі сітки

Ця група методів також відома як Ейлерові методи. На противагу методам на основі частинок, у цих методах усі розрахунки виконуються у вузлах сітки. В цих вузлах сітки і моделюються потоки рідини [11]. В такому випадку кожен з вузлів має набір характеристик, наприклад, таких

як температура, тиск, в'язкість та інші величини залежно від конкретної галузі та задачі.

Також особливістю такого підходу є те, що кожна величина зберігається певним чином. Так типову клітину сітки зображено на рисунку нижче. В цьому випадку таку сітку використовували для моделювання крові, але така клітина є універсальною, її використання стала поворотним для індустрії симуляції рідин. Така структура називається MAC-сіткою [12], а зображення ілюструє саме її тривимірну варіацію.

Використання MAC-сітки надає рішення для проблем класичної сітки. В основному через те що чергування значень швидкості та тиску допомагає більш точно апроксимувати дивергенцію тиску, що в свою чергу впливає на точність його корекції, а тому й на точність всієї симуляції, так як саме операції з тиском вважаються найскладнішим та найбільш чутливим до помилок етапом симуляції.

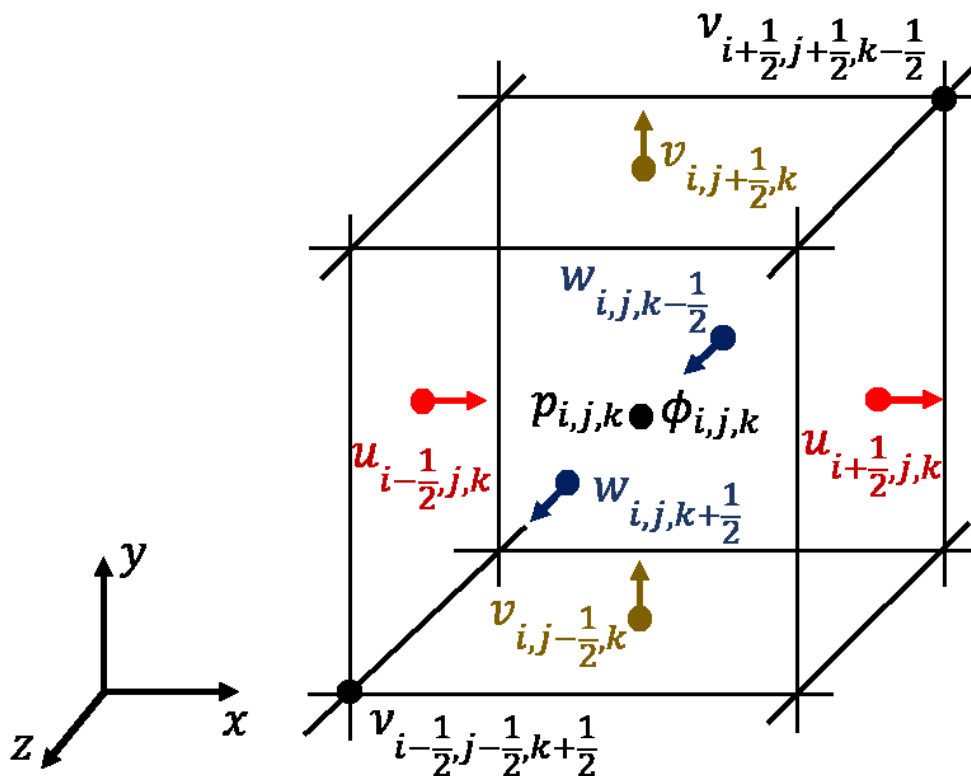


Рис. 4. Приклад MAC-сітки [13]

Загальний алгоритм кроку таких методів складається наступним чином. Спочатку розраховується адвекція, тобто перенос величин, найчастіше основною такою величиною є швидкість. Щоб порахувати адвекцію створюються віртуальні частинки існуючі на минулому кроці та рахують в якому положенні вони були, щоб опинитись в цьому положенні. Це називається зворотне відстеження. Після чого враховується вплив зовнішніх сил, таких як гравітація, на вектор швидкості. Далі проводиться корекція з врахуванням, за потреби, в'язкості та обов'язково тиску. Насамкінець розраховується корекція швидкостей сітки, та задовільнення граничних умов сітки [14].

Наразі ця група методів використовується для інженерних розрахунків в аерокосмічній, автомобільній, енергетичній та будівельній галузях, прогнозування кліматичних змін, руху течій, хвиль, цунамів, штормів, розрахунків взаємодії з газами та задачах газової динаміки.

1.2.3. Гібридні методи

Група методів, що поєднує принципи двох вже представлених категорій. Методи цієї групи одночасно працюють як з сіткою, так і з частинками на кожній ітерації. За загальним принципом всі обчислення кінематики проводяться на частинках, а динаміка на сітці. Дані на кожному кроці передаються між сіткою та частинками [12]. Спрощена схема класичного PIS зображена на рисунку.

В сучасності ця група методів використовується в широкому колі індустрій, таких як:

- автомобільна індустрія, для моделювання кузовної аеродинаміки, будівництво, для розрахунку впливу кліматичних умов, осадових матеріалів та землетрусів,
- ядерна енергетика, для моделювання плазми у реакторі,
- гідроенергетика, для розрахунків забудов та потоків в турбінах, медицина для складних симуляцій роботи органів,

- симуляції в області графіки та анімації, в індустріях кіно, мультфільмах та відеоіграх.

Далі представлено та описано декілька основних методів цієї групи, їх характеристики, відмінності один від одного та особливості.

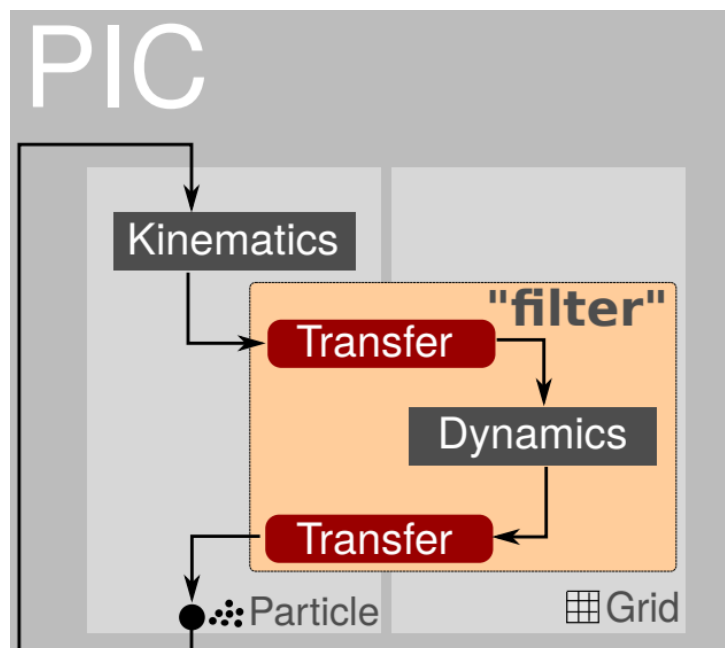


Рис. 5 Схема методу PIC [15]

PIC

PIC буквально розшифровується та перекладається як частинка в клітині. Класичний серед гібридних методів, був представлений ще в п'ятдесятих роках минулого століття для симуляції плазми [16]. Але з тих часів зазнав значних змін в багатьох аспектах, з фізичної, математичної та інженерної точки зору.

Перші версії мали великі проблеми із збереженням об'єму, неточностями апроксимації та швидкодією і ефективністю алгоритму. З роками та розвитком математичних методів, розвитком комп'ютерів та збільшенням кількості досліджень в області симуляції рідин покращення змогли частково вирішити ці проблеми. Прикладами таких покращень стали, наприклад, введення MAC сітки, яку вже було згадано в описі Ейлеревих

методів та експерименти з математичними методами апроксимації окремих етапів стимуляції таких етапів, як корекція тиску тиску [12].

Але незважаючи на великий прогрес у окремих етапах, PIS має стабільну незмінювану загальну схему метода з самого початку свого існування. За схемою спочатку для кожної частинки вираховуються інтерполяційні ваги використовуючи масу та швидкість частинки в даний момент часу. З використанням порахованих ваг вираховують швидкість в фіксованих точках, що в залежності від імплементації знаходяться на гранях або центрах клітин сітки. Далі використовуючи величини в сітці прораховується вплив зовнішніх сил на швидкість клітинки. А в кінці проводять трансформацію швидкості назад до частинок.

Недоліком такого методу є значна втрата кутового моменту. Такі втрати кутового моменту стаються за рахунок особливостей зворотної трансформації від сітки до частинок [14, 16]. що призводить до більш в'язкого, важкого вигляду. Через це, в сучасності, метод PIS у чистому вигляді практично не застосовують на практиці, віддаючи перевагу, більшу сучасним та динамічним методам та модифікаціям.

FLIP

Цей метод є не просто покращенням над класичним PIS, а також вважається одним із основних методів гібридної симуляції рідин в цілому. Хоча для роботи на сучасному ринку він і потребує модифікацій він є стабільною основою для роботи великої кількості програмного забезпечення. На відміну від PIS в цьому методі частинки постійно зберігають швидкість, а від сітки до них надходить тільки значення зміни швидкості. Такі зміни вирішили проблему втрат, але суттєво зменшили стабільність методу додавши шуму через використання нестабільного шляху перенесення швидкостей в якому для розрахунку нової швидкості частинки використовують швидкість минулої ітерації. Через це метод

доволі рідко використовують в чистому вигляді, найчастіше змішуючи з методом PIC [17].

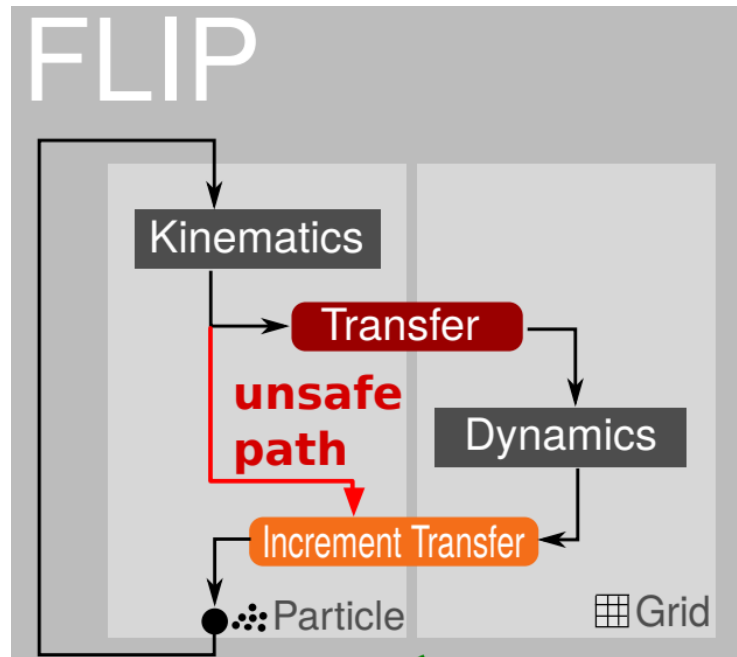


Рис. 6. Схема методу FLIP [15]

APIC

Повна назва *Affine Particle In Cell*. Також є покращенням над методами PIC. Навіть, покращує симуляцію на тому ж етапі, що й FLIP – на перенесені частинок. Але на відміну від FLIP, розробники пішли іншим шляхом, так як підхід FLIP може додавати багато шуму й неточності, що негативно впливає на симуляцію.

Тому в цьому підході замість того, щоб зберігати момент частинки між ітераціями було вирішено змінити сам спосіб переходу між частинками та сіткою, тепер кожна частинка має зберігати матрицю коефіцієнтів, що відповідає локальному градієнту швидкості, яку використовують для афінних трансформацій при переході на сітку [15].

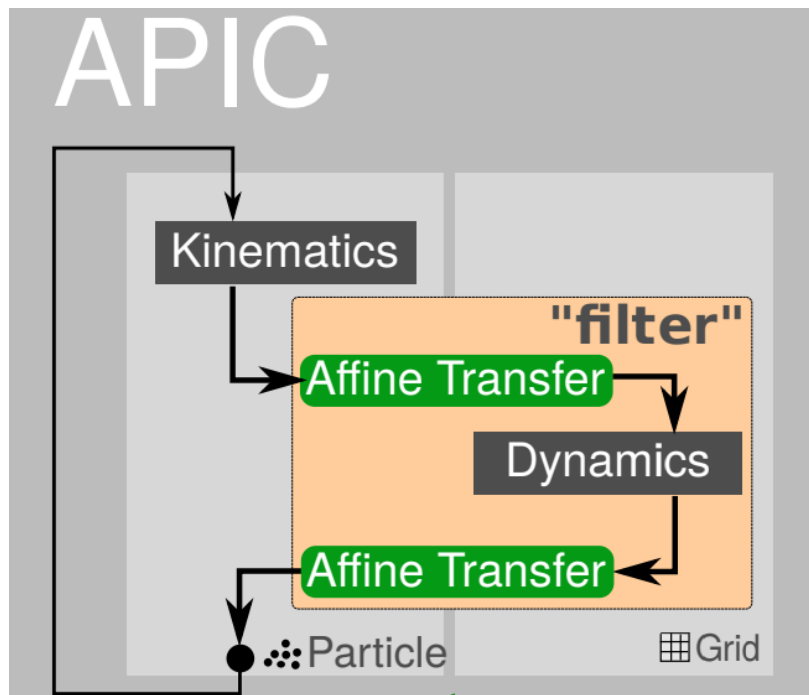


Рис. 7. Схема методу APIC [15]

IPIC

Метод є розширенням над методом APIC та FLIP, як на те вказує назва (Implicit Particle-In-Cell Method) його основною перевагою є збереження імпульсу. Такий підхід робить його найбільш якісним з наразі відомих методів на базі PIC. Але він має великий недолік, в більшості випадків через складність обрахунків його використання в реальному часі не є можливим на даний момент [18].

1.2.4. Методи з використанням штучного інтелекту

Методи з використанням штучного інтелекту є відносно молодим напрямом досліджень у сфері симуляції рідин. Більшість розробок цієї галузі все ще перебувають на стадії досліджень. Згідно дослідження 2023 року симуляції, які повністю спираються на штучний інтелект все ще не демонструють достатньої надійності для використання в більшості сфер. Але часто неймережеві моделі можуть бути вбудовані у програмне забезпечення на основі класичних традиційних методів, для вирішення

окремих задач, або покращення певних етапів, наприклад, вирішення систем лінійних рівнянь.

Для симуляцій, що повністю спираються на методи штучного інтелекту, ця група методів суттєво відрізняється від вже представлених класичних методів за принципом роботи. На відміну від інших вони не намагаються розв'язати рівняння Нав'є-Стокса, натомість використовуючи методи машинного навчання для того щоб навчитись апроксимувати поведінку рідини використовуючи вже існуючі симуляції. Такі методи потребують великої навчальної вибірки та їх реалізація є складною на етапі створення. Крім того такі симуляції є менш точними та стабільними за ті, що використовують класичні методи. Але водночас готові моделі є значно швидші в роботі за інші представлені методи [19, 20].

1.3. Порівняння існуючих рішень

Наразі було розглянуто усі основні групи методів симуляції рідини та детально описано кожну з них. Для кожної з груп приведено її особливості та загальні принципи роботи. Представлено декілька популярних методів або способів всередині цих груп, а також описано приклади їх використання, зазначаючи як загальні індустрії, так і конкретні задачі, що вирішуються цими методами. Надалі існує потреба в групуванні та стисненні викладеної інформації, а отже для подальшої роботи групи методів було порівняно. Для цього було виділено декілька основних характеристик і описано групи методів по кожній з них.

Отже порівняння проходило по таким характеристикам, як стабільність, складність реалізації та швидкодія в роботі. Також для кожної групи було виділено область використання. Готову таблицю в якій приведено подробиці та особливості методу по кожній з характеристик наведено нижче.

Характеристики груп методів симуляції рідин

Група методів	Методи Лагранжа	Методи Ейлера	Гібридні методи	Методи використанням штучного інтелекту
Властивість				
Область використання	Складні поверхні та форми	Великі об'єми рідини, поверхні рідин	Реалістичні симуляції	Швидкі симулятори для створення чорнових варіантів в кіно або іграх
Стабільність	Часто шумні	Стабільні	Стабільні	Нестабільні
Складність реалізації	Простий ітеративний	Простий ітеративний	Складніше за класичні через додаткову складність взаємодії між сіткою та частками.	Складне навчання великою кількістю параметрів
Швидкодія в роботі	Швидкі	Складні	Середні	Дуже швидкі

В таблиці наведено конкретні характеристики кожної з груп, але для подальшого створення висновків було прийнято рішення підсумувати її у таблицю, що представляє та виділяє конкретні переваги та недоліки кожного методу. Такий підхід дозволить більш чітко бачити реальну ситуацію та надасть можливість оцінити кожену групу більш зважено. По такому принципу було створено табл. 2, яка має всього два стовпці недоліків та переваг.

Таблиця 2

Переваги та недоліки груп методів симуляції рідин

	Переваги	Недоліки
Методи Лагранжа	Простота реалізації, можливість працювати із складними формами	Є досить шумними та для реалістичної симуляції потребують великої кількості частинок, що призводить до високих вимог до обчислювальних ресурсів
Методи Ейлера	Стабільні і досить точні, особливо на великих об'ємах	Через обрахунок тільки в конкретних клітинах має проблеми з малими деталями, можуть потребувати велику кількість ресурсів
Гібридні методи	Стабільні і відносно швидкі, тонке налаштування балансу швидкості і точності	Є складнішими в реалізацію через потребу в роботі одразу з сіткою, частинками та переходом між ними.
Методи з використанням штучного інтелекту	Дуже швидка кінцева модель	Обмеженість навчальними даними призводить до низької точності

За такою, більш стислою, таблицею простіше побачити факти про кожен з представлених категорій методів, що робить значно простішим процес аналізу та створення висновків, щодо вибору групи методів з якою більш доцільно та перспективно працювати надалі. Наступним етапом було коротко розглянуто кожен категорію методів окремо та зроблено висновки використовуючи таблицю.

Методи Лагранжа мають перевагу в тому, що вони прості в реалізації та за своєю природою легко підлаштовуються під складні поверхні та форми. Але їх великим недоліком є те, що для реалістичного результату вони потребують великої кількості ресурсів.

Методи Ейлера є гарним рішенням для великих об'ємів рідини та мають переваги, що більше підходять до інженерних розрахунків, тоді як на малих масштабах ця група методів часто втрачає деталі, або потребує забагато ресурсів для обчислення.

Гібридні методи, поєднують переваги обох груп, стабілізуючи частинки за рахунок операцій на сітці та зберігаючи деталі за рахунок частинок. Такий підхід може надавати якісні, реалістичні симуляції, але має додаткову складність реалізації за рахунок постійного переходу, але якісний результат.

Методи з використанням штучного інтелекту досі нові та нестабільні. Автономні моделі використовують у обмеженій кількості галузей і вони непригідні для симуляцій де точність важлива.

Серед виділених груп гібридні методи наразі є найбільш збалансованими між швидкістю та якістю, тому саме їх було обрано для спроби подальше покращити.

1.4. Висновки до розділу

У першому розділі магістерської дисертації було оглянуто актуальність проблеми симуляції рідин, описано індустрії їх використання та конкретні випадки застосування.

Також було розглянуто та проаналізовано методи симуляції рідин. Для цього методи було спочатку розбито по групах, кожену групу окремо описано та виділено ключові особливості. Після чого представлено по декілька методів кожної групи та описано загальні принципи їх роботи. На основі викладеної інформації проаналізовано групи методів для виявлення переваг та недоліків.

У контексті магістерської роботи, основним завданням є розробка алгоритму та програмного забезпечення симуляції рідин, що зможе одночасно давати якісний результат та працювати в реальному часі. Маючи на увазі ці особливості було виділено гібридну групу методів, як саму

перспективну для подальшої роботи, так як саме вона поєднує переваги сітки та частинок, сплачуючи в основному складністю реалізації.

2. РОЗРОБЛЕННЯ СПОСОБУ ДЛЯ СИМУЛЯЦІЇ ПОВЕДІНКИ РІДИН

Цей розділ магістерської дисертації присвячено опису розробці способу симуляції рідин. На початку розділу описано основу задачу розроблюваного способу та вимоги до нього. Далі в тексті описані основні запропоновані модифікації. А також створено детальний опис етапів роботи запропонованого способу, показано його загальний алгоритм.

2.1. Постановка задачі

Основною задачею даного дослідження є розробка такого способу симуляції рідин, який забезпечуватиме високу точність фізичного моделювання та при цьому залишатиметься придатним для виконання в режимі реального часу. Інакше кажучи, мета полягає в досягненні максимально можливої реалістичності при збереженні балансу між якістю симуляції та продуктивністю обчислень.

Робота в реальному часі передбачає, що повна ітерація циклу симуляції, що включає оновлення частинок, клітин сітки та двосторонній трансфер між ними, повинна виконуватися в реальному часі.

Водночас реалістичність поведінки рідини виступає також ключовим критерієм для запропонованого підходу. Оскільки кількісне вимірювання цього критерію є ускладненим, оцінювання проводиться здебільшого за візуальними характеристиками моделі. Основним аспектом, що використовуються для порівняння, можна виділити розподіл частинок. Спосіб не повинен створювати надмірного шуму або артефактів, що призводять до появи частинок у нехарактерних для них областях.

Таким чином, показником якості запропонованого підходу стане порівняння його поведінки з існуючими методами, зокрема таким як PIC або FLIP. Оцінювання здійснюється в однакових умовах, з використанням

розробленого програмного забезпечення, що більш об'єктивно оцінити швидкодію та якість роботи створеної симуляції. Приклад візуального порівняння представлено нижче.

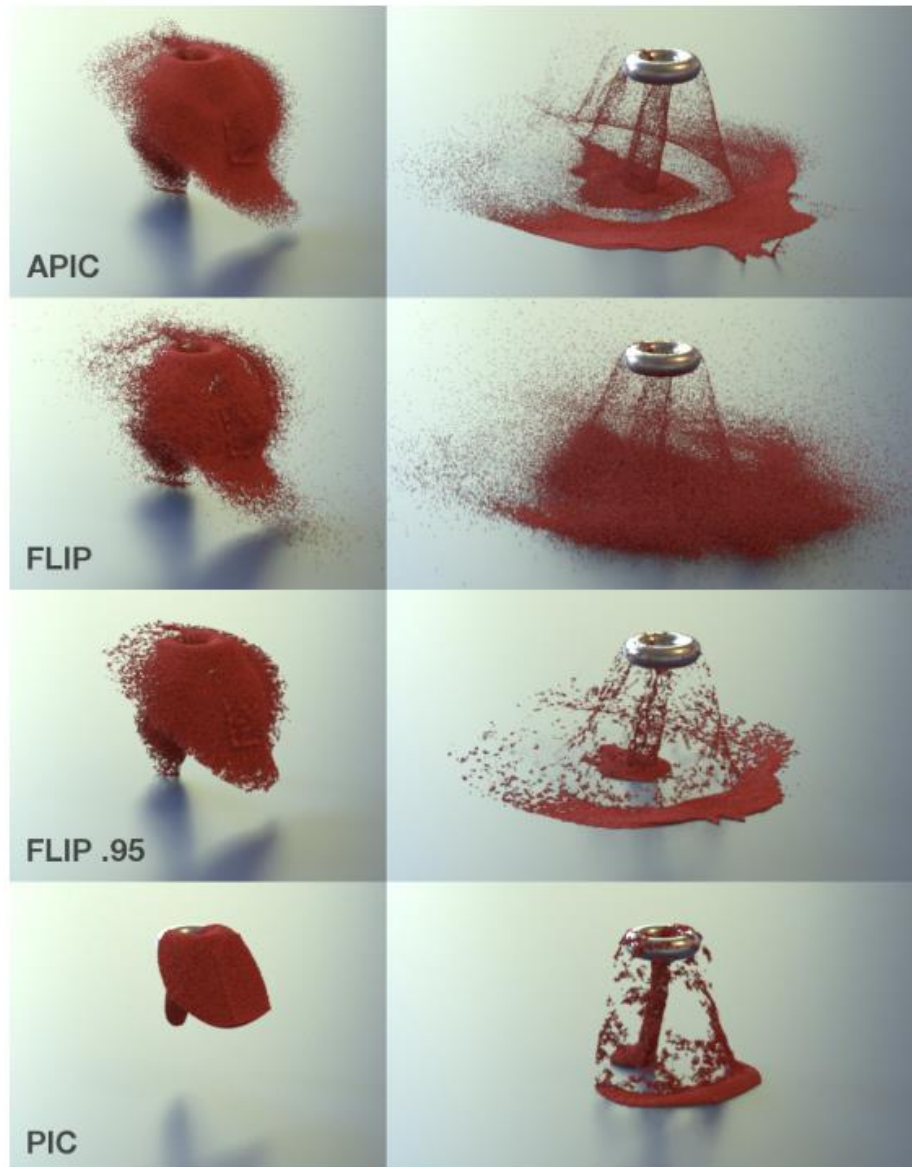


Рис. 8. Зображення порівняння в роботі присвяченій APIC

Даний рисунок взято з роботи в якій представили APIC, а цю ілюстрацію використали для того, щоб показати наскільки зменшився шум в порівнянні з FLIP, PIC та їх комбінацій з постійним коефіцієнтом вкладу FLIP 0.95.

