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

Факультет інформатики та обчислювальної техніки

Автоматизованих систем обробки інформації і управління

«На правах рукопису»
УДК 004.9

«До захисту допущено»
В.о. завідувача кафедри
_____ О.А.Павлов
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Математичне та програмне забезпечення для дослідження властивостей тривимірних клітинних автоматів»

Виконав:
студент VI курсу, групи ІП-72мп
Чередніченко Владислав Олександрович _____

Керівник:
доцент, к.т.н.
Фіногенов О.Д. _____

Консультант:
доц., к.т.н. Ліщук К.І. _____

Консультант:
доц., к.т.н. Залевська О.В. _____

Рецензент:
доц., к.т.н., Гармаш О.В. _____

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

Київ – 2019 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Автоматизованих систем обробки інформації і управління

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ О.А.Павлов

«___»_____20__ р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Чередніченку Владиславу Олександровичу

1. Тема дисертації «Математичне та програмне забезпечення для дослідження властивостей тривимірних клітинних автоматів», науковий керівник дисертації Фіногенов Олексій Дмитрійович, доцент, к.т.н., затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом дисертації _____

3. Об’єкт дослідження: тривимірні клітинні автомати

4. Предмет дослідження: процес еволюції та вплив зовнішніх збуджень на геометричну форму клітинного автомату та його властивості

5. Перелік завдань, які потрібно розробити: проаналізувати існуючі класифікації, правила генерації та візуалізації двовимірних і тривимірних клітинних автоматів; визначити необхідний перелік функцій візуалізації для дослідження клітинних автоматів; спроектувати архітектуру та розробити програмне забезпечення для дослідження тривимірних клітинних автоматах

6. Орієнтовний перелік графічного (ілюстративного) матеріалу структурна схема станів системи, структурна схема бази даних, діаграма класів програмного забезпечення

7. Орієнтовний перелік публікацій 2 публікації

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Графічний	Ліщук К.І., доц.		
Науковий	Залевська О.В., доц.		

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Ґрунтовне ознайомлення з предметною областю, визначення структури магістерської дисертації, вивчення літератури	20 червня 2018 року	
2	Робота над першим розділом магістерської дисертації	1 вересня 2018 року	
3	Проведення наукового дослідження; робота над другим розділом магістерської дисертації	30 вересня 2018 року	
4	Робота над третім розділом магістерської дисертації; підготовка статті за результатами наукового дослідження; розроблення програмного забезпечення	30 жовтня 2018 року	
5	Робота над четвертим розділом магістерської дисертації; завершення роботи над основною частиною магістерської дисертації	15 листопада 2018 року	
8	Оформлення текстової і графічної частин магістерської дисертації	30 листопада 2018 року	

Студент Чередніченко В.О.

Науковий керівник дисертації Фіногенов О.Д.

РЕФЕРАТ

Магістерська дисертація:116 с., 51 рис., 18 табл., 2 додатоки, 19 джерел.

Актуальність. Клітинні автомати – це дискретні динамічні системи, поведінка яких повністю визначається в термінах локальних взаємозв'язків. Вперше був відкритий в 1940-х роках Станіславом Уламом і Джоном фон Нейманом. Хоча деякі автомати вивчалися протягом 1950-х і 1960-х років, вони не були популярними до 1970-х років і гри Конвея «Життя». Область застосування моделей такої системи безмежна: від найпростіших «хрестиківнуликів» до штучного інтелекту. Інтерес до предмета розширився за межі академічної науки, що зумовлено підвищенням обчислювальної потужності комп'ютера і доступності. Найбільш дослідженими є двовимірні клітинні автомати, наприклад гра «Життя» та мураха Ленгтона.

Однак до цього часу залишаються мало дослідженими тривимірні клітинні автомати, хоча і мають величезний потенціал для застосування. Перша причина – складність дослідження автоматів та зберігання тривимірного стану автомату. Крім того, відсутні програмні засоби для проведення такого дослідження. Відображення тривимірних автоматів, їх задання та огляд процесу їх еволюції потребують спеціального функціоналу, який не надається засобами для роботи із двовимірними клітинними автоматами. Тому є необхідність в створенні програмного засобу для роботи із тривимірними клітинними автоматами та дослідження їх властивостей за допомогою створеного програмного забезпечення.

Перевагами використання тривимірних клітинних автоматів є їх універсальність в сфері застосування: від генерації тривимірних об'єктів до моделювання складних молекулярних процесів, фізичних, хімічних явищ та квантових ефектів. Така гнучкість досягається за допомогою дискретності розміру автомату та його часу еволюції, а також вільній можливості формулювати власні правила еволюції, залежно від контексту дослідження.

Метою дослідження є еволюція об’єктів за допомогою тривимірних клітинних автоматів.

Для досягнення поставленої мети необхідно виконати **наступні завдання**:

− проаналізувати існуючі класифікації, правила генерації та візуалізації двовимірних і тривимірних клітинних автоматів;

− визначити необхідний перелік функцій візуалізації для дослідження клітинних автоматів;

− спроектувати архітектуру та розробити програмне забезпечення для дослідження тривимірних клітинних автоматах.

Об’єктом дослідження є тривимірні клітинні автомати.

Предметом дослідження є процес еволюції та вплив зовнішніх збуджень на геометричну форму клітинного автомату та його властивості.

Методи дослідження: методи еволюції клітинних автоматів; обчислювальної математики та комп’ютерної і фрактальної графіки.

Наукова новизна:

− вдосконалено методи дослідження еволюції тривимірних клітинних автоматів під впливом зовнішніх факторів;

− визначення критеріїв присутності елементів регулярності в хаотичній структурі під час динамічного розвитку;

− запропоновані нові способи генерації тривимірних об’єктів та структур.

Практичне значення отриманих результатів:

− розроблено програмне забезпечення, що реалізує основні функції по реалізації еволюції тривимірних автоматів;

− розроблено засоби задання функції впливу під час розвитку динамічного об’єкту;

– розроблено засоби перегляду стану та структури об’єкту під час еволюції.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась на кафедрі автоматизованих систем обробки інформації та управління Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» в рамках теми №0117U000914 «Математичні моделі та технології в СППР»

Апробація: результати роботи доповідалися на науковій конференції «Інформатика та обчислювальна техніка – ІОТ-2018».

Публікації. Наукові положення опубліковані в тезах наукової конференції «Інформатика та обчислювальна техніка – ІОТ-2018» та в фаховому збірнику «Сучасні проблеми моделювання» – 2019. – Вип. 13. (прийнято до друку).

Ключові слова: КЛІТИННИЙ АВТОМАТ, ЕВОЛЮЦІЯ, ЖИТТЯ, ГРА, ФРАКТАЛЬНА ГЕОМЕТРІЯ

Thesis: 116 pages, 51figures, 18 tables, 2 appendices, 19 references.

Topic relevance. Cellular automata are discrete dynamic systems whose behavior is fully determined in terms of local interactions. It was first discovered in the 1940s by Stanislav Ulam and John von Neumann. Although some machines were studied during the 1950s and 1960s, they were not popular until the 1970s and Conway's "Life" games. Interest in the subject has expanded beyond the limits of academic science, due to the increase in computer computing power and availability. The scope of the models of such a system is boundless: from the simplest "naughty crosses" to artificial intelligence. The most studied are two-dimensional cellular automata, such as the game "Life" and the Ant Lang.

However, by this time, three-dimensional cellular automata are still little investigated, although they have tremendous potential for use. The first reason is the complexity of the study of automata and the storage of a three-dimensional state of the machine. In addition, there are no software tools for conducting such research. The reflection of three-dimensional automata, their task and review of the process of their evolution need a special functional that is not provided with tools for working with two-dimensional cellular automata. Therefore, there is a need to create a software tool for working with three-dimensional cellular automata and study their properties with the help of the created program.

The advantages of using three-dimensional cellular automata are their versatility in the application field: from the generation of three-dimensional objects to the simulation of complex molecular processes, physical, chemical phenomena and quantum effects. This flexibility is achieved by the discreteness of the size of the machine and its evolutionary time, as well as the free ability to formulate its own rules of evolution, depending on the context of the study.

The aim of the research is the evolution of objects using three-dimensional cellular automata.

To achieve the goal, we must accomplish the **following tasks**:

- analyze the existing classifications, rules of generation and visualization of two-dimensional and three-dimensional cellular automata;
- to define the necessary list of visualization functions for research of cellular automata;
- design architecture and develop software for the study of three-dimensional cellular automata.

The object of research is the three dimensional cellular automata.

The subject of research is the process of evolution and the effect of external excitations on the geometric form of a cellular automaton and its properties.

Methods of research: methods of evolution of cellular automata; computational mathematics and computer and fractal graphics.

Scientific contribution:

- methods of investigating the evolution of three-dimensional cellular automata under the influence of external factors have been improved;
- determination of the criteria for the presence of elements of regularity in a chaotic structure during dynamic development;
- proposed new ways of generating three-dimensional objects and structures.

The practical value of the results obtained:

- software was developed that implements the main functions for the implementation of the evolution of three-dimensional automata
- means of setting the function of influence during the development of a dynamic object;
- tools for reviewing the condition and structure of the object during evolution have been developed.

Thesis connection to scientific programs, plans, and topics. The work was carried out at the Department of Automated Systems for Information Processing and Management of the National Technical University of Ukraine "Kyiv Polytechnic Institute named after Igor Sikorsky" within the framework of the topic №0117U000914 "Mathematical Models and Technologies in DSS".

Approbation: results of work were reported at the scientific conference "Informatics and Computing - IOT-2018".

Publications. The scientific provisions are published in the theses of the scientific conference "Informatics and Computing - IOT-2018" and in the professional collection "Modern Modeling Problems" - 2019. - Vip. 13. (accepted for publication).

Keywords: CELLULAR AUTOMATION, EVOLUTION, LIFE, GAME, FRACTAL GEOMETRY

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

КА - клітинний автомат

Ітерація - етап генерації клітинного автомату

Конфігурація - значення стану клітинного автомату

GPU (Grafics Processing Unit) – графічний процесор

СЗГ (Central Processing Unit) – центральний процесор

API – прикладний програмний інтерфейс (англ. application programming interface);

SQL (мова структурованих запитів) — це стандартна мова для зберігання, маніпулювання та отримання даних у базах даних

JVM – віртуальна машина Java - основна частина системи Java, так званої Java Runtime Environment (JRE).

ВСТУП

Тема клітинних автоматів дуже актуальна, тому що може привести до розгадок багатьох питань в навколишньому світі. Творець гри «Життя» Конуей, вважав, що наш всесвіт можна уявити клітинним автоматом, який управляє рухом елементарних частинок відповідно до деяких правил.

Клітинні автомати є універсальною математичною моделлю, що застосовується для вирішення проблем та задач у галузях математики, фізики та інформатики. Наприклад, двовимірні клітинні автомати широко застосовуються для моделювання поведінки дорожнього трафіку, для симуляції природних та фізичних явищ, таких як гідродинамічні та газодинамічні течії [16]. Серед ще не вирішених питань можна виділити теорію про самовідновлювані електричні кола.

Тривимірні клітинні автомати мають ще більший потенціал, але насправді не так часто використовуються, так як їх властивості ще не досліджені. Прогресу заважає також відсутність зручних інструментів для дослідження та генерації таких КА.

Метою даної магістерської дисертації є еволюція об’єктів за допомогою тривимірних клітинних автоматів.

Для реалізації поставленої мети необхідно виконати наступні поставленя завдання:

- проаналізувати існуючі класифікації, правила генерації та візуалізації двовимірних і тривимірних клітинних автоматів;
- визначити необхідний перелік функцій візуалізації для дослідження клітинних автоматів;
- спроектувати архітектуру та розробити програмне забезпечення для дослідження тривимірних клітинних автоматах.

1 ОГЛЯД ЛІТЕРАТУЛИ

1.1 Огляд клітинних автоматів

Клітинний автомат - дискретна модель. Включає регулярну решітку осередків, кожен з яких може перебувати в одному з кінцевого безлічі станів, таких як 1 і 0. Решітка може бути будь-якої розмірності. Для кожного осередку визначено безліч осередків, званих околицею. Наприклад, околиця може бути визначена як все осередки на відстані не більше 2 від поточної.

Для роботи клітинного автомата потрібно задати початковий стан всіх клітин на першій ітерації і правила переходу клітини з одного стану в інший. У кожній ітерації автомата правило для кожної комірки синхронно оцінюється, змінюючи відповідним чином стани сітки. Поведінка простого правила показано на рисунку 1.1

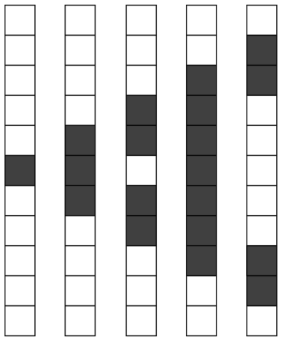

Рисунок 1.1 — Чотири ітерації простого одновимірного клітинного автомату

Кінцева одновимірна сітка ініціалізується з однієї сірої комірки (стан 1) посередині. Кожну ітерацію, клітина стає сірою, якщо в неї є дві сірі клітини - сусіди. В іншому випадку вона повертається або залишається білою (стан 0). Зауважте, що комірка включена до околу.

Якщо розширити цю концепцію до двох або трьох вимірів, то ми отримаємо більше можливостей для вибору околу сусідства. Найбільш поширеними є околи фон Неймана і Мура та їх наслідники, проілюстровані на рисунку 1.2 і 1.3.

Для виконання даної роботи я буду використовувати тривимірний варіант околу Мура, який буде складатися з 26 сусідніх клітин.

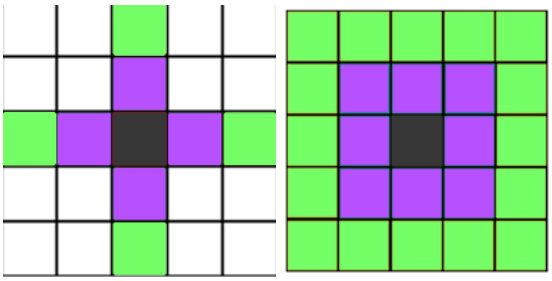

Рисунок 1.2 — Окіл фон Неймана та Мура в двовимірній решітці

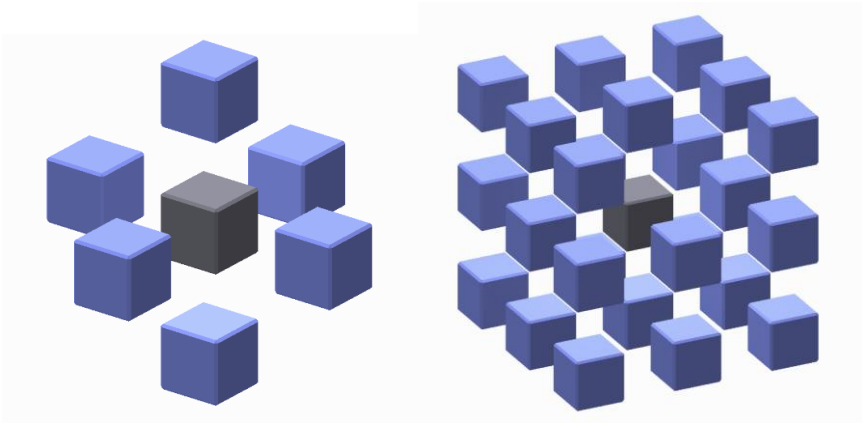

Рисунок 1.3 – Окіл фон Неймана та Мура в тривимірній решітці

Класичні клітинні автомати загалом відповідають наступним критеріям:

– зміна значень всіх клітин відбувається одночасно після обчислення нового стану кожної клітинки решітки. В іншому випадку обчислення стану клітин решітки під час процесу генерації структури наступного стану матиме значний вплив на час отримання результат;

– сітка однорідна. Неможливо розрізнити будь-які два місця на сітці над ландшафтом. Однак на практиці сітка є кінцевим набором клітин (оскільки неможливо виділити необмежену кількість пам’яті для даних). Як наслідок, можуть виникнути прикордонні ефекти: клітини, що стоять на кордонах сітки,

будуть відрізнятися за кількістю сусідів. Щоб уникнути цього, можна ввести періодичні граничні умови;

– взаємодії місцеві. Тільки оточуючі клітини (як правило, суміжні) можуть впливати на цю клітинку;

– безліч станів комірки скінченна. Ця умова необхідна для того, щоб отримати кінцеве значення комірки для нового значення кількості операцій комірки (але це не заважає клітинам зберігати числа з плаваючою точкою для розв'язання прикладних завдань).

Клітинний автомат називається повним, якщо з будь-якого початкового стану можна привести КА у будь-яку задану конфігурацію шляхом зміни значення загального вхідного параметра.

У кожен момент часу кожен елемент КА приймає один стан з кінцевого набору станів. Залежно від цих станів, у більш пізній момент часу набір елементів може зайняти новий стан. Якщо елементи СА відрізняються кількістю можливих станів, то такий клітинний автомат називається полігенним. Але на практиці клітини з еквівалентними наборами можливих станів використовуються в алгебраїчній структурі - лінійних КА.

Елементи можуть бути геометрично розташовані різними способами. Розмір простору може бути довільним, а число елементів - як нескінченне, так і кінцеве. В останньому випадку існує додаткова міра свободи в граничних умовах. Вони можуть бути різними, але на практиці вони використовують постійні у часі (часто нульові) або періодичні граничні умови. У динамічній КА геометрія може змінюватися з часом, і якщо геометрія відрізняється в різних частинах простору, такі клітинні КА називаються неоднорідними.

Для КА визначається поняття сусідів - елементи, від яких залежить елемент автомату. Можна назвати поняття сусідства ключовим до ЦС. Крім того, сусідство розуміється не в геометричному сенсі, а в інформації. Хоча зазвичай інформаційний сенс накладається на геометричний. Сусідство

окремих автоматів задається постійним для кожної окремої граткової машини і визначається спеціальним вектором - індексом сусідства. Як правило, розглядаються d-розмірні регулярні решітки, цілі точки яких розміщуються копіями деякого автомата Мура. Стан елемента в наступний момент часу обчислюється з стану самого елемента і його сусідів. Сусідство в більшій мірі визначається геометрією КА. Для різних цілей можна змінити кількість станів введення елемента. Якщо для кожного елемента КА кількість входів і виходів однакова, це КА називається збалансованим.

Для кожного клітинного автомата визначені певні локальні правила. Відповідно до місцевого правила, стан елемента КА змінюється з часом. КА, в якому локальні правила різні для різних елементів, називається неоднорідним. Місцеве правило може бути недетермінованим, тобто змінюватися з часом або мати випадковий характер. Як правило, це правило встановлює взаємозв'язок між поточним станом елемента та його сусідів і майбутнім станом цього елемента на наступному етапі.

