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

Аналіз зображення за допомогою клітинних автоматів

Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи
Жежерун О. П.
_____
(підпис)
“____” _____ 2022 р.

Виконала студентка
Куренкова О. В.
“____” _____ 2022 р.

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри мультимедійних систем,
Доцент., к. ф.-м. н.
_____О. П. Жежерун
(підпис)
“____” _____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Студентці Куренковій Олені Володимирівні факультету інформатики 4 курсу
ТЕМА Аналіз зображень за допомогою клітинних автоматів
Вихідні дані: імплементація алгоритму фільтрації шуму сіль-перець на зображенні на базі WebGL.
Зміст ТЧ до курсової роботи:

Календарний план
Вступ
1. Теоретичні відомості щодо клітинних автоматів
2. Огляд існуючих методів обробки зображень
3. Шейдери у WebGL
4. Опис імплементації алгоритму
Висновки
Список використаної літератури
Додатки

Дата видачі “____” _____ 2022 р. Керівник _____
(підпис)
Завдання отримала _____
(підпис)

Тема: Аналіз зображень за допомогою клітинних автоматів

Календарний план виконання роботи:

п/п	Назва етапу	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	листопад 2021	
2.	Вивчення предметної області	листопад-грудень 2021	
3.	Вивчення технологій для розробки	січень 2022 р	
4.	Побудова технічного завдання	січень-лютий 2021р	
5.	Написання практичної частини роботи	лютий-березень 2022 р.	
6.	Написання текстової частини роботи	квітень 2022 р	
7.	Перегляд змісту роботи з керівником	квітень-травень 2022	
8.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	травень 2022	
9.	Створення презентації	травень 2022	
10.	Захист роботи	травень 2022	

Студент Куренкова О. В.

Керівник Жежерун О. П.

“_____” _____

Анотація

У даній курсовій роботі розглядається поняття клітинного автомату, його властивостей. Проведено огляд існуючих методів обробки зображення, зокрема алгоритмів пошуку країв та алгоритмів фільтрації шумів. Проаналізована можливість обробки зображення клітинним автоматом та наведені існуючі алгоритми. Було імплементовано алгоритм фільтрації зображення від шуму “сіль та перець” на основі WebGL та описані деталі реалізації.

Ключові слова: клітинний автомат, обробка зображень, фільтрація шумів, медіаних фільтр, шейдери, WebGL.

Зміст

Вступ

Обробка та аналіз зображень актуальна сфера, яка постійно розвивається. Продовжуються пошуки алгоритмів, які спростять обробку зображень, зроблять її більш ефективною або швидшою. У роботі розглядається поняття клітинних автоматів - моделей з простими правилами, але складною поведінкою. Вони прості в імплементації та можуть бути легко розпаралелені. Чи можуть клітинні автомати кинути виклик існуючим алгоритмам, таким як метод Канні для пошуку країв чи медіанний фільтр для прибирання шуму?

Робота складається з чотирьох частин. У першому розділі розглянуті теоретичні відомості щодо клітинних автоматів з прикладами відомих автоматів. У другому розділі описані існуючі методи обробки зображень. Третій розділ присвячений теоретичним деталям шейдерів на базі WebG та аргументації обраної технології. У четвертому розділі наведені деталі імплементації алгоритму фільтрації шуму “сіль та перець” на зображеннях за допомогою клітинних автоматів.

Перелік прийнятих скорочень, умовних позначень та термінів

КА — Клітинний автомат

CPU — англ. central processing unit, центральний процесор

GPU — англ. graphics processing unit, графічний процесор

WebGL (Web Graphics Library) — API на мові JavaScript для візуалізації інтерактивної графіки у будь-якому сумісному браузері

Шейдер (англ. Shader) — у даній роботі це програма, що призначена для запуску на графічному процесорі

GLSL (OpenGL Shading Language) — мова високого рівня для програмування шейдерів.

Канва - HTML елемент <canvas>

Розділ 1. Теоретичні відомості щодо клітинних автоматів

Поняття клітинного автомата було запропоновано у 1940-х фон Нейманом, який був зацікавлений у пошуках систем, які можуть відтворити себе. Клітинний автомат — це проста математична модель для дослідження самоорганізації та самовідтворення. Після створення Гри в Життя Джоном Конвеєм область клітинних автоматів стала більш відомою та популярною. Зачаровуюча простота клітинного автомата, що призводить до створення складних структур є привабливою для багатьох дослідників.

1.1 Визначення та властивості клітинного автомата

Клітинний автомат - це математична модель, простір якої складається з комірок. Кожна з цих клітин перебуває у одному з скінченного набору станів у будь-який момент часу t. У наступний момент часу t+1 стан комірки може змінюватись відповідно до визначених правил переходу. Правило переходу є функцією, що залежить від поточного стану клітинки та станів сусідніх клітинок.[2]

Для ліпшого розуміння розглянемо приклад одновимірного клітинного автомата. Він складається з послідовності комірок. Для кожної клітинки визначено 2 стани: вона може бути біла або чорна та 2 сусіди - правий та лівий. Кожен наступний стан визначений за правилом, яке залежить клітини та її сусідів. Три послідовні клітини, які можуть мати по 2 стани можуть утворити $2^3 = 8$ можливих комбінацій, для яких визначається також вихідний стан клітини. Таке правило можна візуалізувати наступним чином (Рисунок 1.1), де верхній ряд - вхідні стани клітинок, нижній - вихідні. Для такого автомата існує всього $2^8 = 256$ різних правил і їх дослідив та аналізував та класифікував Стівен Вольфрам.[3]

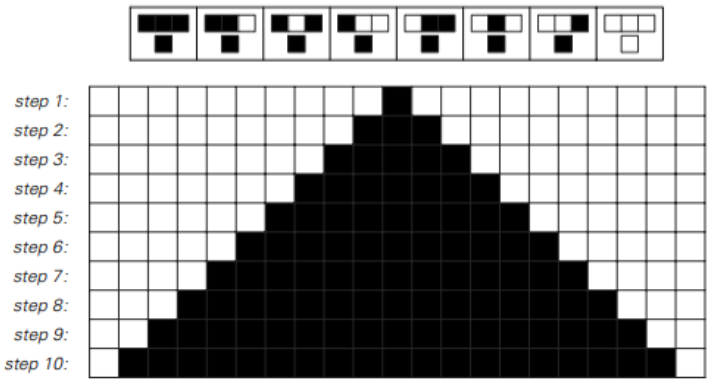

Рисунок 1.1 Правило 254 та перші 10 кроків його виконання

Особливості та переваги клітинних автоматів [5]

1. Незважаючи на прості правила, КА може демонструвати складну поведінку.
2. Зміни КА є локальними, адже кожний елемент залежить тільки від власного околу. Але у той же час локальні зміни з часом можуть вплинути на всю систему.
3. Обчислення для клітинних автоматів доволі прості і можуть бути легко розпаралелені.
4. Для кожної клітини є скінченний набір станів і перехід від одного стану до іншого потребує злічену кількість операцій.

1.2 Види клітинних автоматів

Класифікувати клітинні автомати можна за різними властивостями. Додавання властивостей або зміна певних обмежень відкривають цікаві галузі розвитку клітинних автоматів.

У першу чергу КА можуть варіюватись за розмірністю: одновимірні, двовимірні, тривимірні тощо. А також мож елементи КА можуть набувати різної геометричної форми: трикутниками, чотирикутники, шестикутниками тощо.