2.2. Опис запропонованих модифікацій

Запропонований спосіб моделювання рідини належить до гібридних підходів і передбачає поєднання частинкової та сіткової репрезентації. Основною відмінністю від класичних методів FLIP та APIC є введення додаткового механізму корекції, що базується на врахуванні густини частинок у кожній клітині сітки. Такий підхід дозволяє підвищити стійкість симуляції та покращити збереження імпульсу без значного збільшення шуму [21].

Як було вказано раніше, ключовою проблемою методів сімейства PIC є компроміс між збереженням імпульсу та рівнем шуму. Метод IPIC став суттєвим кроком уперед, запропонувавши перенесення імпульсу частинок замість швидкості, що дозволило покращити збереження кутового моменту [18]. Однак навіть APIC є ресурсомістким, і його стабільна робота в режимі реального часу на обладнанні споживчого класу залишається складним завданням. Метод IPIC, який забезпечує ще кращі властивості перенесення імпульсу, практично неможливо застосувати в реальному часі через надмірну обчислювальну складність.

Щоб досягти подібних переваг без значного падіння продуктивності, в даній роботі пропонується врахування локальної концентрації частинок у клітині сітки. Ідея полягає в тому, що надлишкова кількість частинок у певному регіоні створюватиме ефект, подібний до тиску: клітині призначається вистискаюча сила, що спрямована на вирівнювання густини та запобігання локальній втраті імпульсу. Таким чином формується механізм, який природно витісняє рідину із перенасичених областей, забезпечуючи більш фізично коректну поведінку.

Подібний підхід був започаткований у науковій літературі близько 2019 року, проте його пряме застосування пов'язане зі значними витратами на додаткові обчислення, оскільки потребує розв'язання рівняння Пуассона

вдруге на кожному кроці симуляції. Для забезпечення роботи в реальному часі у даній роботі запропоновано наступне:

- адаптивна активація корекції густини. Цей етап активується лише при наявності високонавантажених регіонів з високою щільністю частинок, що дозволяє уникнути зайвих обчислень,
- використання внутрішніх структур та підходів системи Unreal Engine для інтеграції з рушієм.

2.3. Детальний опис запропонованого способу

Запропонований метод ґрунтується на загальній структурі гібридних підходів до симуляції рідин на основі частинок і сітки. Основна ідея полягає у поєднанні традиційної структури гібридних методів із додатковим етапом контролю густини, який активується лише при наявності областей із надмірною локальною густиною. Такий підхід дозволяє знизити рівень шуму та покращити збереження об'єму рідини, не втрачаючи обчислювальної ефективності, необхідної для роботи в реальному часі.

Основні етапи розроблюваного способу включають в себе:

1. Перенесення значень від частинок до сітки.
2. Розрахунок на сітці:
 - 1) адвекція,
 - 2) обчислення та, за необхідності, корекція густини,
 - 3) додання зовнішніх сил,
 - 4) розрахунок тиску та забезпечення виконання умови нестискаємості.
3. Перенесення швидкості від сітки до частинок.
4. Зміна положення частинок відповідно до нової швидкості.

Далі розглянуто кожен етап більш детально, описано подробиці його роботи. А також наведено формули та пояснено яким фізичним процесам та явищам вони відповідають.

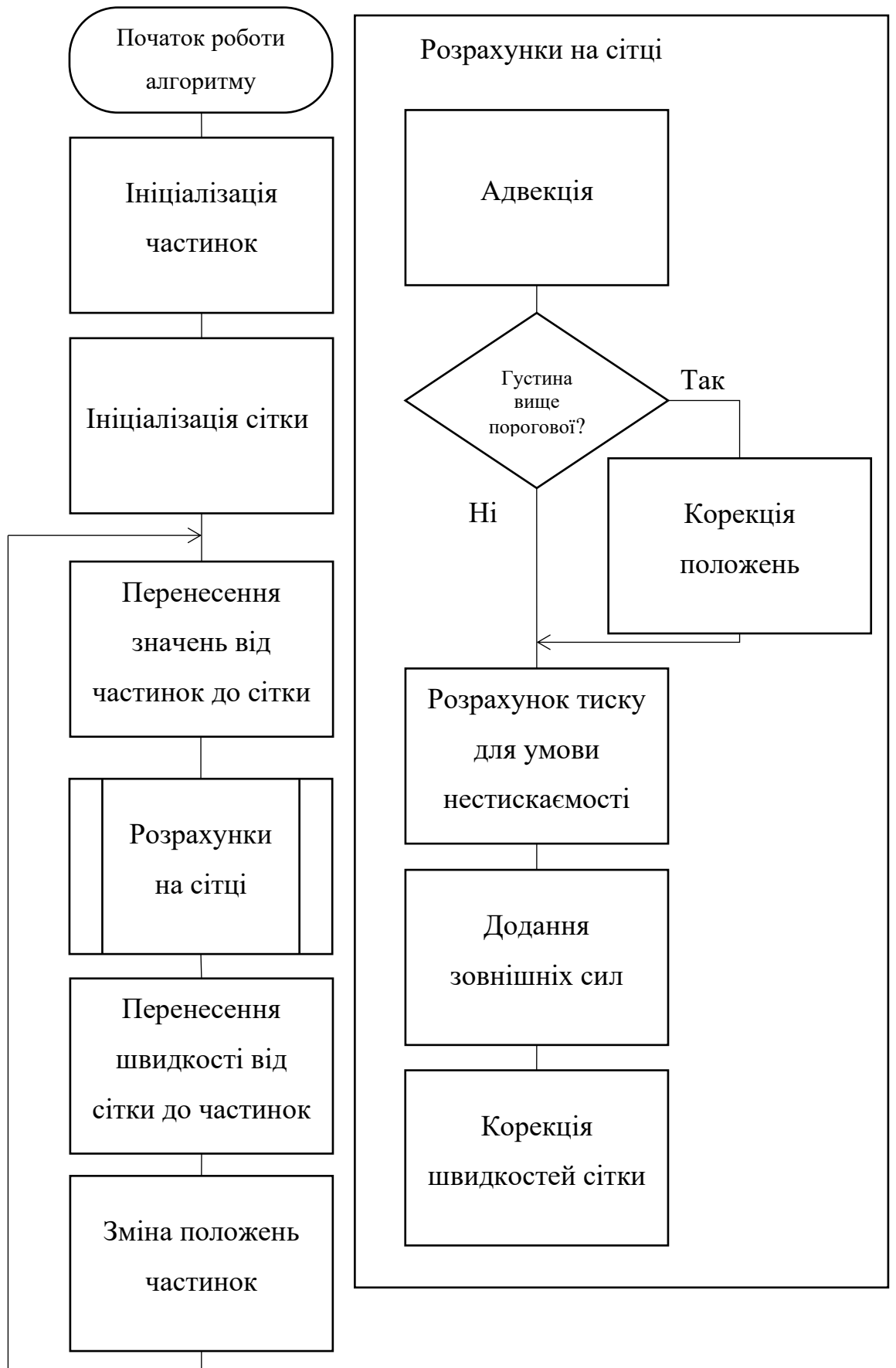


Рис. 9. Загальна блок-схема способу

2.3.1. Перенесення швидкості від частинок до сітки

Перший етап симуляції – це перенесення швидкостей частинок до сітки. а цьому етапі дані з лагранжевої репрезентації (частинок) трансформуються у ейлерову (сіткові комірки). Основною інформацією, що передається, є положення частинок та їх швидкість. Метою цього етапу є побудова поля швидкостей на сітці, яке надалі буде використано для розрахунків тиску, проекції швидкостей та інших кроків симуляції.

Для роботи в гібридних алгоритмах використовують багато принципів з методів Ейлера та Лагранжа. Один з них – це поетапна, або шахова сітка, а точніше її конкретний різновид – MAC сітка. Зображення такої сітки вже було представлено у роботі на рисунку 4 для опису методів розрахунку виключно на сітці. В такій сітці швидкість розбита на три компоненти (вертикальна швидкість, горизонтальна та направлена в глибину). При чому всі три компоненти зберігаються на напівсітці, яка зсунута відносно основної сітки на половину розміру клітини. Відносно основної сітки це виглядає так, що швидкості зберігаються на гранях [12].

Кожна частинка впливає на вісім оточуючих її швидкостей, які зберігаються в кутах напівсітки. Для визначення індексу цих швидкостей для початку визначається в якій клітині основної сітки знаходиться частинка для цього використовується досить проста формула:

$$\vec{i}_g = \text{Floor} \left(\frac{X_p - X_{g_0}}{\Delta x} \right). \quad (3)$$

В цій формулі \vec{i} – це вектор цілих чисел, що зберігає індекси клітини по кожній з осей. X_{g_0} – координати нижнього кута сітки, X_p – координати самої частинки, а Δx розмір клітини сітки. Таким чином знайдений індекс буде як індексом клітини на сітці клітин так і індексом нижнього лівого кута цієї клітини, таким що кожна з координат буде менша або рівна координат інших кутів, що дозволить далі простіше визначити до якої клітини напівсітки належить ця клітина [12].

Таким чином наступним кроком після знаходження індексів буде знайти відстань до знайденого. Тоді в залежності від того де саме в клітинці розташована частинка, ближче до минулої чи наступної клітини, знаходиться частинка обираються швидкості сітки на які вона впливає. Наочно це зображено на рис. 10, де показана сітка двомірних клітин, вертикальні швидкості на цій сітці та частинка, а також зображено на які саме швидкості вона впливає. При чому за основною координатою швидкості, в даному випадку Y , все досить просто, частинки вкладаються у швидкості Y координата яких рівна нижній або верхній грані клітини, тоді як для інших координат принцип ускладнюється, так як частинка може вкладатись як в минулу, так і в наступну клітину в залежності від того відстань для якої ближче [22, 23].

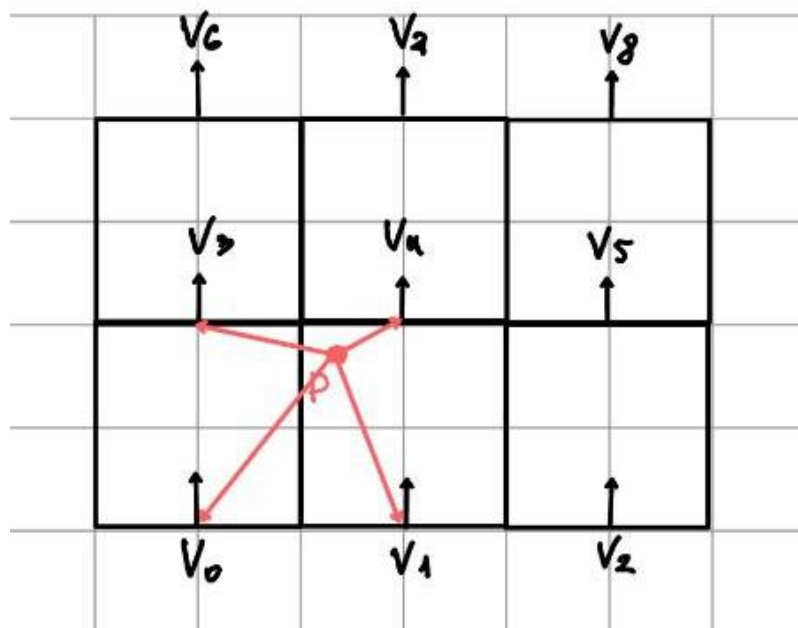


Рис. 10. Вибір на які вертикальні швидкості впливає частинка

Для знаходження до якої клітини ближче визначають дистанцію до лівого кута клітини основної сітки за формулою (4). В цій формулі дистанцію від положення частинки віднімають позицію нижнього кута, тобто кута з найменшими значеннями по всіх осях, визначеної клітини сітки

та ділять на розмір клітини, щоб нормалізувати та отримати відносну дистанцію замість абсолютної.

$$\vec{d} = \frac{\vec{x}_p - \vec{x}_g}{\Delta x} \quad (4)$$

Тоді знаходять індекс лівого кута напівсітки за кожною з координат по формулах:

$$i_{hg \cdot x} = \begin{cases} i_g \cdot x - 1, & d \cdot x < 0.5, \\ i_g \cdot x, & d \cdot x \geq 0.5, \end{cases} \quad (5)$$

$$i_{hg \cdot y} = \begin{cases} i_g \cdot y - 1, & d \cdot y < 0.5, \\ i_g \cdot y, & d \cdot y \geq 0.5, \end{cases} \quad (6)$$

$$i_{hg \cdot z} = \begin{cases} i_g \cdot z - 1, & d \cdot z < 0.5, \\ i_g \cdot z, & d \cdot z \geq 0.5. \end{cases} \quad (7)$$

Далі рахуються вагові коефіцієнти для сітки та напівсітки. Для сітки використовується формула (8). Для напівсітки аналогічно до індексу коефіцієнти залежать від розташування частинки, формули коефіцієнтів представлені у формулах (9), (10) та (11).

$$\vec{w}_g = 1 - \vec{d} \quad (8)$$

$$w_{hg \cdot x} = \begin{cases} 0.5 - d \cdot x, & d \cdot x < 0.5 \\ 1.5 - d \cdot x, & d \cdot x \geq 0.5 \end{cases} \quad (9)$$

$$w_{hg \cdot y} = \begin{cases} 0.5 - d \cdot y, & d \cdot y < 0.5 \\ 1.5 - d \cdot y, & d \cdot y \geq 0.5 \end{cases} \quad (10)$$

$$w_{hg \cdot z} = \begin{cases} 0.5 - d \cdot z, & d \cdot z < 0.5 \\ 1.5 - d \cdot z, & d \cdot z \geq 0.5 \end{cases} \quad (11)$$

Коефіцієнти називаються ваговими так як саме вони далі використовується для визначення вкладу частинки у швидкість на клітині сітки. Загальна формула для збору кожної з швидкостей [4] представлена нижче:

$$u_g = \sum u_p w_g \cdot x * w_{hg \cdot y} * w_{hg \cdot z}, \quad (12)$$

$$v_g = \sum v_p w_{hg \cdot x} * w_g \cdot y * w_{hg \cdot z}, \quad (13)$$

$$w_g = \sum w_p w_{hg \cdot x} * w_{hg \cdot y} * w_g \cdot z. \quad (14)$$

Основною задачею при зборі швидкостей з сітки є отримати швидкість в конкретній фіксованій точці, тому що просте додання всіх

зважених швидкостей не є достатнім через те що кількість частинок починає впливати на значення швидкості та спотворювати її, тому значення швидкості важливо нормалізувати. Для цього аналогічно до швидкостей акумулюють ваги які були використані, щоб їх зібрати, а далі нормалізують кожен зі швидкостей по формулі (15) та аналогічних для швидкостей направлених за іншими осями [4].

$$u = \frac{u}{w} \quad (15)$$

Також на сітці зберігаються значення тиску та густини. Обидва масиви значень по розмірності співпадають із сіткою, так як зазначені величини зберігаються в центрах клітин.

Значення тиску ініціалізується окремо, на кожному кроці всі значення цього масиву виставляються нульовими. І заповнюються лише на кроці корекції, коли потрібно задовільнити умову нестискаємості рідини. Тоді як густина збирається разом із швидкостями з сітки, за аналогічними формулами.

Всі обчислення для роботи напівсітки є досить громіздкими в порівнянні з використанням звичайної сітки, де кожна з величин зберігається в центрі. Але її використання надає можливість отримати значно кращі результати. Отже, причини для розбиття величин між сіткою та напівсіткою пов'язані зі способом підрахунку та корекції значення тиску. Зберігання значень на напівсітці надає перевагу при обчисленні тиску дозволяючи більш точно апроксимувати його дивергенцію.

Виходячи з усього написано було створено рис. 11 на якому зображено клітину сітки, яку використовує даний спосіб. Таким чином виходить, що на напівсітці залишаються тільки швидкості, а на самій сітці в середині клітини записують тиск та густину.

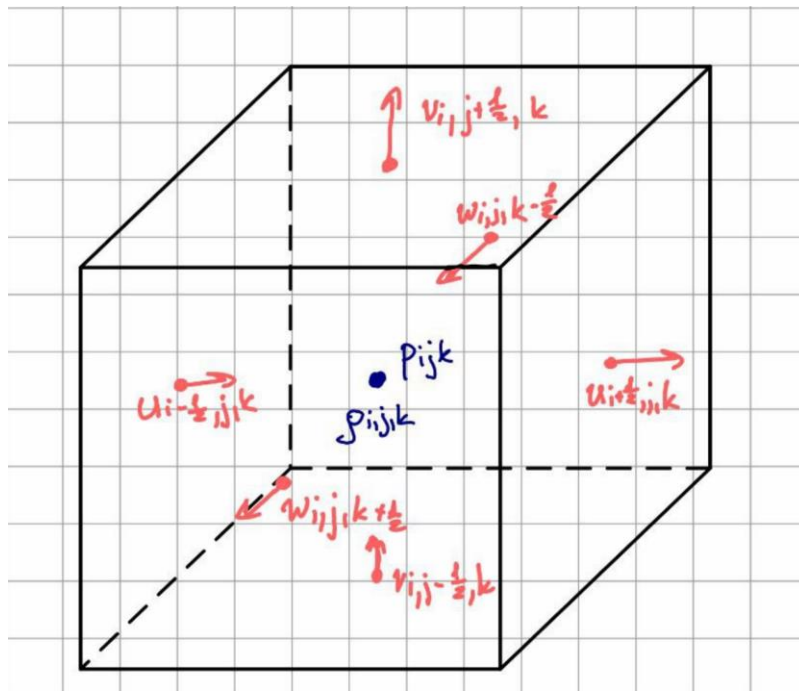


Рис. 11. Клітина сітки для розробленого способу

2.3.2. Розрахунок на сітці

Розрахунки на сітці складають центральний етап симуляції, оскільки саме на цьому кроці виконується дискретизоване наближення фізичних процесів руху рідини. Як було зазначено раніше, рух рідини описується рівняннями Нав'є-Стокса, які включають рівняння руху та умову нестискаємості. У контексті гібридних методів ці рівняння розв'язуються поетапно, із застосуванням чисельних підходів.

Розрахункова частина процесу на сітці поділяється на чотири основні етапи, що відповідають окремим фізичним процесам.

Адвекція

Адвекція є процесом перенесення фізичних величин у рідині під дією її власного руху, тобто загального потоку середовища. Вона описує вплив руху рідинного потоку на окремі частинки або поля швидкості, забезпечуючи передачу імпульсу навіть тим елементам, які не взаємодіють безпосередньо з іншими частинками або твердими перешкодами. Таким чином, адвекція відтворює природний перенос маси та швидкості всередині

середовища, що є фундаментальним для підтримки цілісності та неперервності потоку.

Частина рівняння Нав'є-Стокса, що відповідає цьому явищу представлена у формулі (16). Ця частина формалізує перенесення швидкості та імпульсу рідини під дією її власного руху, забезпечуючи фізично коректне відтворення поведінки потоку.

$$-(\vec{u} \cdot \nabla)\vec{u} \quad (16)$$

У процесі симуляції рідин гібридними методами адвекція реалізується через поле швидкостей, сформоване шляхом перенесення даних з частинок до комірок сітки [24]. Саме швидкість загального потоку в регіоні, що інтерполюється зворотно до частинок є адвекцією.

Обчислення та корекція густини

Цей крок починається одразу після перенесення швидкості з частинок до сітки. Згідно до висунутих покращень, також використовується поріг (17), за яким вирішується наявність потреби в обчисленні всього цього кроку в цілому. Так як він додає значне навантаження на систему і в основному застосовується в даному способі для вирішення ситуацій з високою густиною рідини.

Розроблений поріг одночасно враховує особливості представлення рідини на сітці та частинках. А його застосування дозволяє зекономити ресурси шляхом проведення обрахунків тільки в разі наявності відповідної потреби.

$$\rho_{threshold} = \alpha\rho_0 + \beta \frac{N_i}{N_{max}} \quad (17)$$

Для безпосередньо корекції позицій частинок використовується модифіковане рівняння Пуассона (18), описане в методі Implicit Density Projection, яке вирішується тим ж способом, що й рівняння Пуассона представлене на кроці корекції тиску. За допомогою такого рівняння знаходиться тиск, що задовільняє умови. Після чого тиск підставляється у формулу корекції положень частинок (19) [21].

$$\frac{\Delta t}{\rho_0} \nabla^2 p_2 = \frac{1}{\Delta t} \left(1 - \frac{\rho(t)}{\rho_0} \right) \quad (18)$$

$$\delta x = \delta u \Delta t = - \frac{\Delta t^2}{\rho_0} \nabla p_2 \quad (19)$$

Зовнішні сили

Формально до цього розділу належать усі сили зовнішнього середовища, що діють на рідину, але насправді для реалістичної симуляції достатньо врахувати гравітацію. Таким чином врахування зовнішніх сил проходить шляхом віднімання від вертикальних швидкостей сталої прискорення вільного падіння [4].

Корекція тиску

Основна ідея цього етапу – задовільнити умову нестискаємості, тобто запобігти втрати об'єму рідини. В рівняннях Нав'є-Стокса це відповідає другому рівнянню (2), що вказує на те, що загальна дивергенція повинна залишатись нульовою [23].

Цей етап є одним з найскладніших в симуляції рідин гібридними методами. Він включає в себе декілька підетапів в наступному порядку:

- 1) Розрахунок дивергенції потоків рідини в кожній клітині сітки.
- 2) Розрахунок тиску, такого, що задовільнить умову нестискаємості
- 3) Корекція швидкостей відповідно до знайденого тиску

Отже, робота над етапом починається з розрахунку дивергенції. Загальна формула дивергенції має наступний вигляд:

$$\nabla \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}. \quad (20)$$

Але для роботи з сіткою її апроксимують за формулою центральних кінцевих різниць (21). Крім того, важливо зазначити, що дивергенція апроксимується лише для клітин сітки, що містять рідину.

$$\nabla \vec{u} \approx \frac{u_{i+\frac{1}{2}} - u_{i-\frac{1}{2}}}{\Delta x} + \frac{v_{i+\frac{1}{2}} - v_{i-\frac{1}{2}}}{\Delta x} + \frac{w_{i+\frac{1}{2}} - w_{i-\frac{1}{2}}}{\Delta x} \quad (21)$$

Після розрахунку дивергенції починаються розрахунки безпосередньо тиску. Початковий тиск береться за нульовий, а залишки за дивергенцію взяту зі знаком мінус. А далі використовується метод спряженого градієнта. На початку роботи методу розраховують поріг зупинки. Цей поріг встановлюється по сумі квадратів залишків. Після чого метод спряженого градієнта працює до досягнення значень менше порогових.

Корекція швидкостей є заключним етапом корекції швидкостей поля згідно до тиску. Швидкості коректують за наступними формулами (22) – (24). Після чого переходять до наступного етапу симуляції. Перенесення отриманих швидкостей до частинок [23].

$$u_{i+\frac{1}{2},j,k}^{n+1} = u_{i+\frac{1}{2},j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} \quad (22)$$

$$v_{i,j+\frac{1}{2},k}^{n+1} = v_{i,j+\frac{1}{2},k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x} \quad (23)$$

$$w_{i,j,k+\frac{1}{2}}^{n+1} = w_{i,j,k+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x} \quad (24)$$

2.3.3. Перенесення швидкостей до частинок

Ця частина роботи є зворотною до описаної в пункті 2.3.1 та використовує дуже багато з того, що вже було зроблено при перенесенні швидкостей від частинок до сітки. Головним з повторно використаного є знаходження індексів клітин сітки та напівсітки в яких знаходиться частинка. Алгоритм повністю повторюється.