Основним напрямком дослідження клітинних автоматів є алгоритмічна розв'язність окремих завдань. Крім того беруться до уваги конфігурації початкових станів, в яких клітинний автомат вирішить задану проблему. Наприклад, питання про можливість побудови машини Тьюринга в грі "Життя" залишається відкритим.

1.2 Класифікація клітинних автоматів

Стівен Вольфрам у своїй книзі A New Kind of Science запропонував 4 класи, на які всі клітинні автомати можуть бути поділені в залежності від типу їх еволюції. Класифікація Вольфрама була першою спробою класифікувати самі правила, а не типи поведінки правил окремо. Нижче зазначена ця класифікація в порядку зростання складності.

Клас 1. Результатом еволюції майже всіх початкових умов є швидка стабілізація стану та його гомогенність. Будь-які випадкові конструкції в таких правилах швидко зникають.

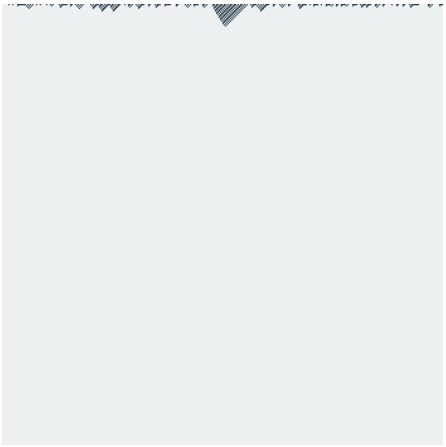

Рисунок 1.4 – Результат еволюції 1 класу автоматів на прикладі правила 40

Клас 2. Результатом еволюції майже всіх початкових умов є швидка стабілізація стану, або виникнення коливань. Більшість випадкових структур в початкових умовах швидко зникає, але деякі залишаються. Локальні зміни в початкових умовах надають локальний характер на подальший хід еволюції системи.

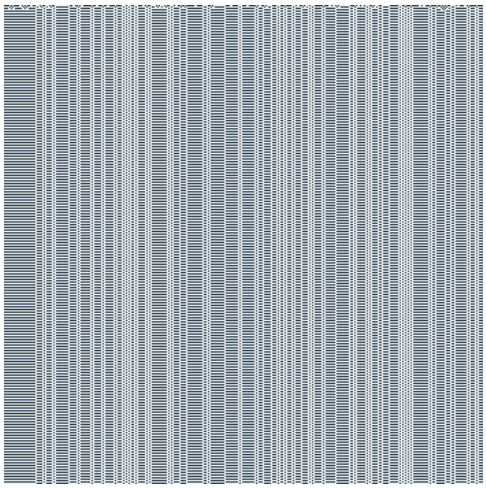

Рисунок 1.5 – Результат еволюції 2 класу автоматів на прикладі правила 33

Клас 3. Результатом еволюції майже всіх початкових умов є псевдо-випадкові, хаотичні послідовності. Будь-які стабільні структури, які виникають, майже відразу ж знищуються оточуючим їх шумом. Локальні зміни в початкових умовах надають широкий, невизначений вплив на хід усієї еволюції системи.

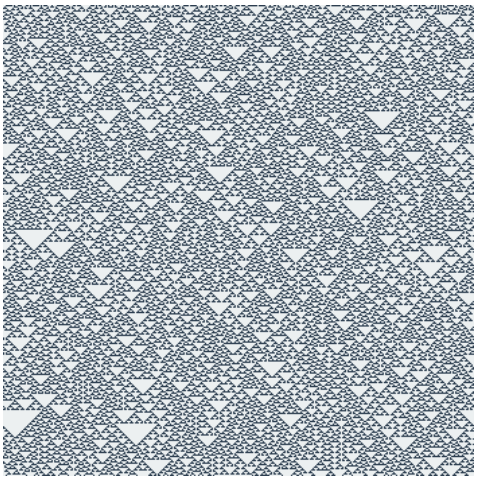

Рисунок 1.6 – Результат еволюції 3 класу автоматів на прикладі правила 22

Клас 4. Результатом еволюції майже всіх правил є структури, які взаємодіють складним і цікавим чином з формуванням локальних, стійких структур, які здатні виживати тривалий час. В результаті еволюції правил цього класу можуть виходити деякі послідовності Класу 2, описаного вище. Локальні зміни в початкових умовах надають широкий, невизначений вплив на хід усієї еволюції системи. Деякі клітинні автомати цього класу мають властивість універсальності по Тьюрингу. Останній факт був доведений для Правила 110 і гри «Життя».

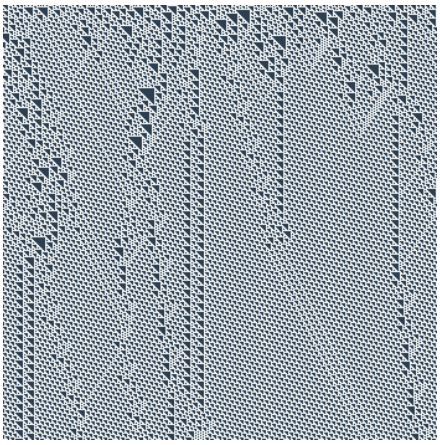

Рисунок 1.7 – Результат еволюції 4 класу автоматів на прикладі правила 193

Крім цього, клітинні автомати також класифіковані за іншими критеріями. За порядком переходу елементів в новий стан клітинні автомати поділяються на синхронні та асинхронні. У синхронних КА всі клітинки переходять у новий стан одночасно за сигналом глобального таймера. При цьому як вхідні стани використовуються старі стани сусідніх клітинок. У асинхронних КА клітинки переходять у новий стан у випадковому порядку, причому новий стан клітинки відразу може використовуватися її сусідами як вхідний.

За можливістю руху клітин автомати поділяються на рухливі та нерухомі. Рухливі КА характеризуються можливістю зміни положення клітинки в решітці під час еволюції системи. У нерухомих КА положення клітини під час еволюції залишається постійним.

За ступенем залежності від випадкових факторів автомати поділяють на детерміновані та імовірнісні. У детермінованих КА стан комірки n+1 в наступний момент часу однозначно визначається станом цієї клітинки і її найближчих сусідів у попередній момент часу. У цьому випадку стан даного елемента в момент часу n +1 є однозначною функцією F від двох змінних — стану цього елемента і суми станів його найближчих сусідів у попередній момент часу n. При такому визначенні клітинний автомат не має пам'яті. КА з

пам'яттю можна отримати, припустивши, що функція F залежить, наприклад, також від стану елемента в ще більш ранній момент часу.

КА, в яких стани комірок в наступний момент часу визначаються на основі деяких ймовірностей, називаються імовірнісними КА (ІКА). У класичних ІКА правила переходів мають абстрактний характер і не пов'язані однозначно з реальними процесами, що відбуваються в модельованій системі. У таких автоматах при моделюванні процесу для кожної клітинки датчиком випадкових чисел генерується випадкове число Q ($0 < Q < 1$), що порівнюється з імовірністю w реалізації цього процесу. Якщо $Q < w$, то процес реалізується.

Іноді використовуються правила, записані у вигляді звичайних диференціальних рівнянь (клас КА-ЗДР).[1] У цьому випадку стани комірок задаються набором змінних, значеннями яких можуть бути будь-які дійсні числа. Для таких автоматів диференціальні рівняння розв'язуються для кожної комірки окремо протягом фіксованого відрізку часу, при цьому кожна клітинка може мати різні початкові умови. Цей клас КА дуже щільно прилягає до диференціальних рівнянь в частинних похідних.

Моделі типу КА-ЗДР займають проміжний стан між КА і ІКА, а також між простими КА і ДР в частинних похідних. Основною ідеєю КА-ЗДР є розбиття модельованої області на рівновеликі комірки і розв'язання системи ЗДР незалежно в кожній клітинці з різними початковими умовами. У деяких моделях просторове розташування комірок неістотне, а в інших кількість сусідніх комірок і розмірність простору відіграють вирішальну роль (випадки поширення хвиль або виникнення стаціонарних просторових структур у нерухомому середовищі). У моделях КА-ЗДР передбачається, що клітинка містить дуже велику кількість частинок, що дозволяє застосовувати ЗДР і неперервні функції. Ця обставина залишає тільки один спосіб для моделювання дифузії, а саме просте опосередкування концентрації по сусіднім коміркам.

За структурою КА поділяють в залежності від кількості вимірів. Найбільш вживані одно- та дво-вимірні.

Як ґратки беруть поле, комірки якого є трикутники, чотирикутники чи шестикутники.

В одновимірному (лінійному) КА решітка являє собою ланцюжок клітинок (одновимірний масив), в якій для кожної з них, крім крайніх, є по два сусіди. Для усунення крайових ефектів решітка «загортається» у тор. Це дозволяє використовувати наступне співвідношення для всіх клітин автомата:

$$y'_i = f(y_{i-1}, y_i, y_{i+1}),$$

де f – функція переходів клітинки;

y'_i – стан і-ої клітинки в наступний момент часу;

y_{i-1} – стан (і-1)-ої клітинки в даний момент часу;

y_i – стан і-ої клітинки в даний момент часу;

y_{i+1} – стан (і +1)-ої клітинки в даний момент часу.

У двовимірному (площинному) КА решітка реалізується двовимірним масивом. У ній кожна клітина має вісім сусідів. Для усунення крайових ефектів решітка так само, як і в попередньому випадку, «загортається» у тор. Це дозволяє використовувати наступне співвідношення для всіх клітинок автомата:

$$y'_{i,j+1} =$$

$$f\left(y_{i,j}, y_{i-1,j}, y_{i-1,j+1}, y_{i,j+1}, y_{i+1,j+1}, y_{i+1,j}, y_{i+1,j-1}, y_{i,j-1}, y_{i-1,j-1}\right).$$

Варіюючи різні параметри, можна отримати КА необхідної конфігурації. Гнучкість конфігурації та універсальність обчислень забезпечили високу популяризацію кліткових автоматів у різних сферах. Свобода у виборі параметрів конфігурації дуже зручна для використання, але це накладає додаткову складність у класифікації та систематизації знань теорії кліткових автоматів. Тим не менше, найбільш використовуване на практиці лише невелике сімейство конфігурацій кліткових автоматів. Як правило,

кожен з них має свою назву. Наведемо невеликий список найбільш використовуваних варіантів конфігурацій.

– Мозаїчний автомат. КА, що використовує у локальному правилі кожного елемента не тільки стан елемента та його сусідів, але і значення загального вхідного параметра, який може змінюватися час від часу. Зміна цього параметра веде до перевизначення набору правил зміни станів у всьому просторі елементів КА. Якщо з будь-якого початкового стану можна привести клітковий автомат в будь-яку задану конфігурацію шляхом варіювання значення загального вхідного параметра, такий КА називають повним.

– Ітеративний автомат. КА, в якому лише один елемент використовує для зміни свого стану значення вхідного параметра.

– Односторонній клітковий автомат. Такий автомат припускає лише односторонню взаємодію елементів. Наприклад, в одновимірному масиві елементів значення кожного елемента залежить лише від його стану і від стану лівого (або правого) сусіда. Незважаючи на удавану вироджуваність звичайного КА, односторонні КА досить універсальні і використовуються для розпізнавання мовних форм.

– Л-система. Цей тип КА використовується для моделювання біологічних систем. Це динамічні КА (як правило, одно- чи двовимірні), в яких з часом один елемент може замінятися декількома або може бути видаленим із системи згідно з заданими правилами.

Відмовостійка система. У таких системах моделюється робота КА в реальних умовах: з деякою ймовірністю кожен елемент КА може перейти в стан, що не відповідає локальному правилом. Завданням є створення алгоритмів, для яких робота КА буде правильною в незалежності від допущених помилок.

Правила зміни стану КА враховують як кількість сусідів у певному стані, так і кількість конкретних станів кожного сусіда. Типи обмежень, які

встановлюються для вираження правил можна згрупувати за такими категоріями:

1.2.1 Тоталістичні клітинні автомати

Тоталістичні клітинні автомати є одним з найпростіших і найбільш широко вивчених автоматів [9].Очевидним фактом є те, що стан клітини цих автоматів визначають підсумовуванням (або середнім) значенням сусідніх комірок для визначення їх стану в наступній ітерації. Зазвичай у них є два можливі стани, живі або мертві (0,1), а правила вказані як такі: 3,4 / 5,6

Цифри ліворуч (3,4) позначають кількість живих сусідів, при яких клітина виживає на наступній ітерації, а ті що справа (5,6), - кількість живих сусідів для відродження клітини. У даному випадку, жива на даній ітерації клітина виживає, якщо вона має 3, або 4 живих сусідів, а мертва оживає, якщо вона має 5 або 6 живих сусідів. У будь-якому іншому випадку клітина гине або залишається мертвою. Застосування правила 1/1 показано на рисунку 1.8.

Рисунок 1.8 – Перші 4 ітерації тоталістичного автомату з правилом 1/1

Клітинний автомат (КА) є основаним на «Житті» (в тому сенсі, що він схожий на гру «Життя» [10]), якщо він відповідає наступним критеріям:

– решітка автомата має два виміри.

– кожна клітина автомата має два стани (умовно їх називають «живий» і "Мертвий", або альтернативно "увімкнено" і "вимкнено")

– окіл кожної клітини визначається околом Мура; тобто він складається з восьми сусідніх клітин до розглянутої і (можливо) самої клітини.

– на кожній ітерації новий стан клітини може бути виражений як функція від кількості сусідніх клітин, що знаходяться в живому стані і від стану самої клітини.

1.2.2 Основані на «Житті»

[6] Основані на «Житті» клітинні автомати є багатостановими КА і узагальнюють «Мозкові правила»

Нехай N - кількість можливих станів, N - 1 - індекс живого стану і 0 мертвого стану. Всі інші стани не розглядаються в обчисленні виживання клітини. Деякій кількості клітин призначається максимальний стан (живий) при ініціалізації, якщо клітина не виживає, на наступній ітерації її стан зменшується на 1.

Ці автомати показують, начебто, живу поведінку, тому краще спостерігаються в русі.

«Мозок Брайана» Брайана Сілвермана зображений на рисунку 1.9. Під час руху структури ковзають і взаємодіють складним та заплутаним чином.

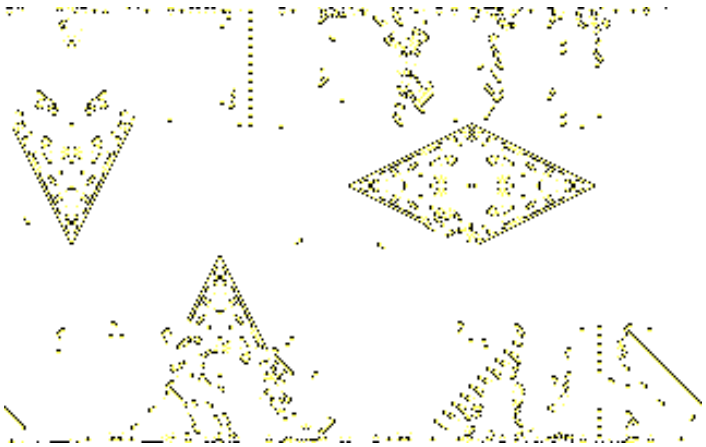

Рисунок 1.9 – Автомат «Мозок Брайана»

1.2.3 Клітинні автомати з пам’яттю

У клітинних автоматах з пам'яттю поточний стан Statei оцінюється як функція m від попередніх станів, з деяким розподілом ваги Wn.

$$State_i = \frac{\sum_{n=i-m}^{i} State_n \times w_n}{\sum_{n=i-m}^{i} w_n}$$

1.2.4 Напрямлені клітинні автомати

Клітинні автомати з спрямованістю є тоталістичними КА, але еволюційна функція має доступ до напрямних предикатів, коли оцінюються сусідні стани, наприклад, підрахунок сусідніх комірок по осі X з станом 1. Розміщення сусідів змінено відповідно до правила автомата.

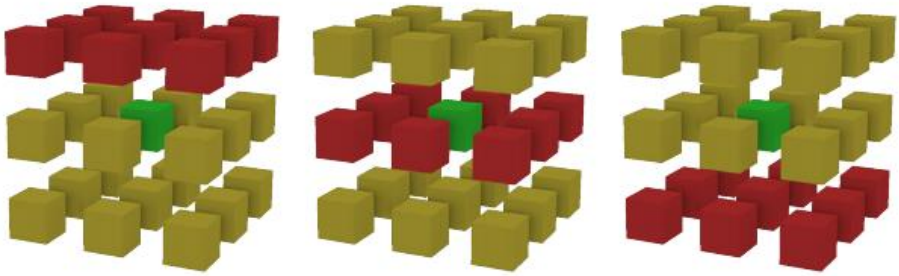

Рисунок 1.10 – Окіл клітини в напрямленому клітинному автоматі

1.3 Найпростіші клітинні автомати

1.3.1 Загальні поняття

Найпростішим нетривіальним клітинним автоматом є одновимірний клітинний автомат з двома можливими станами, а сусідами клітини будуть суміжні з нею клітини. Три клітини (центральна та її сусіди) породжують 23 = 8 комбінацій станів цих клітин. Далі на основі аналізу поточного стану трійки приймається рішення про те, чи буде центральна клітина порожньою чи повною на наступному кроці. Всього існує $2^8 = 256$ можливих правил. Ці 256 правил кодуються відповідно коду Вольфрама - стандартною угодою, правилу, яке було запропоновано Вольфрамом.[9] У деяких статтях ці 256 правил були досліджені та порівняні. Початкова умова для кожного автомату - одна центральна клітина - заповнена, решта - порожні.

Код заснований на спостереженні, що таблиця із зазначенням нового стану кожного осередку в автоматі, в залежності від станів в його околиці, може бути інтерпретована як k-значне число в S-ковій позиційній системі числення, де S - це число станів, що кожна клітинка в автоматі може мати, k = S2n + 1 - число конфігурацій околиці, а n - радіус околиці. Таким чином, код Вольфраму для конкретного правила є числом в діапазоні від 0 до $S^{S^{2n+1}} - 1$, перетворений з S-кової в десяткову систему числення. Вона може бути розрахована наступним чином:

− виписати всі S^{2n+1} можливих станів конфігурації околиці даної клітини;

− інтерпретувати кожну конфігурацію у вигляді числа, як описано вище, впорядкувати їх в порядку убування номерів;

− для кожної конфігурації вкажіть стан, який дана клітина матиме згідно з цим правилом, на наступній ітерації;

− інтерпретувати отриманий список станів знову як S-ковий номер і перетворити це число в десятковій системі. Отримане в результаті десяткове число є кодом Вольфрама.

Для прикладу розглянемо правило 110. $110_{10} = 01101110_{2.}$ Занесемо цифри двійкового представлення до таблиці:

Таблиця 1.1 — Відповідність конфігурації околу клітини на попередній ітерації стану клітини в даній ітерації

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

Залежно від станів сусіда зліва, самої клітини і сусіда праворуч (перший рядок таблиці) на наступному кроці клітина прийме одне з станів, зазначених у другому рядку.