Прийнятим критерієм КА є синхронність - тобто всі елементи КА переходять до наступного стану одночасно. Але якщо кожна комірка змінює свій стан випадково, то такий клітинний автомат називається **асинхронним**.

За класичним визначенням КА є **однорідним** - тобто різних клітинок застосовується однакове правило. У випадку якщо до різних комірок застосовуються різні правила клітинний автомат називається **неоднорідним**. Наприклад, якщо простір КА є обмеженим, то до граничних елементів може застосовуватись відмінне правило. Локальне правило може бути недетермінованим, наприклад, може змінюватись в часі.

КА є **детермінованим**, якщо стан комірки однозначно визначається на кожному кроці. У іншому випадку, якщо стани комірок в наступний момент часу визначаються на основі деяких ймовірностей клітинний автомат називається **ймовірнісним**.

Ще одним класом КА є **тоталістичні** клітинні автомати. Ідея тоталістичного автомату полягає в тому, що стан клітинки залежить не від окремих станів сусідніх клітин, а від середнього значення навколишніх елементів. Якщо стан клітини залежить від суми сусідніх та значення поточної комірки, тоді клітинний автомат називають **зовнішнім тоталістичним**.[7]

Зворотні клітинні автомати - КА у якому кожна конфігурація має унікального попередника, тобто попередній стан будь-якої клітинки перед оновленням можна однозначно визначити з оновлених станів усіх комірок.

У **блокових** автоматах комірки групуютьсяу виключні блоки і перехід до наступного стану відбувається у межах блоку. На кожному кроці розбиття на блоки змінюється і клітина належить лише до одного блоку, але її сусідство змінюється.

Якщо клітини змінюють своє розташування у різні моменти часу, то такий клітинний автомат є **рухомим**.

1.2.1 Класифікація С. Вольфрама

Стівен Вольфрам у книзі “A New Kind Of Science” [3] навів 4 класи конфігурацій клітинних автоматів в залежності від їхньої поведінки.

Перший клас - клас найпростіших автоматів. Сюди відносяться клітинні автомати, які з часом переходять у однорідний або гомогенний стан. Приклад такого автомата наведено на рисунку 1.2 (a) усі клітини з часом стають чорними і більш не змінюються.

До другого класу відносяться ті клітинні автомати, які з часом утворюють стабільні або коливальні структури. Тобто з розвитком моделі усворюється стала структура або вона повторюється через кожні кілька кроків (рисунок 1.2 (b)).

У третьому класі клітинні автомати розвиваються хаотично та псевдо-рандомно (Рисунок 1.2 (c)). Локальні зміни можуть невизначено поширюватись, а сталі структури руйнуються навколишнім шумом.

Четвертий клас КА - найцікавіший, оскільки в ньому поєднуються структурованість та хаотичність. Локальні структури можуть створюватись, рухатись та взаємодіяти з іншими структурами. (Рисунок 1.2 (d))

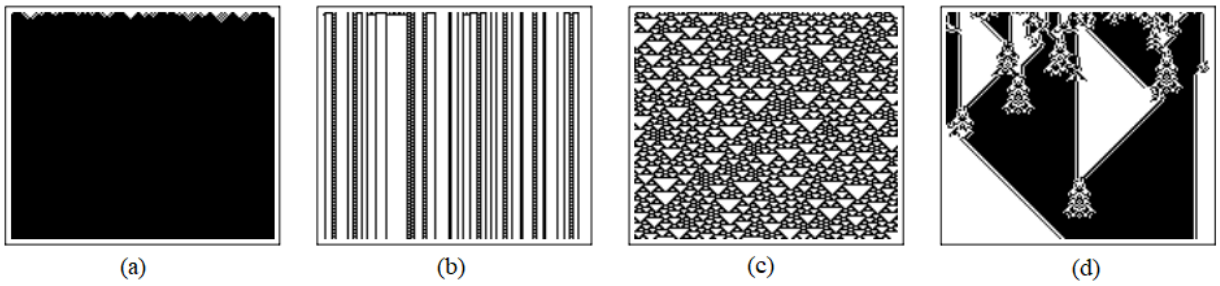

Рисунок 1.2 Класи клітинних автоматів за Стівеном Вольфрамом

1.3 Сусідство у клітинному автоматі

За визначенням правило клітинного автомата залежить від навколишніх елементів. Сукупність навколишніх клітин, що є вхідними даними для функції переходу називають **сусідством** або **окілом**. Формулювання околу може залежати від специфіки клітинного автомата: його розмірності, геометричної структури елементів. Комірки автомата можуть бути трикутними, чотирикутниими, шестикутними тощо.

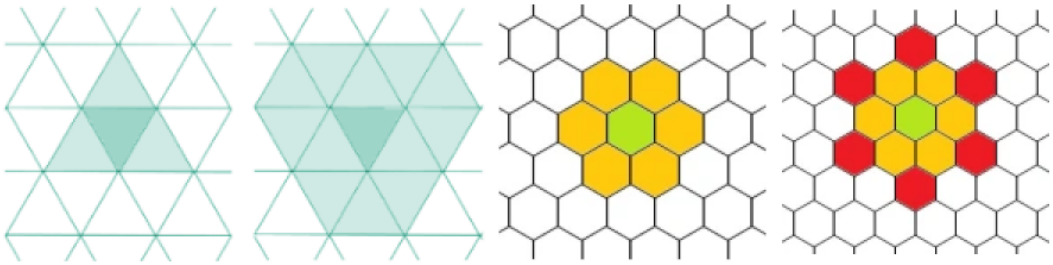

Рисунок 1.3 Можливі околи у двовимірних клітинних автоматах

Оскільки найчастіше елементи клітинного двовимірного автомата представляються у вигляді квадратної решітки, найпопулярнішими околами є окіл фон Неймана та окіл Мура. У тривимірному просторі використовуються їхні тривимірні аналоги (Рисунок 1.4).

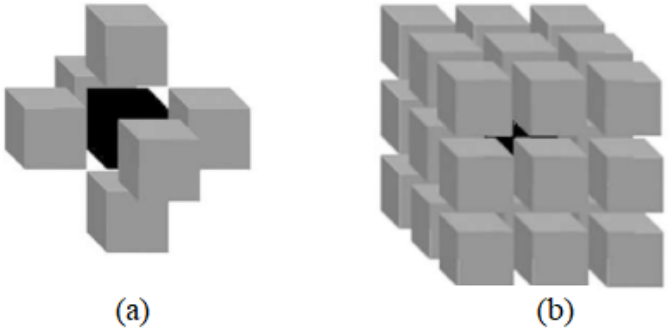

Рисунок 1.4 Окіл фон Неймана та окіл Мура у тривимірному просторі

1.3.1 Сусідство фон Неймана

Сусідство фон Неймана радіуса r визначено рівнянням:

$$N_{x_0,y_0} = \{(x, y) : |x - x_0| + |y - y_0| \leq r\}$$

Клітини, які потрапляють в окіл комірки (x_0, y_0) утворюють ромбоподібну форму. Візуалізацію сусідства фон Неймана наведено у Рисунку 1.5, де зелені клітинки є околом для чорної.