Але при зворотному перенесенні процес перенесення швидкості ускладнюється, так як для кращого результату на цьому етапі часто змішують підходи FLIP та чистого PIC. Це дозволяє отримати переваги обох методів не отримуючи затухання класичного методу та хаотичності FLIP [25]. Цей процес проходить за формулою:

$$v_p^{n+1} = \alpha v_p^{n+1,FLIP} + (1 - \alpha) v_p^{n+1,PIC}. \quad (25)$$

При чому частинка збирає кожен з оточуючих її швидкостей окремо. Для PIC швидкості це проходить за формулою (26). Тобто кожна з компонент швидкості сумується окремо використовуючи дані з сусідніх клітин.

$$v_p^{n+1,PIC} = (\sum u_g w; \sum v_g w; \sum w_g w) \quad (26)$$

FLIP в свою чергу також збирає всі компоненти окремо, але на відміну від PIC, в цьому підході рахують не нову абсолютну швидкість, а зміну швидкості за час, що пройшов за ітерацію.

$$v_p^{n+1,FLIP} = v_p^n + \Delta v_g \quad (27)$$

$$\Delta v_g = (\sum (u_g^{n+1} - u_g^n) w; \sum (u_g^{n+1} - u_g^n) w; \sum (u_g^{n+1} - u_g^n) w) \quad (28)$$

Тоді при з'єднанні підходів отримується наступна формула збору швидкостей з напівсітки:

$$v_p^{n+1} = \alpha(v_p^n - v_g^n) + v_g^{n+1}. \quad (29)$$

2.3.4. Зміна положення частинок

Основною дією на цьому етапі є застосування знайдених на минулих етапах значень. Це найпростіший з усіх етапів, основна його задача – зсунути частинки відповідно до швидкості та часу, що пройшов. Для цього використовується формула [16]:

$$x_p = x_p + \Delta t * v_p. \quad (30)$$

2.4. Висновки до розділу

В розділі було висунуто вимоги до розроблюваного способу симуляції рідин. На основі вимог виділено групу методів до якої належить спосіб, а також обґрунтовано цей вибір. Було детально описано запропоновані модифікації для розробленого способу та виділено різницю з іншими методами. Крім того, було описано переваги створеного способу над іншими методами. Після детального опису створено загальну блок-схему по якій працює алгоритм. Наведено детальний опис кожного з кроків

розробленого способу, показано яким фізичним явищам відповідає кожен з етапів.

3. ПРОЄКТУВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СИМУЛЯЦІЇ ПОВЕДІНКИ РІДИН

Проєктування є важливим етапом при розробці програмного забезпечення, особливо коли існують чіткі вимоги до його швидкодії, а на одному кроці повинні виконуватись, ще декілька десятків інших процесів. Тому на початку процесу проєктування важливо висунути вимоги на основі яких можливо буди оцінити та обрати найбільш відповідний для задачі стек.

При виборі інструментів реалізації важливо враховувати можливі компроміси між швидкодією, простотою написання та читабельністю коду. Неправильна оцінка ситуації на цьому етапі здатна вплинути на якість кінцевої симуляції та можливо призвести до потреби створювати проєкт з самого початку. Тому варіанти вибору потрібно ретельно зважити та аргументувати вибір.

3.1. Вимоги до програмної реалізації

Розроблювана система передусім орієнтована на використання з іншими програмними компонентами. Через це вже на етапі проєктування важливо забезпечити можливість простої інтеграції із зовнішніми системами. Це виражається у встановленні гнучкості важливим пріоритетом при проєктуванні програмного забезпечення.

Гнучкість в даному випадку виражається в декількох аспектах. Насамперед модульність, правильне розбиття дозволить відокремити фізичні та графічні процеси один від одного. А також дозволить гнучко керувати обчислювальними ресурсами, наприклад, відокремлення деяких процесів дозволить позбутися вузьких місць, таких як дорогий трансфер інформації між CPU та GPU, його можливо перенести в окремий потік, який може виконуватись паралельно. Крім того, для простоти інтеграції важливо

не ускладнювати структуру вихідних даних і дотримуватись мінімальної кількості залежностей.

Другим пріоритетом є швидкодія, як вже було зазначено важливим аспектом розробки є можливість роботи в реальному часі, по-перше через вимоги самої прогами, по-друге через те що замала частота оновлень може призводити до візуальних артефактів зв'язаних з збільшенням значення змінної Δt , або затримок, коли користувач буде чітко бачити як рідина застигає між ітераціями. Така вимога вимагає планування можливих оптимізацій вже на початкових етапах розробки. Можливі оптимізації включають спробу передбачити, які операції можуть виконуватись паралельно, які дані можливо передавати між CPU та GPU завчасно, а також вибір найбільш оптимальних структур даних.

Конкретні рішення, щодо даних та можливих оптимізацій було прийнято після визначення інструментів реалізації, коли було виділено, які технології будуть використовуватись та які переваги та недоліки мають саме вони.

3.2. Інструменти реалізації програмного забезпечення (Стек)

Як було вказано вище, вибір інструментів реалізації повинен бути зроблений на перших етапах проєктування, знаючи конкретні інструменти передбачення можливих вузьких місць та областей з потенціалом до оптимізації є простішим. Крім того, вибір є радше очевидним. Більшість програм, що використовують гібридні методи симуляції рідин написані на C, C++ або Python, що використовує C під капотом [26, 27].

В основному такий вибір обумовлений специфікою мов C та C++. Обидві мови є низькорівневими та надають розробнику більше контролю над пам'яттю та потоками, що в свою чергу надає можливість створювати високопродуктивне програмне забезпечення. Хоча, працюючи з цими мовами важливо пам'ятати про те що широкий вибір можливостей також

означає велику кількість вразливостей. Це мови які надають можливість робити критичні помилки повністю перекладаючи відповідальність за це на программіста [28].

Особливістю вибору інструментів візуалізації є вимоги до неї. Важливим аспектом є можливість програмного забезпечення інтегруватись зі сторонньою системою. В якості такої системи було обрано Unreal Engine. Вибір впав на Unreal Engine тому що цей рушій є широко використовуваним у багатьох індустріях і закладена інтеграція з ним надасть можливість отримати велику аудиторію при майбутній розробці стартапу. Крім того, в рушій закладено багато структур та систем, що значно спростять реалізацію програмного забезпечення. Наприклад, і рушію вже існує система візуалізації частинок, яку потрібно лише налаштувати. Використання такої системи також надає механізм передачі змінних між об'єктами симуляції та візуальним представленням.

3.3. Паралельні обчислення

Паралельні обчислення вже давно є стандартом будь-якої індустрії, що потребує високошвидкісних розрахунків. Індустрія симуляції рідин не стала виключенням з цього правила. Більш того, існує досить багато робіт присвячених різним підходам до паралелізації роботи алгоритмів симуляції рідини [29, 30]. На кожному кроці більшість операцій повторюються для кожної частинки або клітини сітки. Задача програмістів написати програму так, щоб дозволити їй ефективно використовувати існуючі ресурси для виконання заданих операцій. А також запобігти великій кількості проблем та ризиків пов'язаних з роботою різних ділянок коду над одним й тими самими даними.

Так як було обрано використовувати Unreal Engine, то також було використано вбудовані інструменти для паралелізму, зокрема такі інструкції як `ParallelFor`, що надають можливості для паралелізму даних,

тобто виконанню одних й тих самих операцій на різних даних. Такий підхід дозволить значно підвищити швидкодію програми.

3.4. Архітектура розроблюваної системи

Головним принципом проектування розроблюваної системи є забезпечення високої гнучкості та модульності. Так як саме гнучкість стане великою перевагою для майбутньої роботи з програмним забезпеченням, а модульність дозволить тестувати, покращувати та розроблювати кожен з модулів не спираючись на інші, що надасть більшу кількість свободи та можливостей, таких як додання нових модулів у майбутньому для розробників. Тому користуючись цими принципами було обрано створити програмне забезпечення у вигляді плагіну до відомого рушія Unreal Engine. Це дозволяє інтегрувати систему в різні проекти на базі рушія без значних змін у вихідному коді або структурі проекту. Відповідно, система реалізована у вигляді окремого плагіну, що спрощує підключення до будь-якого проекту, незалежно від його призначення чи масштабу. Такий підхід також забезпечує можливість оновлення або заміни компонентів без порушення роботи інших систем, що є ключовим аспектом сучасної архітектури програмного забезпечення.

Плагін містить в собі декілька основних компонентів: допоміжні функції симуляції рідини, зв'язуючи класи, що забезпечують інтеграцію з рушієм та використовують підготовлені функції для симуляції рідин, систему візуалізації з використанням вбудованого плагіну Niagara, інтеграція з системою, що дозволяє редагувати змінні симуляції та працювати з нею без знання коду – Blueprints. Завдяки такому поділу на окремі компоненти реалізується чітке розділення відповідальності, що спрощує модифікацію, тестування та масштабування системи. Окремі модулі можна розвивати незалежно один від одного, що підвищує гнучкість плагіну та його здатність до адаптації під нові завдання або інтеграції з

іншими системами рушія. Нижче розглянуто кожен з компонентів детальніше.

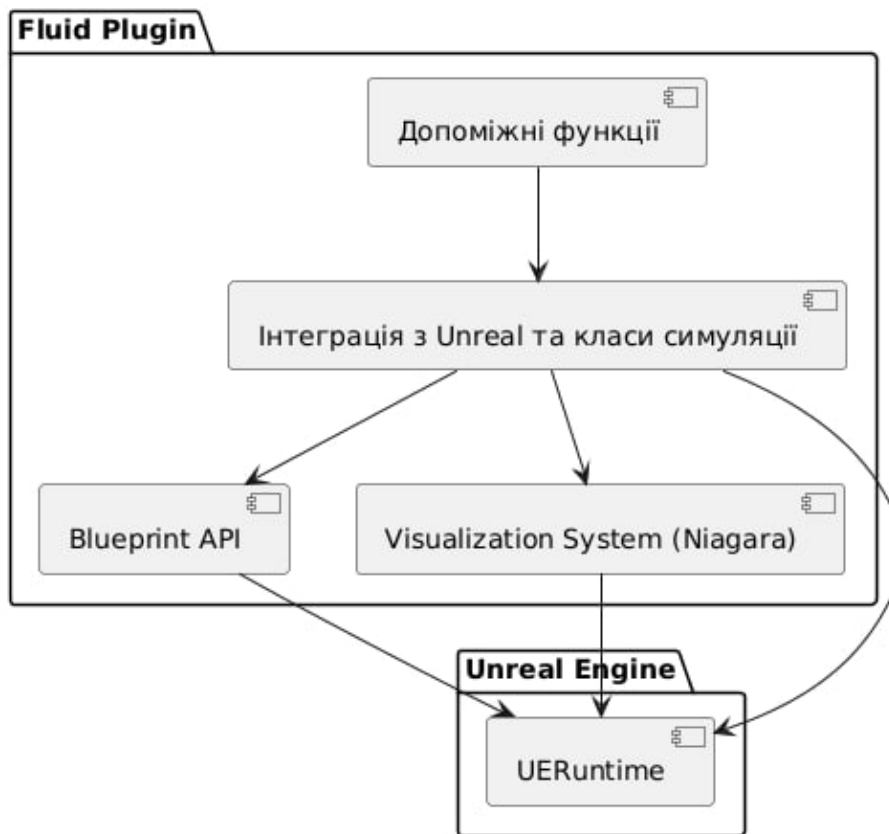


Рис. 12. Архітектура розробленого програмного забезпечення

3.4.1. Допоміжні функції

Окремо винесені функції, призначені для використання безпосередньо в класах симуляції рідини. Ці функції на повну використовують можливості C++ оптимізуючи виконання.

Прикладами допоміжних функцій є функції конвертації між пласкими та тривимірними індексами, або обрахунок кількості сусідів клітини. В цілому це функції які часто використовують та існує можливість їх винести за межі основних класів.

Серед важливого функціоналу мови який вони використовують є ручне завдання `force inline` функцій та використання вказівників. `Force inline`

дозволяє встроювати невеликі, але дуже важливі функції до коду під час компіляції. Такий функціонал є дуже важливим так як може підвищити швидкість в рази [31], особливо на таких функціях як конвертація між плоскими та тривимірними координатами, які визиваються тисячі раз на цикл. Що ж до вказівників, то це не менш важливий механізм унікальний для мов сімейства c. Так як саме можливість передавати та редагувати значення по вказівникам робить ці мови настільки швидкими. Це працює через те що операції запису та зчитування є одними з найбільш дорогих простих операцій, а при стандартній передачі значення до функції та запису її результату до змінної мало того що додатково зчитати початкові параметри та переписати значення, що повернулось, а ще й динамічно виділити пам'ять під додаткові змінні. Механізм вказівників же значно спрощує цей процес дозволяючи уникнути надлишкових обчислень.

3.4.2. Класи симуляції та інтеграції з Unreal Engine

Unreal Engine має свою внутрішню ієрархію класів, дотримання якої надає можливість співпрацювати з рушієм ефективно. Так, наприклад, для створення об'єкту, що може бути розміщений на сцені потрібно реалізувати наслідування від класу Actor так як він має необхідний набір полів та функцій для взаємодією з класом "level" – сценою та іншими об'єктами розміщеними у просторі [32].

Крім того Unreal використовує свою систему макросів, за рахунок якої працює велика система різноманітних механізмів рушія в тому числі рефлексія. Тому, класи що тісно працюють з рушієм і повинні відображатись в редакторі повинні бути правильно відмічені в кодї. Так кожен клас повинен використовувати макро UCLASS та його опцію Blueprintable, щоб працювати з системою Blueprints, що дозволяє не технічним спеціалістам працювати з нащадками класів створених в кодї за допомогою візуального інтерфейсу. Ця система та принцип роботи з нею більш детально описаний в пункті 3.4.4.

Серед інших макросів обов'язковими є GENERATED_BODY, UFUNCTION, UPROPERTY. Перший, генерує додатковий код. А два інших забезпечують систему рефлексії.



Рис. 13. Налаштування змінних у системі Blueprint

Також, Unreal додає нові структури та типи даних працювати з якими вигідно з точки зору оптимізації робочого процесу. Наприклад, стандартні програми на с++ часто використовують вбудовані бібліотеки для реалізації такого типу даних, як динамічні масиви. Але при використанні Unreal Engine кращим варіантом буде використовувати TArray – вбудоване рішення від Unreal Engine. Воно є не менш оптимізованим ніж стандартний масив, але при цьому гарантує злагоджену роботу з будь-якими частинами рушія. Так, наприклад, можливо буде передати позиції частинок до системи візуалізації використовуючи всього декілька рядків коду для налаштування. Крім того, самі позиції будуть мати формат вектора – структури, що включає в себе три числа з плаваючою точкою, підтримує більшість векторних операцій. Та оптимізована під стандартні випадки використання таких структур [32].

3.4.3. Візуалізація

Візуалізацію симуляції забезпечує система для візуалізації частинок Niagara. Ця система збудована таким чином, що надає можливість ефективно передавати ряд параметрів до візуальної системи. В список таких параметрів входять вектори, числа, текстові змінні [33].

Крім того, важливим аспектом Niagara є гнучкість створюваної системи. Використовуючи цей інструмент досить легко налаштувати пайплайн так, що налаштування зовнішнього вигляду частинок та візуалізації в цілому повністю незалежні від технічних аспектів, які передаються на вхід. Крім того, простота додання шейдерів та зміни моделей частинок надає широкі можливості для налаштування під конкретні проєкти.

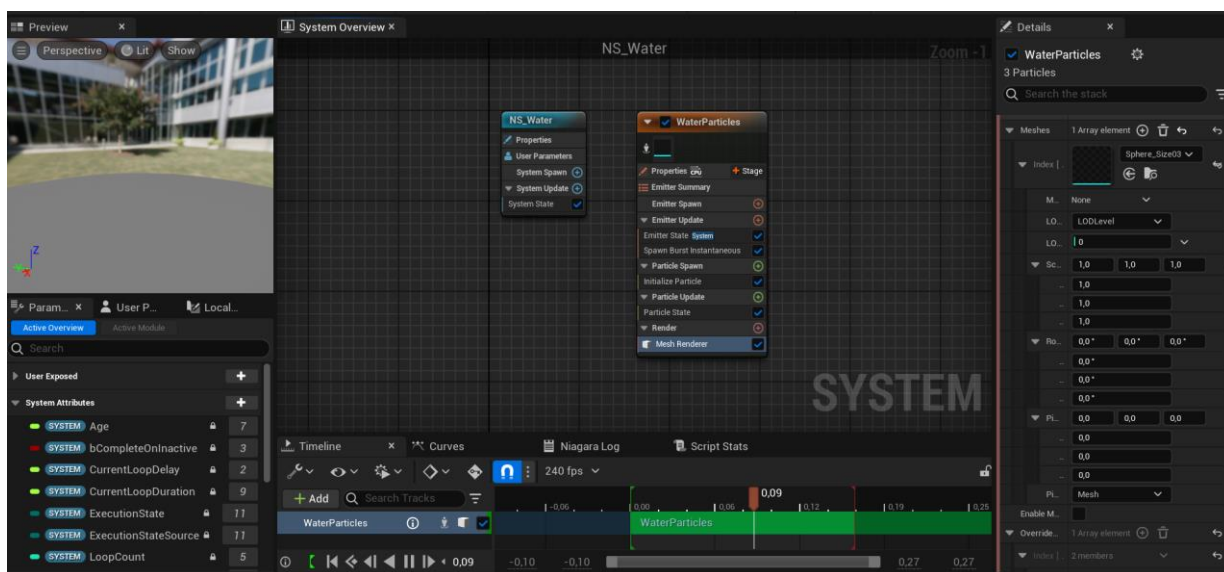


Рис. 14. Приклад інтерфейсу Niagara

3.4.4. Blueprints

Unreal Engine має особливу систему, що дає велику кількість можливостей для співпраці між різними командами розробки – Blueprints. Ця система дозволяє створювати більш візуальні та доступні нащадки класів, що в свою чергу надає можливість дизайнерам, аніматорам та іншим технічним та нетехнічним спеціалістам співпрацювати ефективно [34]. Крім того, такі системи надають можливість більш гнучкого та простого налаштування, так наприклад можливо змінити будь-який з параметрів не втручаючись в код та не перезавантажуючи систему. І такі налаштування доступні для кожного об'єкту симуляції на сцені окремо. Тому створення

blueprint симуляції було важливим кроком для покращення простоти роботи з плагіном.

Також, важливо зазначити, що система має можливість візуального скриптингу, що розширює можливості людей працюючих з рушієм. Візуальний скриптинг часто використовується для швидких прототипів програмістами, але також має значення для менш інженерно налаштованих спеціалістів. Вона надає їм можливість самостійно імплементувати або прототипувати більш просту частину логіки. Така незалежність допоможе пришвидшити процес створення продукту і його тестування, а також стане великим плюсом при виборі інструменту роботи.

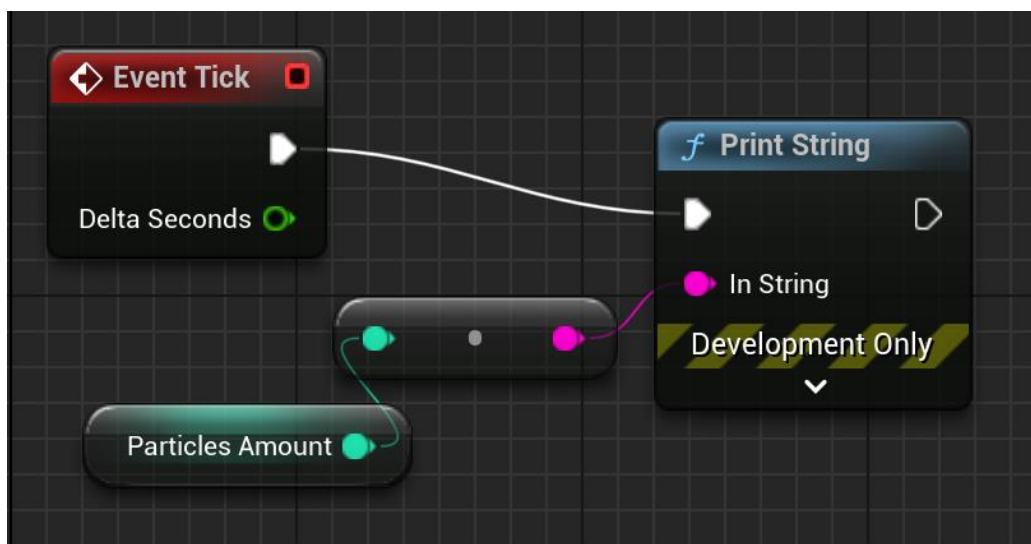


Рис. 15. Візуальна система скриптингу Blueprint

3.5. Висновки до розділу

В розділі було висунуто чіткі вимоги до розроблюваного програмного забезпечення. На основі висунутих вимог було порівняно можливі інструменти для реалізації, описано запропоновані можливості. Після чого було також описано обрані інструменти реалізації та обґрунтовано чому було обрано саме ці.

Крім того, було створено архітектуру програмного забезпечення. Так саме програмне забезпечення було створено у вигляді плагіну, який працює з рушієм Unreal Engine.

Кожен з інструментів реалізації та архітектурних модулів було детально описано. Зазначено можливі оптимізації та потенційні проблеми. Описано загальну взаємодію між компонентами та внутрішній принцип роботи кожного з компонентів окремо. А також показано інтерфейс роботи з модулями з якими взаємодіє користувач.

4. АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ РОЗРОБЛЕНОГО СПОСОБУ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Основною метою розробки способу було створення покращення до існуючого методу FLIP. Такого, що він зберігатиме свою швидкодію, але при цьому зможе вирішувати ситуації з рідиною високої густини залишаючи або приводячи загальний об'єм рідини в нормі. Отже тестування полягатиме в двох етапах. Перший – верифікація працездатності, показати, що програма дійсно може привести великий об'єм в норму. Створити візуальний приклад. Другий етап – проведення тестів швидкодії програми.

4.1. Верифікація роботи програмного забезпечення

Первинне тестування та верифікація працездатності включає в себе запуск програми з порогом та перевірка на задовільнення умови швидкодії та візуальна верифікація, що програма здатна привести густину до норми. Для цього було обрано ситуацію, коли частинки сконцентровані в одному місці. Це вид ситуацій, з якими не може впоратись FLIP, оскільки потребують додання шару корекції густини. Параметри симуляції наведені в таблиці.

Таблиця 3

Параметри симуляції

Параметр	Значення
Кількість частинок	120000
Вага частинки	0.07
Розмір клітини сітки	25×25×25

Вигляд симуляції на початку роботи та через деякий час після корекції густини зображено на рисунках 15 та 16.

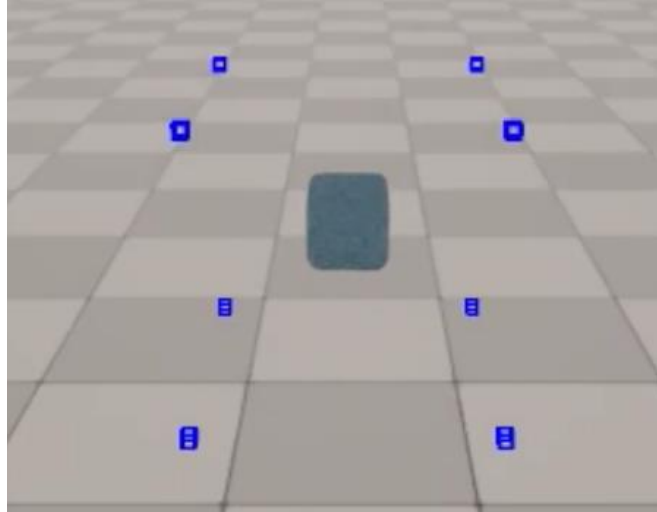


Рис. 16. Надмірна концентрація рідини на початку роботи програми

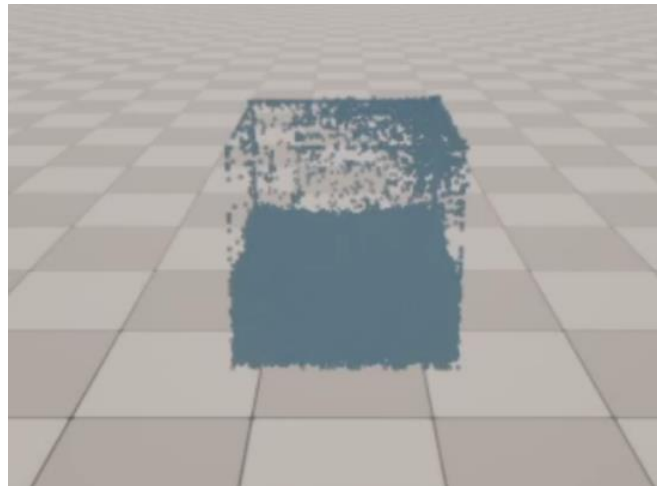


Рис. 17. Результат роботи програми

Видно, що з часом, частинки поступово вирівнюються. Що свідчить про те що розроблене програмне забезпечення з використанням розробленого способу здатне видати результат, який би не зміг видати стандартний FLIP алгоритм.