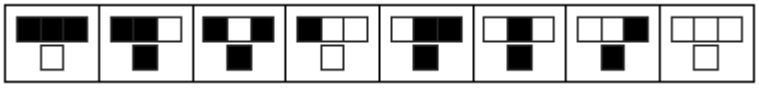

Рисунок 1.11 — Відповідність конфігурації околу клітини на попередній ітерації стану клітини в даній ітерації

1.3.2 Правило 110

Одне з найцікавіших правил. Вольфрам відносить його до класу 4, але в залежності від початкових умов воно може вести себе як представник класу 1, 2, 3 або 4. Правило 110 - елементарний одновимірний клітинний автомат з поведінкою, що знаходиться на межі хаосу і стабільності. В цьому відношенні Правило 110 ідентично грі «Життя». Відомо, що Правило 110 є Тьюринг-повним, що означає, що будь-яка обчислювальна процедура може бути реалізована за допомогою цього клітинного автомата.

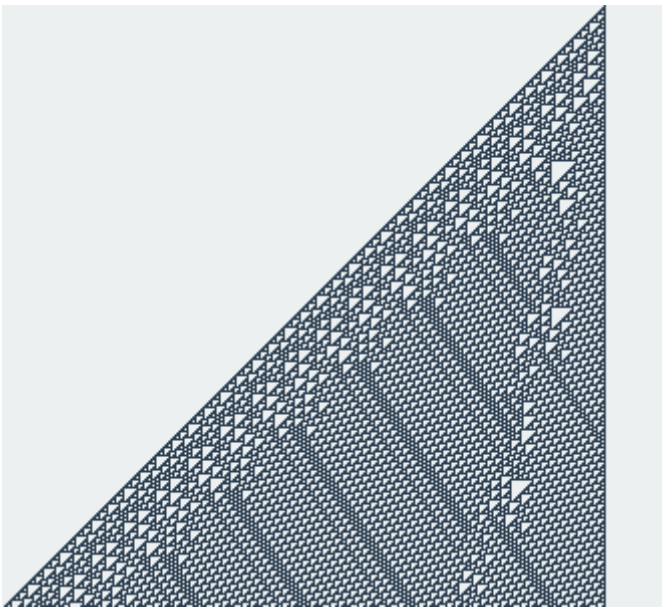

Рисунок 1.12 — Еволюція автомата за правилом 110, починаючи з однієї точки

На лівій межі трикутника є періодичні структури, у правій частині стабільний однорідний стан, а в центральній і правій частинах трикутника хаотичні структури, що переплітаються з нестійкою періодичною.

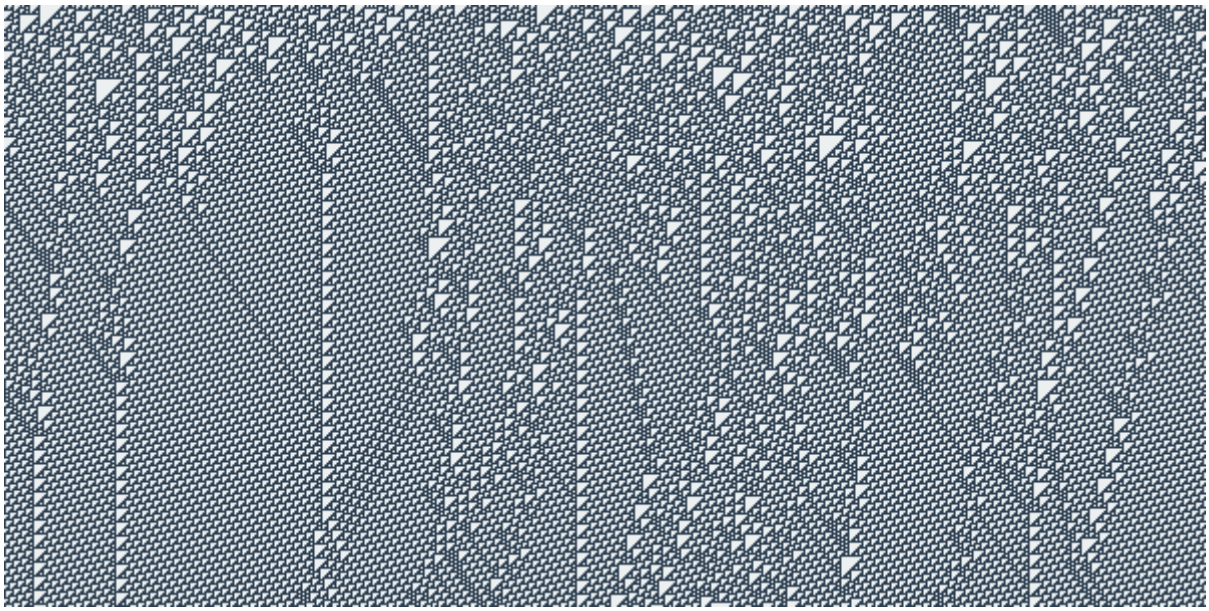

Рисунок 1.13 — Еволюція автомата за правилом 110, з першим станом, хаотично заповненим живими клітинами на 50%

Такий варіант також показує періодичні (що цікаво, з різними періодами) і хаотичні структури.

1.3.3 Фрактали

Є цілий ряд клітинних автоматів (правила 18, 22, 126, 161, 182, 218), які, розвиваючись з однієї точки, породжують фрактальні зображення. Наприклад, рисунок правила 22 - це трикутник Паскаля по модулю 2, що являється дискретним аналогом Серветки Серпінського.

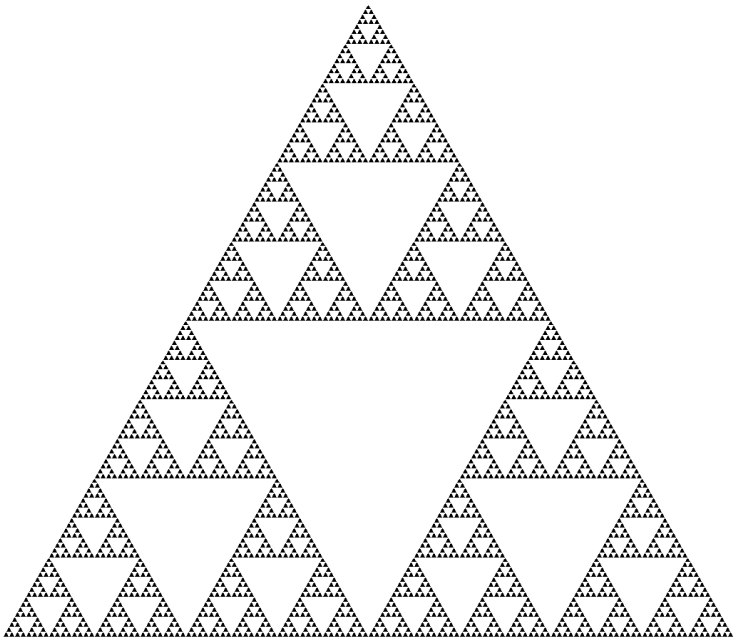

Рисунок 1.14 — Трикутник Серпінського

Можна досягти інвертованого варіанту Трикутника Серпінського, якщо реалізувати Правило 161.

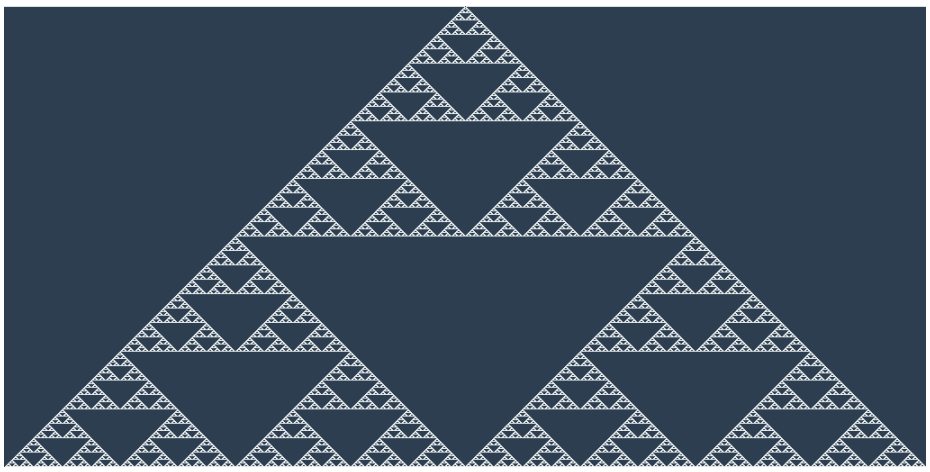

Рисунок 1.15 — Правило 161

1.4 Гра «Життя»

Гра «Життя» (англійською Conway's Game of Life) - клітинний автомат, винайдений англійським математиком Джоном Конвеєм в 1970 році.

Джон Конвей зацікавився проблемою, запропонованою в 1940-х роках відомим математиком Джоном фон Нейманом, який намагався створити

гіпотетичну машину, яка може відтворювати сама себе.[9] Джону фон Нейману вдалося створити математичну модель такої машини з дуже складними правилами. Конвей спробував спростити ідеї, запропоновані Нейманом, і врешті-решт йому вдалося створити правила, які стали правилами гри «Життя». Вперше опис цієї гри було опубліковано в жовтневому (1970 рік) випуску журналу Scientific American, в рубриці «Математичні ігри» Мартіна Гарднера (Martin Gardner).[5]

Гра “Життя” є двовимірним клітинним автоматом та, як і багато інших розглянутих автоматів, має квадратну сітку та два можливих стани клітини: “жива” та “мертва”. Існують два основних правила, що керують роботою цього клітинного автомату: умова загибелі та умова народження:

– якщо пуста (“мертва”) клітина має рівно 3 сусіда, на наступному кроці вона заповнюється (переходить в “живий” стан);

– якщо заповнена (“жива”) клітина має менше 2 чи більше 3 сусідів, вона “вмирає” (стає пустою) від “самотності” чи “перенаселення” на наступному кроці.

Обидві ці умови зав’язані на одному критерії — кількості сусідів. Вона визначається як кількість живих клітин серед восьми сусідніх (зверху, знизу, ліворуч, праворуч та в кутах). При цьому стан клітини на наступному кроці визначається станом сусідей та поточної клітини на даному кроці. Таким чином цей автомат можна віднести до синхронних.

Гра “Життя” відноситься до клітинних автоматів 4 класу, містячи складні постійні структури та хаотичне розповсюдження клітин навіть при незначних змінах. Було доведено, що цей клітинний автомат є Тьюринг-повним, тобто за допомогою нього можна реалізувати розрахунок майже будь-якої математичної операції. Його дослідженням займаються вже декілька поколінь вчених, знаходячи все складніші стійкі структури, що мають певні цікаві особливості.

Певна ділянка поля, в якій заповнені конкретні клітини, називається конфігурацією клітинного автомату. Далі буде розглянуто різновиди цих конфігурацій та стійких структур, а також назви, які їм були дані в процесі дослідження.

1.5 Конфігурації клітинних автоматів

1.5.1 Натюрморти

Натюрморти - конфігурації «Життя» або іншого клітинного автомата, які не змінюються в процесі еволюції. Іншими словами, натюрморт є осциллятором періоду 1 (осциллятори будуть розглянуті пізніше).[5]

Існує кілька близьких за змістом термінів, що позначають конфігурації, що не змінюються в процесі еволюції (конфігурації, які є власними батьками). Відмінності між ними пов'язані з відповіддю на питання, зазначені нижче.

– Чи вважається натюрмортом конфігурація, що складається з двох незалежних натюрмтортов (наприклад, двох блоків на досить великій відстані один від одного)?

– Чи вважається натюрмортом конфігурація, що складається з двох частин, кожну з яких можна видалити так, що друга частина залишиться батьком себе?

В існуючих словниках і онлайн-енциклопедіях наводяться наступні визначення:

– стійкий зразок (англ. Stable pattern) - об'єкт, який є власним батьком;

– натюрморт (англ. Still life, strict still life) - стійкий об'єкт, який є кінцевим і непустим, який не може бути розділений на дві стійкі частини;

– псевдонатюрморт (англ. Pseudo still life) - стійкий об'єкт, який не є натюрмортом, в якому присутня хоча б одна мертва клітина, що має більше трьох сусідів загалом, але менше трьох сусідів у кожному зі складових об'єкт натюрмортів.

Точне визначення «стійкості» представляє інтерес в контексті перерахування натюрмортів: наприклад, згідно з наведеними визначеннями, кількість стійких конфігурацій розміру 8 (тобто складаються з 8 живих клітин) в «Житті» нескінченно, тому що пара блоків на будь-якій відстані один від одного є стійкою; проте, кількість натюрмортів обмеженого розміру вважається кінцевою. Наведемо два приклади.

Рисунок 1.16 — Псевдонатюрморт та “строгий” натюрморт

Ліворуч на рисунку 1.16 - псевдонатюрморт в «Життя». Видалення одного з островів не впливає на стабільність другого острова.

Праворуч - «строгий» натюрморт. Стабільність кожного з островів залежить від наявності іншого острова.

Дослідниками встановлено число натюрмортів і псевдонатюрмортів розміру не вище 24 клітин. Завдання визначення типу стійкої конфігурації (натюрморт, псевдонатюрморт) вирішується за поліноміальний час шляхом пошуку циклів в пов'язаному кососиметричному графі. Далі розглянемо конкретні натюрморти.

Найбільш поширений натюрморт - блок - конфігурація у формі квадрата 2 × 2. Два блоки, розміщені в прямокутнику 2 × 5, утворюють бі-блок - найпростіший псевдонатюрморт. Блоки використовуються в якості складових частин у безлічі складних пристроїв, наприклад, в планерній рушниці Госпера. На рисунку 1.17 ліворуч зображено блок, праворуч — бі-блок.

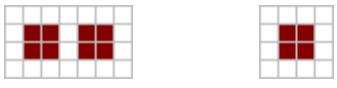

Рисунок 1.17 — Блок та бі-блок

Другий за поширеністю натюрморт - вулик. Вулики часто виникають четвірками в конфігурації, що називають пасікою. Вулик та пасіку зображено на рисунку 1.18 ліворуч та праворуч відповідно.

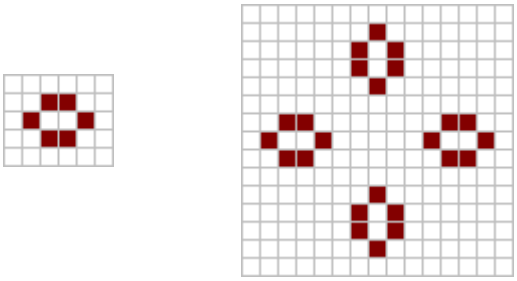

Рисунок 1.18 — Вулик та пасіка

Існує велика кількість інших відомих натюрмортів, що відрізняються за розміром та частотою появи. Деякі з них зображені на рисунках 1.19 та 1.20.

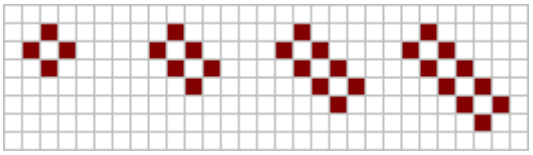

Рисунок 1.19 — Ящик, баржа та подовжені баржи

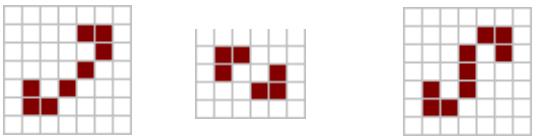

Рисунок 1.20 — Приклади інших натюрмортів гри “Життя”

Існують й особливі різновиди натюрмортів, такі як пожирач та відбивач. Вони будуть розглянуті пізніше.

1.5.2 Осцилятори

Осцилятор - конфігурація клітинного автомата, яка після кінцевого числа поколінь повторюється в тій же самій орієнтації і позиції. Іншими словами, осцилятор - це будь-який зразок, який є попередником самого себе. Еволюція осцилятора триває як завгодно довго.[7]

Залежно від контексту, космічні кораблі також можуть вважатися осцилляторами, але зазвичай вони розглядаються окремо. Мінімальна кількість поколінь, через яке осцилятор повертається у вихідну конфігурацію, називається періодом осцилятора. Осцилятор з періодом 1 зазвичай називається стійкою конфігурацією, так як він не змінюється. Таким чином, натюрморти є підмножиною осциляторів.

В «Житті» кінцеві осцилятори відомі для всіх періодів, крім 19, 23, 38 і 41. Хоча існують осцилятори періоду 34, всі відомі приклади вважаються тривіальними, оскільки вони складаються з по суті окремих компонент, осцилюючих з меншими періодами. Наприклад, осцилятор з періодом 34 можна отримати шляхом розміщення у всесвіті двох незалежних осциляторів з періодами 2 і 17. Осцилятор вважається нетривіальним, якщо він містить хоча б одну клітку, період осциляції якої дорівнює періоду осцилятора.

Наведемо приклади осциляторів. Найпоширенішим осцилятором є “блимавка”, та вона має період 2.

Рисунок 1.21 — Блимавка

Серед інших осциляторів можна виділити крест – осцилятор з періодом

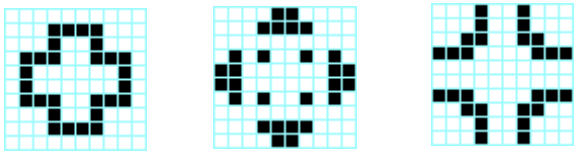

Рисунок 1.22 — Крест

Прикладом осцилятору с більшим періодом може слугувати, зокрема, фумарола. Її період складає 5 кроків.

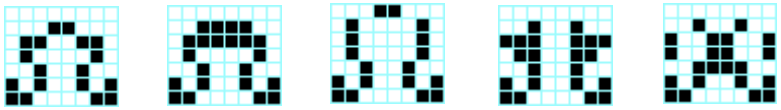

Рисунок 1.23 — Фумарола

Знайдено й велику кількість інших осциляторів (наприклад, галактика Кока з періодом 8 чи пентадекатлон з періодом 15). Зокрема, для створення нового осцилятору достатньо об’єднати два вже відомих осциляторів в одну конфігурацію. Таким чином, можна зробити висновок, що на нескінченному полі кількість можливих осциляторів також нескінченна.

Осцилятори є прикладом більш складної стабільної структури, яка зазвичай відповідає 2 класу клітинних автоматів, проте здатна існувати у клітинному автоматі, що проявляє ознаки хаотичності.

1.5.3 Космічні кораблі

Конфігурація «Життя» або іншого клітинного автомату називається космічним кораблем, якщо через певну кількість поколінь вона знову з'являється без доповнень або втрат, але зі зміщенням відносно вихідного положення. Найменше таке число поколінь - період космічного корабля.[5]

Першим виявленим космічним кораблем став планер. Планер був знайдений під час відстеження еволюції R-пентаміно (стартової комбінації з п’яти сусідніх заповнених клітин, що розпочинає найдовшу еволюцію серед усіх можливих таких комбінацій) в 1970 році Річардом Гаєм. Еволюцію планеру можна побачити на рисунку 1.24.