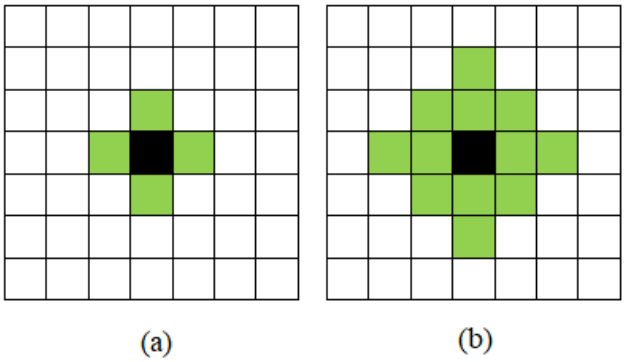

Рисунок 1.5 Сусідство фон Неймана радіуса 1 (a) та радіуса 2 (b)

1.3.2 Сусідство Мура

Сусідство Мура радіуса r визначається рівнянням:

$$N_{x_0,y_0} = \{(x, y) : |x - x_0| \leq r,\ |y - y_0| \leq r\}$$

Клітини, які потрапляють в окіл комірки (x_0, y_0) утворюють квадратну форму. Візуалізацію сусідства фон Неймана наведено у Рисунку 1.6, де зелені клітинки є околом для чорної.

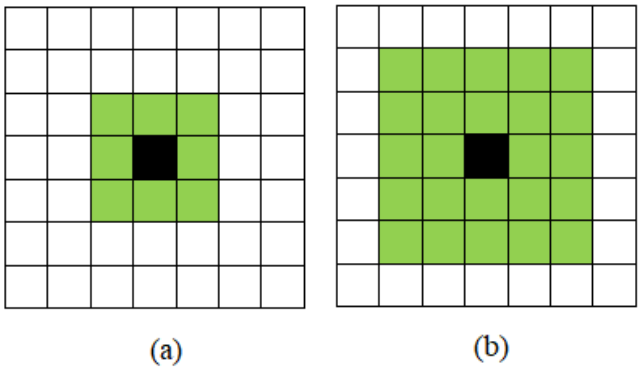

Рисунок 1.6 Сусідство Мура радіуса 1 (a) та радіуса 2 (b)

1.4 Відомі клітинні автомати

1.4.1 Гра в Життя

У 1970 році Гра в Життя була представлена Джоном Конвеєм і одразу викликала хвилю зацікавленості. Причиною було те що прості правила гри

породжували надиво комплексні структури. Чи можна назвати Гру в Життя справжньою грою доволі філософськи питання, але сам Джон називав її грою без гравця (0-player game).[9] Гра в Життя - безкінечний двовимірний тоталістичний клітинний автомат, де у кожної комірки є всього 2 можливі стани - жива або мертва, та 8 сусідів за сусідством Мура. Правила переходу дуже прості:

1. Будь-яка жива клітина з 2 або 3 живими сусідами залишається живою.
2. Будь-яка мертва клітина з трьома живими сусідами стає живою.
3. Усі інші живі клітини гинуть у наступному поколінні.
4. Усі інші мертві клітини залишаються мертвими.

Ці прості правила призводять до виникнення величезної кількості різноманітних форм, які можна класифікувати[10]:

- Стійкі фігури, що лишаються незмінними після кожної ітерації
- Періодичні фігури, стан яких повторюється через деяку кількість поколінь.
- Рухливі фігури, стан яких повторюється, але при цьому фігура зсувається у просторі.
- Гармати - періодичні фігури, які породжують рухливі фігури.
- Паротяги - рухливі фігури, внаслідок руху яких залишаються стійкі або періодичні фігури.
- Пожирачі - стійкі або періодичні фігури, які можуть “з’їсти” рухливу фігуру при зіткненні.
- Довгожителі - фігури, що довго змінюються, перш ніж стабілізуватися і перетворитися на групу постійних або періодичних фігур.

Рисунок 1.7 Гармата, що створює глайдери (зліва) та приклад розвитку випадкової популяції клітин (справа)

1.4.2 Мураха Ленгтона

Ще один цікавий клітинний автомат простими правилами і надиво комплексною поведінкою - мураха Ленгтона. Мураху можна також вважати двовимірною машиною Тюрінга з 2 символами і 4 станами.[11] Ця модель теж є нескінченним двовимірним автоматом з квадратними комірками, які можуть бути білого або чорного кольору. В одній з клітин знаходиться «мураха», яка на кожному кроці може рухатися в одному з чотирьох напрямків в клітинку, сусідню по стороні. Правила для мурашки:

1. Якщо вона знаходиться на чорній клітинці - повернути на 90° вліво, змінити колір клітинки на білий і зробити крок вперед на наступну клітинку.
2. Якщо вона знаходиться на білій клітинці - повернути на 90° вправо, змінити колір квадрату на чорний, зробити крок вперед на наступну клітинку.

Незалежно від початкового стану поля, мураха спершу рухається доволі хаотично, але в якийсь момент мураха починає будувати магістраль з періодичністю в 104 кроки і рухається так нескінченно.

Рисунок 1.8 Стан після 11000 кроків. Червоний піксель позначає мураху

1.4.3 Wireworld

Wireworld — це клітинний автомат, запропонований Браяном Сільверманом у 1987 році. Цей автомат має прості правила та є повним за Тюрінгом.[12] Wireworld - це двовимірний клітинний автомат з квадратними комірками, які мають 4 різних стани та 8 сусідів (окіл Мура). Стани яких може набувати клітина:

1. Порожня клітина - чорна
2. Провідник - позначення жовтогарячим кольором
3. Голова електрона - позначення синім кольором
4. Хвіст електрона - позначення білим кольором

На кожному кроці клітини підпорядковуються таким правилам:

1. Порожня клітина завжди залишається порожньою
2. Голова електрона завжди стає хвостром електрона
3. Хвіст електрона завжди стає провідником
4. Якщо на сусідніх клітинках є рівно 1 або 2 голови електронів, провідник стає головою електрона. У інших випадках залишається провідником.

За допомогою Wireworld можна легко моделювати логічні операції. Рисунок 1.8 є прикладом симуляції діоду: у верхньому прикладі електрони вільно проходять по провіднику, а у нижньому - ні.

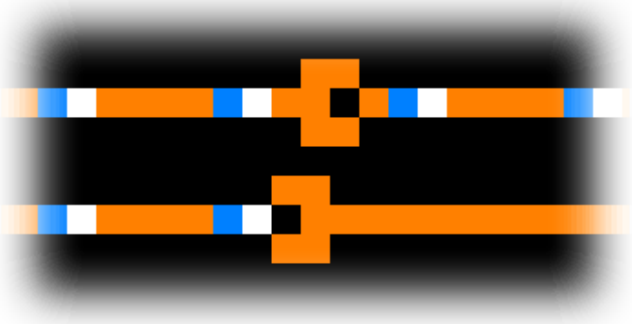

Рисунок 1.9 Симуляція діоду у Wireworld

1.5 Висновки до розділу 1

У цьому розділі були розглянуті теоретичні відомості щодо клітинних автоматів. Спершу було наведено визначення та пояснення на прикладі елементарного одновимірного автомата. Також були розглянуті різні види клітинних автоматів в залежності від різних характеристик та розглянута класифікація Стівена Вольфрама. Були наведені приклади різних сусідств та формалізовані окіл вон Неймана та Мура. У кінці розділу були розглянуті деякі з відомих клітинних автоматів: Гра в Життя, мураха Ленгтона та Wireworld.