4.2. Порівняння роботи за різних порогів

За результатами дослідження можливо сказати, що розроблений спосіб здатен вирішувати ситуації з рідиною високої густини. Але існує

також питання швидкодії, наскільки програма розроблений спосіб покращує швидкодію програми в порівнянні з вже існуючим, що не має порогу.

Такі тести було проведено та занесено результати до таблиці.

Таблиця 4

Швидкість роботи програми

	Поріг відключено	Адаптивний поріг
Середній час видачі кадру	0.0451	0.0357

Отримані значення свідчать, що введення порогу призвело до пришвидшення симуляції на 20.84%, що є важливим часом для такого типу симуляцій.

4.3. Висновки до розділу

Висока швидкість роботи: експериментальні результати продемонстрували ефективність додання адаптивного критерію для підключення шару корекції густини до класичних методів симуляції рідин у реальному часі, що дозволило успішно розв'язувати ситуації, з якими традиційні підходи не справляються та підвищити ефективність методу на 20.84%.

5. ПОБУДОВА БІЗНЕС МОДЕЛІ

В цьому розділі розроблено та описано бізнес модель стартап проєкту основним продуктом якого є розроблений плагін. Було виділено зацікавлені та проаналізовано їх. Описано унікальну ціннісну пропозицію. Пораховано потенційні доходи й витрати. На основі цих даних було створено канву бізнес моделі.

В рамках роботи цей розділ є завершальним. Його наповнення зв'язує наукову роботу і бізнес, що є дуже важливим для роботи, так як саме поєднання між цими сферами надає найбільш прогрес. Перенесення нових технологій у бізнес дозволяє виводити їх на ринок та давати змогу користуватись ними простим користувачам, що в свою чергу приносить фінанси на майбутні дослідження та розвиток сфери.

5.1. Зацікавлені сторони

Головна ідея аналізу зацікавлених сторін – це виявлення тих чий інтереси та потреби потрібно враховувати в першу чергу, тобто найбільш зацікавлених. Для цього спочатку оцінюється вплив та зацікавленість кожного. Логічно, що розробник має максимальний вплив та зацікавленість у проєкті. Далі все трішки складніше, високий вплив матиме основна аудиторія, тобто працівники ігрової та кіноіндустрії. При чому кіноіндустрія буде менш зацікавленою так як навіть враховуючи, що у кіно вже звикли використовувати технології реального часу це все ще не буде для них першим пріоритетом. Архітектори ж, хоча й зацікавлені у використанні подібних симуляцій в основному використовують їх для презентацій своїх проєктів і для них таке рішення часто буде надлишковим. Власники рушія є досить впливовими, але не сильно зацікавленими, саме вони диктують правила по яким працює основна платформа розповсюдження плагінів і сам рушій. Але хоча їм і важливо постійне

розширення рушія, один плагін це капля у морі. Науковці мають високий вплив так як саме їх дослідження в першу стають фундаментом нової системи, але вони не є кінцевими клієнтами. Зацікавлені сторони та їх сфери також перелічено у табл. 5.

Таблиця 5

Зацікавлені сторони

Зацікавлена сторона	Сфера зацікавленості	Очікування від проєкту
Розробник	Розробник в першу чергу зацікавлений в успіху свого проєкту. А в даному випадку від цього також залежить його диплом.	Успішний гнучкий проєкт спроможний на ринкову конкуренцію за рахунок використання нових методів та підходу до розробки.
Програмісти ігор	Симуляція рідини в іграх це складний процес і часто доцільним рішенням є вибір готового плагіну замість написання нового. Крім того, часто з точки зору фінансів (коштів на покупку плагіну та коштів в людино-годинах на свою розробку) виграє покупка готового плагіну.	В першу чергу очікується можливість роботи в реальному часі та гарна оптимізація. А також можливість налаштування виключно потрібних функцій.
Інді розробники ігор	Інді розробка є специфічною, тому що одна людина часто бере на себе одразу декілька ролей у проєкті. За такого навантаження пророблення кожної частини проєкту вручну є неможливим.	Стабільно працюючий, недорогий плагін який легко вбудувати у свій проєкт.

3D художники, художники по оточенню	Працюють як у індустрії ігор, так і в індустрії кіно. Для них основною задачею є створення візуально привабливої картини на екрані. Використання плагіну надасть їм можливість швидко створити будь-які водні поверхні та їх взаємодію, що в деяких проєктах зекономить сотні годин роботи.	Плагін з детальним налаштуванням візуальних ефектів для фінального рендеру.
Дизайнер рівнів	Займається створенням рівнів в іграх, слідкує за роботою механік на рівні та їх збалансованістю. Плагін дозволить швидко прототипувати рівні з водою сильно економлячи час та гроші.	Робота з коробки, легкість додавання у проєкт.
Архітектори	Часто використовують Unreal Engine та близький до нього продукт Twinmotion для візуалізацій своїх проєктів. При візуалізації їм важливо показати свій проєкт у середовищі.	Робота з коробки, без потреби у будь-яких додаткових діях.
Власники рушія Unreal Engine (Компанія Epic Games)	Компанії вигідно коли їх рушій розвивається. Чим більше плагінів та контенту створеного конкретно під їх рушій, тим більше користувачів в них буде взагалі. І тим більше вони зароблять на цьому грошей.	Робочій плагін, що буде додатковою перевагою для вибору їх рушія.

Науковці у сфері симуляції рідин	Застосування та покращення методів вигідно для розвитку науки. Крім того вдалі покращення можуть принести більшу розповсюдженість тим чи іншим методам.	Імплементация та вдосконалення методів симуляції рідин
----------------------------------	---	--

Згідно таблиці найбільший вплив мають люди, що тим чи іншим чином приймають участь у розробці ігор. Тому найбільш правильним рішенням буде представляти плагін на ігрових конференціях та gamedev спільнотах. Крім того ефективною стратегією буде комунікація з розробниками наприклад на дискорд серверах по розробці ігор, де можливо зібрати інформацію чого найбільше очікують спеціалісти у сфері і можливо взяти ідею для того як вигідно виділитись серед конкурентів.

Також важливими фігурами є власники компанії Epic Games та науковці у сфері симуляції рідин. Вони не є кінцевими клієнтами, але як від них залежить робота плагіну. Так і вони залежать від неї. У випадку компанії постійний розвиток їх рушія у вигляді плагінів приваблює більшу аудиторію, що є прямим джерелом фінансів для компанії. У випадку вчених, то практичне використання та покращення методів є сенсом їх роботи у першу чергу. Винайдення технології без практичного застосування немає сенсу, а отже дослідження також не матимуть фінансування. Звісно є важливим задовольняти основні вимоги цих зацікавлених сторін

5.2. Унікальна ціннісна пропозиція рішення та конкурентні переваги

Оскільки унікальна ціннісна пропозиція залежить не тільки від властивостей продукту, а й від споживачів, то в першу чергу потрібно описати хто є споживачем. Враховуючи, що є плагіном, що будуть використовувати при подальшій розробці, то можливо сказати, що

основним клієнтом в такому випадку є в першу чергу бізнес, що створює свої продукти використовуючи Unreal Engine. Покупка плагіну для них є більш доцільною ніж розробка складної системи з залученням декількох спеціалістів.

Отриманий портрет клієнту включає в себе бізнес різного розміру, а отже їх цілі можуть відрізнятись. Загально, незалежно від розміру бізнесу існує потреба у максимальній оптимізації плагіну, так як симуляція рідини буде не єдиним процесом працюючим у реальному часі, а тому він повинен займати якнайменшу кількість ресурсів. Звідси виходить, що гнучкість також важлива для всіх споживачів, так як підлаштовуватись під різні типи обладнання потрібно як великим проектам так і інді. Відмінності існують у двох ключових потребах: ціна та гнучкість візуалізації. Великі проекти часто потребують більших можливостей кастомізації, таку потребу закривають два фактори. Перший - це можливість за потреби розширення класів плагіну та додання потрібних особливостей. Другий – можливість детального налаштування рідини, використання інноваційних методів має під собою врахування досить великої кількості факторів, які можливо винести в налаштування, щоб надати можливість досягти того, що симуляція буде виглядати саме так як того хоче дизайнер. Менші проекти більше звертають увагу на ціну і легкість влаштування в проект, ця потреба легко закривається запропонованим проектом. Крім того, можливо ввести декілька видів прав використання плагіну в залежності від розміру прибутку проекту, що дозволить проекту бути гнучким та краще адаптуватись під можливості і потреби клієнта. Отже, унікальною ціннісною пропозицією є гнучкий спосіб симуляції рідин, що готовий підлаштовуватись під різні потреби клієнтів.

До конкурентних переваг відносяться наступні пункти.

- Відсутність необхідності залучати декількох спеціалістів для розробки складної системи симуляції рідин зменшить витрати на їх утримання.

- Легкість підключення до проєкту. Розробка у формі плагіну і використання внутрішньої торгівельної платформи надасть змогу використовувати програмне забезпечення без внесення докорінних змін в інші частини проєкту.
- Використання інноваційних методів дозволяє плагіну бути більш оптимізованим для ефективної роботи у реальному часі, що має вирішальне значення у багатьох проєктах.
- Легкість використання. Використання стандартного інтерфейсу для роботи з об'єктами в середині рушія дозволяє легко використовувати програмне забезпечення без потреби у додатковому навчанні.

5.3. Доходи та витрати

При розробці продуктового стартапу важливо розуміти, що витрати починаються задовго до отримання будь-якого прибутку. Продукт повинен пройти усі стадії планування та більшість стадій розробки перед тим як вийти на ринок. І навіть після отримання першого прибутку знадобиться час, щоб покрити витрати за час розробки. Отже для порівняння буде створено таблиці витрат на час розробки і таблиці прибутку лише на час починаючи з виходу продукту на ринок, так як на час розробки виторг буде нульовим.

Досить важко оцінити конкретну кількість ліцензій, що будуть продані, особливо маючи на увазі відсутність статистики по продуктах на майданчику. Тому для оцінки було вирішено проконсультуватись на форумах розробників, які мали досвід випуску продуктів та готові їм поділитись. Інформація досить різна, але багато розробників сходяться у тому, що багато залежить від якості та специфічності продукту, крім того в перший місяць випуску товар продається найбільш активно та в подальшому може підніматись у пошуку за рахунок регулярних оновлень. Також розробники часто кажуть про 1-2 продажі в день в перший місяць та

сильне падіння продажів у другий з подальшою стабільністю. Тому, було зроблено припущення, що за рахунок новизни продукту у перший квартал буде зроблено найбільше продажів – 70. В другому кварталі ситуація повинна стабілізуватись і вийти на 30 продажів за квартал. До 3 кварталу будуть зібрана перша статистика по відгукам та виправлено більшість релізних помилок, що має під собою велике оновлення, що приведе до зросту продажів. І в 4 кварталі починається сезон знижок (чорна п'ятниця, кіберпонеділок, Різдво і т.д.), робити знижки в такий час є вигідною маркетинговою стратегією. Підсумовуючи все це було створено таблицю:

Таблиця 6

План доходу від продажу товарів та послуг на 2026 р

№	Назва статті доходу	1квартал 2026 р.	2квартал 2026 р.	3квартал 2026 р.	4квартал 2026 р.	Всього за 2026 р.
1	Кількість ліцензій (шт.)	70	30	45	45	190
2	Ціна ліцензії (тис. грн)	1	1	1	0.8	
3	Виторг від ліцензій (тис. грн)	70	30	45	36	181

Видатки діляться на 2 проміжки часу: до та після випуску продукту. До випуску продукту основний вклад іде на дослідну роботу, що включає в себе покупку спеціалізованих видань та статей за темою та безпосередньо час розробника. Після час розробника залишається, але основні методи вже імплементовані, а підтримка не потребує досліджень. Але до витрат додаються податки, розмір яких було взято як ПДФО (18%) помножений на очікуваний прибуток. Щодо розробки, то стартапом займається його розробник і робота оцінюється в його часі, що важко перевести в грошовий

еквівалент, особливо маючи на увазі, важкість оцінки кількості часу, що піде на проєкт.

Результати розрахунку проілюстровано у табл. 7 та 8. Таким чином сумарні витрати досягають 37.08 тис. грн. А прибуток сягає 143.92 тис. грн.

Таблиця 7

Структура та розрахунок витрат (грн) на дослідну роботу на 2025 р

№	Назва статті витрат	4квартал 2024 р.	1квартал 2025 р.	2квартал 2025 р.	3квартал 2025 р.	4квартал 2025 р.	Всього за 2025 р.
1	Дослідна робота (тис. грн)	0	2	2	0.5	0	4.5

Таблиця 8

Розрахунок податкових платежів (грн)

№	Назва статті витрат	1квартал 2026 р.	2квартал 2026 р.	3квартал 2026 р.	4квартал 2026 р.	Всього за 2026 р.
1	Податкові платежі (тис. грн)	12,6	5,4	8,1	6,48	32,58

5.4. Бізнес-модель

Побудова бізнес-моделі є завершальним етапом розробки стартапу. На цьому кроці всі дані збираються та стягуються до однієї таблиці. Цей етап є дуже важливим через те, що саме надає можливість побачити модель майбутнього бізнесу в цілому, краще зрозуміти його та виявити потенційні проблеми.

Для роботи було обрано шаблон канви бізнес-моделі, що є відомим шаблоном для побудови бізнес-моделей. Він включає в себе наступні сегменти:

- проблема – проблема яку вирішує продукт створюваного бізнесу,
- інноваційний спосіб/метод – новий спосіб, який бізнес використовує у своєму продукті,
- унікальна ціннісна пропозиція – чим продукт цікавий для споживача, яка його цінність для нього,
- зацікавлені сторони – люди, що зацікавлені у продукті,
- ринки – ті в кого виникає проблема,
- рішення – основні концепції, підхід до рішення проблеми,
- програмний продукт – основний продукт стартапу,
- конкурентні переваги – чим продукт виділяється серед аналогів,
- канали збуту – як продукт доходить до клієнтів,
- потоки доходів – як саме бізнес приносить гроші,
- структура витрат – можливі статті витрат.

Отже підсумовуючі підрозділи 5.1 – 5.3 можливо виділити наступні факти, що були використані для побудови бізнес моделі.

Проблема: Складність процесу симуляції рідини в реальному часі.

Зацікавлені сторони:

- розробник,
- програмісти ігор,
- інші розробники ігор,
- 3D художники, художники по оточенню,
- дизайнер рівнів,
- архітектори,
- власники рушія Unreal Engine (Компанія Epic Games),
- науковці у сфері симуляції рідин.

Рішення. Представленим рішенням є програмне забезпечення у вигляді плагіну до рушію Unreal Engine. Що розроблене з використанням нового інноваційного способу симуляції рідини, має можливості для гнучкого налаштування, як самої симуляції, так і її візуалізації. А також надає можливість легкого підключення до проєкту за рахунок формату розробки.

Ринок. Ринок, як споживачі продукту буде спиратись на людей, що використовують рушій Unreal Engine при створенні своїх продуктів. Клієнтами будуть бізнеси різних розмірів, але здебільшого малі та середні бізнеси, що мають менший бюджет на створення своєї розробки, але все ще її потребують.

Канали збуту. Основним каналом збуту стане внутрішній торгівельний майданчик Unreal Engine. Майданчик створений компанією Epic Games для поширення різного виду ресурсів для роботи з рушієм, що є вигідним, як для компанії так і для споживачів. Майданчик постійно розвивається та активно використовується більшою частиною розробників на цьому рушії, що гарантує, що продукт побачить саме цільова аудиторія.

Унікальна ціннісна пропозиція. Основною ціннісною пропозицією є програмне забезпечення, що реалізує інноваційний спосіб симуляції рідини в реальному часі з можливістю гнучкого налаштування та різними типами прав в залежності від прибутку проєкту.

Конкурентні переваги. Відносно створення власних систем – це відсутність необхідності розробляти складну систему залучаючи до цього штат розробників. А також легкість підключення та використання. Відносно інших – це гнучке налаштування та використання інноваційного способу для симуляції рідин.

Потоки доходів. Основними доходами є доходи від продажу прав на використання плагіну.

Структура витрат. Утримання розробника під час розробки та підтримки проєкту після введення в експлуатацію. Оплата комунальних послуг та інтернету. Підтримка робочого обладнання.

Тоді загальна бізнес модель має наступний вигляд, представлений у табл. 9.

Таблиця 9

Канва бізнес моделі

Проблема	Інноваційна технологія, спосіб, метод	Унікальна ціннісна пропозиція	Зацікавлені сторони	Ринок
Складність процесу симуляції рідини в реальному часі	Поєднання ключових переваг методів FLIP та Implicit Density Projection з використанням порогу для створення більш досконалого способу симуляції рідин.	Програмне забезпечення, що реалізує іноваційний спосіб симуляції рідини в реальному часі з можливістю налаштування та легкою інтеграцією до проєкту.	Програмісти ігор Інді розробники ігор 3D художники, художники по оточенню Архітектори Власники рушія Unreal Engine (Компанія Epic Games) Науковці у сфері симуляції рідин	Бізнеси різних розмірів, особливо компанії для розробки ігор, що використовують Unreal Engine у своїх проєктах.

Рішення	Програмний продукт		Конкурентні переваги	Канали збуту
Програмне забезпечення у вигляді плагіну до Unreal Engine з можливістю гнучкого налаштування.	Плагін до Unreal Engine для симуляції рідини в реальному часі		<p>1. Відсутність необхідності залучати декількох спеціалістів</p> <p>2. Легкість підключення до проєкту.</p> <p>3. Використання інноваційних методів.</p> <p>4. Гнучкість</p> <p>5. Легкість використання.</p>	Внутрішній торговельний майданчик.
Структура витрат		Потоки доходів		
Утримання розробника під час розробки та підтримки проєкту після введення в експлуатацію. Оплата комунальних послуг та інтернету. Підтримка робочого обладнання.		Доходи від продажу прав на використання плагіну.		

5.5. Висновки до розділу

В п'ятому розділі було розглянуто перспективи стартап проєкту на базі розробленого способу та програмного забезпечення. Для цього було

проведено аналіз зацікавлених сторін та виявлено найбільш впливові з них. Розглянуто унікальну ціннісну пропозицію проєкту, відмічено його сильні сторони. А також розраховано потенційні прибутки.

ВИСНОВКИ

Ця магістерська дисертація присвячена створенню способу та програмного забезпечення для симуляції рідин в реальному часі.

На початку роботи було описано актуальність дослідження, описано приклади використання технологій цієї галузі в реальному світі. Описано особливості та потреби до моделювання поведінки рідин. Далі в ході проведення дослідження було проведено аналіз існуючих методів та способів симуляції. Існуючі рішення було розбито на декілька груп та пояснено переваги й недоліки кожної. Для кожної групи також було наведено типове практичне використання та декілька прикладів програмного забезпечення. На основі проведеного аналізу було обрано працювати з методами гібридної групи, яка характеризується балансом між якістю та швидкістю, а також найчастіше використовується для симуляцій в реальному часі. Після чого було виділено недоліки методів цієї групи, а саме FLIP та Implicit Density Projection, на основі яких було обрано напрям дослідження та модифікації.

Було детально описано запропоновані модифікації. Основою для якого стала корекція положень частинок за перевищення адаптивного порогу густини. Крім того, було описано загальний алгоритм способу, які етапи він має. Кожен з етапів було детально описано та пояснено, виділено вхідні та вихідні дані. Пояснено порядок етапів та фізичні явища, що їм відповідають.

Також на основі запропонованого способу було розроблене програмне забезпечення. Для програмного забезпечення було висунуто вимоги, на основі яких обрано та обґрунтовано вибір інструментів розробки. Було описано архітектуру програмного забезпечення та його частини. Саме програмне забезпечення було розроблено у вигляді плагіну до рушія Unreal Engine, що одразу надає великий інструментарій для роботи, просту інтеграцію з іншими проєктами та торговий майданчик разом з великою

аудиторією для подальшої дистрибуції. Плагін містить в собі декілька основних компонентів: допоміжні функції симуляції рідини, зв'язуючі класи, що забезпечують інтеграцію з рушієм та використовують підготовлені функції для симуляції рідин, систему візуалізації з використанням вбудованого плагіну Niagara, інтеграція з системою, що дозволяє редагувати змінні симуляції та працювати з нею без знання коду – Blueprints. Кожен з компонентів та його функції також було описано більш розгорнуто у відповідних підрозділах.

На завершальному етапі розробки було проведено експериментальне дослідження розробленого способу. Метою якого було показати можливість вирішувати проблеми з якими не можуть впоратись класичні методи, а також оцінити швидкодію розробленого способу з використання створеного програмного забезпечення. В результаті було доведено, що використання висунутих модифікацій призвело до збільшення швидкодії на 20.84%.