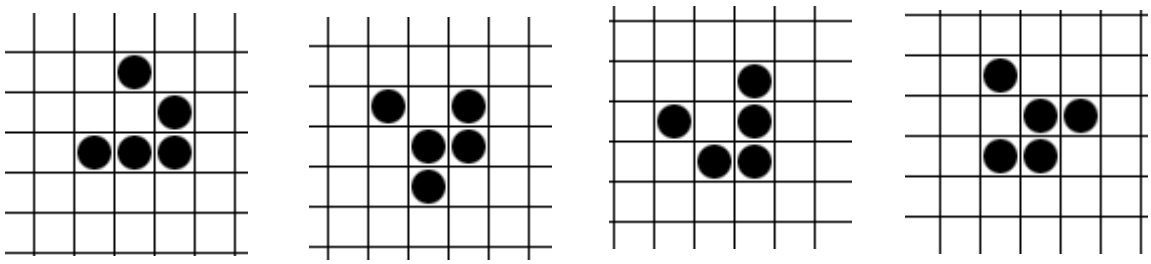

Рисунок 1.24 — Планер

Космічний корабель періоду p, Який переміщується на (m, n) протягом його періоду, де m ≥ n, має тип (m, n) / p. Як було доведено Конуеєм в 1970 році, p ≥ 2m + 2n.

Космічні кораблі, що рухаються по горизонталі або вертикалі, називаються ортогональними кораблями. Якщо рух космічного корабля відбувається по діагоналі під кутом 45 °, такий корабель називається діагональними. Космічні кораблі, що рухаються під іншими кутами, називаються косими або наклонними. У 2010 році був сконструйований перший наклонний космічний корабель типу (5120, 1024) / 33699586.

Швидкістю світла в заданому клітинному автоматі називають найбільшу швидкість поширення інформації. Швидкість світла в «Житті» дорівнює швидкості переміщення шахового короля - швидкості в одну клітку за покоління по горизонталі, вертикалі або діагоналі. Зазвичай швидкість світла позначається літерою c.

Швидкість космічного корабля визначається відношенням відстані зміщення до періоду. Часто швидкість виражається через c. Так, швидкість планера в «Життя» дорівнює c / 4, так як він переміщується на одну клітину по діагоналі за чотири покоління. Найпростіший ортогональний космічний корабель, ВКК, рухається зі швидкістю c / 2.

У загальному випадку, якщо космічний корабель в двовимірному клітинному автоматі на квадратній сітці переміщується на вектор (x, y) через n поколінь, його швидкість дорівнює v = c * max (| x |, | y |) / n.

Найбільш поширеними різновидами космчних кораблів, окрім планерів, є легкі, середні та важкі космічні кораблі, зображені на рисунку 1.25 (починаючи з лівого краю).

Рисунок 1.25 — Космічні кораблі

Крім цього, існує велика кількість більших космічних кораблів, як ортогональних, так і діагональних. Наприклад, ортогональний космічний корабель, виявлений Тімом Коу 11 червня 2016 року, що називається “макаронний монстр” - це перший космічний корабель швидкості 3c / 7. Він зображений на рисунку 1.26.

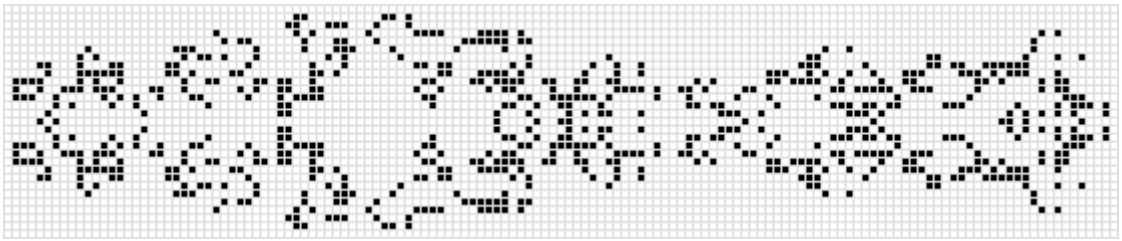

Рисунок 1.26 — “Макаронний монстр”

Конфігурація, що здатна змінювати напрямок руху космічного корабля, не руйнуючись при цьому, називається рефлектором. Конфігурацію, що здатна знищити космічний корабель і відновитися після реакції, називають пожирачем.

Рушниця - нерухома конфігурація, періодично випускає космічний корабель. Найвідоміша та перша знайдена рушниця — планерна рушниця Госпера.

Рисунок 1.27 — Планерна рушниця Госпера

Космічні кораблі можуть слугувати засобом передачі інформації. Здатність планерів передавати інформацію була основою для твердження, що гра “Життя” є Т’юринг-повною. Деякі космічні кораблі здатні на більше, ніж просто рухатись у просторі, зберігаючи свою структуру: вони здатні нескінченно створювати натюрморти чи інші кораблі в процесі руху, або навіть створювати рушниці, таким чином швидкість їх зростання прискорюється з часом. Такі складні структури і стали причиною, чому гра “Життя” досі цікавить математиків по всьому світу, а нові конфігурації з унікальними характеристиками знаходяться навіть після десятків років з винаходу цього клітинного автомату.

Існує й велика кількість варіацій гри “Життя”, в якій змінюється кількість сусідів, що необхідна для народження клітини, або її загибелі від самотності чи перенаселення. Також може змінюватись структура самого поля або його розмірність, додаватись випадкові елементи тощо.

1.6 Аналіз існуючих рішень

1.6.1 Аналіз реалізацій двовимірних автоматів

Існує велика кількість програмних реалізацій різних клітинних автоматів. Найбільш поширені реалізації автоматів, таких як мурашник Ленгтона і гра «Життя». Реалізація цих автоматів у вигляді веб-сторінки на мові JavaScript особливо поширена. Зокрема, прикладом реалізації мурашок Лангтона є реалізація Фабріса Вайнберга. Він складається з веб-сторінки, на якій розміщено вихідний код програми та її фактичного виконання у веб-переглядачі. Програма дозволяє запускати класичну версію мурашок, а також розширені опції з великою кількістю станів. У той же час, користувачеві надається можливість будь-якої з раніше заданих опцій, що мають певні цікаві можливості, або встановити правило власною рукою, що дозволяє реалізувати будь-який варіант мурашок Ленгтона на квадратному полі. Реалізація Вайнберга також дозволяє вибрати швидкість, з якою мураха рухається

автоматично, враховуючи більш незначні кроки або загальну картину. Рисунки 1.28-1.29 показують кілька скріншотів цієї програми.

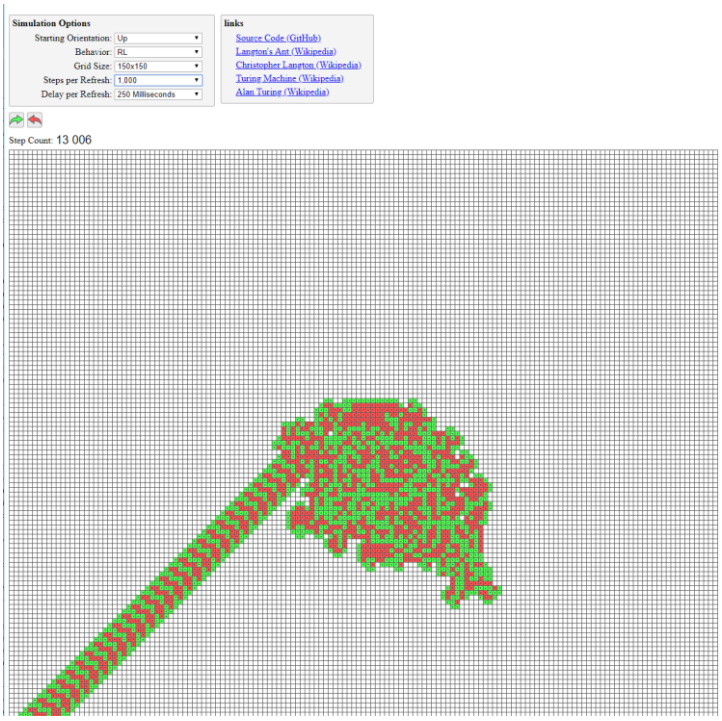

Рисунок 1.28 — Результат еволюції мурашника Ленгтона за правилом RL

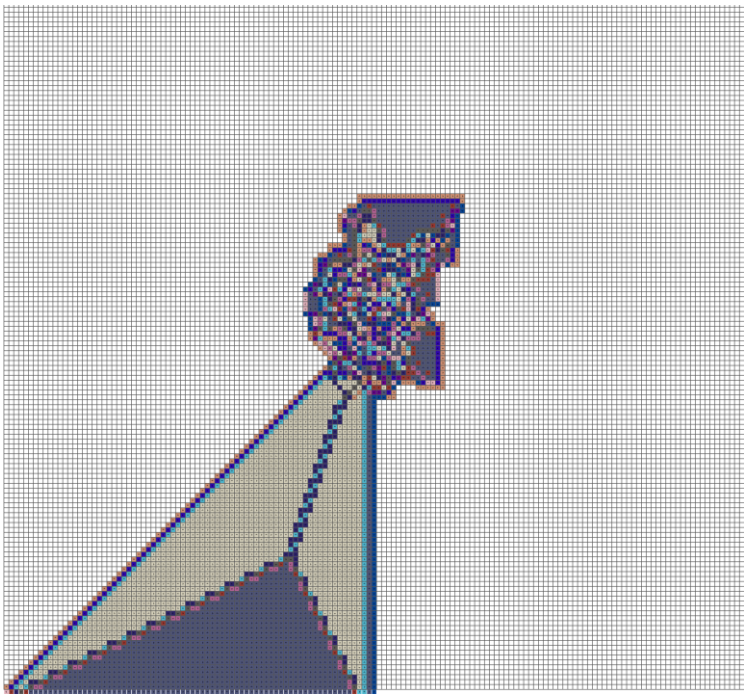

Рисунок 1.29 — Результат еволюції мурашника Ленгтона за правилом RRLLLRLLLRRR

У випадку ж гри “Життя” різноманітних реалізацій ще більше. Гру “Життя” програмно реалізовували ще на старих комп’ютерах 70-х років, проте зараз можливо створити реалізацію зі зручним інтерфейсом та додатковими можливостями. Одним з прикладів таких програм може слугувати реалізація на веб-сайті michurin.ru, що становить з себе JavaScript варіант автомату з можливістю налаштування розміру поля, швидкості переходу до нового стану, покроковому огляду еволюції, а також можливістю обрати одну з заздалегідь створених конфігурацій, серед яких велика кількість натюрмортів, осциляторів та космічних кораблів, а також рушниця Госпера. Крім того, як і багато інших реалізацій автомату, він дозволяє змінювати стан клітин під час виконання, вносячи корективи в роботу автомату чи, наприклад, продовжуючи еволюцію, що вже стала тривіальною.

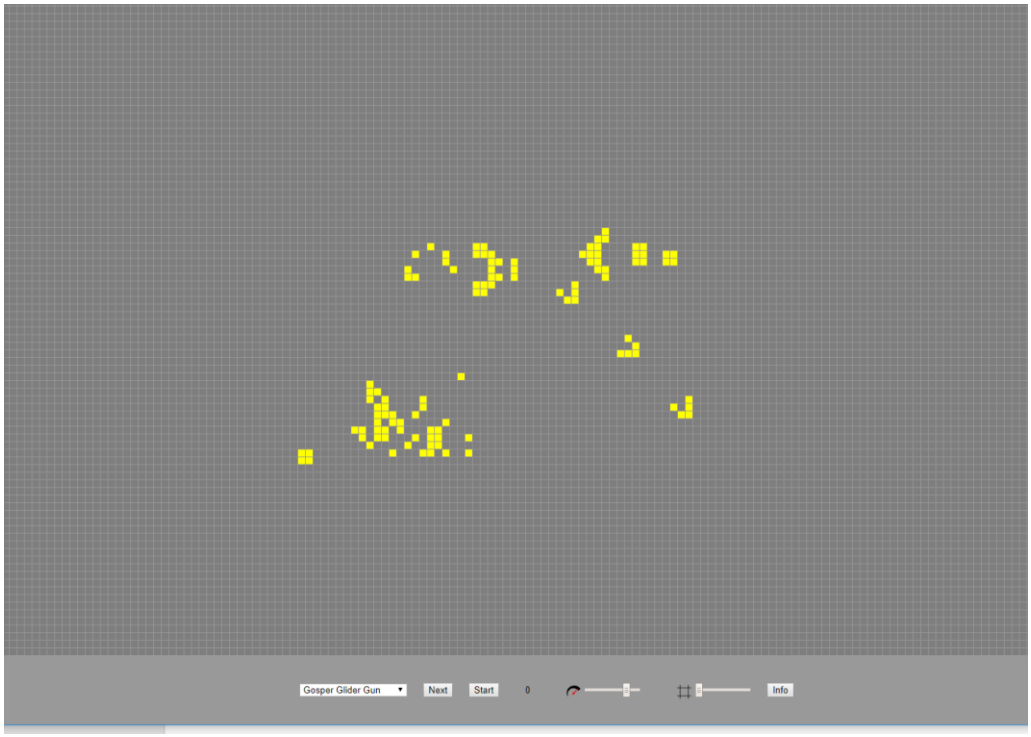

Рисунок 1.30 – Результат еволюції гри «Життя» з випадковими початковими умовами та рушницею Госпера зверху

Ще одним яскравим прикладом реалізації гри “Життя” може слугувати веб-сторінка на сайті pmav.eu. Ця реалізація не має певних можливостей, що присутні у попередній, проте має свої переваги. До недоліків реалізації варто

віднести відсутність можливості налаштовувати розмір поля та швидкість переходу до нового стану (хоча ці дані й зазначаються на самій сторінці). Також варто відзначити значно менший набір конфігурацій, які можна швидко задати за допомогою інтерфейсу. Однак дана програма все ж має свої переваги; зокрема, в ній є можливість змінювати кольорову палітру відображення, зовнішній вигляд сітки, що розмежовує клітини, а також дозволяє відображати клітини, що раніше вже були живими, але на даний момент мертві, особливим кольором. Таким чином, цей застосунок більш підходить для розгляду розповсюдження конфігурації в процесі своєї еволюції, надаючи можливість отримати більш широке розуміння цього процесу.

На рисунку 1.31 зображено кінцевий результат еволюції одного з шаблонів, наданих в даній реалізації. Салатовим кольором позначені клітини, що зараз є пустими, проте були заповнені в попередніх поколіннях.

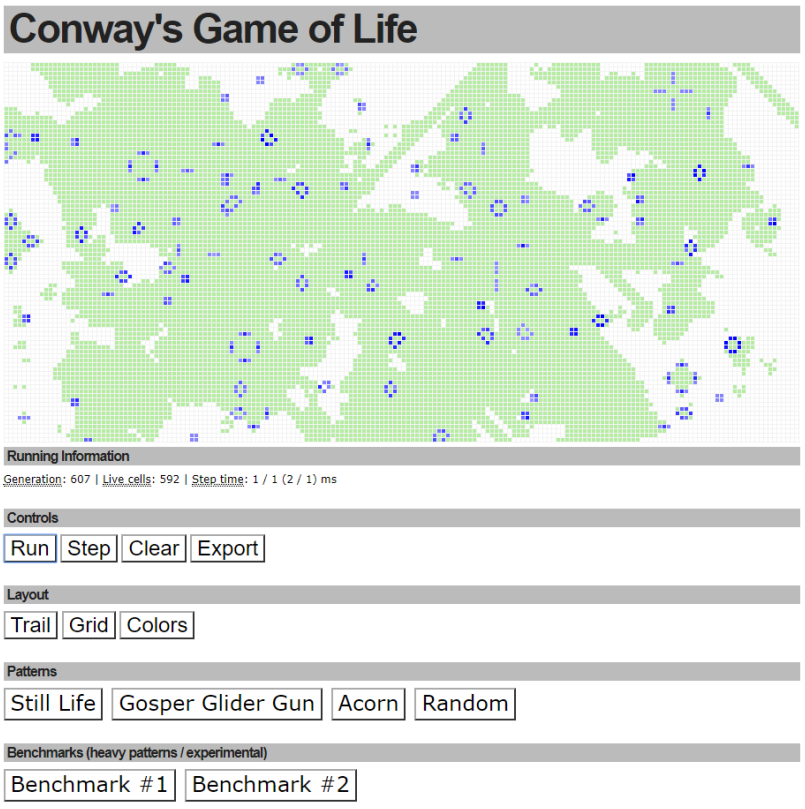

Рисунок 1.31 – Результат еволюції гри «Життя» з випадковими початковими умовами

Висновок до розділу

В даному розділі було:

− проаналізовано існуючі класифікації, правила генерації та візуалізації двовимірних і тривимірних клітинних автоматів;

− вивчено найбільш поширені реалізації двовимірних та тривимірних клітинних автоматів, зокрема гру «Життя» та мураху Ленгтона

− детально розглянуто існуючі засоби для дослідження клітинних автоматів.

В результаті було встановлено, що більшість існуючих програмних засобів для клітинних автоматів не відповідають встановленим вимогам або взагалі не пристосовані для роботи із тривимірними автоматами.

На основі проведеного дослідження виявлена потреба в розробці нових методів дослідження еволюції тривимірних клітинних автоматів під впливом зовнішніх факторів та розробці нового програмного забезпечення, що реалізує увесь необхідний функціонал та надасть зручний інтерфейс для роботи з ним.

2 МАТЕМАТИЧНЕ ОБГРУНТУВАННЯ

2.1 Основна ідея роботи

Тривимірні клітинні автомати є галуззю, в якій створено значну кількість програмних реалізацій, проте цей різновид клітинних автоматів все ж доволі слабо досліджений. Порівняно з двовимірними клітинними автоматами, тривимірні дозволяють реалізовувати набагато більшу множину правил, при цьому певні стійкі структури є складнішими за їх двовимірні аналоги, адже вони мають бути стабільними в трьох вимірах замість двох. Через це пошук набору правил, що надасть клітинному автомату властивості двовимірних аналогів, зазначених у попередньому розділі, ускладнюється. Існуючі реалізації тривимірних клітинних автоматів не надають достатнього функціоналу для їх дослідження та пошуку автоматів 4 класу, що виказують ознаки хаотичності та формування стабільних структур.

З цієї причини виникає необхідність в створенні нового програмного застосунку, здатного задовільнити потреби, що виникають при розгляді тривимірних клітинних автоматів. Зазначимо, яких саме цілей необхідно досягти при розробці даного застосунку.