Розділ 2. Огляд існуючих методів обробки зображень

Камери сприймають світ інакше на відміну від людського ока, тож для того, щоб цифрове зображення якісніше відображало реальність необхідна додаткова обробка. Обробка зображень потрібна для виокремлення важливої інформації та загального поліпшення зображення, надання йому художньої цінності. Для попередньої обробки зображення використовуються різні підходи в залежності від потреб. Можна виділити такі основні кроки:

1. Геометричні перетворення: зсув, поворот, масштабування тощо
2. Піксельні перетворення: конвертація у різні кольорові простори, корекція кольору.
3. Фільтрація: прибирання шуму, знаходження країв
4. Сегментація зображення, перетворення Фур’є

Надалі будуть розглянуті лише деякі алгоритми фільтрації.

Основна мета фільтрації - виокремити на зображенні потрібну інформацію або прибрати навпаки непотрібну, зайву. Багато фільтраційних алгоритмів працюють за принципом згортки. Ідея полягає у тому, що до кожного пікселя зображення обраховується нове значення з урахуванням сусідніх значень та матриці згортки.

Клітинні автомати також можуть бути застосовані для обробки зображення, адже їхня структура дуже схожа на структуру зображення. Навіть концепція згортки звучить дуже схоже до визначення клітинного автомата: комірка КА - піксель, а його стан - значення з поточного кольорового простору. Так само можуть застосовуватись сусідства фон Неймана або Мура, а правило переходу - це обрахування наступного стану як суми сусідніх пікселів з коефіцієнтами визначеними у згортці. От тільки у клітинних автоматів є перевага, адже можна застосувати набагато ліпші і цікавіші правила переходу.

2.1 Пошук контурів

Виявлення країв — це метод обробки зображень для виявлення точок цифрового зображення з різкими змінами яскравості зображення. Ці точки, “розриви” яскравості назвивають краями або межами зображення.

2.1.1 Оператор Собеля

Оператор Собеля - це дискретний диференціальний оператор, що обчислює наближене значення градієнта чи норми градієнта для яскравості зображення.[13] Оператор використовує ядра 3×3, з якими згортає зображення для обчислення наближених значень часткових похідних по горизонталі та по вертикалі. Обчислення відбувається наступним чином. Нехай A - вихідне зображення, а Gx та Gy — два зображення, де кожна точка містить часткові похідні по x та по y відповідно. (* - операція згортки).

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A, \; G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Для кожної точки зображення наближене значення градієнта G обчислюється через наближенні значення часткових похідних та також напрямок градієнта:

$$G = \sqrt{G_x^2 + G_y^2}, \; \Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

де, наприклад, кут Θ рівний нулю для вертикальної границі, в якої темна сторона зліва.

2.1.2 Метод Канні.

Метод Канні є найбільш ефективним і широко використовуваним алгоритмом пошуку країв. Він складається з наступних кроків:

1. Конвертувати зображення у кольоровий простір відтінків сірого.
2. Зменшити шум, наприклад застосувавши фільтр Гауса.

3. Вирахувати значення градієнтів для пікселів, застосувавши наприклад оператор Собеля.
4. Відсіяти не максимальні значення.
5. Застосування подвійного порогу.
6. Гістерезис країв. На цьому етапі Залишаються тільки пікселі меж: залишаються “сильні” пікселі та ті “слабкі” пікселі, як є дотичними до “сильних”.

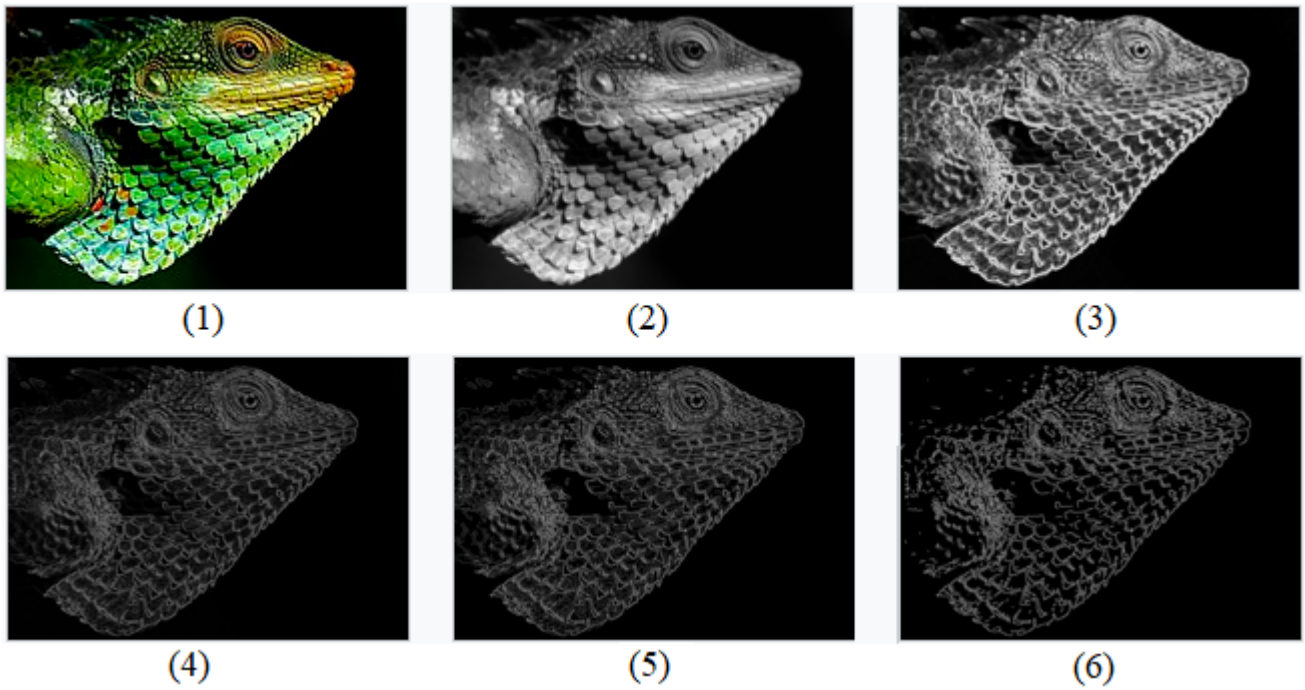

Рисунок 2.1 Кроки методу Канні

2.1.3 Обробка клітинним автоматом

Алгоритм пошуку країв клітинними автоматами застосовується до зображень у бінарному кольоровому просторі або у просторі відтінків сірого. Є пропозиції різних алгоритмів: в одному випадку до бінарного зображення застосовується набір оптимальних правил переходу[14], а іншому - правило переходу визначено рівнянням та надає пікселю значення відповідно до визначених класів сірого.[15]

2.2 Фільтрація шуму

Мета фільтрації шуму - прибрати шумні пікселі, які викривляють інформацію на зображенні.

2.2.1 Види шумів

Перш ніж розглянути алгоритми, які прибирають шум варто зазначити види шумів та їхні особливості. Деякі алгоритми мають відмінні результати з одним видом шуму, але не призначені для інших.