Останнім етапом стала розробка бізнес-моделі стартап проєкту, основними продуктом якого є розроблене програмне забезпечення. Для цього було описано зацікавлені сторони, виділено їй інтерес та стратегію роботи з ними. Виділено конкурентні переваги, що дозволяють продукту виділитись серед інших. На основі чого створено унікальну ціннісну пропозицію. Далі було детально розписано доходи та витрати. Останнім кроком було створено загальну бізнес модель, що стисло викладає основну інформацію про створений стартап проєкт.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Zhu Y., Bridson R. Animating sand as a fluid. *ACM Transactions on Graphics*. 2005. Vol. 24, no. 3. P. 965–972. Режим доступу: <https://doi.org/10.1145/1073204.1073298>. Дата доступу 17.12.2025.
2. Frost B., Stomakhin A., Narita H. Moana: Performing Water. *SIGGRAPH '17: Special Interest Group on Computer Graphics and Interactive Techniques Conference* article No.: 30, p. 1 - 2, Los Angeles California. New York, NY, USA, 2017. Режим доступу: https://alexey.stomakhin.com/research/siggraph2017_gretchen.pdf. Дата доступу 17.12.2025.
3. Wadsley J. W., Keller B. W., Quinn T. R. Gasoline2: a modern smoothed particle hydrodynamics code. *Monthly Notices of the Royal Astronomical Society*. 2017. Vol. 471, no. 2. P. 2357–2369. Режим доступу: <https://doi.org/10.1093/mnras/stx1643>. Дата доступу 17.12.2025.
4. Bridson R. *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, 2018.
5. Deep Fluids: A Generative Network for Parameterized Fluid Simulations / B. Kim et al. *Computer Graphics Forum*. 2019. Vol. 38, no. 2. P. 59–70. Режим доступу: <https://doi.org/10.1111/cgf.13619>. Дата доступу 17.12.2025.
6. Shadden S. C., Arzani A. Lagrangian Postprocessing of Computational Hemodynamics. *Annals of Biomedical Engineering*. 2014. Vol. 43, no. 1. P. 41–58. Режим доступу: <https://doi.org/10.1007/s10439-014-1070-0>. Дата доступу 17.12.2025.
7. Muller, M., Charypar, D. & Gross, M. Particle-based fluid simulation for interactive applications, *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* P. 154 - 159 – 2003. Режим доступу:

- <https://dl.acm.org/doi/10.5555/846276.846298>. Дата доступа 17.12.2025.
8. Samulyak R., Wang X., Chen H.-C. Lagrangian particle method for compressible fluid dynamics. *Journal of Computational Physics*. 2018. Vol. 362. P. 1–19. Режим доступа: <https://doi.org/10.1016/j.jcp.2018.02.004>. Дата доступа 17.12.2025.
 9. Smoothed particle hydrodynamics: Methodology development and recent achievement / C. Zhang et al. *Journal of Hydrodynamics*. 2022. Vol. 34, no. 5. P. 767–805. Режим доступа: <https://link.springer.com/article/10.1007/s42241-022-0052-1>. Дата доступа 17.12.2025.
 10. Position based dynamics / M. Müller et al. *Journal of Visual Communication and Image Representation*. 2007. Vol. 18, no. 2. P. 109–118. Режим доступа: <https://doi.org/10.1016/j.jvcir.2007.01.005>. Дата доступа 17.12.2025.
 11. Structure-preserving discretization of incompressible fluids / D. Pavlov et al. *Physica D: Nonlinear Phenomena*. 2011. Vol. 240, no. 6. P. 443–458. Режим доступа: <https://doi.org/10.1016/j.physd.2010.10.012>. Дата доступа 17.12.2025.
 12. Harlow F. H., Welch J. E. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *Physics of Fluids*. 1965. Vol. 8, no. 12. P. 2182. Режим доступа: <https://doi.org/10.1063/1.1761178>. Дата доступа 17.12.2025.
 13. A Computational Model of Thrombus Growth Based on Level Set Method / C. Ma et al. *IEEE Access*. 2021. Vol. 9. P. 100769–100780. Режим доступа: <https://doi.org/10.1109/ACCESS.2021.3091556>. Дата доступа 17.12.2025.
 14. Stam J. Stable fluids. SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques,

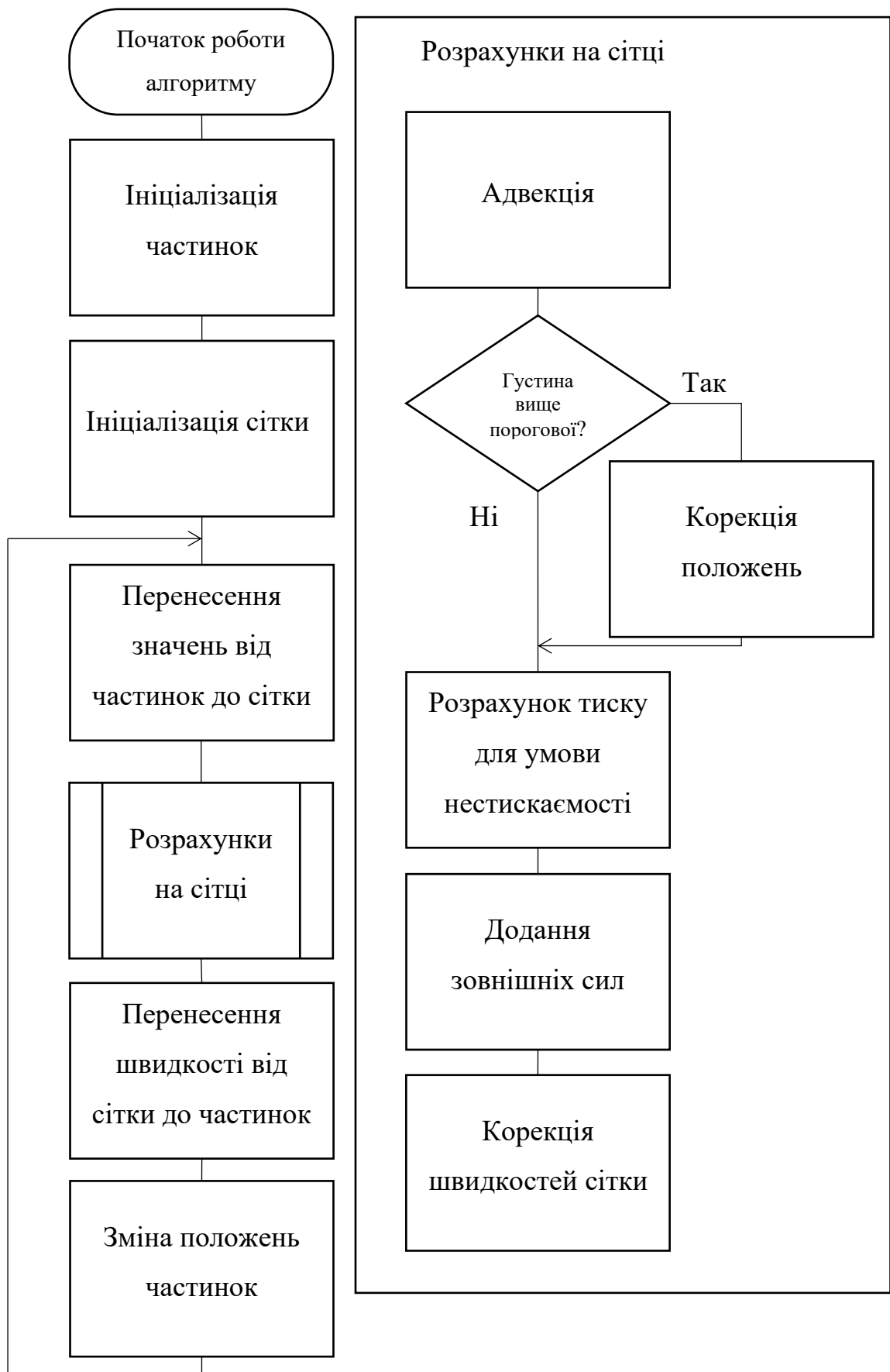
- Pages 121 – 128, New York, USA, 1999. Режим доступа: <https://doi.org/10.1145/311535.311548>. Дата доступа 17.12.2025.
15. The affine particle-in-cell method / C. Jiang et al. ACM Transactions on Graphics. 2015. Vol. 34, no. 4. P. 1–10. Режим доступа: <https://doi.org/10.1145/2766996>. Дата доступа 17.12.2025.
 16. Evans, Martha W. and Harlow, F. H. (1957). “The Particle-in-Cell Method for Hydrodynamic Calculations,” Los Alamos Scientific Laboratory, New Mexico.
 17. Brackbill J. U., Kothe D. B., Ruppel H. M. Flip: A low-dissipation, particle-in-cell method for fluid flow. Computer Physics Communications. 1988. Vol. 48, no. 1. P. 25–38. Режим доступа: [https://doi.org/10.1016/0010-4655\(88\)90020-3](https://doi.org/10.1016/0010-4655(88)90020-3). Дата доступа 17.12.2025.
 18. The Impulse Particle - In - Cell Method / S. Sancho et al. Computer Graphics Forum. 2024. Режим доступа: <https://doi.org/10.1111/cgf.15022>. Дата доступа 17.12.2025.
 19. Machine Learning in Fluid Dynamics–Physics-Informed Neural Networks (PINNs) Using Sparse Data: A Review / M. El Hassan et al. Fluids. 2025. Vol. 10, no. 9. P. 226. Режим доступа: <https://doi.org/10.3390/fluids10090226>. Дата доступа 17.12.2025.
 20. Recent Advances in Computational Fluid Dynamics Using Artificial Intelligence, Machine Learning, and Deep Learning: A Comprehensive Review / T. Rajashekaraiyah et al. Journal of Mines, Metals and Fuels. 2025. P. 3469–3481. Режим доступа: <https://doi.org/10.18311/jmmf/2025/50326>. Дата доступа 17.12.2025.
 21. Implicit Density Projection for Volume Conserving Liquids / T. Kugelstadt et al. IEEE Transactions on Visualization and Computer Graphics. 2019. P. 1. Режим доступа: <https://doi.org/10.1109/TVCG.2019.2947437>. Дата доступа 17.12.2025.

22. Solving viscoelastic free surface flows of a second-order fluid using a marker-and-cell approach / M. F. Tomé et al. *International Journal for Numerical Methods in Fluids*. 2007. Vol. 53, no. 4. P. 599–627. Режим доступа: <https://doi.org/10.1002/fld.1298>. Дата доступа 17.12.2025.
23. Foster N., Metaxas D. *Realistic Animation of Liquids*. *Graphical Models and Image Processing*. 1996. Vol. 58, no. 5. P. 471–483.
24. Ships, splashes, and waves on a vast ocean / L. Huang et al. *ACM Transactions on Graphics*. 2021. Vol. 40, no. 6. P. 1–15. Режим доступа: <https://doi.org/10.1145/3478513.3480495>. Дата доступа 17.12.2025.
25. Ando R., Tsuruno R. A particle-based method for preserving fluid sheets. the 2011 ACM SIGGRAPH/Eurographics Symposium, Vancouver, British Columbia, Canada, 5–7 August 2011. New York, New York, USA, 2011. Режим доступа: <https://doi.org/10.1145/2019406.2019408>. Дата доступа 17.12.2025.
26. Briand L., Jourden H., Pérache M. Julia versus C++ Kokkos for performance portable Cartesian CFD solvers on heterogeneous architectures. *The International Journal of High Performance Computing Applications*. 2025. Режим доступа: <https://doi.org/10.1177/10943420251341179>. Дата доступа 17.12.2025.
27. Performance analysis of GPU accelerated meshfree q-LSKUM solvers in Fortran, C, Python, and Julia / N. R. Mamidi et al. 2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC), Bengaluru, India, 18–21 December 2022. 2022. Режим доступа: <https://doi.org/10.1109/HiPC56025.2022.00031>. Дата доступа 17.12.2025.
28. LLAMA: The low - level abstraction for memory access / B. M. Gruber et al. *Software: Practice and Experience*. 2022. Режим доступа: <https://doi.org/10.1002/spe.3077>. Дата доступа 17.12.2025.

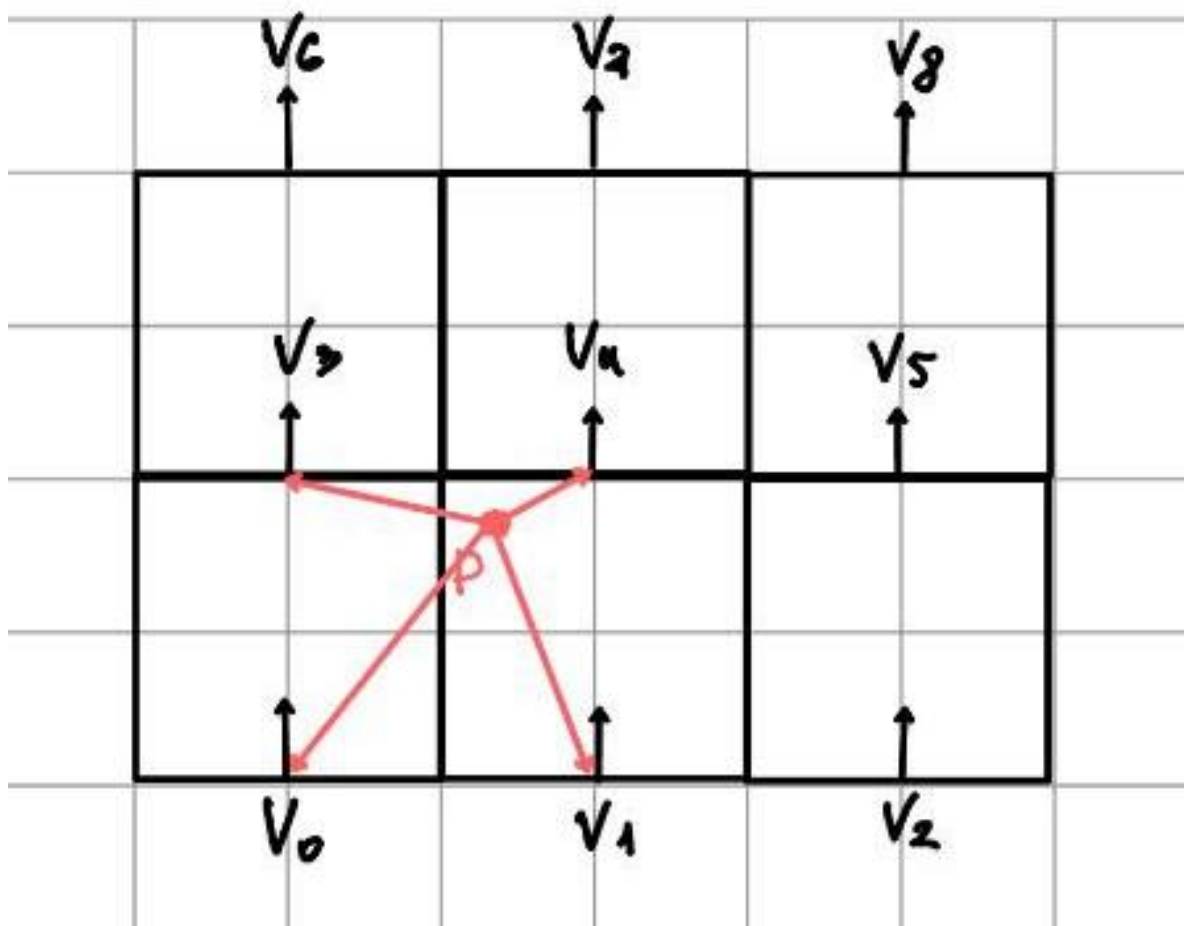
29. Micale D., Bracconi M., Maestri M. Increasing Computational Efficiency of CFD Simulations of Reactive Flows at Catalyst Surfaces through Dynamic Load Balancing. Режим доступа: <https://doi.org/10.1021/acsengineeringau.3c00066>. Дата доступа 17.12.2025.
30. Bessonov O. Technological Aspects of the Hybrid Parallelization with OpenMP and MPI. Lecture Notes in Computer Science. Cham, 2017. P. 101–113.
31. ClangJIT: Enhancing C++ with Just-in-Time Compilation / H. Finkel et al. 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Denver, CO, USA, 22 November 2019. 2019. Режим доступа: <https://doi.org/10.48550/arXiv.1904.08555>. Дата доступа 17.12.2025.
32. Unreal Engine Documentation. Dev EpicGames. Режим доступа: <https://dev.epicgames.com/documentation>. Дата доступа 17.12.2025.
33. Tan T. W. Harnessing the Power of Niagara: Practical Examples in Unreal Engine 5. Game Development with Unreal Engine 5 Volume 1. Berkeley, CA, 2024. P. 277–409. Режим доступа: https://doi.org/10.1007/978-1-4842-9824-4_7. Дата доступа 17.12.2025.
34. Nixon D. Blueprints. Beginning Unreal Game Development. Berkeley, CA, 2020. P. 115–164.

ДОДАТКИ

Додаток 1
Копії графічних матеріалів

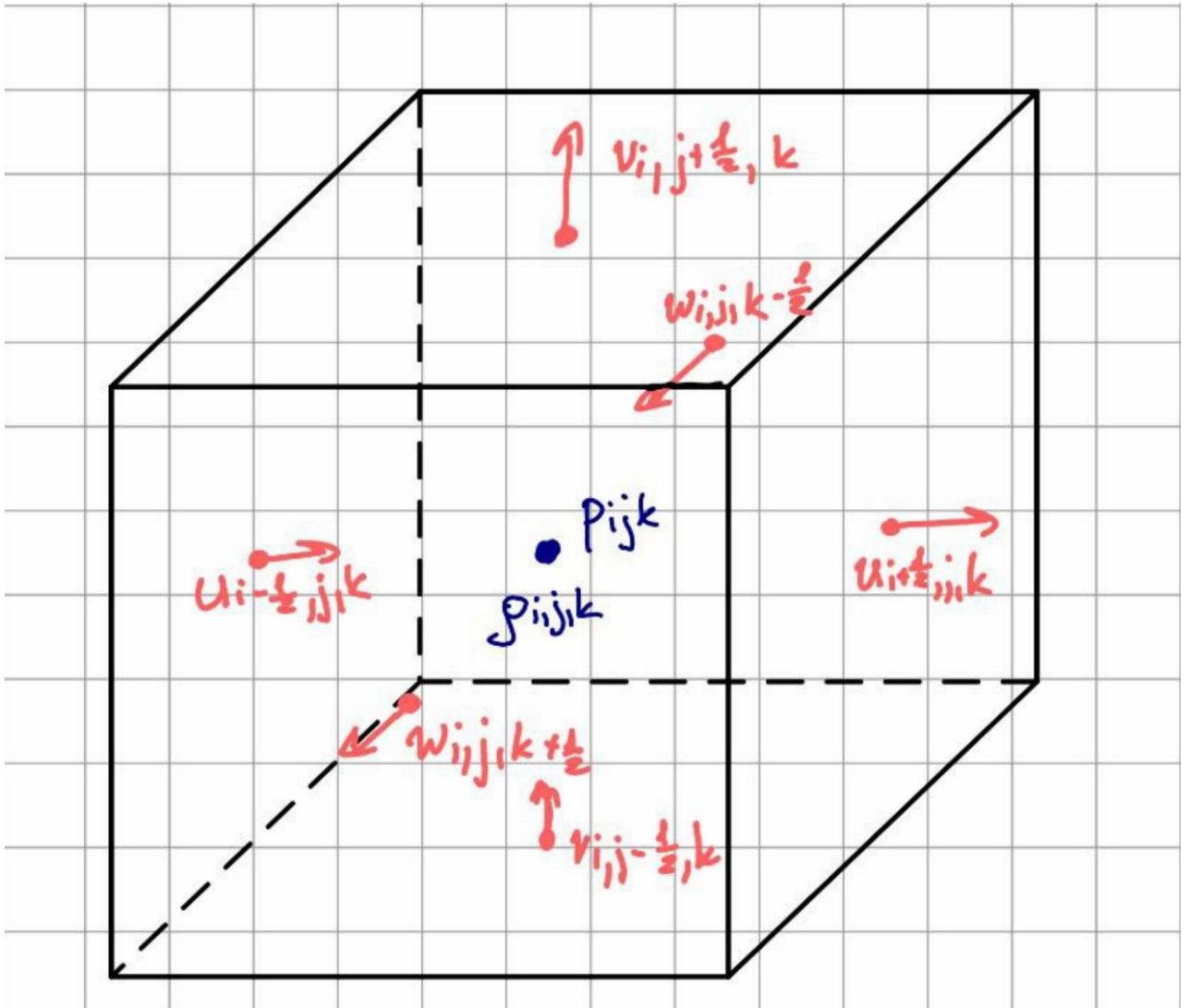


Загальна блок-схема способу
Седухіна А.Д., група КП-41мп



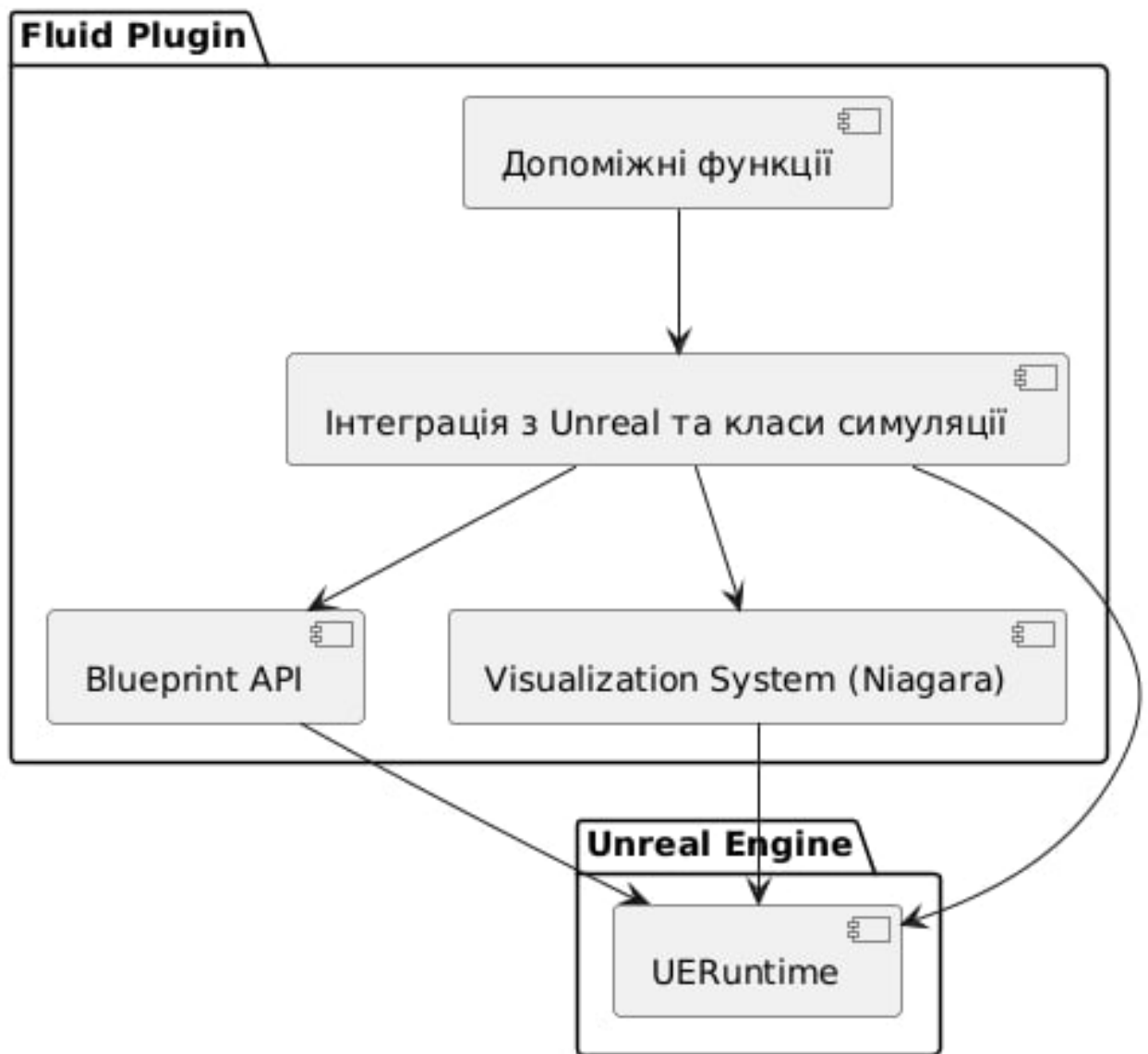
Вибір швидкостей сітки на які впливає частинка

Седухіна А.Д., група КП-41мп



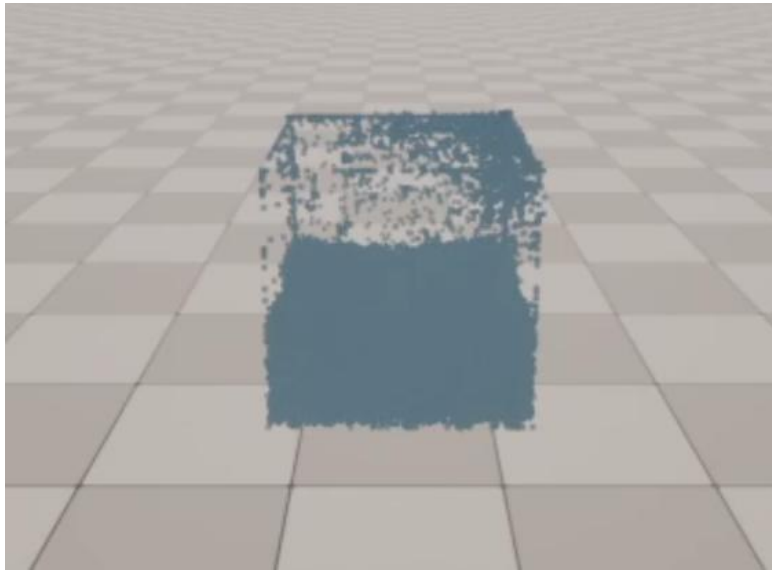
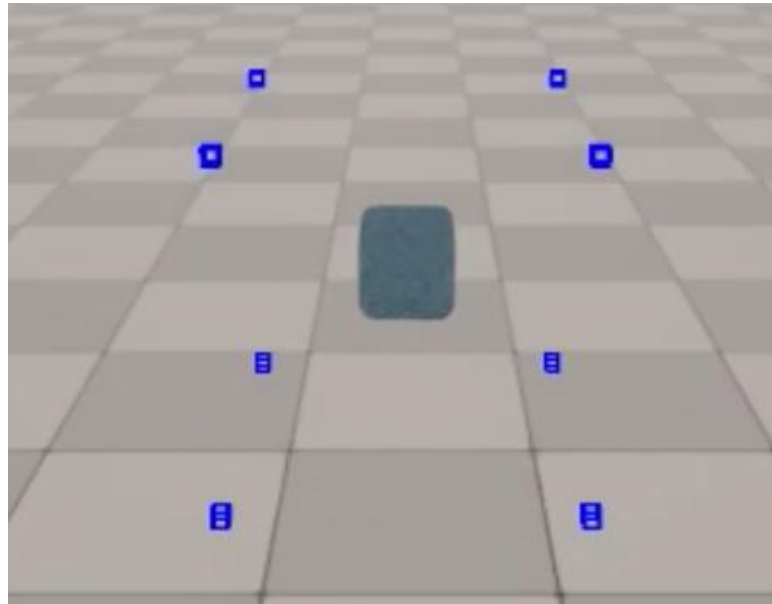
Клітина сітки для розробленого способу

Седухіна А.Д., група КП-41мп



Архітектура розробленого програмного забезпечення

Седухіна А.Д., група КП-41мп



Приклад результату симуляції
Седухіна А.Д., група КП-41мп

Проблема Складність процесу симуляції рідини в реальному часі	Інноваційна технологія, спосіб, метод Поєднання ключових переваг методів FLIP та Implicit Density Projection з використанням порогу для створення більш досконалого способу симуляції рідин.	Унікальна ціннісна пропозиція Програмне забезпечення, що реалізує іноваційний спосіб симуляції рідини в реальному часі з можливістю налаштування та легкою інтеграцією до проекту.	Зацікавлені сторони Програмісти ігор Інді розробники ігор 3D художники, художники по оточенню Дизайнери рівнів Архітектори Власники рушія Unreal Engine (Компанія Epic Games) Науковці у сфері симуляції рідин	Ринок Бізнеси різних розмірів, особливо компанії для розробки ігор, що використовують Unreal Engine у своїх проектах.
Рішення Програмне забезпечення у вигляді плагіну до Unreal Engine з можливістю гнучкого налаштування.	Програмний продукт Плагін до Unreal Engine для симуляції рідини в реальному часі		Конкурентні переваги 1. Відсутність необхідності залучати декількох спеціалістів 2. Легкість підключення до проекту. 3. Використання іноваційних методів. 4. Гнучкість 5. Легкість використання.	Канали збуту Внутрішній торговельний майданчик.
Структура витрат Утримання розробника під час розробки та підтримки проекту після введення в експлуатацію. Оплата комунальних послуг та інтернету. Підтримка робочого обладнання.		Потоки доходів Доходи від продажу прав на використання плагіну.		