Основною метою роботи програми є обрахунок еволюції тривимірних клітинних автоматів. У відриві від способу зображення клітинного автомату, обрахунок еволюції становить з себе процес визначення, які клітини мають загинути, а які — зародитись на наступній ітерації. Зазвичай для визначення наступного стану автомату використовуються чітко визначені правила, що у даному випадку базуються на кількості заповнених сусідів відповідної клітини. Так як розроблюваний застосунок підтримує ймовірнісні клітинні автомати, то клітина може зародитись, якщо вона має кількість сусідів, що дорівнює випадково визначеному для конкретного випадку числу, яке знаходиться в заздалегідь визначеному діапазоні. Границі загибелі клітини від самотності та перенаселення також визначаються випадково в кожному окремому випадку на основі заданих меж значень. Окрім цього, автомат може

мати різний розмір, хоча й завжди матиме лише кубічну форму. Також важливо коректно обраховувати наступну ітерацію: зміни, що відбуваються при переході до наступної ітерації, не мають впливати на кінцевий результат, адже клітини, які змінять свій стан, визначаються лише на основі попереднього стану. Підтримка даного функціоналу є основною для застосунку, проте й найпростішою в розробці: загалом цей функціонал реалізується чітко заданими простими алгоритмами переходу до наступного стану.

Даний застосунок має підтримувати коректне відображення процесу еволюції тривимірного клітинного автомату. Одною з основних функціональних можливостей застосунку, який потребується для досліждення, є можливість відображення як поточного стану автомату, так і процесу переходу до наступного стану. Основною проблемою в даному випадку є той факт, що, на відміну від двовимірних автоматів, відобразити на екрані відразу весь тривимірний автомат немає можливості. Тому необхідно вирішити проблему його відображення таким чином, щоб залишити можливість цілком розглянути структуру клітинного автомату незалежно від ітерації. При цьому велика кількість клітин з декількома можливими станами може ускладнити розуміння структури поточної ітерації та її зміни при переході до наступної ітерації. Тому необхідно забезпечити таку форму відображення, при якій окремі клітини можна чітко розрізнити, навіть при великій їх кількості, а перехід до наступного кроку дає чітке представлення, яка частина клітин змінила свій стан.

Розпізнавання кінцевого стану клітинного автомату є проблемою, що постає не лише перед тривимірними, але й перед двовимірними клітинними автоматами. Вона полягає в тому, що клітинні автомати 4 класу часто припиняють свою еволюцію, переходячи до стабільного стану, що містить лише стійкі структури (наприклад, натюрморти та осцилятори у грі “Життя”). Проте зазвичай це не означає, що стан клітин автомату більше не змінюється.

Зазвичай еволюція стає тривіальною, коли всі зміни відбуваються лише серед циклічних структур, що повторюються через певну кількість ітерацій. При формуванні великої кількості подібних структур повне повторення стану автомату може відбутись через декілька сотень ітерацій чи навіть більше. Тому розпізнавання цього етапу еволюції є складною задачею — необхідно порівняти поточний стан клітинного автомату з усіма попередніми ітераціями, адже якщо стан автомату повністю повторює один з минулих станів — це є головною ознакою зациклення. Застосунок повинен вирішувати цю задачу методом, що дозволить обчислювати тисячі ітерацій та перевіряти нетривіальність еволюції навіть для великих обмежених клітинних автоматів.

Необхідно надати користувачу можливість змінювання стану клітин автомату в реальному часі. Однією з основних проблем програмних реалізацій тривимірних клітинних автоматів є складність їх задання за допомогою графічного інтерфейсу. Навіть якщо проблема відображення тривимірного автомату вже вирішена, можливість змінювання стану клітин у трьох вимірах потребує особливого підходу.

Застосунок повинен надавати можливість зберігати поточний стан клітинного автомату та завантажувати його в разі необхідності. Подібний функціонал дозволить зберігати знайдені стабільні структури чи конфігурації, здатні до довготривалої еволюції, що значно перевищує складністю початковий стан.

Розглянемо детальніше, яким саме чином планується вирішити вищезазначені проблеми.

2.2 Зображення стану автомату

Відображення поточного стану тривимірного автомату зазвичай виконується за допомогою тривимірних моделей, що зображають окремі клітини. Зокрема, у випадку клітинного автомату з двома можливими станами клітини, наявність в певній позиції моделі вказує на “живий” стан відповідної

клітини, а відсутність моделі — на “мертвий” стан. Розповсюдженим способом зображення заповненої клітини є звичайна модель кубу з певною текстурою та шейдерами для більш деталізованого відображення стану автомату.

При подібному способі відображення стану виникає необхідність прийняти певні рішення, що впливатимуть на те, наскільки легко буде розрізнити окремі клітини та загалом зрозуміти структуру автомату. Зазвичай доцільно розміщувати моделі впритул один до одного, таким чином представляючи неперервну послідовність клітин. Наявність між клітинами проміжку може спростити розпізнавання меж клітин, проте ускладнить розгляд пустих клітин, адже через наявність цього проміжку зрозуміти, скільки пустих клітин знаходиться між двома заповненими, становиться складніше. Ще одним можливим варіантом є явне зображення сітки навіть між пустими клітинами. Це остаточно вирішує проблему з підрахунком кількості пустих клітин між двома заповненими, однак створює іншу проблему: через велику кількість зайвих ліній розглядати структуру клітин, що знаходяться далеко від краю автомату, становиться набагато складніше. Через це подібний підхід не є доцільним. Проте для чіткого розпізнавання меж заповнених клітин ребра кубу відмічені чорним кольором, що дозволяє не використовувати проміжки між клітинами. На рисунку 2.1 можна побачити приклад подібної системи відображення:

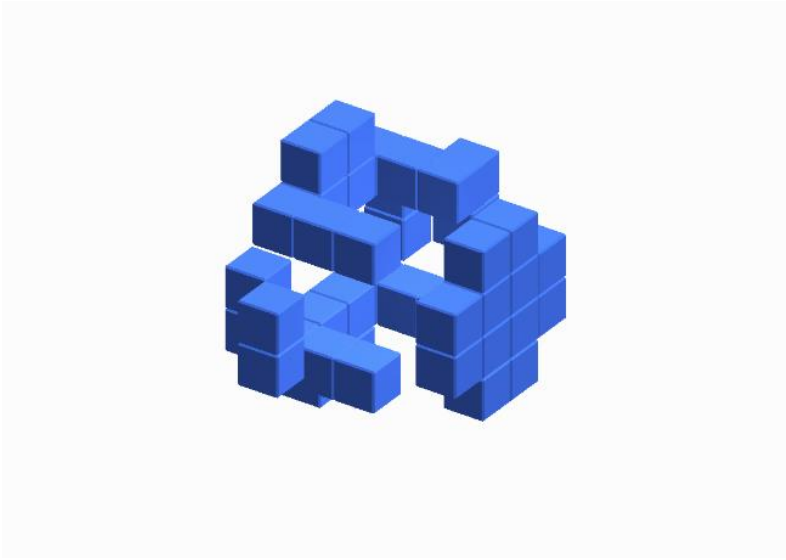

Рисунок 2.1 — Приклад тривимірного відображення клітинного автомату

Однак для реалізації повноцінного засобу розгляду структури тривимірних автоматів недостатньо лише задати тривимірну модель його стану. Дуже важливо також надати засоби зміни точки огляду. Вочевидь, для детального огляду стану автомату необхідна можливість сповільнювати та пришвидшувати, а також зовсім зупиняти процес переходу до наступної ітерації. Проте незалежно від швидкості зміни ітерацій функціонал для огляду автомату з будь-якого кута є обов’язковим. Найпростіший спосіб реалізації цього функціоналу — повернення камери навколо центру клітинного автомату. Наближення та віддалення зображення зручно контролювати за допомогою коліщатка миші. Повернення ж зображення можна реалізувати за допомогою руху миші. Можливим варіантом є автоматичне повернення за кутами Ейлера при руху миші, але при цьому виникає проблема використання інших елементів інтерфейсу: для натискання певної кнопки чи інтеракції з іншими елементами інтерфейсу необхідно рухати мишею, проте це викликатиме поворот кута огляду, що ускладнить роботу з програмою. Тому більш доцільною системою є поворот у вказаному напрямку при затисканні лівої кнопки миші. Це дозволить вільно повертати зображення в будь-якому напрямку інтуїтивно зрозумілими рухами, при цьому не заважаючи

користуватись інтерфейсом програми. У призупиненому стані це дозволить детально розгледіти поточну ітерацію, а під час роботи автомату дозволить побачити процес переходу до нового стану з кута, який можна динамічно обирати в процесі роботи.

Проте навіть такі засоби можуть виявитись недостатніми для візуалізації внутрішньої структури автомату. При великій кількості заповнених клітин зовнішні клітини можуть закрити огляд внутрішніх, при цьому зміна кута огляду не допоможе. Тому важливим функціоналом, який потребує застосунок, є можливість відобразити лише один шар структури автомату. Найбільш зручним варіантом реалізації цієї функції є можливість обрати один з чотирьох форматів відображення: тривимірний та по одному формату на кожну можливу ортогональну площину відображення (вісі X/Y, X/Z та Y/Z відповідно). При відображенні в одній з ортогональних площин усі клітини стають невидимі, окрім одного шару клітин у відповідній площині. При цьому камеру необхідно розташувати перпендикулярно площині, що розглядається, таким чином формуючи псевдодвовимірне відображення автомату. Вибір конкретного шару для відображення можна реалізувати за допомогою повзунка, при цьому перехід від одного шару до іншого дозволятиметься робити динамічно, навіть під час роботи автомату. Завдяки цьому застосунок дозволить розглянути структуру автомату будь-якої складності у зручному форматі, у тому числі динамічно під час роботи автомату. Приклад відображення окремого шару автомату зображений на рисунку 2.2.

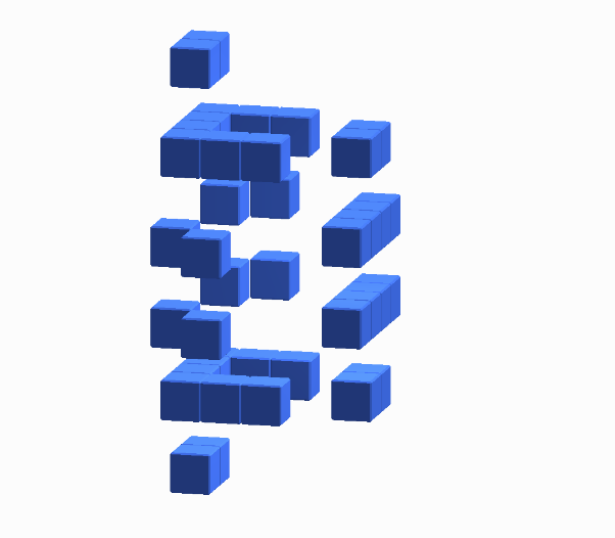

Рисунок 2.2 — Приклад відображення автомату, розбитого на шари

Окрім вищезазначеного функціоналу, програма також підтримуватиме можливість відображення клітин автомату різними кольорами. Детальніше про те, яким чином клітинам буде задаватись необхідний колір, описано нижче. Подібну можливість можна використовувати декількома способами. Найтривіальнішим з них є помічення важливих клітин. У випадку використання кольорової ідентифікації клітин окремі клітини чи навіть регіони можна відобразити певним кольором. Це дозволить чітко бачити дану клітину чи регіон навіть на фоні сотень чи тисяч інших клітин. Наприклад, користувача цікавить стан конкретної клітини в процесі еволюції конфігурації. Він може позначити цю клітину окремим кольором, щоб чітко вирізняти її на фоні інших клітин. Далі користувач може розпочати еволюцію системи. Якщо розгляду клітини заважають інші клітини, доцільним є використання відображення лише шару, який містить цю клітину. Позначення ж цілого регіону одним кольором може бути використано для відстеження, чи заповнена хоч одна клітина з цього регіону, або при розгляді конкретної ділянки еволюції системи.

Іншим варіантом використання системи кольорової ідентифікації клітин є наслідування кольору при переході до наступного стану клітинного автомату. При застосування цього режиму якщо зароджується певна клітина

при переході до наступної ітерації, то колір даної клітини визначається як середнє значення кольорів клітин-сусідів із випадковим зміщенням. Початкове задання кольору клітин можна провести до запуску еволюції. Після цього початкова кольорова матриця розповсюдиться автоматом, створюючи фрактальні кольорові візерунки. Це можна використовувати для формування тривимірних графічних структур, що будуть становити з себе випадковий результат еволюції початкової конфігурації. Також подібну кольорову систему можна використовувати для імітації різноманітних тривимірних процесів, наприклад, змішування різнорідних речовин. У цьому випадку кожна речовина буде позначена окремим кольором, а клітини проміжних кольорів відображатимуть пропорції та ступінь змішування речовин у даному регіоні.

Наведемо приклад структури, яку можна отримати шляхом використання кольорової диференціації клітин, на рисунку 2.3.

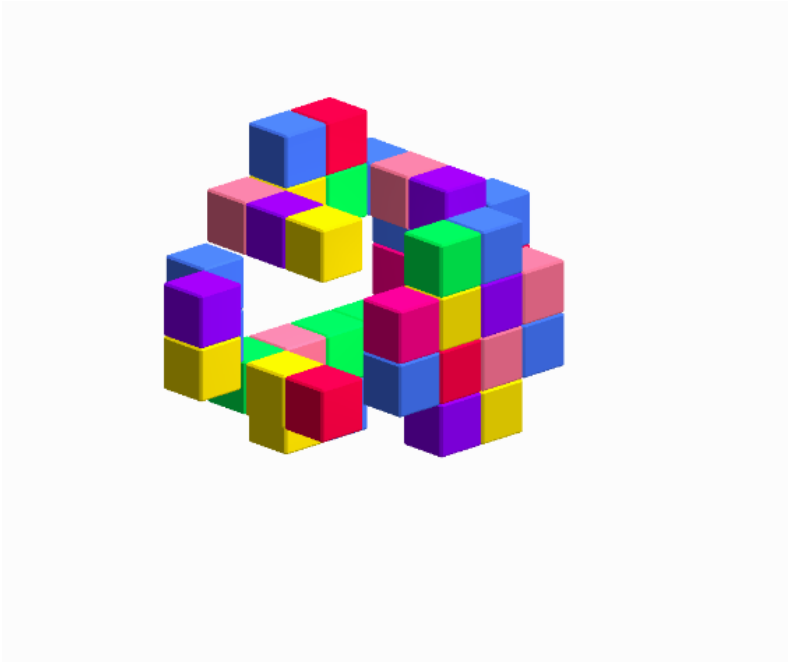

Рисунок 2.3 — Приклад структури із кольоровим маркуванням клітин

2.3 Розпізнавання переходу автомату до тривіального стану

При дослідженні властивостей клітинного автомату одним з критеріїв оцінки може слугувати час еволюції конфігурації, тобто кількість ітерацій від

початкової до переходу в тривіальний стан. Тривіальним станом зазвичай називають стан, при якому подальша еволюція системи є легко передбачуваною. Наприклад, автомат знаходиться в тривіальному стані, якщо він містить лише стабільні структури, що не змінюються з часом (“натюрморти” в термінах гри “Життя”), циклічні структури (“осцилятори”) та циклічні структури, що рухаються в певному напрямку (“космічні кораблі”). Проблема визначення тривіального стану має бути вирішена для надання функціоналу з підрахунку часу еволюції конфігурації, що дозволить детальніше дослідити особливості тривимірного клітинного автомату.

Навіть для детермінованого автомату реалізація подібного функціоналу породжує певні проблеми. Найпростішим способом перевірки тривіальності системи є перевірка, чи змінився стан хоча б однієї клітини автомату. Якщо цього не сталося, це означає, що автомат або пустий, або складається лише з натюрмортів. Проте подібний підхід не дозволяє визначити перехід до тривіального стану, що містить циклічні структури. Тому цей метод необхідно розширити, порівнюючи поточний стан з усіма попередніми станами автомату. Якщо поточний стан повністю відповідає одному з попередніх, то у випадку детермінованого автомату це означатиме зациклювання еволюції, адже з кожного стану можна перейти лише в один наступний стан. Проте певні еволюції можуть тривати тисячі ітерацій, що у випадку великого розміру автомату може викликати проблему: кожна наступна ітерація потребуватиме все більше й більше часу на обрахунок. Тому необхідний ефективний спосіб порівняння станів автомату, що дозволить обраховувати подібні довготривалі еволюції.

Доволі простим рішенням проблеми є перетворення стану автомату в послідовність бітів для подальшого побітового порівняння поточного стану з попередніми. Так як клітинний автомат, що розглядається, може приймати лише один з двох можливих станів, то стан клітини можна описати одним бітом. В цьому випадку побітове порівняння двох значень є найшвидшим

способом. Проте навіть у цьому випадку особливо великі автомати з довготривалою еволюцією можуть викликати проблеми на великій швидкості обрахунку чи слабкій конфігурації комп’ютера. Тому для прискорення обрахунку подібних еволюцій пропонується формувати хеш-значення поточного стану, та порівнювати саме його. Формування хеш-значення з поточного стану також займає певний час та має виконуватись на кожній ітерації, проте це прискорить порівняння двох станів між собою, яке на пізніших ітераціях виконуватиметься тисячі разів на кожному кроці, тому час на формування хешу стану буде виправданим. Так як ймовірність, що два різних стану автомату матимуть однаковий хеш, є майже нульовою, то порівняння хешу достатньо, щоб визначити, чи зациклився клітинний автомат, перейшовши таким чином у тривіальний стан.

Якщо ж клітинний автомат перейшов у тривіальний стан з наявністю “космічних кораблів”, тобто циклічних структур, що постійно рухаються в певному напрямку, цей варіант додатково розглядати немає необхідності. Якщо в процесі руху “космічний корабель” опиниться достатньо близько від іншої стабільної структури, то еволюція може продовжитись, таким чином цей стан не є тривіальним. Якщо ж “корабель” рухатиметься без перешкод, він з часом досягне краю автомату та більше не зможе зберігати свою постійність. Таким чином, скінченність автомату виключає необхідність ускладнення обрахунку переходу до тривіального стану у випадку виникнення рухомих стабільних структур.

Прикладами тривіальних станів клітинних автоматів можуть слугувати натюрморти з гри “Життя”.

2.4 Заповнення початкових умов клітинного автомату

Можливість задання стну клітин є обо’язковим функціоналом програми. Він дозволить досліджувати окремі, заздалегідь задані конфігурації на предмет їх стабільності та відповідності певним критеріям (наприклад,

великий час еволюції). Однак процес задання стану клітин тривимірного автомату ускладнюється тим, що користувач не може бачити усієї структури відразу та не здатен рухати покажчик у трьох вимірах одночасно. Перша проблема вирішується засобами відображення тривимірних автоматів, що були описані в одному з попередніх підрозділів. Друга проблема ж є розвитком цих ідей з додаванням елементів інтерактивності.