1. **Шум Гауса.** Його також називають електронним шумом, оскільки він виникає в підсилювачах або детекторах. Гаусів шум викликаний природними джерелами, такими як теплові коливання атомів та дискретною природою випромінювання тепла. Характеризується рівномірною спектральною щільністю, нормально розподіленим значенням амплітуди і адитивним способом впливу на сигнал. Гаусовий шум зазвичай порушує значення сірого в цифрових зображеннях.
2. **Імпульсний шум, “Сіль та перець”.** Такий шум може виникати через несправність елементів пікселів у датчиках камери або помилки в процесі оцифровки або зберігання. Шум “сіль та перець” характерний появою білих та чорних пікселів на зображенні.
3. **Періодичний шум.** Цей шум створюється внаслідок втручання сигналів електроніки(наприклад сигнал живлення) під час отримання зображення. Характерний просторово-залежною та синусоїдальної природою, з’являється з певною частотою.

Рисунок 2.2 (a) — шум Гауса, (b) — шум "сіль та перець", (c) — періодичний шум

2.2.2 Медіанний фільтр

Найпростішим алгоритмом фільтрації шуму є Box blur - кожному пікселю присвоюється середнє арифметичне значення навколишніх пікселів. Розмиття Гауса є трошки більш ефективним, оскільки вплив навколишніх пікселів корелюється певними коефіцієнтами.

Концепція медіанного фільтру теж доволі проста: замінити кожний піксель на медіанне значення серед навколишніх пікселів. Сукупність навколишніх пікселів називають "ковзаючим вікном". Чим більший радіус такого вікна, тим більша вибірка значень пікселів. Медіанний фільтр доволі ефективний, особливо проти імпульсних шумів. Але хоч він простий концептуально та має гарні результати, він потребує більше часу або ресурсів для обрахування. Наївною імплементацією було б на кожному кроці сортувати значення пікселів для пошуку медіани, тоді складність алгоритму $O(r^2 \log r)$, де r - радіус вікна.[17] Можливо поліпшити складність алгоритму до константного часу, але все одно потрібні додаткові ресурси, необхідно додатково зберігати гістограму у пам'яті.

Недоліком усіх наведених алгоритмів є те, що вони можуть розмивати контури на зображенні, особливо якщо застосовується великий радіус "ковзаючого вікна".

2.2.3 Обробка клітинним автоматом

Розглянемо метод фільтрації імпульсного шуму за допомогою клітинного автомата. Ідея доволі проста: якщо клітина є шумом, то замінити її на будь-яку сусідню, яка не є шумом. Також можливо замінити на моду в околі, тобто те значення, яке найчастіше зустрічається в околі.

Нехай C(i,j) - піксель вхідного зображення, δ - функція переходу у наступний момент часу t+1, яку можна подати у загальному вигляді

$$C(i,j)^{t+1} = \delta\left(C(i,j)^t, C\left(i', j'\right)^t\right),$$ де

$C\left(i', j'\right) \in N_M(i,j,r)$ - пікселі з околу.

Розглядаються різні сусідства та різний порядок перебору сусідніх клітинок. Узагальнене сусідство наведено у Рисунку 2.3, де ψ - ідентифікує конкретну сусідню помірку. Сусіди ψ1-4 - утворюють окіл вон Неймана радіусу 1, ψ1-12 - радіусу 2. Сусіди ψ1-8 - утворюють сусідство Мура радіусу 1, ψ1-24 - радіусу 2.

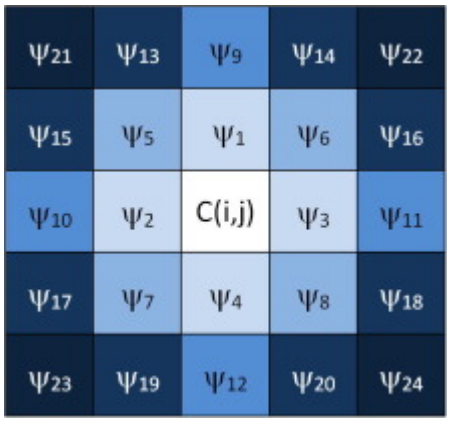

Рисунок 2.3 Окіл комірки C(i,j) [2]

Найпростіший випадок - розглянути лише верхню та нижню комірки відносно C(i,j). Для кожного пікселя зображення, якщо його значення є кольоровим екстремумом (0 - чорний, або 255 - білий), тоді перевірити чи є C(i-1,j) екстремумом. Якщо ні, то привласнити це значення поточному пікселю. У іншому випадку перевірити аналонічно C(i+1,j). У випадку, коли обидва пікселі не задовольняють умову, залишити поточний піксель без змін. Внаслідок одного кроку такого

клітинного автомата шуму стає помітно менше або зникає зовсім, якщо вхідне зображення було мало зашумлене. Достатньо зробити кілька ітерацій і отриманий результат буде негіршим за медіанний фільтр.

Недоліком такого алгоритму може бути вертикальна пропагація: коли через скупчення шумних пікселів, для заміни використовується одне й те саме значення, і воно візуально вертикально “розростається”. Таку пропагацію наглядно видно на Рисунку 2.4 (d) та (e). Для усунення цього недоліку алгоритм розглядає більший набір пікселів перед заміною, наприклад перебрати усі 24 пікселі в околі Мура з радіусом 2.

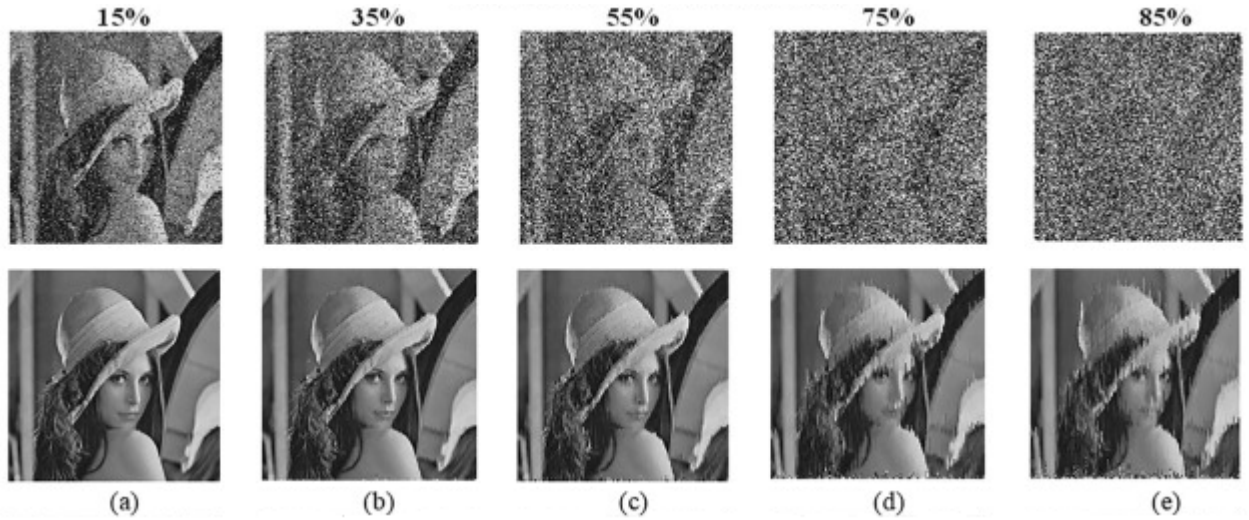

Рисунок 2.4 Результат застосування наведеного фільтру на зображеннях з різним ступенем зашумленості

2.3 Висновки до розділу 2

У другому розділі були розглянуті існуючі алгоритми обробки зображень. Більш детально були розглянуті методи для пошуку контурів та методи фільтрації шумів. Серед розглянутих алгоритмів були оператор Собеля, метод Канні, медіанний фільтр. Також були наведені альтернативні методи обробки клітинними автоматами.