Бізнес-модель

Седухіна А.Д., група КП-41мп

Додаток 2
Лістинг програми

```
===== FluidSim\Private\Fluid.cpp =====
```

```
// Fill out your copyright notice in the Description page of Project Settings.
```

```
#include "Fluid.h"
```

```
#include "NiagaraDataInterfaceArrayFunctionLibrary.h"
```

```
#include "NiagaraFunctionLibrary.h"
```

```
#include "NiagaraComponent.h"
```

```
#include "IndexingFunctions.hpp"
```

```
#include "ParticleHelperFunctions.hpp"
```

```
#include <DebugFunctions.hpp>
```

```
int it = 0;
```

```
float sum = 0;
```

```
// Sets default values
```

```
AFluid::AFluid()
```

```
{
```

```
    // Set this actor to call Tick() every frame. You can turn this off to improve performance if  
    you don't need it.
```

```
    PrimaryActorTick.bCanEverTick = true;
```

```
}
```

```
float AFluid::GetTotalAmountOfCellsPositions() const
```

```
{
```

```
    return GridPositionsAmount.X * GridPositionsAmount.Y * GridPositionsAmount.Z;
```

```
}
```

```
void AFluid::SpawnParticles()
```

```
{
```

```
    FVector ParticleSpawnMin = GridPositions[0] + FVector(0.5);
```

```
    FVector ParticleSpawnMax = GridPositions.Last() - FVector(0.5);
```

```
    //FVector Diapazone = ParticleSpawnMax - ParticleSpawnMin;
```

```
    //ParticleSpawnMin += 0.4f * Diapazone;
```

```
    //ParticleSpawnMax -= 0.4f * Diapazone;
```

```
    ParticlePositions.Reserve(ParticlesAmount);
```

```
    while (ParticlePositions.Num() < ParticlesAmount)
```

```
    {
```

```
        FVector Position(
```

```
            FMath::FRandRange(ParticleSpawnMin.X, ParticleSpawnMax.X),
```

```
            FMath::FRandRange(ParticleSpawnMin.Y, ParticleSpawnMax.Y),
```

```
            FMath::FRandRange(ParticleSpawnMin.Z, ParticleSpawnMax.Z)
```

```
        );
```

```
        ParticlePositions.Add(Position);
```

```
        ParticleVelocities.Add(FVector(0));
```

```
    }
```

```
}
```

```

void AFluid::BeginPlay()
{
    Super::BeginPlay();

    // Grid initialization
    //
    // Setting grid positions amount
    GridPositionsAmount = GridCellsAmount + FIntVector(1);

    // Initializing Grid values
    // Half grid velocities, half grid weights, pressure, density, particles in cell, divergence
    ZeroGrid();

    // Creating Grid cells
    // First is at the bottom
    GridPositions.Reserve(GetTotalAmountOfCellsPositions());
    for (size_t i = 0; i < GridPositionsAmount.X; i++)
    {
        for (size_t j = 0; j < GridPositionsAmount.Y; j++)
        {
            for (size_t k = 0; k < GridPositionsAmount.Z; k++)
            {
                GridPositions.Add(GetActorLocation() + (FVector(i, j, k) *
CellSize));
            }
        }
    }

    ParticlesInCellMax = PerfectWaterDensity * CellSize * CellSize * CellSize;

    // Creating Particles
    SpawnParticles();

    // Particle Visualization parametrs set
    ParticlesVisualizationComponent =
UNiagaraFunctionLibrary::SpawnSystemAtLocation(GetWorld(), ParticlesVisualization,
FVector(0), GetActorRotation());
    ParticlesVisualizationComponent->SetIntParameter("ParticlesAmount",
ParticlesAmount);
    UNiagaraDataInterfaceArrayFunctionLibrary::SetNiagaraArrayPosition(ParticlesVisualiz
ationComponent, "ParticlesPositions", ParticlePositions);

    it = 0;
    sum = 0;
}

// Called every frame
void AFluid::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    double StartTime = FPlatformTime::Seconds();

```

```

/* DENSITY PROJECTION*/
// Collecting only densities first
ParticlesToGrid(false);
DensityProjection(DeltaTime);

// Collecting new density and speed
/* P2G */
ParticlesToGrid(true);

/*EXTERNAL FORCES*/
ApplyExternalForces(DeltaTime);

HalfGridVelocitiesXCorrectedP = HalfGridVelocitiesX;
HalfGridVelocitiesYCorrectedP = HalfGridVelocitiesY;
HalfGridVelocitiesZCorrectedP = HalfGridVelocitiesZ;

/*PRESSURRE PROJECTION*/
PressureProjection(DeltaTime);

/* G2P */
GridToParticles(0.05f);
PushParticles(DeltaTime);

UE_LOG(LogTemp, Log, TEXT("Max Density: %f"), FMath::Max(GridDensity));
UE_LOG(LogTemp, Log, TEXT("Min Density: %f"), FMath::Min(GridDensity));
UE_LOG(LogTemp, Log, TEXT("Max GridPressure: %f"), FMath::Max(GridPressure));
UE_LOG(LogTemp, Log, TEXT("Min GridPressure: %f"), FMath::Min(GridPressure));

ZeroGrid();

// Updating visual system
UNiagaraDataInterfaceArrayFunctionLibrary::SetNiagaraArrayPosition(ParticlesVisualizationComponent, "ParticlesPositions", ParticlePositions);

// Efficiency log
double EndTime = FPlatformTime::Seconds();
double Elapsed = EndTime - StartTime;
UE_LOG(LogTemp, Warning, TEXT("Iteration %d completed in %f seconds"), it, Elapsed);
sum += Elapsed;
UE_LOG(LogTemp, Warning, TEXT("All iterations completed in %f seconds"), sum);
it++;
}

void AFluid::DrawGrid()
{
    FVector OffsetToBoxCenter = FVector(CellSize/2);
    for (size_t i = 0; i < GridCellsAmount.X; i++)
    {
        for (size_t j = 0; j < GridCellsAmount.Y; j++)
        {

```

```

        for (size_t k = 0; k < GridCellsAmount.Z; k++)
        {
            DrawDebugBox(GetWorld(), GridPositions[i *
GridPositionsAmount.Y * GridPositionsAmount.Z + j * GridPositionsAmount.Z + k] +
OffsetToBoxCenter, FVector(CellSize / 2), GetActorRotation().Quaternion(), FColor::Blue, false,
5, 0, 0.3);
        }
    }
}

```

```

void AFluid::DrawGridPoints()
{
    for (FVector Pos : GridPositions)
    {
        DrawDebugPoint(GetWorld(), Pos, 22, FColor::Purple, true, 10);
    }
}

```

```

void AFluid::DrawGridCorners()
{
    FVector OffsetToBoxCenter = FVector(CellSize / 2);
    for (size_t i = 0; i < GridCellsAmount.X; i += GridCellsAmount.X - 1)
    {
        for (size_t j = 0; j < GridCellsAmount.Y; j += GridCellsAmount.Y - 1)
        {
            for (size_t k = 0; k < GridCellsAmount.Z; k += GridCellsAmount.Z - 1)
            {
                DrawDebugBox(GetWorld(), GridPositions[i *
GridPositionsAmount.Y * GridPositionsAmount.Z + j * GridPositionsAmount.Z + k] +
OffsetToBoxCenter, FVector(CellSize / 2), GetActorRotation().Quaternion(), FColor::Blue, false,
5, 0, 0.3);
            }
        }
    }
}

```

```

// Used for pressure and density
float AFluid::GetNeighboursSum(int X, int Y, int Z, const TArray<float>* Array,
EOnBoundaryValue BoundaryVal) const
{
    float Sum = 0;

    float Coef = (BoundaryVal == EOnBoundaryValue::Neighbour) ? 2.f : 1.f;

    // ---- X neighbors ----
    if (X == 0)
    {
        Sum += Coef * (*Array)[FlattenIndex(FIntVector(X + 1, Y, Z), GridCellsAmount)];
    }
    else if (X == GridCellsAmount.X - 1)
    {

```

```

        Sum += Coef * (*Array)[FlattenIndex(FIntVector(X - 1, Y, Z), GridCellsAmount)];
    }
    else
    {
        Sum += (*Array)[FlattenIndex(FIntVector(X - 1, Y, Z), GridCellsAmount)];
        Sum += (*Array)[FlattenIndex(FIntVector(X + 1, Y, Z), GridCellsAmount)];
    }

    // ---- Y neighbors ----
    if (Y == 0)
    {
        Sum += Coef * (*Array)[FlattenIndex(FIntVector(X, Y + 1, Z), GridCellsAmount)];
    }
    else if (Y == GridCellsAmount.Y - 1)
    {
        Sum += Coef * (*Array)[FlattenIndex(FIntVector(X, Y - 1, Z), GridCellsAmount)];
    }
    else
    {
        Sum += (*Array)[FlattenIndex(FIntVector(X, Y - 1, Z), GridCellsAmount)];
        Sum += (*Array)[FlattenIndex(FIntVector(X, Y + 1, Z), GridCellsAmount)];
    }

    // ---- Z neighbors ----
    if (Z == 0)
    {
        Sum += Coef * (*Array)[FlattenIndex(FIntVector(X, Y, Z + 1), GridCellsAmount)];
    }
    else if (Z == GridCellsAmount.Z - 1)
    {
        Sum += Coef * (*Array)[FlattenIndex(FIntVector(X, Y, Z - 1), GridCellsAmount)];
    }
    else
    {
        Sum += (*Array)[FlattenIndex(FIntVector(X, Y, Z - 1), GridCellsAmount)];
        Sum += (*Array)[FlattenIndex(FIntVector(X, Y, Z + 1), GridCellsAmount)];
    }

    return Sum;
}

void AFluid::PushParticles(float DeltaTime)
{
    const FVector UpperCorner = GridPositions.Last();
    const FVector LowerCorner = GridPositions[0];
    ParallelFor(ParticlePositions.Num(), [&](int32 Index)
    {
        FVector Pos = ParticlePositions[Index];
        FVector NewPos = Pos + ParticleVelocities[Index] * DeltaTime;
        ForceParticlePositionAndVelocityInBoundaries(UpperCorner, LowerCorner,
NewPos, ParticleVelocities[Index], Pos);
        ParticlePositions[Index] = NewPos;
    });
}

```

```

    });
}

void AFluid::ApplyExternalForces(float DeltaTime)
{
    const float ParticleGravity = GravityConst * DeltaTime * ParticleMassInGramms;
    // Adding external forces: Gravity
    ParallelFor(HalfGridVelocitiesZ.Num(), [&](int32 Index)
    {
        HalfGridVelocitiesZ[Index] += ParticleGravity;
    });
}

void AFluid::ZeroGrid()
{
    HalfGridVelocitiesX.Reset();
    HalfGridVelocitiesY.Reset();
    HalfGridVelocitiesZ.Reset();

    HalfGridWeightsX.Reset();
    HalfGridWeightsY.Reset();
    HalfGridWeightsZ.Reset();

    GridPressure.Reset();
    GridPressure_0.Reset();
    GridPressure_1.Reset();
    GridDensity.Reset();
    ParticlesInCell.Reset();
    GridDensityPotential_0.Reset();
    GridDensityPotential_1.Reset();
    GridRhs.Reset();
    StepDirection.Reset();
    StepSize.Reset();

    Adiaq.Reset();
    Ax.Reset();
    Ay.Reset();
    Az.Reset();

    HalfGridVelocitiesX.AddDefaulted((GridCellsAmount.X + 1) * GridCellsAmount.Y *
GridCellsAmount.Z);
    HalfGridVelocitiesY.AddDefaulted((GridCellsAmount.Y + 1) * GridCellsAmount.X *
GridCellsAmount.Z);
    HalfGridVelocitiesZ.AddDefaulted((GridCellsAmount.Z + 1) * GridCellsAmount.Y *
GridCellsAmount.X);

    HalfGridWeightsX.AddDefaulted((GridCellsAmount.X + 1) * GridCellsAmount.Y *
GridCellsAmount.Z);
    HalfGridWeightsY.AddDefaulted((GridCellsAmount.Y + 1) * GridCellsAmount.X *
GridCellsAmount.Z);
    HalfGridWeightsZ.AddDefaulted((GridCellsAmount.Z + 1) * GridCellsAmount.Y *
GridCellsAmount.X);
}

```

```

GridPressure.AddDefaulted(GetTotalAmountOfCellsPositions());
GridPressure_1.AddDefaulted(GetTotalAmountOfCellsPositions());
GridPressure_0.AddDefaulted(GetTotalAmountOfCellsPositions());
GridPressure_1.AddDefaulted(GetTotalAmountOfCellsPositions());
GridDensity.AddDefaulted(GetTotalAmountOfCellsPositions());
ParticlesInCell.AddDefaulted(GetTotalAmountOfCellsPositions());
GridDensityPotential_0.AddDefaulted(GetTotalAmountOfCellsPositions());
GridDensityPotential_1.AddDefaulted(GetTotalAmountOfCellsPositions());
GridRhs.AddDefaulted(GetTotalAmountOfCellsPositions());
StepDirection.AddDefaulted(GetTotalAmountOfCellsPositions());
StepSize.AddDefaulted(GetTotalAmountOfCellsPositions());

Adiag.AddDefaulted(GetTotalAmountOfCellsPositions());
Ax.AddDefaulted(GetTotalAmountOfCellsPositions());
Ay.AddDefaulted(GetTotalAmountOfCellsPositions());
Az.AddDefaulted(GetTotalAmountOfCellsPositions());
}

void AFluid::ClampBoundaryVelocities()
{
    for (size_t j = 0; j < GridCellsAmount.Y; j++)
    {
        for (size_t k = 0; k < GridCellsAmount.Z; k++)
        {
            int HalfGridXIndexZero = FlattenIndex(FIntVector(0, j, k),
FIntVector(GridCellsAmount.X + 1, GridCellsAmount.Y, GridCellsAmount.Z));
            HalfGridVelocitiesX[HalfGridXIndexZero] = 0;

            int HalfGridXIndexLast = FlattenIndex(FIntVector(GridCellsAmount.X, j,
k), FIntVector(GridCellsAmount.X + 1, GridCellsAmount.Y, GridCellsAmount.Z));
            HalfGridVelocitiesX[HalfGridXIndexLast] = 0;
        }

        for (size_t i = 0; i < GridCellsAmount.X; i++)
        {
            int HalfGridZIndexZero = FlattenIndex(FIntVector(i, j, 0),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y, GridCellsAmount.Z + 1));
            HalfGridVelocitiesZ[HalfGridZIndexZero] = 0;

            int HalfGridZIndexLast = FlattenIndex(FIntVector(i, j,
GridCellsAmount.Z), FIntVector(GridCellsAmount.X, GridCellsAmount.Y, GridCellsAmount.Z
+ 1));
            HalfGridVelocitiesZ[HalfGridZIndexLast] = 0;
        }
    }

    for (size_t i = 0; i < GridCellsAmount.X; i++)
    {
        for (size_t k = 0; k < GridCellsAmount.Z; k++)
        {

```

```

        int HalfGridYIndexZero = FlattenIndex(FIntVector(i, 0, k),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1, GridCellsAmount.Z));
        HalfGridVelocitiesY[HalfGridYIndexZero] = 0;

        int HalfGridYIndexLast = FlattenIndex(FIntVector(i, GridCellsAmount.Y,
k), FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1, GridCellsAmount.Z));
        HalfGridVelocitiesY[HalfGridYIndexLast] = 0;
    }
}
}

```

PRAGMA_DISABLE_OPTIMIZATION

```

void AFluid::PressureProjection(float DeltaTime)
{
    CalculateDivergence();
    Residuals = GridRhs;
    CalculatePressure_Conjugate(DeltaTime);
    CorrectVelocitiesByPressure(DeltaTime);
}

```

```

void AFluid::CalculateDivergence()
{
    for (size_t x = 0; x < GridCellsAmount.X; x++)
    {
        for (size_t y = 0; y < GridCellsAmount.Y; y++)
        {
            for (size_t z = 0; z < GridCellsAmount.Z; z++)
            {
                int GridIndex = FlattenIndex(FIntVector(x, y, z),
FIntVector(GridCellsAmount));

                if (GridDensity[GridIndex] > 1.e-6)
                {
                    int HalfGridXIndex = FlattenIndex(FIntVector(x, y, z),
FIntVector(GridCellsAmount.X + 1, GridCellsAmount.Y, GridCellsAmount.Z));
                    int HalfGridYIndex = FlattenIndex(FIntVector(x, y, z),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1, GridCellsAmount.Z));
                    int HalfGridZIndex = FlattenIndex(FIntVector(x, y, z),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y, GridCellsAmount.Z + 1));

                    int HalfGridXIndex1 = FlattenIndex(FIntVector(x + 1, y, z),
FIntVector(GridCellsAmount.X + 1, GridCellsAmount.Y, GridCellsAmount.Z));
                    int HalfGridYIndex1 = FlattenIndex(FIntVector(x, y + 1, z),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1, GridCellsAmount.Z));
                    int HalfGridZIndex1 = FlattenIndex(FIntVector(x, y, z + 1),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y, GridCellsAmount.Z + 1));

                    GridRhs[GridIndex] -=
HalfGridVelocitiesX[HalfGridXIndex1];
                    GridRhs[GridIndex] -=
HalfGridVelocitiesY[HalfGridYIndex1];

```

```

        GridRhs[GridIndex]           -=
HalfGridVelocitiesZ[HalfGridZIndex1];

        if (x != 0)
        {
            GridRhs[GridIndex]       +=
HalfGridVelocitiesX[HalfGridXIndex];
        }
        if (y != 0)
        {
            GridRhs[GridIndex]       +=
HalfGridVelocitiesY[HalfGridYIndex];
        }
        if (z != 0)
        {
            GridRhs[GridIndex]       +=
HalfGridVelocitiesZ[HalfGridZIndex];
        }

        GridRhs[GridIndex] = GridRhs[GridIndex] / CellSize;
    }
}

//UE_LOG(LogTemp, Log, TEXT("Max divergence: %f"),
FMath::Max(GridDivergence));
//UE_LOG(LogTemp, Log, TEXT("Min divergence: %f"), FMath::Min(GridDivergence));
}

void AFluid::CalculatePressure_Jacobi(float DeltaTime)
{
    const int IterationNum = 100;

    TArray<float>* CurrentPressureArray = &GridPressure_0;
    TArray<float>* PreviousPressureArray = &GridPressure_1;

    const float Scale = -CellSize * CellSize/DeltaTime;

    for (size_t iteration = 0; iteration < IterationNum; iteration++)
    {
        for (size_t i = 0; i < GridCellsAmount.X; i++)
        {
            for (size_t j = 0; j < GridCellsAmount.Y; j++)
            {
                for (size_t k = 0; k < GridCellsAmount.Z; k++)
                {
                    int32 FlatIndex = FlattenIndex(FIntVector(i, j, k),
GridCellsAmount);

                    (*CurrentPressureArray)[FlatIndex] =
(GetNeighboursSum(i, j, k, PreviousPressureArray, EOnBoundaryValue::Neighbour) -
GridRhs[FlatIndex] * Scale * GridDensity[FlatIndex] * PressureCalcConst) / 6;
                }
            }
        }
    }
}

```

```

    }
}

if (CurrentPressureArray == &GridPressure_0)
{
    CurrentPressureArray = &GridPressure_1;
    PreviousPressureArray = &GridPressure_0;
}
else
{
    CurrentPressureArray = &GridPressure_0;
    PreviousPressureArray = &GridPressure_1;
}
}

GridPressure_0 = *CurrentPressureArray;
//UE_LOG(LogTemp, Log, TEXT("Max Pressure: %f"), FMath::Max(GridPressure_0));
//UE_LOG(LogTemp, Log, TEXT("Min Pressure: %f"), FMath::Min(GridPressure_0));
}

void AFluid::CorrectVelocitiesByPressure(float DeltaTime)
{
    // unext = u - (deltat/rho)(pnext - p) / deltax.
    const float Scale = DeltaTime / CellSize;
    //ParallelFor(GridCellsAmount.X, [&](int32 i)
        for (size_t i = 0; i < GridCellsAmount.X; i++)
        {
            //ParallelFor(GridCellsAmount.Y, [&](int32 j)
                for (size_t j = 0; j < GridCellsAmount.Y; j++)
                {
                    for (size_t k = 0; k < GridCellsAmount.Z; k++)
                    {
                        int32 FlatIndex = FlattenIndex(FIntVector(i, j, k),
GridCellsAmount);

                        float Density = GridDensity[FlatIndex];
                        //Density = (Density > 0) ? Density : 0.005;
                        if (Density > VerySmallNumber)
                        {
                            int32 NeighbourCount =
CountNeighbours(FIntVector(i, j, k), GridCellsAmount);
                            float ScaledPressure = GridPressure_0[FlatIndex] *
Scale / (Density * NeighbourCount);

                            int32 XVelocityIndex = FlattenIndex(FIntVector(i, j,
k), FIntVector(GridCellsAmount.X + 1, GridCellsAmount.Y, GridCellsAmount.Z));
                            int32 YVelocityIndex = (FlattenIndex(FIntVector(i, j,
k), FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1, GridCellsAmount.Z)));
                            int32 ZVelocityIndex = (FlattenIndex(FIntVector(i, j,
k), FIntVector(GridCellsAmount.X, GridCellsAmount.Y, GridCellsAmount.Z + 1)));

                            HalfGridVelocitiesXCorrectedP[XVelocityIndex] -=
ScaledPressure;

```



```

    }
    float beta = sigma / sigma_old;

    for (size_t x = 0; x < GridCellsAmount.X; x++)
    {
        for (size_t y = 0; y < GridCellsAmount.Y; y++)
        {
            for (size_t z = 0; z < GridCellsAmount.Z; z++)
            {
                int GridIndex = FlattenIndex(FIntVector(x, y, z),
FIntVector(GridCellsAmount));
                GridRhs[GridIndex] = Residuals[GridIndex] + beta *
Residuals[GridIndex];
            }
        }
    }
}

```

```

void AFluid::CalculateStepDirectionAndSize(float sigma)
{
    for (size_t x = 0; x < GridCellsAmount.X; x++)
    {
        for (size_t y = 0; y < GridCellsAmount.Y; y++)
        {
            for (size_t z = 0; z < GridCellsAmount.Z; z++)
            {
                int GridIndex = FlattenIndex(FIntVector(x, y, z),
FIntVector(GridCellsAmount));

                if (GridRhs[GridIndex] != 0)
                {
                    float GridRhsSum = 0.f;
                    int Neighbours = 0;
                    GetFluidNeighboursDirectionSum(GridRhsSum,
Neighbours, x, y, z);
                    StepDirection[GridIndex] = Neighbours *
GridRhs[GridIndex] - GridRhsSum;
                    StepSize[GridIndex] = sigma / GridRhs[GridIndex] *
StepDirection[GridIndex];
                }
            }
        }
    }
}