Головні проблеми, які потрібно вирішити при реалізації даного функціоналу, це спосіб відображення поточної позиції покажчику та спосіб зручного переміщення покажчика у тривимірному просторі. При відображенні поточної позиції покажчику необхідно розглянути два варіанти: коли покажчик знаходиться на заповненій клітині, та коли він знаходиться на пустій клітині. Доволі репрезентативним способом зображення покажчика є періодичне миготіння відповідної клітини. Якщо клітина пуста, на ній періодично з’являтиметься напівпрозорий куб, чия прозорість поступово мінятиметься від нуля до певної величини, відображаючи таким чином поточну обрану клітину. Якщо ж клітина заповнена, відбуватиметься зворотній процес: періодично куб в даній клітині ставатиме напівпрозорим, поступово змінюючи прозорість від повністю непрозорого до певної величини та назад.

Друга проблема ж вирішується шляхом використання двовимірної репрезентації одного з шарів автомату. Так як програма містить функціонал з відображення певного шару автомату, до нього можна додати можливість обирати конкретну клітину курсором миші чи стрілками клавіатури. Змінювати ж шар, на якому необхідно встановити стан клітини, можна за допомогою повзунка чи коліщатка миші. Подібна система дозволить обрати будь-яку клітину та змінити її стан зручним чином. Доцільним також є надання пожливості затиснути ліву кнопку миші та, проводячи по конкретному шару, заповнювати всі клітини, по яким проведено курсором. Це дозволить

заповнювати великі масиви клітин швидше, ніж при індивідуальному їх заданні.

Окрім цього, необхідно надати можливість задання кольору клітин. Для цього можна використати спеціальний режим задання кольору. В цьому режимі замість зміни стану клітини відбуватиметься вибір даної клітини задля задання її кольору. Після вибору клітини спеціальна панель дозволить обрати колір цієї клітини на палітрі чи безпосередньо повзунками. Одночасно може бути обрана лише одна клітина.

2.5 Завантаження та збереження стану клітинного автомату

Окрім задання випадкового початкового стану, необхідно також надати функціонал зі збереження та завантаження певних конфігурацій. Якщо користувач досліджує не загальні властивості певного автомату, а властивості конкретних конфігурацій, в нього може виникнути необхідність багатократно задавати одну й ту саму конфігурацію. Збереження дозволить задати складну конфігурацію лише один раз, використовуючи випадкове задання чи безпосереднє заповнення окремих клітин, а далі зберегти конфігурацію для подальшого користування. Також збереження поточного стану може бути корисним для перезапуску еволюції, що дозволить повернути автомат в початковий стан до початку еволюції.

Завантаження поточного стану заповнить необхідні клітини згідно збереженої конфігурації, відновлюючи стан автомату. Інформацію про стан автомату можна зберігати шляхом запам’ятовування правил поточного автомату, його розмірності та бітового представлення станів клітин, подібного до того, що пропонувалось використовувати для підрахунку переходу до тривіального стану. Крім того, доцільним є збереження кольорової ідентифікації клітин автомату. Задання необхідної кольорової схеми автомату з метою зручного відображення клітинного автомату чи використання

наслідування кольору може зайняти значну кількість часу, тому можливість зберегти кольорову конфігурацію може зекономити цей час.

Зберігати таку інформацію можна у базі даних, що дозволить зручно отримувати конфігурації за різними критеріями, такими як назва, час створення, розмірність та правила автомату. Іншим варіантом є збереження стану методом серіалізації. Подібний підхід дозволить створювати зручні в розповсюдженні файли поточної конфігурації, що міститимуть усі необхідні дані. Такі файли можна копіювати на зовнішній носій чи передати мережею Інтернет для розгляду та використання іншими користувачами. Також подібний підхід не потребує використання бази даних, що спростить встановлення застосунку. Крім того, використання локальних файлів дозволить використовувати застосунок без підключення до мережі Інтернет.

Завантаження стану потребуватиме коректної обробки даних з серіалізованого файлу. Цього можна досягти шляхом перевизначення масивів, що зберігають тривимірні моделі клітин, з подальшим заданням їх стану та кольору. Також необхідною є можливість переглянути доступні збережені файли станів. У випадку даної програми відображується список доступних у спеціальному каталогу файлів за іменами, вибір одного з яких дозволить завантажити відповідну конфігурацію.

Висновки до розділу

В даному розділі розглянуто функціонал, необхідний для повноцінного дослідження тривимірних клітинних автоматів, та варіанти реалізації інтерфейсу користувача, що є найбільш зручними для виконання поставленої задачі. Зокрема, було описано варіанти вирішення таких проблем, як:

- визначення необхідного переліку функцій візуалізації для дослідження клітинних автоматів;
- дослідження автомату при втручанні сторонніх факторів.

Встановлено, застосунок надає повноцінний інструментарій для роботи з тривимірними клітинними автоматами: обрахунок їх еволюції, зручну систему відображення стану автомату, систему кольорової диференціації клітин, спрощену систему визначення переходу до тривіального стану, інструментарій по заданню стану та кольору клітини, а також можливість запису та завантаження стану клітинного автомату разом із правилами.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Інформаційне забезпечення

Розробка програмної реалізації потрібна для того щоб мати змогу провести дослідження та продемонструвати результатів, що були отримані під час дослідження. Перед тим як провести розробку програмного забезпечення, необхідно виконати підготовку у вигляді таких складових:

- зібрати дані про потреби ринку в реалізації тривимірних клітинних автоматів та платформи розгортання даної реалізації, якщо потреба в ній є;
- провести аналіз основних існуючих аналогів для того щоб виявити який функціонал необхідно реалізувати, а який, можливо, розширити чи вдосконалити. Після цього необхідно згрупувати та систематизувати вимоги до програмного забезпечення та розробити архітектуру, структуру даних та продумати деталі реалізації.

Аналіз ринку показав, що для дослідження тривимірних клітинних автоматів в основному використовується програмне забезпечення, що розгортається на стаціонарних комп’ютерах, а ніша в мобільних реалізаціях залишається доволі порожньою. Тому для проведення дослідження тривимірних клітинних автоматів необхідно розробити мобільний додаток, за допомогою якого ми зможемо генерувати, переглядати, зберігати та завантажувати конфігурації КА.

Крім того, мобільна платформа була вибрана ще й тому, що ринок мобільних додатків дуже стрімко розвивається, і вся основна робота поступово переходить на смартфони, планшети чи інші портативні пристрої. Більш того, реалізація додатку для генерації клітинних автоматів для мобільної платформи дає змогу для широкого подальшого застосування методів та алгоритмів, визначених в даній роботі у сфері розробки мобільних симуляцій природних явищ і фізичних процесів.

3.2 Засоби розробки додатку

Перш ніж створювати будь-яке програмне забезпечення, потрібно обрати стек технологій, що вплине на подальшу розробку архітектури та деталі реалізації.

Додаток був розроблений за допомогою офіційного інтегрованого середовища розробки (IDE) Android Studio для платформи Android. Android Studio використовується в основному для розробки мобільних додатків, включає інтегрований емулятор ОС Android, відладчик, систему контроля версій (VCS), засоби моніторингу стану мобільного додатку та засоби тестування.

Для розробки алгоритму, архітектури клітинного автомату та роботи з структурами даних використовувались такі мови програмування:

- Java – одна з самих найпопулярніших ООП мов програмування, представлена в 1995 році, являється також основною мовою для розробки мобільних додатків на платформі Android. В розробці була використана Java 8
- Kotlin – мова програмування, що поступово приходить на зміну мові Java, як основній в сфері розробки мобільних додатків. Вперше представлена в 2011 році, об'єктно – орієнтована мова програмування, що працює поверх JVM.

Візуалізація станів, ітерацій, перегляду клітинних автоматів розроблена, використавуючи технологію OpenGL ES (Embedded Systems). Це безкоштовний, міжплатформовий API для розробки передової 2D і 3D графіки на вбудованих і мобільних системах - включаючи консолі, смартфони, прилади та транспортні засоби. Вона складається з чітко визначеного підмножини десктопного OpenGL, адаптованого для малопотужних пристроїв, і забезпечує гнучкий і потужний інтерфейс між програмним забезпеченням і апаратним для прискорення графіки.

Для розробка системи збереження та завантаження даних використовувалась така СУБД як SQLite – реляційна система керування базою даних. Широко застосовується для збереження даних в мобільних додатках.

Крім основних технологій, були використані такі бібліотеки:

- Room Persistence Library – обгортка для зручної роботи з базою SQLite;
- Glide – відображення двовимірного зображення в віджетах;
- QuadFlask:colorpicker – віджет вибору кольору.

3.3 Архітектура програмного забезпечення

Перш за все необхідна визначити ключові компоненти додатку, що реалізують бізнес логіку додатку. Структурну схему класів цих компонентів представлено на 1 листі Додатку Б.

- Rule – клас, який являє собою представлення правила генерування клітинного автомату. Інтерфейс класу складають методи для отримання наступної ітерації автомату. Окрім того, клас інкапсулює функціонал для обчислення стану клітини на даному етапі еволюції, містить дані про поведінку КА.
- GameInstance – клас – обгортка над основними методами OpenGL ES. Містить методи життєвого циклу клітинного автомату.
- ModelRenderBuilder – зберігає графічні дані автомату, а саме список кольорів та векторів нормалей вершин. Функціонал: перебудова графічних даних на основі нової структури автомату, перенесення графічних даних з ОЗУ в пам’ять графічного процесора (GPU); додавання, видалення, зміна кольору клітини для відображення.
- Presenter – аналізує вхідні сигнали від представлення (view) та викликає керує ним. Функціонал включає керування анімацією віджетів, зміною стану екрану, переходами між екранами, навігацією по додатку.

− GraphicsRenderer – клас, що реалізує функціонал для обробки та підготовки графічних атрибутів. Сюди входить обчислення освітлення, матриць повороту та положення моделі на екрані, компіляція шейдерів, передача атрибутів в шейдери; утримання мосту комунікації між пам’яттю графічного процесора та мобільного додатку; попередня обробка зображення перед виведенням на екран; перетворення двовимірних координат точки доторкання екрана в тривимірний простір.

− RenderCubeMap – спеціально розроблена колекція, на зразок ArrayList в Java, але адаптована під потреби клітинного автомату. Містить тривимірну структуру клітинного автомату, список всіх живих клітин, кількість живих клітин. Функціонал: додавання клітини, видалення, зміна кольору; формування колекції RenderCubeMap на основі списку живих клітин; знаходження клітини за координатами та навпаки; перетворення колекції в список.

Важливим етапом є специфікація основних методів програмного забезпечення, що являється мобільним додатком. Під час розробки додатку її буде використано як основу для функціоналу та структури класів. Результат такої специфікації можна побачити в таблиці 3.1.

Таблиця 3.1 – Специфікація методів застосунку

Клас	Метод	Параметри	Дія
Rule	nextIterations	cumeMap	Метод для обчислення наступнох ітерації стану автомату. Результатом роботи є список живих клітин на наступній ітерації.
	countStatus	cube, neighbors	Обчислює статус клітини на наступній ітерації, тобто її стан, кількість пройдених ітерацій.
	countColor	cube, neighbors	Визначає колір клітини на наступній ітерації.
	neighbours	cube, map	Повертає список клітин – сусідів заданої клітини.
GameInstance	create		Наслідує один з трьох найважливіших методів OPengl ES. Виконує підготовку екрану до відображення графічних об’єктів. Ініціалізує правило генерації клітинного

Продовження таблиці 3.1

Клас	Метод	Параметри	Дія
			автомату, структура даних.
	Render		Наслідує один з трьох найважливіших методів OPengl ES. Опрацьовує команди, що надходять від користувача та на основі них відображує представлення клітинного автомату. Відображає на дисплеї клітини автомату, що є живими в даній ітерації.
ModelRenderBuilder	handleTouch	touchResult	Обробляє точку дотику до екрану та інтерпретує її в тривимірний простір.
	viewMode	isViewMode	Перемикає стан автомату в режим перегляду.
	build		На основі живих клітин автомату генерує графічні дані про автомат для

Продовження таблиці 3.1

Клас	Метод	Параметри	Дія
			відображення даних на дисплеї.
	draw		Відображення на екрані згенерованих графічних даних даних.
	addNewCube	renderCube	Додає нову живу клітину до структури автомату.
	paintCube	center, color	Зміна кольору живої клітини.
	deleteCube	center	Видалення клітини з структури автомату.
Presenter	saveFragmentSave Pressed		Зберігає поточний стан клітинного автомату до бази даних.
	editPressed		Відкриває екран редагування автомату.
	closeEditPressed		Закриває екран редагування автомату.
	loadPressed		Відкриває вікно перегляду збережених автоматів.

Продовження таблиці 3.1

Клас	Метод	Параметри	Дія
GraphicsRenderer	calculateModelMatrix		Обчислення матриці, що відповідає за розташування об’єкта на сцені та його кут нахилу.
	calculateLightMatrices		Обчислення матриць, що відповідають за освітлення на сцені (ambient та directional).
	setFigureUniforms	shader	Перенесення згенерованих даних моделі автомату на GPU.
	takeScreenshot		Робить знімок екрану для використання в покращенні наглядності відображення збережених клітинних автоматів.
RenderCubeMap	getLayer	height	Повертає список живих клітин автомату на визначеній висоті height.

Продовження таблиці 3.1

Клас	Метод	Параметри	Дія
	getCubeAt	coords	Повертає клітину за координатами, якщо така існує.
	add	renderCube	Додати клітину до колекції.
	remove	renderCube	Видалити клітину з колекції.
	cubeExists	renderCube	Перевіряє чи існує клітина за заданими координатами .
	cubeInBounds	renderCube	Перевіряє, чи входить клітина з заданими координатами до меж решітки клітинного автомату
	fromCubeList	cubeList, automataRadius	Повертає колекцію CubeMap, створену зі списку живих клітин

Програмний код додатку детально розписаний у додатку А.

3.4 Вимоги до технічого забезпечення

Програмне забезпечення являє собою мобільний додаток. Для того щоб розгорнути додаток, знадобиться :

- мобільний пристрій на Android з підтримкою Opengl ES 2.0 та вище;
- ОС Android 4.4 та вище;
- оперативна пам’ять 2гб та більше;
- вільне місце в пам’яті пристрою 30мб.;
- CPU Qualcomm Snapgragon 845 та вище;
- GPU Adreno 630 та вище.

3.5 Керівництво користувача

Керівництво користувача розробляється для опису роботи додатку в різних сценаріях користування. В цьому розділі розглянемо декілька сценаріїв використання розробленого застосунку. Для запуску додатку, першим кроком необхідно його встановити на мобільний пристрій. Для цього необхідно скопіювати .apk файл додатку на мобільний пристрій та запустити його. Після встановлення додатку та запуску, користувач повинен побачити інтерфейс, аналогічний тому, який зображений на рисунку 3.4.

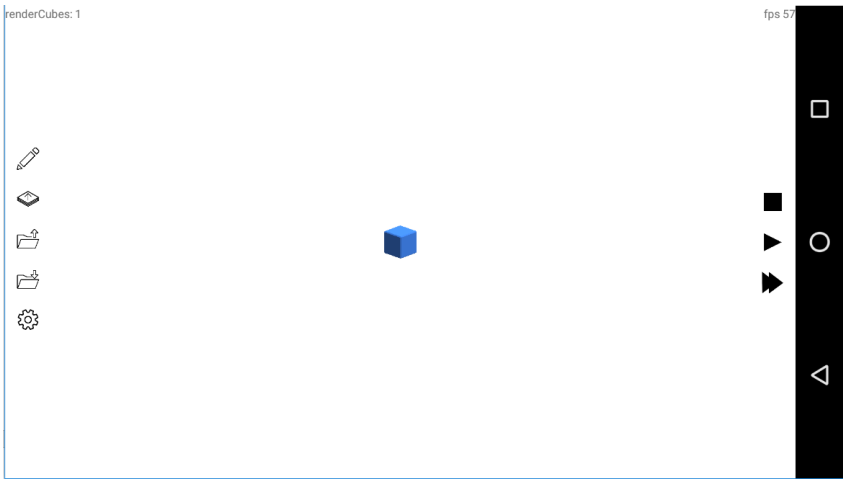

Рисунок 3.4 – Інтерфейс додатку

Зліва на екрані знаходиться панель інструментів, зверху вниз:

- режим редагування моделі автомату. Може бути використаний для задання початкового стану КА вручну або редагування автомату на певній ітерації;
- режим перегляду автомату. В даному режимі клітинний автомат можна переглянути, розбивши його на горизонтальні шари;
- завантаження раніше збереженого стану клітинного автомату. Перехід до екрану завантаження клітинного автомату, призупинення генерації;
- збереження стану клітинного автомату. Перехід до екрану збереження стану клітинного автомату та ведення назви;
- налаштування. Включає налаштування правил генерації, генерацію кольору за принципом Мандельброта та наслідування кольору.

Справа на екрані розташована панель керування процесом еволюції клітинного автомату. Зверху вниз:

- стоп – зупинити генерацію та перейти до початкового стану;
- спарт \ пауза – розпочати генерацію автомату та призупинити;
- прискорити – зменшує час ітерації та пришвидшує еволюцію.

3.5.1 Робота з конкретною конфігурацією автомату

Розглянемо декілька сценаріїв використання розробленого застосунку. Нехай користувач хоче розглянути процес еволюції конкретної конфігурації автомату із відомими правилами переходу до нового стану. При цьому використовуватиметься можливість зберегти цей початковий стан для більш зручного багатократного огляду еволюції без необхідності повторно задавати початковий стан. Опишемо послідовність дій, що користувач має при цьому зробити.

У першу чергу користувачу необхідно налаштувати автомат під відповідні правила, для яких його конфігурація передбачена. Для цього можна

вибрати режим налаштування та перейти до екрану налаштувань. Далі необхідно встановити значення розміру поля, кількості сусідів, необхідних для зародження клітини (мінімум та максимум), кількості сусідів, менше якої клітина загине від самотності, та кількості, більше якої клітина загине від перенаселення (також мінімум та максимум у обох випадках).