Розділ 3. Шейдери у WebGL

Для імплементації практичної частини були використані шейдери на базі WebGL. У цьому розділі будуть розглянути деякі теоретичні відомості та специфікації щодо обраних технологій а також описані переваги такого підходу.

3.1 Паралельні GPU обчислення

Обробка зображення — процес ресурсоємкий, оскільки обчислення потрібно зробити для кожного пікселя. Більшість операцій оброблюється “мозком” пристрою - центральним процесором. Центальний процесор (CPU) займається обчисленнями, має контрольний блок, який займається послідовним виконанням задач. Також CPU взаємодіє і іншими частинами пристрою: пам’яттю, засобами вводу та виводу тощо. Процесор дуже швидко обробляє дані послідовно, оскільки має кілька ядер з високою тактовою частотою. Він може швидко перемикатись між задачами і створювати враження паралелізму, але він все таки розрахований для послідовних обчислень.

Натомість графічний процесор складається з сотень невеликих легких ядер. Поодинці кожне ядро GPU поступається ядрам процесора, але вони чудово обраховують паралельні задачі, незалежно одне від одного. Графічний процесор першочергово розрахований для обробки графіки. Іншою особливістю графічного процесора є спеціальні математичні функції, які прискорені за допомогою апаратного забезпечення, тому складні математичні операції вирішуються безпосередньо мікрочіпами, а не програмним забезпеченням.

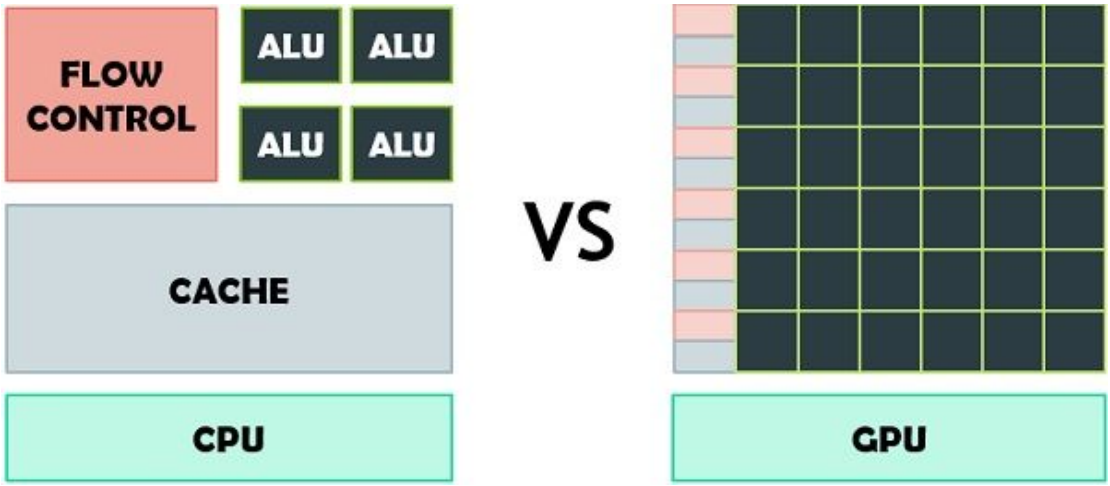

Рисунок 3.1 Різниця між CPU та GPU

Властивості клітинних автоматів дозволяють гарно розпаралелити процес обробки зображення. Критерій одночасного переходу всіх комірок до наступного стану ніби натякає на паралельні обчислення. Зміни для кожної комірки обраховуються локальним правилом, тож різні частини зображення можуть бути обчислені паралельно та незалежно, що пришвидшить обробку зображення. Отож наведена реалізація клітинного автомата розрахована на обрахування за допомогою графічного прискорення.

3.2 Основні поняття WebGL

WebGL (Web Graphics Library) — JavaScript (API) для реалізації інтерактивної веб-графіки. Він дозволяє рендерити графіку за допомогою графічного процесора прямо в HTML канву. Іншими словами WebGL - це засіб растеризації векторних об’єктів для відображення на екрані за допомогою графічного процесора (GPU). Що відрізняє його від інших інструментів, так це якість і складність візуальних елементів, які він може відтворити.

Технологія WebGL, створена KhronosGroup, є прямим нащадком OpenGL ES. Ця інноваційна технологія в даний час використовується в 3D веб-дизайні, інтерактивних іграх, симуляції фізики, візуалізації даних та ілюстраціях.

Для використання WebGL необхідно надати код, який буде виконуватись графічним процесором. Програма складається з 2 частин: вершинного шейдеру та фрагментного шейдеру.

Шейдер - це програма, яка розрахована для виконання графічним процесором. Є кілька різних мов, якими можуть бути написані шейдери, але у даній роботі використовується GLSL. **GLSL** (OpenGL Shading Language) — мова високого рівня для програмування шейдерів, синтаксис якої схожий на мову C.

Вершинний шейдер обраховує координати вершин об’єктів, а фрагментний шейдер визначає колір пікселів. Додаткові поняття, які необхідні у роботі з шейдерами:

- **Атрибути (attribute)** — це глобальні змінні, як передаються від застосунку до вершинного шейдеру і використовуються для обчислення координат вершин.
- **Константи (uniform)** — це глобальні змінні, як передаються від застосунку до шейдеру будь-якого типу. Всередині шейдеру ця змінна є константою. Відмінність між uniform та const визначеннями зминних полягає у тому, що uniform константи передаються в шейдер, а const - встановлюється на етапі компіляції.
- **Змінні (varying)** — Змінні які передаються від вершинного шейдера до фрагментного.
- **Текстури** — це зображення, як передаються до шейдеру. Умовно кажучи, це такі собі “шпалери”, які шейдер може використати і накласти на об’єкт.
- **Буфери** та **фреймбуфери**: буфер — це об’єкт, який може зберігати піскельні дані або зображення. Фреймбуфер трошки складніший і може зберігати більше інформації.

3.3 Висновки до розділу 3

У третьому розділі були порівняні CPU та GPU обчислення та наведені переваги реалізації розпаралеленого алгоритму. Також у цьому розділі описані деякі основні поняття WebGL та шейдерів.

Розділ 4. Опис імплементації алгоритму

4.1 Опис практичної частини

У якості алгоритму для імплементації був обраний алгоритм фільтрації шуму “сіль та перець” за допомогою клітинного автомата. Концепція алгоритму описана у розділі 2.2.3. Мета практичної частини - створити міні-застосунок, до можна побачити результат обробки зображення клітинним автоматом. У якості технологій було обрано мову програмування JavaScript та інтерфейс WebGL, шейдери ж написані мовою GLSL. Основні пункти реалізації:

1. У результаті необхідно мати можливість бачити вхідне і оброблене зображення.
2. Бажано мати можливість змінювати зображення, завантажувати власне, для зручнішого тестування алгоритму.
3. Бажано мати можливість додати шум на зображення для полегшення процесу тестування. Опціонально, бажано інтерактивно контролювати кількість доданого шуму.
4. Бажано мати наглядний результат роботи інших алгоритмів (box blur/медіанний фільтр або фільтр Гауса) над одним і тим самим зображенням, для порівняння ефективності цих фільтрів.