```

```

void AFluid::GetFluidNeighboursDirectionSum(float& Sum, int& NotSolidNeighboursAmount,
int X, int Y, int Z)
{
    if (X > 0)
    {
        NotSolidNeighboursAmount++;
    }
}

```

```

        int Index = FlattenIndex(FIntVector(X - 1, Y, Z), GridCellsAmount);
        if (GridDensity[Index] > VerySmallNumber)
        {
            Sum += GridRhs[Index];
        }
    }
    if (X < GridCellsAmount.X - 1)
    {
        NotSolidNeighboursAmount++;
        int Index = FlattenIndex(FIntVector(X + 1, Y, Z), GridCellsAmount);
        if (GridDensity[Index] > VerySmallNumber)
        {
            Sum += GridRhs[Index];
        }
    }
    if (Y > 0)
    {
        NotSolidNeighboursAmount++;
        int Index = FlattenIndex(FIntVector(X, Y - 1, Z), GridCellsAmount);
        if (GridDensity[Index] > VerySmallNumber)
        {
            Sum += GridRhs[Index];
        }
    }
    if (Y < GridCellsAmount.Y - 1)
    {
        NotSolidNeighboursAmount++;
        int Index = FlattenIndex(FIntVector(X, Y + 1, Z), GridCellsAmount);
        if (GridDensity[Index] > VerySmallNumber)
        {
            Sum += GridRhs[Index];
        }
    }
    if (Z > 0)
    {
        NotSolidNeighboursAmount++;
        int Index = FlattenIndex(FIntVector(X, Y, Z - 1), GridCellsAmount);
        if (GridDensity[Index] > VerySmallNumber)
        {
            Sum += GridRhs[Index];
        }
    }
    if (Z < GridCellsAmount.Z - 1)
    {
        NotSolidNeighboursAmount++;
        int Index = FlattenIndex(FIntVector(X, Y, Z + 1), GridCellsAmount);
        if (GridDensity[Index] > VerySmallNumber)
        {
            Sum += GridRhs[Index];
        }
    }
}

```



```

        });
    });
}

GridDensityPotential_0 = *CurrentDensityPotentialArray;

//UE_LOG(LogTemp, Log, TEXT("Max DensityPotential: %f"),
FMath::Max(GridDensityPotential_0));
//UE_LOG(LogTemp, Log, TEXT("Min DensityPotential: %f"),
FMath::Min(GridDensityPotential_0));
}

void AFluid::CorrectParticlesPositionsAccordingToDensity()
{
    const FVector UpperCorner = GridPositions.Last();
    const FVector LowerCorner = GridPositions[0];

    ParallelFor(ParticlePositions.Num(), [&](int32 Index)
    {
        FVector ParticleLocation = ParticlePositions[Index];

        // Finding cell in which particle is now
        FVector CellIndex = FVector(FMath::FloorToInt((ParticleLocation.X
- LowerCorner.X) / CellSize),
FMath::FloorToInt((ParticleLocation.Y - LowerCorner.Y) /
CellSize),
FMath::FloorToInt((ParticleLocation.Z - LowerCorner.Z) /
CellSize));

        // Left corner of cell where particle located
        FVector GridLocation = GridPositions[FlattenIndex(CellIndex,
GridPositionsAmount)];

        FVector ParticleVelocity = ParticleVelocities[Index];

        // Weight coefficient = (Xp - Xgrid)/CellCize
        // Weights to Lower Grid point
        FVector WeightsToLowGrid = FVector(1) - ((ParticleLocation -
GridLocation) / CellSize);

        FVector DistanceToCellLeftCorner = (ParticleLocation - GridLocation) /
CellSize;

        // If close to smaller point then -1
        FVector HalfGridIndex = FVector(
            CellIndex.X - (DistanceToCellLeftCorner.X < 0.5 ? 1 : 0),
            CellIndex.Y - (DistanceToCellLeftCorner.Y < 0.5 ? 1 : 0),
            CellIndex.Z - (DistanceToCellLeftCorner.Z < 0.5 ? 1 : 0)
        );

        FVector HalfGridLeftCornerWeights = FVector(

```

```

        DistanceToCellLeftCorner.X > 0.5 ? (1.f -
(DistanceToCellLeftCorner.X)) : DistanceToCellLeftCorner.X,
        DistanceToCellLeftCorner.Y > 0.5 ? (1.f -
(DistanceToCellLeftCorner.Y)) : DistanceToCellLeftCorner.Y,
        DistanceToCellLeftCorner.Z > 0.5 ? (1.f -
(DistanceToCellLeftCorner.Z)) : DistanceToCellLeftCorner.Z
    );

    FVector CurrentWeights = HalfGridLeftCornerWeights;
    for (size_t x = (HalfGridIndex.X > -1 ? 0 : 1); x < ((HalfGridIndex.X ==
GridCellsAmount.X - 1) ? 1 : 2); x++)
    {
        CurrentWeights.Y = HalfGridIndex.Y > -1 ?
HalfGridLeftCornerWeights.Y : 1 - HalfGridLeftCornerWeights.Y;
        for (size_t y = (HalfGridIndex.Y > -1 ? 0 : 1); y < ((HalfGridIndex.Y
== GridCellsAmount.Y - 1) ? 1 : 2); y++)
        {
            CurrentWeights.Z = HalfGridIndex.Z > -1 ?
HalfGridLeftCornerWeights.Z : 1 - HalfGridLeftCornerWeights.Z;
            for (size_t z = (HalfGridIndex.Z > -1 ? 0 : 1); z <
((HalfGridIndex.Z == GridCellsAmount.Z - 1) ? 1 : 2); z++)
            {
                int32 DensityIndex =
FlattenIndex(FIntVector(HalfGridIndex.X + x, HalfGridIndex.Y + y, HalfGridIndex.Z + z),
FIntVector(GridCellsAmount.X, GridCellsAmount.Y, GridCellsAmount.Z));
                FVector NewPos = ParticleLocation -
CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z *
GridDensityPotential_0[DensityIndex];
                FVector Vel(0);

                ForceParticlePositionAndVelocityInBoundaries(UpperCorner, LowerCorner, NewPos, Vel,
ParticleLocation);

                ParticlePositions[Index] = NewPos;
                CurrentWeights.Z = 1 - CurrentWeights.Z;
            }
            CurrentWeights.Y = 1 - CurrentWeights.Y;
        }
        CurrentWeights.X = 1 - CurrentWeights.X;
    }
});
}

// Particles velocities are collected from grid
// Physically: advection term
void AFluid::ParticlesToGrid(bool CollectSpeed)
{
    const FVector LowestGridCorner = GridPositions[0];

    FCriticalSection Mutex;
    ParallelFor(ParticleVelocities.Num(), [&](int32 Index)
    {
        FVector ParticleLocation = ParticlePositions[Index];

```

```

// Finding cell in which particle is now
FIntVector CellIndex = FIntVector(FMath::FloorToInt((ParticleLocation.X -
LowestGridCorner.X)/ CellSize),

FMath::FloorToInt((ParticleLocation.Y - LowestGridCorner.Y) / CellSize),

FMath::FloorToInt((ParticleLocation.Z - LowestGridCorner.Z) / CellSize));

// Left corner of cell where particle located
FVector      GridLocation      =      GridPositions[FlattenIndex(CellIndex,
GridPositionsAmount)];

FVector ParticleVelocity = ParticleVelocities[Index];

// Weights to Lower Grid point
FVector GridCellLeftCornerWeights = FVector(1) - ((ParticleLocation -
GridLocation) / CellSize);

FVector DistanceToCellLeftCorner = (ParticleLocation - GridLocation) / CellSize;

// If close to smaller point then -1
FIntVector HalfGridIndex = FIntVector(
    CellIndex.X - (DistanceToCellLeftCorner.X < 0.5 ? 1 : 0),
    CellIndex.Y - (DistanceToCellLeftCorner.Y < 0.5 ? 1 : 0),
    CellIndex.Z - (DistanceToCellLeftCorner.Z < 0.5 ? 1 : 0)
);

// If point is close to left low corner then half grid index
// DistanceToCellLeftCorner.X < 0.5 ? (1.f - (DistanceToCellLeftCorner.X + 0.5f)) :
(1.f - (DistanceToCellLeftCorner.X - 0.5f)),
FVector HalfGridCellLeftCornerWeights = FVector(
    DistanceToCellLeftCorner.X < 0.5 ? (0.5f - (DistanceToCellLeftCorner.X)) :
(1.5f - (DistanceToCellLeftCorner.X)),
    DistanceToCellLeftCorner.Y < 0.5 ? (0.5f - (DistanceToCellLeftCorner.Y)) :
(1.5f - (DistanceToCellLeftCorner.Y)),
    DistanceToCellLeftCorner.Z < 0.5 ? (0.5f - (DistanceToCellLeftCorner.Z)) :
(1.5f - (DistanceToCellLeftCorner.Z))
);

FScopeLock Lock(&Mutex);
// Density collection
CollectDensity(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex);

//if (CollectSpeed)
{
    // X speed collection
    if (ParticleVelocity.X > MinSpeedToCollect)
    {
        CollectXVelocities(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex, ParticleVelocity.X);
    }
}

```

```

    }
    // Y speed collection
    if (ParticleVelocity.Y > MinSpeedToCollect)
    {
        CollectYVelocities(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex, ParticleVelocity.Y);
    }
    // Z speed collection
    if (ParticleVelocity.Z > MinSpeedToCollect)
    {
        CollectZVelocities(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex, ParticleVelocity.Z);
    }
    }
    Lock.Unlock();
});

NormalizeHalfGridVelocities();
}

```

```

void AFluid::NormalizeHalfGridVelocities()
{
    // Normalization
    ParallelFor(HalfGridVelocitiesX.Num(), [&](int32 Index)
    {
        if (HalfGridWeightsX[Index] > 0)
        {
            HalfGridVelocitiesX[Index] = HalfGridVelocitiesX[Index] /
HalfGridWeightsX[Index];
        }
    });
    ParallelFor(HalfGridVelocitiesY.Num(), [&](int32 Index)
    {
        if (HalfGridWeightsY[Index] > 0)
        {
            HalfGridVelocitiesY[Index] = HalfGridVelocitiesY[Index] /
HalfGridWeightsY[Index];
        }
    });
    ParallelFor(HalfGridVelocitiesZ.Num(), [&](int32 Index)
    {
        if (HalfGridWeightsZ[Index] > 0)
        {
            HalfGridVelocitiesZ[Index] = HalfGridVelocitiesZ[Index] /
HalfGridWeightsZ[Index];
        }
    });
}

```

```

void AFluid::CollectDensity(const FIntVector& CellIndex, const FVector& WeightsToLowGrid,
const FVector& HalfGridCellLeftCornerWeights, const FIntVector& HalfGridIndex)
{

```

```

    FVector CurrentWeights = HalfGridCellLeftCornerWeights;
    for (size_t x = (HalfGridIndex.X > -1 ? 0 : 1); x < ((HalfGridIndex.X ==
GridCellsAmount.X - 1) ? 1 : 2); x++)
    {
        CurrentWeights.Y = HalfGridIndex.Y > -1 ? HalfGridCellLeftCornerWeights.Y : 1
- HalfGridCellLeftCornerWeights.Y;
        for (size_t y = (HalfGridIndex.Y > -1 ? 0 : 1); y < ((HalfGridIndex.Y ==
GridCellsAmount.Y - 1) ? 1 : 2); y++)
        {
            CurrentWeights.Z = HalfGridIndex.Z > -1 ?
HalfGridCellLeftCornerWeights.Z : 1 - HalfGridCellLeftCornerWeights.Z;
            for (size_t z = (HalfGridIndex.Z > -1 ? 0 : 1); z < ((HalfGridIndex.Z ==
GridCellsAmount.Z - 1) ? 1 : 2); z++)
            {
                int32 DensityIndex = FlattenIndex(FIntVector(HalfGridIndex.X + x,
HalfGridIndex.Y + y, HalfGridIndex.Z + z), FIntVector(GridCellsAmount.X, GridCellsAmount.Y,
GridCellsAmount.Z));
                ParticlesInCell[DensityIndex]++;
                GridDensity[DensityIndex] += ParticleMassInGramms *
CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z;
                CurrentWeights.Z = 1 - CurrentWeights.Z;
            }
            CurrentWeights.Y = 1 - CurrentWeights.Y;
        }
        CurrentWeights.X = 1 - CurrentWeights.X;
    }
}

```

```

void AFluid::CollectXVelocities(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const float ParticleVelocityX)
{
    FVector CurrentWeights = FVector(GridCellLeftCornerWeights);
    for (size_t x = 0; x < 2; x++)
    {
        CurrentWeights.Y = HalfGridIndex.Y > -1 ? HalfGridCellLeftCornerWeights.Y : 1
- HalfGridCellLeftCornerWeights.Y;
        for (size_t y = (HalfGridIndex.Y > -1 ? 0 : 1); y < ((HalfGridIndex.Y ==
GridCellsAmount.Y - 1) ? 1 : 2); y++)
        {
            CurrentWeights.Z = HalfGridIndex.Z > -1 ?
HalfGridCellLeftCornerWeights.Z : 1 - HalfGridCellLeftCornerWeights.Z;
            for (size_t z = (HalfGridIndex.Z > -1 ? 0 : 1); z < ((HalfGridIndex.Z ==
GridCellsAmount.Z - 1) ? 1 : 2); z++)
            {
                int32 XVelocityIndex = FlattenIndex(FIntVector(CellIndex.X + x,
HalfGridIndex.Y + y, HalfGridIndex.Z + z), FIntVector(GridCellsAmount.X + 1,
GridCellsAmount.Y, GridCellsAmount.Z));
                HalfGridVelocitiesX[XVelocityIndex] += ParticleVelocityX *
CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z;
                HalfGridWeightsX[XVelocityIndex] += CurrentWeights.X *
CurrentWeights.Y * CurrentWeights.Z;
            }
        }
    }
}

```

```

        CurrentWeights.Z = 1 - CurrentWeights.Z;
    }
    CurrentWeights.Y = 1 - CurrentWeights.Y;
}
CurrentWeights.X = 1 - CurrentWeights.X;
}
}

void AFluid::CollectYVelocities(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const float ParticleVelocityY)
{
    FVector CurrentWeights = FVector(GridCellLeftCornerWeights);
    for (size_t y = 0; y < 2; y++)
    {
        CurrentWeights.X = HalfGridIndex.X > -1 ? HalfGridCellLeftCornerWeights.X : 1
- HalfGridCellLeftCornerWeights.X;
        for (size_t x = (HalfGridIndex.X > -1 ? 0 : 1); x < ((HalfGridIndex.X ==
GridCellsAmount.X - 1) ? 1 : 2); x++)
        {
            CurrentWeights.Z = HalfGridIndex.Z > -1 ?
HalfGridCellLeftCornerWeights.Z : 1 - HalfGridCellLeftCornerWeights.Z;
            for (size_t z = (HalfGridIndex.Z > -1 ? 0 : 1); z < ((HalfGridIndex.Z ==
GridCellsAmount.Z - 1) ? 1 : 2); z++)
            {
                int32 YVelocityIndex = FlattenIndex(
                    FIntVector(HalfGridIndex.X + x, CellIndex.Y + y,
HalfGridIndex.Z + z),
                    FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1,
GridCellsAmount.Z)
                );
                HalfGridVelocitiesY[YVelocityIndex] += ParticleVelocityY *
CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z;
                HalfGridWeightsY[YVelocityIndex] += CurrentWeights.X *
CurrentWeights.Y * CurrentWeights.Z;
                CurrentWeights.Z = 1 - CurrentWeights.Z;
            }
            CurrentWeights.X = 1 - CurrentWeights.X;
        }
        CurrentWeights.Y = 1 - CurrentWeights.Y;
    }
}

void AFluid::CollectZVelocities(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const float ParticleVelocityZ)
{
    FVector CurrentWeights = FVector(GridCellLeftCornerWeights);
    for (size_t z = 0; z < 2; z++)
    {

```

```

        CurrentWeights.Y = HalfGridIndex.Y > -1 ? HalfGridCellLeftCornerWeights.Y : 1
- HalfGridCellLeftCornerWeights.Y;
        for (size_t y = (HalfGridIndex.Y > -1 ? 0 : 1); y < ((HalfGridIndex.Y ==
GridCellsAmount.Y - 1) ? 1 : 2); y++)
        {
            CurrentWeights.X = HalfGridIndex.X > -1 ?
HalfGridCellLeftCornerWeights.X : 1 - HalfGridCellLeftCornerWeights.X;
            for (size_t x = (HalfGridIndex.X > -1 ? 0 : 1); x < ((HalfGridIndex.X ==
GridCellsAmount.X - 1) ? 1 : 2); x++)
            {
                int32 ZVelocityIndex = FlattenIndex(
                    FIntVector(HalfGridIndex.X + x, HalfGridIndex.Y + y,
CellIndex.Z + z),
                    FIntVector(GridCellsAmount.X, GridCellsAmount.Y,
GridCellsAmount.Z + 1)
                );
                HalfGridVelocitiesZ[ZVelocityIndex] += ParticleVelocityZ *
CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z;
                HalfGridWeightsZ[ZVelocityIndex] += CurrentWeights.X *
CurrentWeights.Y * CurrentWeights.Z;
                CurrentWeights.X = 1 - CurrentWeights.X;
            }
            CurrentWeights.Y = 1 - CurrentWeights.Y;
        }
        CurrentWeights.Z = 1 - CurrentWeights.Z;
    }
}

```

```

void AFluid::GridToParticles(float FlipRatio)
{
    const FVector LowestGridCorner = GridPositions[0];
    ParallelFor(ParticleVelocities.Num(), [&](int32 Index)
    {
        FVector InitialParticleVelocity = ParticleVelocities[Index];
        ParticleVelocities[Index] = FVector(0);

        FVector ParticleLocation = ParticlePositions[Index];
        FIntVector CellIndex = FIntVector(FMath::FloorToInt((ParticleLocation.X
- LowestGridCorner.X) / CellSize),
            FMath::FloorToInt((ParticleLocation.Y - LowestGridCorner.Y) /
CellSize),
            FMath::FloorToInt((ParticleLocation.Z - LowestGridCorner.Z) /
CellSize));

        // Left corner of cell where particle located
        FVector GridLocation = GridPositions[FlattenIndex(CellIndex,
GridPositionsAmount)];

        FVector GridCellLeftCornerWeights = FVector(1) - ((ParticleLocation -
GridLocation) / CellSize);
        FVector DistanceToCellLeftCorner = (ParticleLocation - GridLocation) /
CellSize;
    }
}

```

```

// If close to smaller point then -1
FIntVector HalfGridIndex = FIntVector(
    CellIndex.X - (DistanceToCellLeftCorner.X < 0.5 ? 1 : 0),
    CellIndex.Y - (DistanceToCellLeftCorner.Y < 0.5 ? 1 : 0),
    CellIndex.Z - (DistanceToCellLeftCorner.Z < 0.5 ? 1 : 0)
);

FVector HalfGridCellLeftCornerWeights = FVector(
    DistanceToCellLeftCorner.X < 0.5 ? (0.5f -
(DistanceToCellLeftCorner.X)) : (1.5f - (DistanceToCellLeftCorner.X)),
    DistanceToCellLeftCorner.Y < 0.5 ? (0.5f -
(DistanceToCellLeftCorner.Y)) : (1.5f - (DistanceToCellLeftCorner.Y)),
    DistanceToCellLeftCorner.Z < 0.5 ? (0.5f -
(DistanceToCellLeftCorner.Z)) : (1.5f - (DistanceToCellLeftCorner.Z))
);

HalfgridXVelocitiesToParticles(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex, Index, InitialParticleVelocity.X, FlipRatio);
HalfgridYVelocitiesToParticles(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex, Index, InitialParticleVelocity.Y, FlipRatio);
HalfgridZVelocitiesToParticles(CellIndex, GridCellLeftCornerWeights,
HalfGridCellLeftCornerWeights, HalfGridIndex, Index, InitialParticleVelocity.Z, FlipRatio);
});
}

```

```

void AFluid::HalfgridXVelocitiesToParticles(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const int32 ParticleIndex, const float ParticleVelocityX, float FlipRatio)
{
    FVector CurrentWeights = FVector(GridCellLeftCornerWeights);
    for (size_t x = 0; x < 2; x++)
    {
        CurrentWeights.Y = HalfGridIndex.Y > -1 ? HalfGridCellLeftCornerWeights.Y : 1
- HalfGridCellLeftCornerWeights.Y;
        for (size_t y = (HalfGridIndex.Y > -1 ? 0 : 1); y < ((HalfGridIndex.Y ==
GridCellsAmount.Y - 1) ? 1 : 2); y++)
        {
            CurrentWeights.Z = HalfGridIndex.Z > -1 ?
HalfGridCellLeftCornerWeights.Z : 1 - HalfGridCellLeftCornerWeights.Z;
            for (size_t z = (HalfGridIndex.Z > -1 ? 0 : 1); z < ((HalfGridIndex.Z ==
GridCellsAmount.Z - 1) ? 1 : 2); z++)
            {
                int32 XVelocityIndex = FlattenIndex(
                    FIntVector(CellIndex.X + x, HalfGridIndex.Y + y,
HalfGridIndex.Z + z),
                    FIntVector(GridCellsAmount.X + 1, GridCellsAmount.Y,
GridCellsAmount.Z));
                // up = f(up - uold) + anew
                ParticleVelocities[ParticleIndex].X += (FlipRatio *
(ParticleVelocityX - HalfGridVelocitiesX[XVelocityIndex])) +

```

```

HalfGridVelocitiesXCorrectedP[XVelocityIndex]) * CurrentWeights.X * CurrentWeights.Y *
CurrentWeights.Z;
        CurrentWeights.Z = 1 - CurrentWeights.Z;
    }
    CurrentWeights.Y = 1 - CurrentWeights.Y;
}
CurrentWeights.X = 1 - CurrentWeights.X;
}
}

```

```

void AFluid::HalfgridYVelocitiesToParticles(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const int32 ParticleIndex, const float ParticleVelocityY, float FlipRatio)

```

```

{
    FVector CurrentWeights = FVector(GridCellLeftCornerWeights);
    for (size_t y = 0; y < 2; y++)
    {
        CurrentWeights.X = HalfGridIndex.X > -1 ? HalfGridCellLeftCornerWeights.X : 1
- HalfGridCellLeftCornerWeights.X;
        for (size_t x = (HalfGridIndex.X > -1 ? 0 : 1); x < ((HalfGridIndex.X ==
GridCellsAmount.X - 1) ? 1 : 2); x++)
        {
            CurrentWeights.Z = HalfGridIndex.Z > -1 ?
HalfGridCellLeftCornerWeights.Z : 1 - HalfGridCellLeftCornerWeights.Z;
            for (size_t z = (HalfGridIndex.Z > -1 ? 0 : 1); z < ((HalfGridIndex.Z ==
GridCellsAmount.Z - 1) ? 1 : 2); z++)
            {
                int32 YVelocityIndex = FlattenIndex(
                    FIntVector(HalfGridIndex.X + x, CellIndex.Y + y,
HalfGridIndex.Z + z),
                    FIntVector(GridCellsAmount.X, GridCellsAmount.Y + 1,
GridCellsAmount.Z)
                );
                ParticleVelocities[ParticleIndex].Y += (FlipRatio *
(ParticleVelocityY - HalfGridVelocitiesY[YVelocityIndex]) +
HalfGridVelocitiesYCorrectedP[YVelocityIndex])
                * CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z;
                CurrentWeights.Z = 1 - CurrentWeights.Z;
            }
            CurrentWeights.X = 1 - CurrentWeights.X;
        }
        CurrentWeights.Y = 1 - CurrentWeights.Y;
    }
}

```

```

void AFluid::HalfgridZVelocitiesToParticles(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const int32 ParticleIndex, const float ParticleVelocityZ, float FlipRatio)

```

```

{
    FVector CurrentWeights = FVector(GridCellLeftCornerWeights);
    for (size_t z = 0; z < 2; z++)
    {

```

```

        CurrentWeights.Y = HalfGridIndex.Y > -1 ? HalfGridCellLeftCornerWeights.Y : 1
- HalfGridCellLeftCornerWeights.Y;
        for (size_t y = (HalfGridIndex.Y > -1 ? 0 : 1); y < ((HalfGridIndex.Y ==
GridCellsAmount.Y - 1) ? 1 : 2); y++)
        {
            CurrentWeights.X = HalfGridIndex.X > -1 ?
HalfGridCellLeftCornerWeights.X : 1 - HalfGridCellLeftCornerWeights.X;
            for (size_t x = (HalfGridIndex.X > -1 ? 0 : 1); x < ((HalfGridIndex.X ==
GridCellsAmount.X - 1) ? 1 : 2); x++)
            {
                int32 ZVelocityIndex = FlattenIndex(
                    FIntVector(HalfGridIndex.X + x, HalfGridIndex.Y + y,
CellIndex.Z + z),
                    FIntVector(GridCellsAmount.X, GridCellsAmount.Y,
GridCellsAmount.Z + 1)
                );
                ParticleVelocities[ParticleIndex].Z += (FlipRatio *
(ParticleVelocityZ - HalfGridVelocitiesZ[ZVelocityIndex]) +
HalfGridVelocitiesZCorrectedP[ZVelocityIndex]) \
                    * CurrentWeights.X * CurrentWeights.Y * CurrentWeights.Z;
                CurrentWeights.X = 1 - CurrentWeights.X;
            }
            CurrentWeights.Y = 1 - CurrentWeights.Y;
        }
        CurrentWeights.Z = 1 - CurrentWeights.Z;
    }
}

```

```

===== FluidSim\Private\FluidSim.cpp =====
// Copyright Epic Games, Inc. All Rights Reserved.