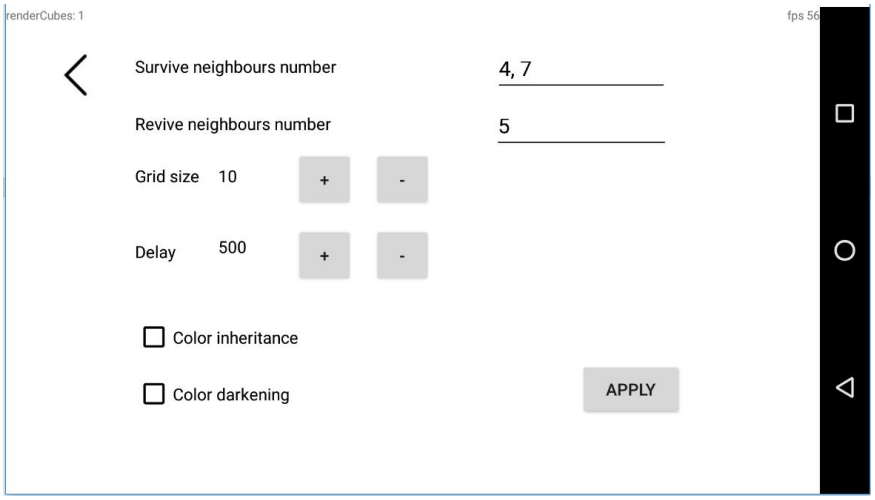

Рисунок 3.5 – Екран налаштувань клітинного автомату

Наступним кроком користувачу необхідно задати стан клітин початкової конфігурації. Для цього йому необхідно перейти в режим редагування, натиснувши на відповідний інструмент на боковій панелі.

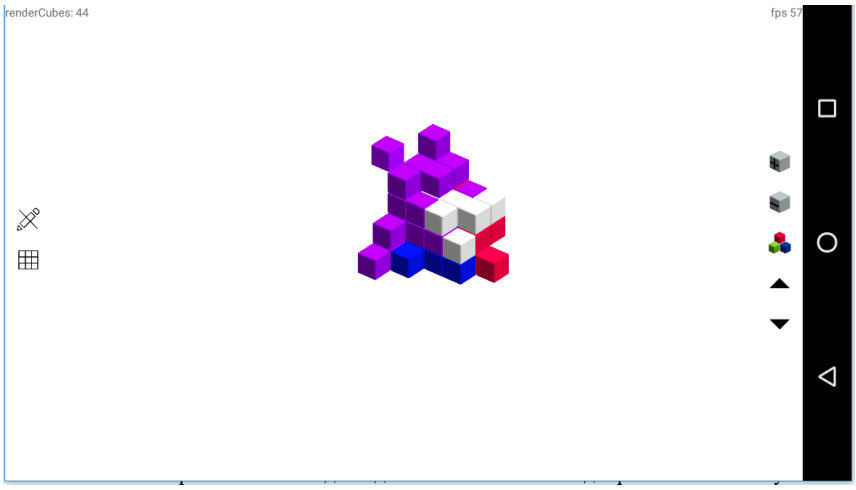

Рисунок 3.6 – Екран редагування стану клітинного автомату

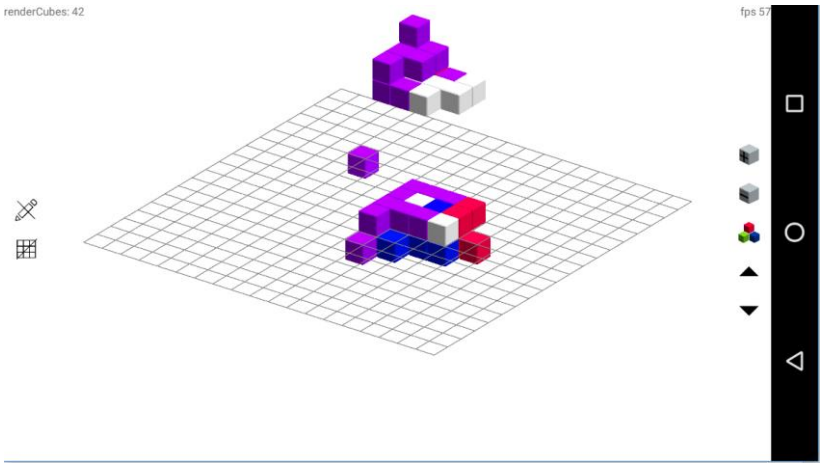

Рисунок 3.7 – Екран редагування стану клітинного автомату з ввімкненою решіткою

Ліворуч на панелі інструментів розміщені інструменти:

- вихід з режиму редагування ;
- відобразити \ вимкнути допоміжну решітку.

Праворуч:

- додати клітину;
- видалити клітину;
- змінити колір клітини;
- перемістити решітку на шар вище \ нижче.

Вибрати один з двох форматів відображення окремого шару. У будь-якому з цих форматів користувач може змінювати стан всіх клітин поточного шару. Номер поточного шару користувач може визначити шляхом переміщення відповідного повзунку. Після вибору потрібного шару необхідно навести курсор миші на поле автомату. Поточна позиція курсору підсвічуватиметься шляхом періодичного змінення прозорості клітини, на яку курсор наведено. Натискання лівої кнопки миші змінить поточний стан клітини на протилежний і відповідним чином змінить відображення автомату. Важливо відмітити, що перемикач “Pick Color” має бути виключеним для можливості змінити стан клітин автомату. Користувач може використовувати

різні формати відображення для більш зручного задання станів клітин. Наприклад, при заданні одних клітин доцільнішим є використання виду спереду для кращого розуміння, які клітини вже розташовані на цьому шарі, проте для задання інших зручнішим може виявитись вид зверху. Також користувачу може знадобитись тривимірне відображення, щоб побачити, як виглядає вже задана на даний момент структура та визначитись, стани яких ще клітин варто змінити.

Після завершення задання станів клітин користувачу варто зберегти поточну конфігурацію. Для цього необхідно натиснути кнопку “Save Configuration”, що викличе появу діалогового вікна запису конфігурації у файл. У вікні, що появилось після натискання кнопки, у текстовому полі необхідно ввести ім’я файлу. Варто відмітити, що збереження конфігурації у файл з ім’ям файлу, що вже існує і присутній у тому самому каталогу, видалить попередній файл. Тому якщо це не є метою користувача, необхідно обрати унікальне ім’я файлу. Після цього необхідно натиснути кнопку зберегти, що відкриє вікно збереження автомату.

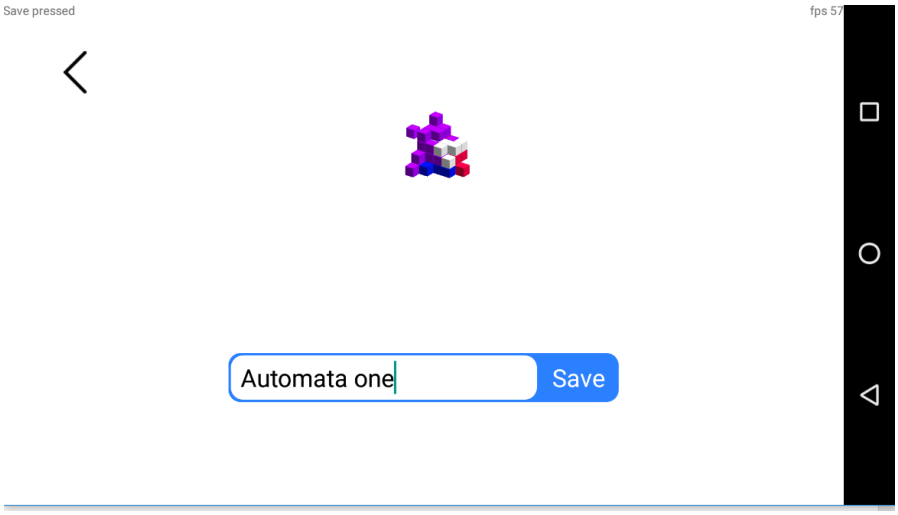

Рисунок 3.8 – Екран збереження стану клітинного автомату

Поки вікно відкрите, взаємодія із моделлю автомату та іншими елементами інтерфейсу неможлива, а еволюція автомату автоматично призупиняється, тому під час збереження стан автомату не може змінитись.

Натиснувши кнопку «Save», ви викличете команду збереження автомату, після виконання якої віксно буде автоматично закрито.

Нарешті, для запуску еволюції конфігурації користувач може натиснути кнопку “Start”. Після цього автомат розпочне переходити до нового стану відповідно із заданими правилами автомату та початковою конфігурацією. Перехід до нового стану користувач може відслідковувати за допомогою анімації зміни стану клітини, поверненням автомату у зручну позицію, наближенням та віддаленням камери та вибором іншого формату відображення чи поточного шару. Для цього рекомендується призупинити еволюцію автомату натисканням кнопки “Stop”. Швидкість еволюції можна регулювати відповідною кнопкою на правій панелі, що дозволить пришвидшити еволюцію на менш значних ділянках та сповільнити на найбільш цікавих для дослідження ділянках.

Після завершення огляду еволюції створеної конфігурації його можна запустити повторно, завантаживши збережену раніше початкову конфігурацію. Для цього необхідно натиснути кнопку “Load”, що викличе появу діалогового вікна. У ньому необхідно обрати у списку файл з відповідним даній конфігурації ім’ям чи зображенням та натиснути кнопку “Load”.

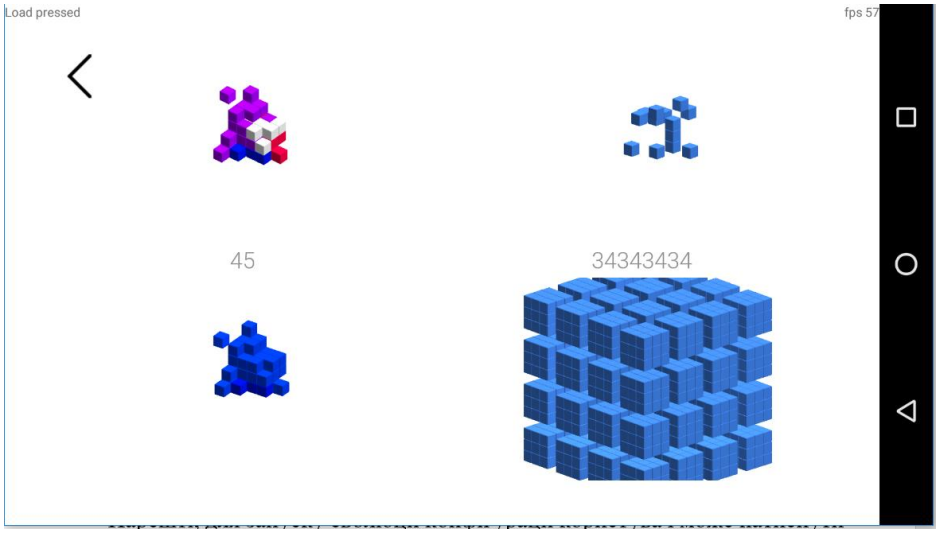

Рисунок 3.9 – Екран завантаження стану клітинного автомату

Після цього автомат змінить стан на завантажений з файлу, а у полях задання правил автомату встановляться відповідні значення, діалогове вікно ж при цьому закриється. Далі із завантаженим автоматом можна працювати по описаній вище схемі.

3.5.2 Виявлення автомату з заданими властивостями

Нехай мета користувача — знайти такі правила переходу до нової ітерації, що наділяють автомат необхідними властивостями. Перевірку цих властивостей користувач виконує шляхом порівняння еволюції одної конфігурації на різних правилах автомату. Більшість етапів будуть аналогічні першому сценарію, тому будуть описані скорочено.

Спочатку користувач має створити базовий файл конфігурації. Йому необхідно задати правила автомату за схемою, аналогічною попередньому сценарію використання, та встановити початковий стан усіх клітин. Після цього користувачу варто зберегти конфігурацію у новому файлі.

Далі користувач може розпочинати дослідження еволюції цієї конфігурації. Цей процес також проходить аналогічно попередньому сценарію використання, проте на цей раз користувач звертатиме увагу на різницю в еволюції конфігурації при даному набору правил та інших наборах. На тих етапах, що суттєво відрізняються між різними правилами, користувач може сповільнити чи призупинити еволюцію автомату для більш детального огляду.

Після завершення еволюції користувач може знову завантажити початковий варіант та змінити певні значення правил автомату на основі отриманих даних. Далі слідує повторення попереднього етапу для нових правил. Цей цикл може продовжуватись, поступово наближаючи користувача до конфігурації, яка його влаштує.

Коли користувач знайшов необхідні значення правил, він може зберегти ці правила у тому ж самому базовому файлі або іншому файлі, щоб далі

використовувати цей набір правил, наприклад, для подальшого дослідження властивостей автомату загалом чи поведінки окремих конфігурацій.

3.5.3 Наслідування кольору клітин, принцип множини Мандельброта

Розглянемо сценарій використання застосунку, за якого користувач бажає проаналізувати еволюцію певної конфігурації автомату, в якій задано колір початкових клітин, з використанням наслідування кольору. Загалом даний варіант схожий на дослідження звичайної конфігурації, однак містить певні додаткові етапи. Користувач необхідно провести наступну послідовність дій.

Задати правила автомату та конфігурацію (на даному етапі — лише стан клітин). Цей процес детально описаний в першому сценарії використання. Альтернативним варіантом є завантаження конфігурації з файлу, що вже містить коректно задані стани клітин та правила автомату. Цей процес також аналогічний іншим сценаріям.

Далі необхідно встановити колір кожної клітини. Процес встановлення кольору клітин детально описаний у попередньому сценарії. Але у даному випадку необхідно задавати лише колір тих клітин, що знаходяться у “живому” стані в початковій конфігурації — колір інших клітин визначатиметься на основі кольору їх сусідів під час їх зародження. Після задання кольорів доцільно зберегти конфігурацію у файл.

Наступний крок — розпочати еволюцію автомату. При цьому на кожній ітерації усі нові клітини, що з’являтимуться на полі, будуть мати колір, визначений як середнє значення кольорів клітин-сусідів із невеликим випадковим відхиленням. Таким чином дві групи клітин в початковій конфігурації, що мають різний колір, в процесі еволюції створять перехідну область між ними, що сформує градієнтну зміну кольору від однієї області до

іншої. Розгляд цього процесу може відбуватись тими ж методами, що й у інших сценаріях використання.

Еволюцію автомату можна призупинити з метою зміни кольорів клітин. Наприклад, користувач вирішив змінити колір клітин в певній області чи додати нову область іншого кольору. Для цього використовуються вже описані раніше засоби. У цьому разі після продовження еволюції результат зміниться, додаючи до автомату нові градієнти, що походять із зміненої чи доданої області. Таким чином користувач може розглянути, яким чином на систему впливатиме зовнішнє втручання та різка зміна умов.

Висновок до розділу

В розділі «Опис програмної реалізації» надане інформаційне забезпечення. Після аналізу ринку визначено платформу, під яку буде розроблятися додаток.

Визначені технології, що будуть використовуватися при розробці мобільного додатку. Зокрема обрано Android Studio як середовище розробки додатку, Java - основна та Kotlin - допоміжна мова програмування бізнес логіки. Для відображення графіки обрано бібліотеку OpenGL ES.

Визначена специфікація основних методів, що використовуватимуться для реалізації візуалізації для дослідження тривимірного клітинного автомату, а також в редакторі автомату для редагування його стану на етапі генерації. Структурна схема класів наведена в додатку Б на листі 1

Спроектовано архітектуру та розроблено програмне забезпечення для дослідження тривимірних клітинних автоматів. Також створена модель бази даних для зберігання стану клітинного автомату. Структурна схема наведена в додатку Б на листі 2.

Отримана архітектура повністю відповідає поставленим вимогам. Ми отримали новий спосіб генерації тривимірних об’єктів та структур.

4 ОГЛЯД РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМИ

4.1 Результати виконання програми

Розглянемо для прикладу еволюцію клітинного автомату без наслідування кольорів, початковий стан якого заданий вручну. Правило задамо 1,2,3,4/6,7,8. Це означає, що клітина виживатиме при 1-4 живих сусідах та буде відроджуватися при 6-8 живих сусідах. У всіх інших випадках клітина загине.

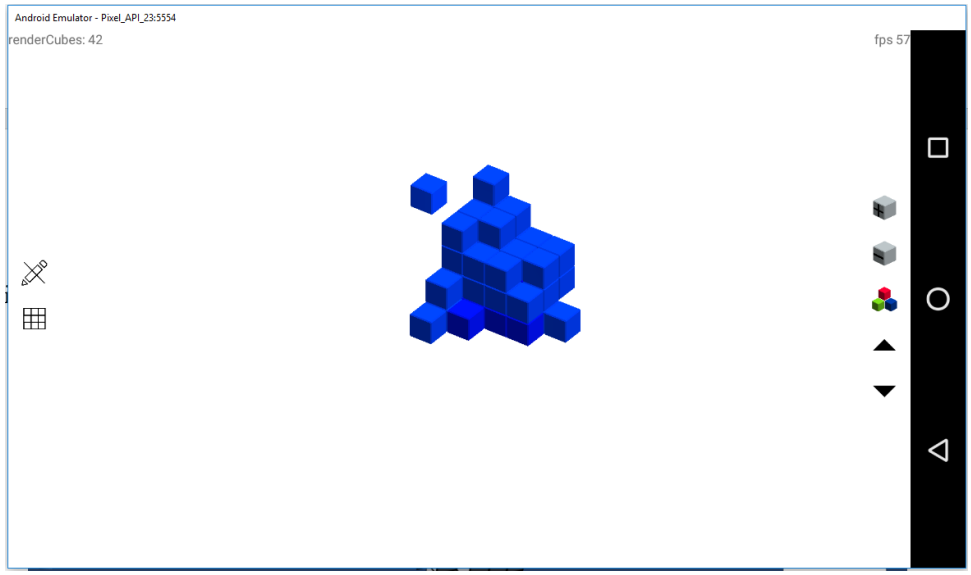

Рисунок 4.1 – Початкова конфігурація автомату

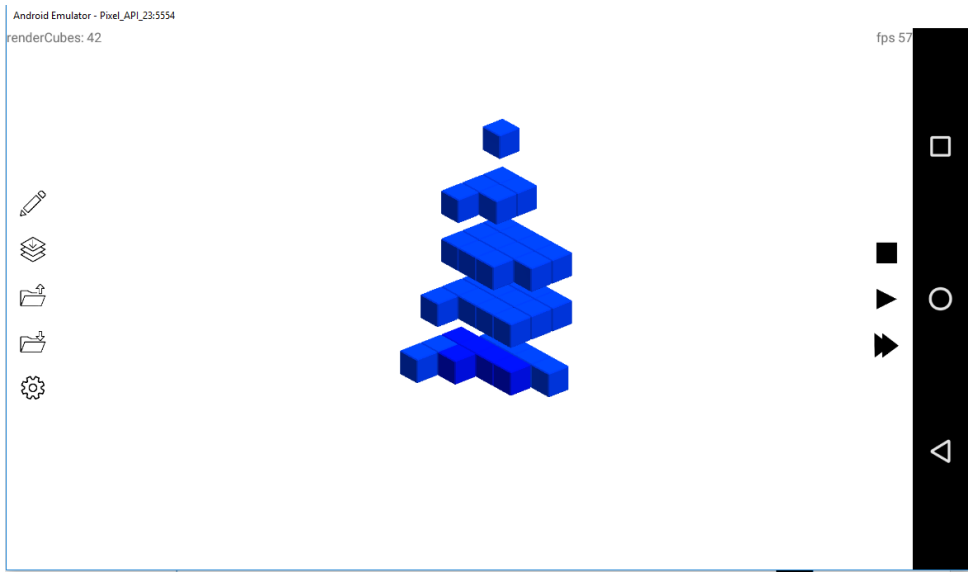

Рисунок 4.2 – Початкова конфігурація автомату в режимі перегляду

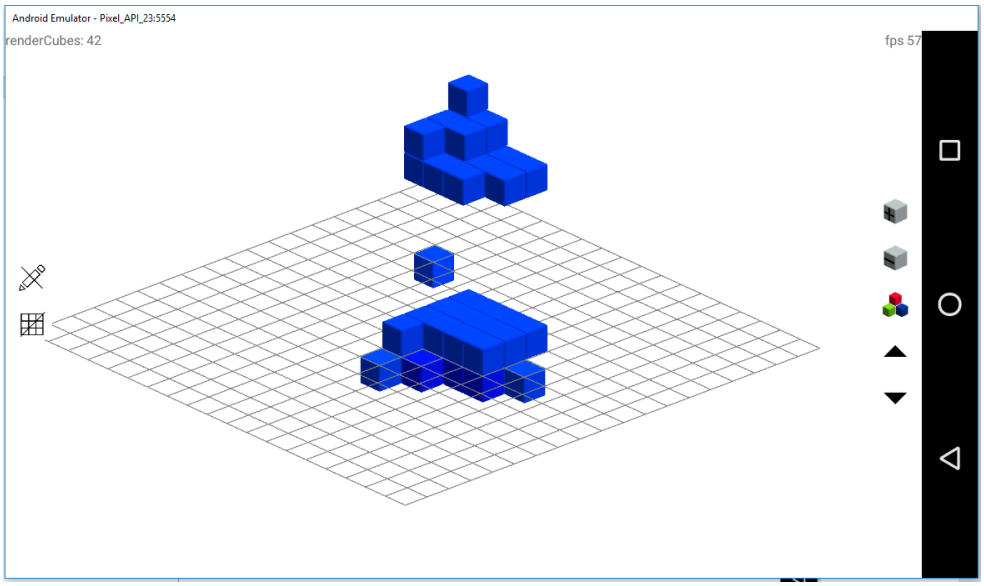

Рисунок 4.3 – Початкова конфігурація автомату в режимі редагування

Після виконання 9 ітерацій, за встановленим правилом, клітинний автомат перейшов до стабільного стану.