Результатом є імплементації є невеличка веб-сторінка, на якій відображені описані пункти. Результати зашумлення, використання box blur та результат роботи КА алгоритму відображенні HTML елементами <canvas>. Завантаження файлів та контролювання шуму реалізовані засобами JavaScript і їх реалізацію буде опущено.

Робота з WebGL виглядає наступним чином:

1. Спершу створюється WebGL контекст у межах канви, на які будуть відображатись потрібні зміни.
2. Створюється програма з 2 шейдерів: вершинного та фрагментного. Для всіх програм був використаний однаковий вершинний шейдер, який просто створює

площину, пікселі якої будуть промальовуватись вже фрагментним шейдером. Код вершинного шейдеру надано у Додатку А, фрагментні шейдери будуть детальніш описані у наступних підрозділах.

3. У створену програму потрібно додати текстуру. Саме текстурою передається вхідне зображення у програму шейдеру.
4. Також створюються фреймбуфери: вони зберігатимуть проміжний стан роботи клітинного автомата.
5. На наступному етапі за потреби передаються атрибути (attribute) та константні змінні (uniform).
6. Окремо визначається перелік дій, який необхідно робити щоразу при перемальовці - renderLoop. У цьому методі може відбуватись перевизначення змінних, які передаються шейдерам, промальовка об’єктів, рендер у текстуру тощо.
7. Визначений renderLoop передати до методу requestAnimationFrame. Екран бразерів оновлюється з певною частотою, зазвичаю близько 60 раз на секунду. Кожного разу, коли екран буде готовий перемальовуватись, додатково виконаються дії передані до методу requestAnimationFrame.

4.2 Особливості написання шейдерів

Підхід до написання шейдерів трохи не стандартний та змушує замислитись про процес обробки зображення з іншого боку. Зазвичай алгоритми проходяться по зображенню, як по двовимірному масиву, та обраховують значення кожного i, j-того пікселя. Шейдер же пишеться ніби для конкретного пікселя, який не знає свого положення, яке значення він мав у попередній момент часу або поточного стану пікселів навкруги. Саме з такою незалежністю обчислень розпаралелення обробки зображень взагалі можливе. Ще однією особливістю розробки шейдерів є те, що всі координати чи значення нормалізуються у діапазон від 0 до 1. Таким чином, якщо

наприклад є необхідність встановити пісель зеленого кольору, це виглядатиме наступним чином:

gl_FragColor = vec4(0.0,1.0,0.0,1.0); де gl_FragColor - вбудована змінна для зміни кольору, а її значення вектор з чотирма значеннями, які відповідають червоному, зеленому, синьому та альфа (значення прозорості) каналам.

4.3 Створення шуму

Перш ніж прибирати шум з зображення, треба мати зашумлене зображення. Для простоти і зручності тестування, мною був створений додатковий алгоритм, який додає на зображення білі та чорні пікселі - тобто шум “сіль та перець”.

Вхідними параметрами для фрагментного шейдеру, що додає шум на зображення є текстура - тобто саме зображення та міра зашумленості, яка є значенням від 0 до 1.

Шум зазвичай виникає випадковим чином, тож необхідно для кожного пікселя рандомно обрати чи буде він зашумленим та випадковим чином обрати чи він буде сіллю чи перцем. У реалізації було застосовано псевдо-випадкову функцію, яка дозволяє генерувати випадкові значення. Для кожного пікселя відповідно генерується 2 значення:

- одне, яке округлюється до 0 або 1 і визначає колір, шуму
- друге - ймовірність конвертації пікселя у шумний, яка порівнюється із мірою зашумленості. Наприклад міра зашумленості g становить 20%, тобто $g = 0.2$. Ймовірність того, що піксель не змінить свого значення p становить 0.1. У результаті $p < g = 0.1 < 0.2$ і піксель змінить своє значення на зашумлене.

Приклади роботи шейдеру наведено у Рисунку 4.1. Повний код фрагментного шейдера, що відповідає за додавання шуму на зображення наведено у Додатку Б.

Рисунок 4.1 Додавання шуму "сіль та перець" до кольорового зображення Ленна (a) — 0% шуму, (b) — 10% шуму, (c) — 50% шуму

4.4 Реалізація клітинного автомата

Методологія написання шейдерів доволі підходяща для реалізації клітинних автоматів, оскільки по суті у шейдері прописується тільки логіка правила переходу для одного пікселя. Необхідним параметром для такого шейдеру є текстура, яка містить в собі початкове зображення або попередній стан автомату. Ще одним важливим параметром є розмір зображення, який потрібен для знаходження значень клітин в околі. Наприклад координати верхнього сусіднього пікселя

vec2 upCoords = coord + vec2(0.0, 1.0)/u_resolution; де coord - це нормалізовані координати поточної клітинки, u_resolution - розмір зображення, тоді другий доданок є нормалізованим зсувом на один піксель (vec2 визначає вектор зсуву).

Знаходження кольорового значення пікселя з координатами coord тестури u_texture відбувається методом texture2D:

vec4 pixel = texture2D(u_texture, coord);

Повний код шейдера наведено у Додатку В. Алгоритм фільтрації клітинним автоматом з такими простими правилами має гарні результати вже на першій ітерації - може майже повністю прибрати 10% шуму на кольоровому зображенні, результат

наведено у Рисуноку 4.2. Обраний для порівняння Box Blur розмиває імпульсний шум і зображення виглядає неякісно. Це приклад того, як цей алгоритм не розрахований для прибирання імпульсного шуму.

Рисунок 4.2 (a) — зображення з 10% шуму, (b) — результат роботи Box Blur, (c) — результат одного кроку клітинного автомата

Для подальших ітерацій КА необхідно зберегти поточний стан та передати його у той самий шейдер. Для цього використовується техніка рендерінгу у текстуру. WebGL надає можливість не тільки одразу відображати результат на канві, а й зберігати його у фреймбуфер. Також з фреймбуфера можливо передати дані до шейдеру. Проте WebGL не дозволяє одночасно використати буфер для зчитування і запису. Тому для реалізації ітерації клітинного автомату використовуються 2 фреймбуфери, які постійно чергуються між собою: в одному зберігається поточний стан, а в інший записується наступний.

4.5 Висновки до розділу 4

У четвертому розділі описані деталі імплементації обраного алгоритму на базі WebGL. Вказані особливості сприйняття та написання шейдерів. Також наведено більш детальні пояснення щодо двух частин програми: генерація шуму “сіль та перець” та реалізація власне фільтрації створеного шуму за допомогою клітинного автомата.

Висновки

У даній роботі було розглянуте поняття клітинного автомата, зазначені можливі види та наведені приклади відомих клітинних автоматів. Клітинні автомати мають прості правила, але можуть утворювати неочікувано складні структури.

У другому розділі були розглянуті існуючі алгоритми обробка зображень, зокрема алгоритми пошуку країв та алгоритми фільтрації шумів. Були наведені приклади алгоритмів реалізованих за допомогою клітинних автоматів.

Були розглянути теоретичні аспекти вибраних для реалізації практичної частини технологій: WebGL та шейдерів. Клітинні алгоритми доволі просто розпаралелюються, а шейдери - програми, що виконуються на графічних процесорах. У роботі були описані деталі імплементації, наведені результати роботи обраного алгоритму.