```

```

#include "FluidSim.h"

```

```

#define LOCTEXT_NAMESPACE "FFluidSimModule"

```

```

void FFluidSimModule::StartupModule()

```

```

{
    // This code will execute after your module is loaded into memory; the exact timing is
specified in the .uplugin file per-module
}

```

```

void FFluidSimModule::ShutdownModule()

```

```

{
    // This function may be called during shutdown to clean up your module. For modules that
support dynamic reloading,
    // we call this function before unloading the module.
}

```

```

#undef LOCTEXT_NAMESPACE

```

```

IMPLEMENT_MODULE(FFluidSimModule, FluidSim)

```

```

===== FluidSim\Public\Fluid.h =====

```

// Fill out your copyright notice in the Description page of Project Settings.

```
#pragma once
```

```
#include "CoreMinimal.h"  
#include "GameFramework/Actor.h"  
#include "FluidSimConstants.h"  
#include "Fluid.generated.h"
```

```
UCLASS()
```

```
class FLUIDSIM_API AFluid : public AActor  
{
```

```
    GENERATED_BODY()
```

```
public:
```

```
    // Sets default values for this actor's properties  
    AFluid();
```

```
    FORCEINLINE float GetTotalAmountOfCellsPositions() const;
```

```
protected:
```

```
    virtual void BeginPlay() override;
```

```
    // GRID
```

```
    // Grid parameters
```

```
    UPROPERTY(EditAnywhere)
```

```
    FVector GridCellsAmount = FVector(2);
```

```
    // Grid corners positions stored, so amount of positions will be CellsAmount + 1
```

```
    // Sets automatically in BeginPlay
```

```
    FVector GridPositionsAmount;
```

```
    UPROPERTY(EditAnywhere)
```

```
    float CellSize = 1;
```

```
    // Grid Variables
```

```
    /*
```

```
    *
```

```
        +-----+
```

```
        / .Vy / |
```

```
+-----+ |
```

```
|   |.
```

```
| . |Vx+1
```

```
| Vz | /
```

```
+-----+
```

```
    *
```

```
        Each storing corresponding velocity component
```

```
    */
```

```
    // Half Grid variables
```

```
    TArray<float> HalfGridVelocitiesX;
```

```

TArray<float> HalfGridVelocitiesY;
TArray<float> HalfGridVelocitiesZ;

TArray<float> HalfGridVelocitiesXCorrectedP;
TArray<float> HalfGridVelocitiesYCorrectedP;
TArray<float> HalfGridVelocitiesZCorrectedP;

TArray<float> HalfGridWeightsX;
TArray<float> HalfGridWeightsY;
TArray<float> HalfGridWeightsZ;

// Grid Corners variables
TArray<FVector> GridPositions;

// Grid per cell variables
// Previous grid pressure needs to be stored, so framebuffer principle used
TArray<float> GridPressure_0;
TArray<float> GridPressure_1;

TArray<float> GridPressure;

TArray<float> GridDensity;
TArray<int32> ParticlesInCell;
TArray<float> GridDensityPotential_0;
TArray<float> GridDensityPotential_1;

UPROPERTY(EditAnywhere)
int ParticlesInCellMax = 0;

// RHS = - GridDivergence
TArray<float> GridRhs;
TArray<float> Residuals;

TArray<float> StepDirection;
TArray<float> StepSize;

TArray<float> Preconditioner;

TArray<float> Adiaq;
TArray<float> Ax;
TArray<float> Ay;
TArray<float> Az;

void DrawGrid();
void DrawGridPoints();
void DrawGridCorners();

// PARTICLES
//
// Particle Params
UPROPERTY(EditAnywhere)
int ParticlesAmount;

```

```

UPROPERTY(EditAnywhere)
float ParticleMassInGramms = 1;

// Particle Vars
TArray<FVector> ParticlePositions;
TArray<FVector> ParticleVelocities;

// Particles Visualisatoion
UPROPERTY()
class UNiagaraSystem* ParticlesVisualization;

UPROPERTY()
class UNiagaraComponent* ParticlesVisualizationComponent;

void SpawnParticles();
void PushParticles(float DeltaTime);

void ApplyExternalForces(float DeltaTime);
void ZeroGrid();
void ClampBoundaryVelocities();

void PressureProjection(float DeltaTime);
void CalculateDivergence();
void CalculatePressure_Conjugate(float DeltaTime);
void CalculateStepDirectionAndSize(float sigma);
void GetFluidNeighboursDirectionSum(float& Pressure, int&
NotSolidNeighboursAmount, int X, int Y, int Z);
void CalculatePressure_Jacobi(float DeltaTime);
void CorrectVelocitiesByPressure(float DeltaTime);

bool DensityProjection(float DeltaTime);
bool IsDensityProjectionRequired(float alpha, float beta);
void CalculateDensityCorrectionPotential();
void CorrectParticlesPositionsAccordingToDensity();

void ParticlesToGrid(bool CollectSpeed);
void NormalizeHalfGridVelocities();
void CollectDensity(const FIntVector& CellIndex, const FVector& WeightsToLowGrid,
const FVector& HalfGridLeftCornerWeights, const FIntVector& HalfGridIndex);
void CollectXVelocities(const FIntVector& CellIndex, const FVector&
WeightsToLowGrid, const FVector& HalfGridLeftCornerWeights, const FIntVector&
HalfGridIndex, const float ParticleVelocityX);
void CollectYVelocities(const FIntVector& CellIndex, const FVector&
WeightsToLowGrid, const FVector& HalfGridLeftCornerWeights, const FIntVector&
HalfGridIndex, const float ParticleVelocityY);
void CollectZVelocities(const FIntVector& CellIndex, const FVector&
WeightsToLowGrid, const FVector& HalfGridLeftCornerWeights, const FIntVector&
HalfGridIndex, const float ParticleVelocityZ);

void GridToParticles(float FlipRatio);

```

```

        void HalfgridXVelocitiesToParticles(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const int32 ParticleIndex, const float ParticleVelocityX, float FlipRatio);
        void HalfgridYVelocitiesToParticles(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const int32 ParticleIndex, const float ParticleVelocityY, float FlipRatio);
        void HalfgridZVelocitiesToParticles(const FIntVector& CellIndex, const FVector&
GridCellLeftCornerWeights, const FVector& HalfGridCellLeftCornerWeights, const FIntVector&
HalfGridIndex, const int32 ParticleIndex, const float ParticleVelocityZ, float FlipRatio);

```

```
public:
```

```

    // Called every frame
    virtual void Tick(float DeltaTime) override;
    float GetNeighboursSum(int X, int Y, int Z, const TArray<float>* PressureArray,
EOnBoundaryValue BoundaryVal) const;
};

```

```

.
===== FluidSim\Public\FluidSim.h =====
// Copyright Epic Games, Inc. All Rights Reserved.

```

```
#pragma once
```

```

#include "CoreMinimal.h"
#include "Modules/ModuleManager.h"

```

```

class FFluidSimModule : public IModuleInterface
{
public:

```

```

    /** IModuleInterface implementation */
    virtual void StartupModule() override;
    virtual void ShutdownModule() override;
};

```

```

.
===== FluidSim\Public\FluidSimConstants.h =====
#pragma once

```

```

// const FVector GravityVector = FVector(0, 0, -10);
// cm/s^2
const float GravityConst = -980.f;

```

```
const float PressureCalcConst = 1.f;
```

```
const float MinSpeedToCollect = 1e-6;
```

```
const float VerySmallNumber = 1e-5;
```

```

// grams / cm^3S
const float PerfectWaterDensity = 1.f;

```

```

// Trace Channels
#define ECC_Fluid ECC_GameTraceChannel6

```

```
UENUM(BlueprintType)
enum class EOnBoundaryValue : uint8
{
    Neighbour,
    Zero
};
```

Додаток 3
Копія презентації



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО”



ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ

СПОСІБ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ СИМУЛЯЦІЇ ПОВЕДІНКИ РІДИН В РЕАЛЬНОМУ ЧАСІ

Доповідач: Седухіна Аліна Дмитрівна

Науковий керівник: к.ф.-м.н., доц. Нещадим Олександр Михайлович

Київ – 2025

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ



Симуляція рідин є однією з найскладніших задач комп'ютерного моделювання, попри тривалу історію її розвитку. Вона широко застосовується у VFX, ігрових движках та інженерних обчисленнях. Повний опис динаміки рідин вимагає розв'язання рівнянь Нав'є–Стокса, що належать до нерозв'язаних проблем тисячоліття, тому практичні системи спираються на чисельні апроксимації як єдиний реалістичний підхід. Тому якісні апроксимації є надзвичайно важливими та наразі залишаються єдиним практичним засобом симуляції рідин.



ПОСТАНОВКА ЗАДАЧІ

Мета роботи: підвищити швидкодію симуляції рідин у реальному часі шляхом упровадження адаптивного порогу для шару корекції густини, із використанням мови програмування C++ та Unreal Engine API.

Об'єкт дослідження: процес генерації даних для симуляції поведінки рідин

Предмет дослідження: методи та способи генерації даних для симуляції поведінки рідин.

ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

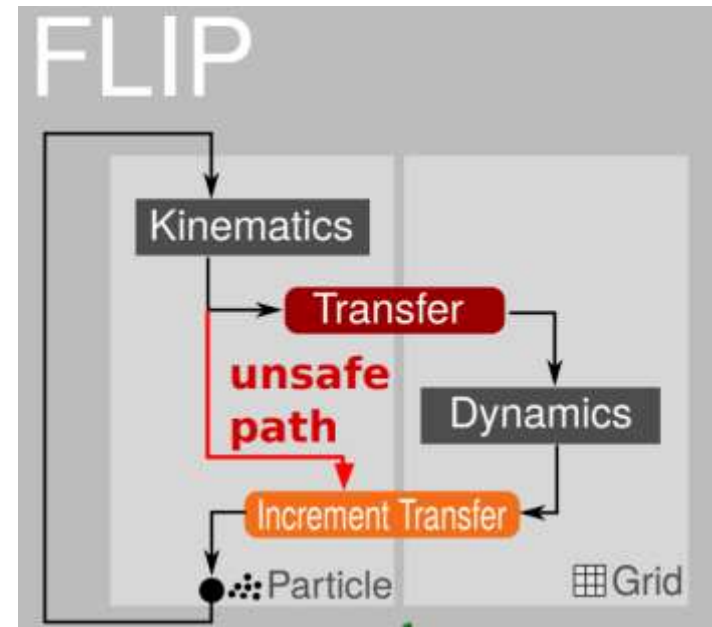
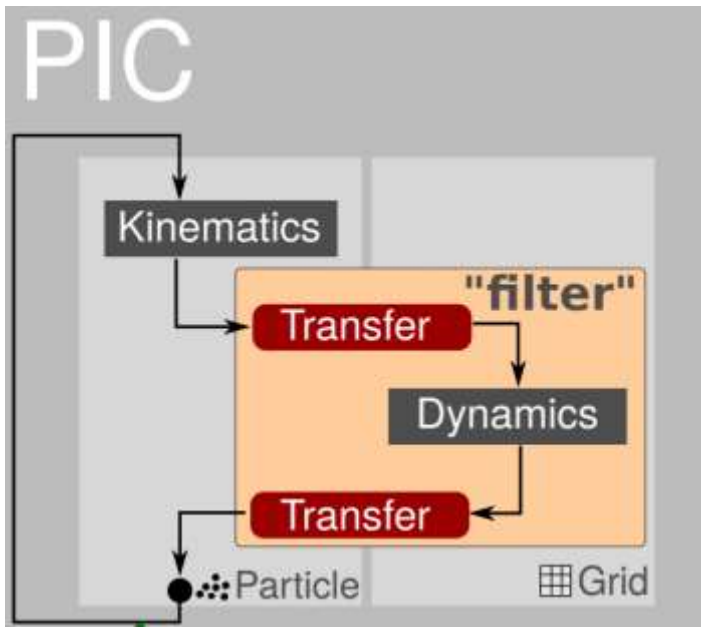


	Переваги	Недоліки
Методи на основі частинок	Простота реалізації, можливість працювати із складними формами	Є досить шумними та для реалістичної симуляції потребують великої кількості частинок, що призводить до високих вимог до обчислювальних ресурсів
Методи на основі сітки	Стабільні і досить точні, особливо на великих об'ємах	Через обрахунок тільки в конкретних клітинах має проблеми з малими деталями, можуть потребувати велику кількість ресурсів
Гібридні методи	Стабільні і відносно швидкі, тонке налаштування балансу швидкості і точності	Є складнішими в реалізацію через потребу в роботі одразу з сіткою, частинками та переходом між ними.
Методи з використанням штучного інтелекту	Дуже швидка кінцева модель	Обмеженість навчальними даними призводить до низької точності

Програмне забезпечення	Метод	Категорія методу	Призначення
OpenFOAM	Модифіковані кінцеві об'єми	На основі сітки	Інженерні розробки
ANSYS Fluent	Модифіковані кінцеві об'єми	На основі сітки	Інженерні розробки
STAR CCM+	Модифіковані кінцеві об'єми	На основі сітки	Дослідження астрофізики
CoolBM	Метод решіткових рівнянь Больцмана	На основі сітки	Дослідження мультифазних рідин
SPLisHSPlasH	Модифікований SPH	На основі частинок	Дослідження, графіка
SPH Fluid Simulator	SPH	На основі частинок	Навчання
SmartSim (розширення OpenFOAM)	Методи машинного навчання	ШІ	
Blender Mantaflow	FLIP/модифікований FLIP	Гібридні	Візуалізації
Houdini FLIP solver	FLIP/APIC	Гібридні	Візуальні ефекти



ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ





НЕДОЛІКИ ІСНУЮЧИХ ПІДХОДІВ

- Відсутність можливості стабільної симуляції в регіонах з високою густиною. Класичні методи не забезпечують коректну роботу при локальних перевищеннях густини, що призводить до нестабільності та появи артефактів.
- Недостатня швидкодія при постійному використанні шару корекції густини. Методи корекції густини підвищують стабільність, але потребують значних обчислювальних ресурсів, що робить їх непридатними для симуляцій у реальному часі.

ОПИС ЗАПРОПОНОВАНОГО СПОСОБУ



Запропонований спосіб покликаний збільшити швидкодію симуляції рідини з використанням шару корекції густини шляхом додання критерію активації шару корекції.

Спосіб полягає в активації корекції густини лише за наявності високонавантажених областей з високою щільністю частинок, що дозволяє уникнути зайвих обчислень

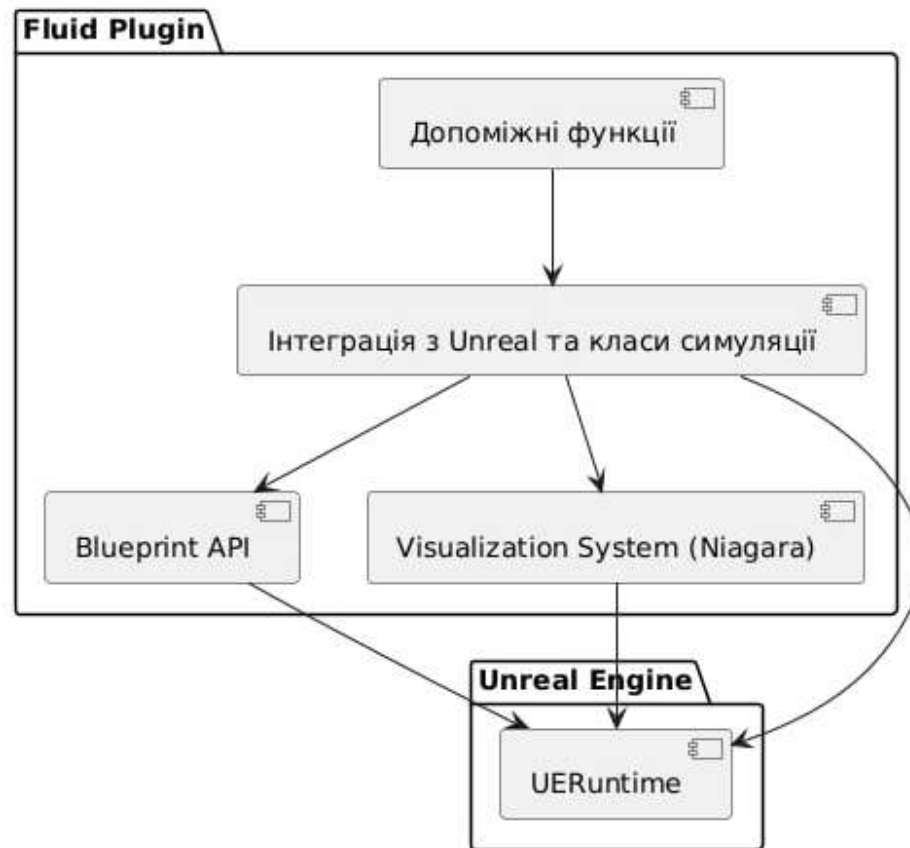
ОПИС ЗАПРОПОНОВАНОГО СПОСОБУ



Застосування такого порогу враховує дискретність представлення рідини та забезпечує уникнення зайвих обчислень.

$$\rho_{threshold} = \alpha\rho_0 + \beta \frac{N_i}{N_{max}}$$

СХЕМА РОЗРОБЛЕНОГО ЗАСТОСУНКУ З ВИКОРИСТАННЯМ СПОСОБУ





ОПИС ЗАПРОПОНОВАНОГО СПОСОБУ

Основний C++ інструментарій:

- Вказівники для роботи з внутрішніми масивами без копіювання
- Передача даних за посиланням / адресою для мінімізації накладних витрат
- Встроювання критичних функцій

Основний Unreal інструментарій:

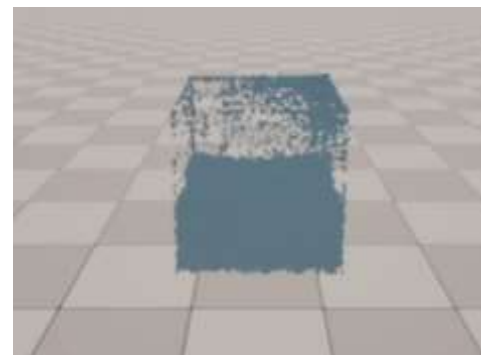
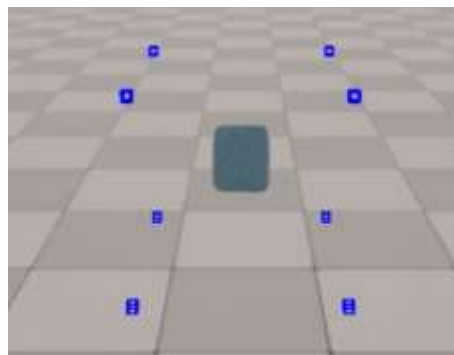
- Використання API рушія для паралельних задач
- Робота з внутрішніми контейнерами та структурами даних
- Використання існуючих пайплайнів для внутрішньої роботи та інтеграції з системою візуалізації



АНАЛІЗ РЕЗУЛЬТАТІВ

	Поріг відключено	Адаптивний поріг
Середній час видачі кадру	0.0451	0.0357

Отримані значення свідчать, що введення порогу призвело до пришвидшення симуляції в середньому на 20.84% , що є значним покращенням для такого типу симуляцій.



НАУКОВА НОВИЗНА

- Вперше запропоновано метод симуляції поведінки рідин, характерною рисою якого є застосування адаптивного порогу до шару корекції густини, що дозволило прискорити симуляцію в середньому на 20.84%.

ПРАКТИЧНА ЗНАЧУЩІСТЬ

Запропонований підхід може бути імплементований розробниками як у вже існуючих програмних рішеннях, так і під час створення нових застосунків. Крім того, програмну реалізацію методу виконано у вигляді плагіна для Unreal Engine, що забезпечує можливість його використання широким колом користувачів: розробниками ігор, архітекторами, фахівцями зі створення візуальних ефектів та іншими спеціалістами.

СТАРТАП ПРОЄКТ

До конкурентних переваг відносяться наступні пункти:

- Відсутність необхідності залучати декількох спеціалістів для розробки складної системи симуляції рідин зменшить витрати на їх утримання.
- **Легкість підключення до проєкту.** Розробка у формі плагіну і використання внутрішньої торгівельної платформи надасть змогу використовувати програмне забезпечення без внесення докорінних змін в інші частини проєкту.
- **Використання інноваційного** способу дозволяє плагіну бути більш оптимізованим для роботи
- **Легкість використання.** Використання структур та контейнерів рушія, а також створення спеціальних класів Blueprint дозволяє легко використовувати програмне забезпечення без потреби у додатковому навчанні.

СТАРТАП ПРОЄКТ

Проблема Складність процесу симуляції рідини в реальному часі	Інноваційна технологія, спосіб, метод Поєднання ключових переваг методів FLIP та Implicit Density Projection з використанням порогу для створення більш досконалого способу симуляції рідин.	Унікальна ціннісна пропозиція Програмне забезпечення, що реалізує інноваційний спосіб симуляції рідини в реальному часі з можливістю налаштування та легкою інтеграцією до проєкту.	Зацікавлені сторони Програмісти ігор Інді розробники ігор 3D художники, художники по оточенню Дизайнери рівнів Архітектори Власники рушія Unreal Engine (Компанія Epic Games) Науковці у сфері симуляції рідин	Ринок Бізнеси різних розмірів, особливо компанії для розробки ігор, що використовують Unreal Engine у своїх проєктах.
Рішення Програмне забезпечення у вигляді плагіну до Unreal Engine з можливістю гнучкого налаштування.	Програмний продукт Плагін до Unreal Engine для симуляції рідини в реальному часі		Конкурентні переваги 1. Відсутність необхідності залучати декількох спеціалістів 2. Легкість підключення до проєкту. 3. Використання інноваційних методів. 4. Гнучкість 5. Легкість використання.	Канали збуту Внутрішній торгівельний майданчик.
Структура витрат Утримання розробника під час розробки та підтримки проєкту після введення в експлуатацію. Оплата комунальних послуг та інтернету. Підтримка робочого обладнання.		Потоки доходів Доходи від продажу прав на використання плагіну.		

АПРОБАЦІЯ

Доповідь на науковій конференції:

Нещадим О.М., Седухіна А.Д. СПОСІБ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ СИМУЛЯЦІЇ РІДИН В РЕАЛЬНОМУ ЧАСІ // Прикладна математика та комп'ютинг ПМК' 2025 : збірник тез доповідей Вісімнадцятої наукової конференції магістрантів та аспірантів (Київ, 19-21 листопада 2025 р.). - Київ : КПІ ім. Ігоря Сікорського, 2025. - С. 396-400.



ВИСНОВКИ

Висока швидкість роботи: експериментальні результати продемонстрували, що додання адаптивного критерію для підключення шару корекції густини до класичних методів симуляції рідин у реальному часі дозволяє підвищити швидкодію, що робить можливим успішно розв'язувати ситуації, з якими традиційні підходи не справляються.

Можливості для практичного застосування: Запропонований підхід може бути імплементований розробниками як у вже існуючих програмних рішеннях, так і під час створення нових застосунків. Крім того, програмну реалізацію методу виконано у вигляді плагіна для Unreal Engine, що забезпечує можливість його використання широким колом користувачів: розробниками ігор, архітекторами, фахівцями зі створення візуальних ефектів та іншими спеціалістами.



Дякую за увагу!