Список використаної літератури

1. Ilachinski A. Cellular automata: A discrete universe. Singapore : World Scientific, 2001. 808 p.
2. Jeelani Z., Qadir F. Cellular automata-based approach for salt-and-pepper noise filtration. Journal of King Saud University - Computer and Information Sciences. 2018. URL: https://doi.org/10.1016/j.jksuci.2018.12.006 (date of access: 20.05.2022).
3. Wolfram S. A New Kind of Science. S. l : Wolfram Media Incorporated. 1197 p.
4. The Wolfram Atlas of Simple Programs. The Wolfram Atlas of Simple Programs. URL: http://atlas.wolfram.com/ (date of access: 20.05.2022).
5. Shukla A. P. Training Cellular Automata for Image Edge Detection. Romanian Journal Of Information Science And Technology. Vol. 19, 4, p. 338–359
6. Nayak D. R., Patra P. K., Mahapatra A. A Survey On Two Dimensional Cellular Automata And Its Application In Image Processing.
7. Amrogowicz S., Zhao Y., Zhao Y. An edge detection method using outer Totalistic Cellular Automata. Neurocomputing. 2016. Vol. 214. P. 643–653. URL: https://doi.org/10.1016/j.neucom.2016.05.092 (date of access: 20.05.2022).
8. Shiffman D. The Nature of Code: Simulating Natural Systems with Processing. The Nature of Code, 2012. 520 p.
9. Numberphile. Inventing Game of Life (John Conway) - Numberphile, 2014. YouTube. URL: https://www.youtube.com/watch?v=R9Plq-D1gEk (date of access: 20.05.2022).
10. Життя (гра) – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Життя_(гра) (дата звернення: 20.05.2022).
11. Мураха Ленгтона – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Мураха_Ленгтона (дата звернення: 20.05.2022).

12. What is Wireworld? URL: https://www.quinapalus.com/wires0.html (date of access: 20.05.2022).
13. Оператор Собеля – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Оператор_Собеля (дата звернення: 20.05.2022).
14. RanjanNayak D., Kumar Sahu S., Mohammed J. A Cellular Automata based Optimal Edge Detection Technique using Twenty-Five Neighborhood Model. International Journal of Computer Applications. 2013. Vol. 84, no. 10. P. 27–33. URL: https://doi.org/10.5120/14614-2869 (date of access: 20.05.2022).
15. Wongthanavasu S., Tangvoraphonkchai V. Cellular Automata-Based Algorithm and its Application in Medical Image Processing. 2007 IEEE International Conference on Image Processing, San Antonio, TX, USA, 16 September – 19 October 2007. 2007. URL: https://doi.org/10.1109/icip.2007.4379241 (date of access: 20.05.2022).
16. Boyat A. K., Joshi B. K. A Review Paper : Noise Models in Digital Image Processing. Signal & Image Processing : An International Journal. 2015. Vol. 6, no. 2. P. 63–75. URL: https://doi.org/10.5121/sipij.2015.6206 (date of access: 20.05.2022).
17. Perreault S., Hebert P. Median Filtering in Constant Time. IEEE Transactions on Image Processing. 2007. Vol. 16, no. 9. P. 2389–2394. URL: https://doi.org/10.1109/tip.2007.902329 (date of access: 20.05.2022).
18. WebGL Is a Key to Stunning Web Projects - Visartech Blog. Visartech: Custom Software Development Services. URL: https://www.visartech.com/blog/interactive-3d-graphics-with-webgl/ (date of access: 20.05.2022).
19. The Book of Shaders. The Book of Shaders. URL: https://thebookofshaders.com/ (date of access: 20.05.2022).

Додатки

Додаток А

Код використаного вершинного шейдеру (vertex.glsl).

```
precision mediump float; // необхідно встановити точність чисел з плаваючою точкою
attribute vec2 position; // вхідна змінна, 2 координати вершини
attribute vec2 vertCoord; // вхідна змінна, 2 координати

varying vec2 coord; // ця змінна буде передана до фрагментного шейдеру

// Основна функція шейдеру
void main() {
    coord = vertCoord; // Передаємо отримане значення до фрахментного шейдеру
      gl_Position = vec4(position, 0.0, 1.0); // встановлюємо положення вершини, зо
допомогою
}
```

Додаток Б

Код фрагментного шейдеру, який додає шум “сіль та перець” до зображення (applyNoise.glsl).

```
precision mediump float; // необхідно встановити точність чисел з плаваючою точкою
uniform sampler2D u_texture0; // вхідний параметр - текстура, зображення, до якого буде додано шум
uniform float gate; // Міра зашумленості від 0 до 1. Якщо 0, то шум не застосовується

varying vec2 coord; // Змінна отримана від вершинного шейдера, позначає координати поточного пікселя

// Псевдо випадкова функція
float rand(vec2 st) {
    return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) * 47758.5453123);
}

// Функція, що генерує нормально розподілені випадкові значення.
// За допомогою параметра seed можна отримати різні випадкові значення, для однакових значень вектора st
float urand(vec2 st, float seed) {
    float t = fract(seed);
    float rnd = rand(st + 0.07 * t);
    return rnd;
}

// Основна функція, яка буде виконуватись
void main() {
    // беремо значення пікселя текстури u_trxture0 у точці з координатами coord
    vec4 image = texture2D(u_texture0, coord);
    // Генеруємо чорний або білий піксель шуму для даної координати
    float noise = step(0.5, urand(coord, 1.0));
    // Визначаємо ймовірність з якою поточний піксель перетвориться на шумний
    float change = urand(coord, 0.8);
    // Вихідний піксель або стає азашумленим або незмінним
    // відповідно до визначених міри зашумленості та ймовірності
    gl_FragColor = change < gate ? vec4(vec3(noise), 1.0) : image;
}
```

Додаток В

Код фрагментного шейдера, який реалізує фільтрацію зображення за допомогою КА (f3_filter.glsl).

```
precision mediump float; // необхідно встановити точність чисел з плаваючою точкою
uniform sampler2D u_texture0; // вхідний параметр - текстура, зображення, до якого буде додано шум
uniform vec2 u_resolution; // вхідний параметр - розміри канви

varying vec2 coord; // Змінна отримана від вершинного шейдера, позначає координати поточного пікселя

// Перевірка чи є вхідний піксель екстремумом , тобто чорним або білим пікселем
bool isNoise(vec3 color) {
    return color == vec3(0.0)|| color == vec3(1.0);
}

// Основна функція, яка буде виконуватись
void main() {
    // беремо значення пікселя текстури u_trxture0 у точці з координатами coord
    vec3 color = texture2D(u_texture0, coord).rgb;
    // Якщо поточний піксель є "шумним"
    if (isNoise(color)) {
        // Знаходимо координати верхнього пікселя
        vec2 upCoords = vec2(0.0, - 1.0) / u_resolution;
        // Знаходимо стан верхнього пікселя
        vec3 up = texture2D(u_texture0, coord +upCoords ).rgb;
        // Аналогічно з нижнім
        vec2 downCoords = vec2(0.0, 1.0) / u_resolution;
        vec3 down = texture2D(u_texture0, coord + downCoords).rgb;

        // Якщо верхній піксель не є шумним, привласнюємо його значення поточному
        if (!isNoise(up))color = up;
        // Або аналогічно з нижнім
        else if (!isNoise(down))color = down;
    }

    // Встановлюємо колір вихідному пікселю
    gl_FragColor = vec4(color.rgb, 1.0);
}
```