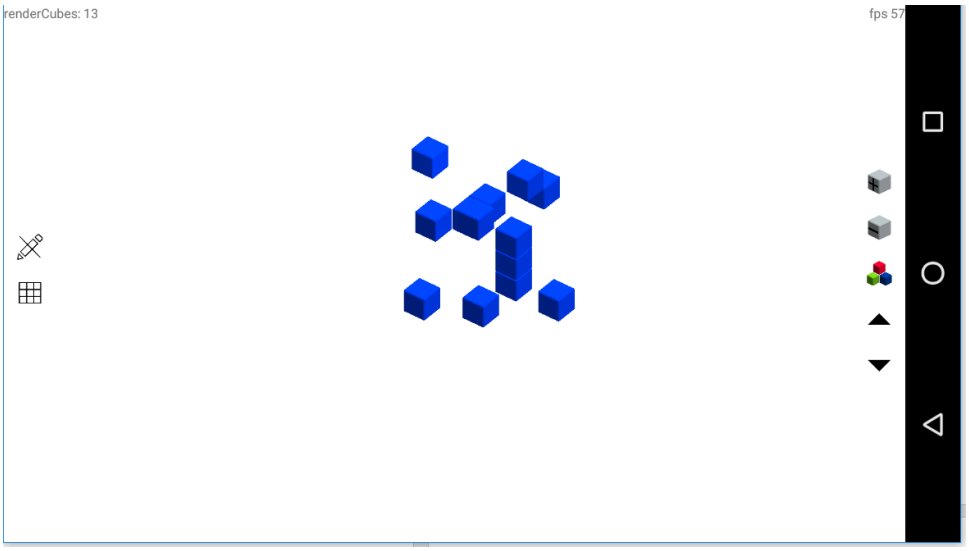

Рисунок 4.4 – Автомат перейшов до стабільного стану

4.2 Дослідження статистичних даних

Проведемо експерименти над довільно згенерованими клітинними автоматами та занесемо результати до таблиці. Якщо автомат перейшов до стабільного стану – результатом є кількість ітерацій від початкового стану до даного. Якщо автомат не виказує ознак переходу до стабільного стану – поле

залишимо порожнім. Для даної серії експериментів виберемо розмір поля 5 х 5 клітин.

Правило генерації автомата: клітина виживає при наявності в неї 4, 5, 9 живих сусідів. Клітина відроджується з мертвого стану за умови присутності 7,8,3, 2 живих клітин сусідів. При будь – якому іншому випадку клітина гине або залишається мертвою

Таблиця 4.1 — Статистичні дані часу еволюції автомату з розміром поля 5 х 5. Вимірювання 1

к-ть живих клітин	к-ть клітин для виживання	к-ть клітин для відродження	№ експерименту											
			1	2	3	4	5	6	7	8	9	10	11	12
10	4, 5, 9	7, 8, 3, 2	9	5	8	9	11	9	10	9	9	9	5	7
12	4, 5, 9	7, 8, 3, 2	13	14	13	18	10	9	13	7	15	11	13	8
14	4, 5, 9	7, 8, 3, 2	14	15	16	15	12	14	4	10	16	17	12	19
16	4, 5, 9	7, 8, 3, 2	16	20	6	18	15	14	10	20	16	20	14	12
18	4, 5, 9	7, 8, 3, 2	17	19	12	17	14	50	12	14	13	10	17	18
20	4, 5, 9	7, 8, 3, 2	18	21	19	19	18	22	6	18	3	17	19	22
22	4, 5, 9	7, 8, 3, 2	24	42	12	22	8	10	26	28	25	9	17	18
24	4, 5, 9	7, 8, 3, 2	9	26	18	45	60	16	26	33	31	22	19	20
26	4, 5, 9	7, 8, 3, 2	61	41	11	22	5	13	27	40	35	19	17	31
28	4, 5, 9	7, 8, 3, 2	110	67	8	124	141	58	63	12	141	203	122	98
30	4, 5, 9	7, 8, 3, 2	233	135	224	456	187	247	314	217	331	401	302	516

Побудуємо графік залежності часу ітерації від розміру автомату

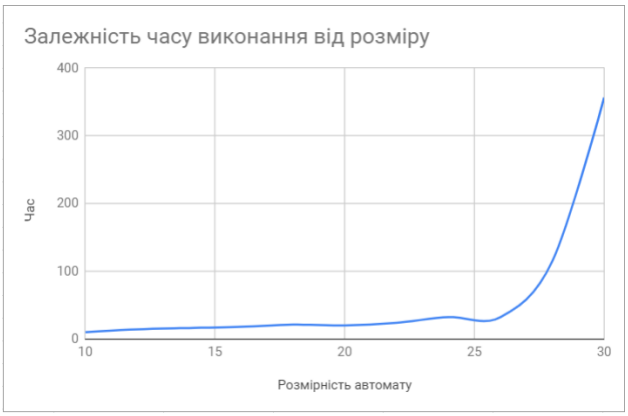

Рисунок 4.5 — Графік залежності часу ітерації від розміру автомату

З отриманих результатів можна зробити висновок, що за допомогою даного правила генерації клітинного автомата можна привести до стабільного стану будь – які конфігурації автоматів, розмірністю від 10 до 30 клітин. Причому час переходу автомату до стабільного стану значно зростає, починаючи з 28 клітин.

Розглянемо цей же випадок, але для поля розміром 10 х 10 клітин

Таблиця 4.2 — Статистичні дані часу еволюції автомату з розміром поля 10 х 10. Вимірювання 2

к-ть живих клітин	к-ть клітин для виживання	к-ть клітин для відродження	№ експерименту											
			1	2	3	4	5	6	7	8	9	10	11	12
10	4, 5, 9	7, 8, 3, 2												
12	4, 5, 9	7, 8, 3, 2												
14	4, 5, 9	7, 8, 3, 2												
16	4, 5, 9	7, 8, 3, 2												
18	4, 5, 9	7, 8, 3, 2					7				3			
20	4, 5, 9	7, 8, 3, 2		12		2			5				11	
22	4, 5, 9	7, 8, 3, 2	18	15	3			5	15	11		14	10	16
24	4, 5, 9	7, 8, 3, 2	13		20	21		16			6	5		17
26	4, 5, 9	7, 8, 3, 2	14	18	9		12		8	16		27	22	25
28	4, 5, 9	7, 8, 3, 2	17	2	26	19	7				21	15	11	15
30	4, 5, 9	7, 8, 3, 2	26	31	18	22		30	3	23		27	15	17

Рисунок 4.6 — Графік залежності часу ітерації від розміру автомату при розмірі решітки 10 х 10

Проаналізуємо результат. Отримані дані значно відрізняються від попередніх. При розмірності від 10 до 16 клітин, клітинний автомат просто вимирає від самотності. Лише починаючи з 18 клітин на полі, знаходяться конфігурації, здатні вижити та сформувати стабільні структури.

Так як заповнення початкового стану є випадковим, то розміщення клітин в автоматі можна вважати рівномірним. Щільність d можна порахувати, поділивши кількість початкових клітин живих на загальну кількість клітин.

Для варіанту з 30 клітинами $d = 30 / (10^3) = 0.03$. Тобто ймовірність появи живої клітини становить 3%. Таким чином, ймовірність появи у живої клітини живого сусіда, а значить і виживання клітини становить 26 * 3 = 78%. Це підтверджує результат, записаний в таблиці.

Розглянемо варіант першого вимірювання, але зменшимо умови виживання клітини, та розширимо умови відродження.

Таблиця 4.3 — Статистичні дані часу еволюції автомату з розміром поля 5 х 5. Вимірювання 3

к-ть живих клітин	к-ть клітин для виживання	к-ть клітин для відродження	№ експерименту											
			1	2	3	4	5	6	7	8	9	10	11	12
10	4, 5	7, 8, 3, 2, 6	109	67	40	78	56	30	68	34	212	678	86	17
12	4, 5	7, 8, 3, 2, 6	151	163	340	135	332	879	275	165	185	213	115	157
14	4, 5	7, 8, 3, 2, 6	276	290	198	178	156	196	299	369	201	172	256	230
16	4, 5	7, 8, 3, 2, 6	358	467	204	2434	350		293	356	375	902	356	207
18	4, 5	7, 8, 3, 2, 6	689	998		1407	751		6576		247		4567	2456
20	4, 5	7, 8, 3, 2, 6	1006	656	2568		3006	994	1056			1323		6578
22	4, 5	7, 8, 3, 2, 6			6678			5692		4242			2314	
24	4, 5	7, 8, 3, 2, 6	19034		20567			19367				18076		
26	4, 5	7, 8, 3, 2, 6	16745							17634				
28	4, 5	7, 8, 3, 2, 6												
30	4, 5	7, 8, 3, 2, 6												

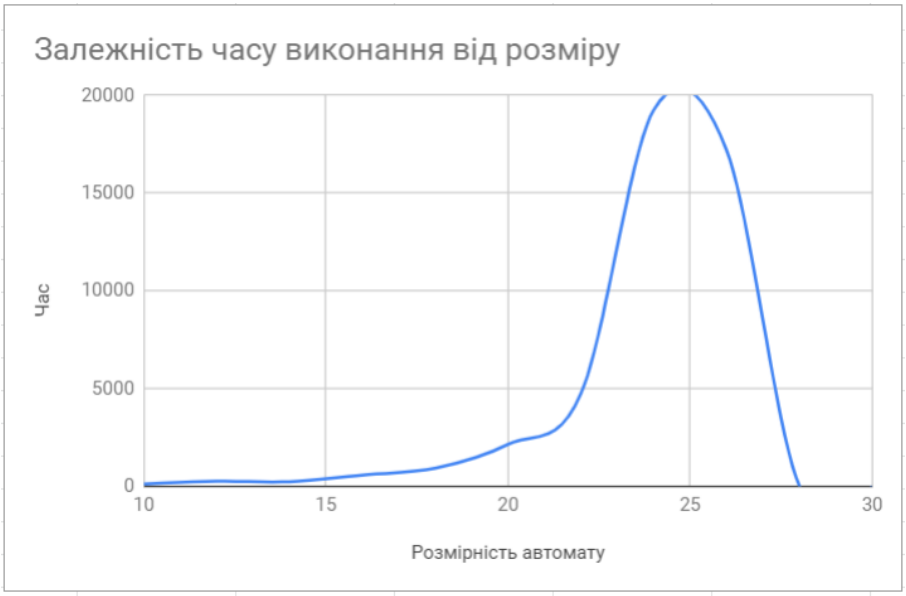

Рисунок 4.7 — Графік залежності часу ітерації від розміру автомату при розмірі решітки 5 x 5. Вимірювання 3

Як видно з результату та графіку, клітинний автомат при досягненні критичної точки в 25 клітин починає вимирати. Пояснення цьому є те, що після досягнення критичної точки, клітинний автомат поводить себе так: трапляється перенаселення, на наступній ітерації вимирає багато клітин, але так як умови для відродження сприятливі – далі слідує ітерація, що призводить знову до перенаселення. І так до безкінечності. Графік показує, що чим більша початкова щільність автомату – тим більше часу потрібно для переходу в стабільний стан, або взагалі не перейти до нього, заповнивши майже автомат живими клітинами, що будуть безкінечно вимирати.

На основі цього можна зробити висновок, що ймовірнісні тривимірні автомати можуть відноситись до класу 3 навіть при їх тенденції до заповнення усього поля. Ймовірнісний елемент автомату надає автомату можливість не переходити до тривіального стану навіть при повторенні певних ітерацій. Однак даний автомат не можна назвати автоматом 4 класу через неможливість появи “космічних кораблів” чи навіть простих осциляторів, та надмірну випадковість змін, що відбуваються в автоматі, що заважає появі більш систематизованих структур.

Висновок до розділу

Додаток реалізує основні функції по реалізації тривимірних клітинних автоматів, а саме:

- дозволяє генерувати тривимірні КА;
- задавати правила переходу до наступної ітерації;
- призупиняти та пришвидшити генерацію;
- еволюція автоматів відображається на дисплеї за допомогою тривимірних графічних структур;
- можливість вибору способу відображення.

Додаток містить всі необхідні засоби для впливу на еволюцію динамічних об’єктів:

- засоби редагування конфігурації автомату;
- засоби задання нових правил генерації на конкретній ітерації. Є можливість вибору способу редагування.

Аналіз статистичних даних дозволив зробити висновок, що пошук автоматів 4 класу не є найбільш перспективним серед ймовірнісних автоматів. Це пов’язано із тим, що формування стабільних структур за такими умовами ускладнюється, і хоча автомати й виказують хаотичну поведінку та мають великий час еволюції, це не дозволяє віднести їх до 4 класу.

5 РОЗРОБКА СТАРТАП ПРОЕКТУ «РУШІЙ ГЕНЕРАЦІЇ СИМУЛЯЦІЇ ЛАНДШАФТУ»

5.1 Опис ідеї

Розглянувши в попередніх розділах новий метод дослідження тривимірних клітинних автоматів, в цьому розділі буде проведено аналіз стартап проекту рушія генерації симуляції ландшафту.

Ідея проекту пролягає в розробці рушія, за допомогою якого можна симулювати та генерувати природні ландшафти гірської місцевості та рельєфу, використовуючи методи та алгоритми розглянуті в попередніх розділах.

Таблиця 5.1 – Опис ідеї стартап проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Створення системи на основі удосконаленого алгоритму випадкової генерації тоталістичних тривимірничх клітинних автоматів	Генерація дандшафту в моделюванні фізичних систем	Зручний та швидкий спосіб моделювання фізичних явищ та законів
	Генерація ландшафту в відеоіграх	Зручний спосіб генерації тривимірного світу відеогри, що буде унікальним при його кожному новому створенні

Отже, пропонується новий спосіб симуляції ландшафту, який значно полегшує роботу тривимірним відображенням реального ландшафту.

Далі проводимо аналіз потенційних техніко-економічних переваг ідеї порівняно із пропозиціями конкурентів:

− визначаємо перелік техніко-економічних властивостей та характеристик ідеї;

− визначаємо попереднє коло конкурентів (проектів-конкурентів) або товарів-замінників чи товарів-аналогів, що вже існують на ринку, та проводимо збір інформації щодо значень техніко-економічних показників для ідеї власного проекту та проектів-конкурентів відповідно до визначеного вище переліку;

− проводимо порівняльний аналіз показників: для власної ідеї визначено показники, що мають а) гірші значення (W, слабкі); б) аналогічні (N, нейтральні) значення; в) кращі значення (S, сильні) (табл. 5.2).

Таблиця 5.2 — Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона	S (сильна сторона)
		Мій проект	Instant Terra	World Machine			
1.	Можливість динамічної генерації ландшафту в реальному часі	+	–	–			+
2.	Редагування ландшафту вручну	+	+	+			+

Продовження таблиці 5.2

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Instant Terra	World Machine			
3.	Накладення текстур та карти нормалей	Вимагає підключення сторонніх бібліотек	Інтегровано в систему	Інтегровано в систему	+		
4.	Можливість автоматичної генерації за заданими вхідними даними	+	-	+		+	
5.	Мобільна версія	+	-				+
6.	Створення воксельних ландшафтів	+	-				+

Отже, проаналізувавши два аналоги, я виявив, що мій продукт має переваги в сфері динамічної генерації об’єктів та використанні його в режимі реального часу, що дозволить застосувати процедурну генерацію та генетичні алгоритми для генерування ландшафту під час гри чи симуляції. Крім того, значним плюсом є наявність мобільної версії, адже ринок мобільних відеоігр та додатків розширюється значними темпами сьогодні.

Єдиним мінусом є накладення текстур та кольору. Це ускладнено тим що створений ландшафт дискретний та складається з кубів.

5.2 Аналіз ринкових можливостей запуску стартап проекту

Визначимо ринкові можливості, які можна використати під час ринкового впровадження проекту, та ринкові загрози, які можуть перешкодити його реалізації.

Це дозволяє розрахувати складність впровадження проекту на масовий ринок. Спочатку проведемо аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку (таблиця 5.3).

Таблиця 5.3 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	2
2	Загальний обсяг продаж, грн/ум.од	1000
3	Динаміка ринку (якісна оцінка)	Стагнує
4	Наявність обмежень для входу (вказати характер обмежень)	Невелика кількість клієнтів.
5	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	120%

Надалі визначаємо потенційні групи клієнтів, їх характеристики, та формуємо орієнтовний перелік вимог до товару для кожної групи (табл. 5.4).

Таблиця 5.4 – Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Потреба в рушії для генерації ландшафту відеовідеогри	Компанії розробники ігрового програмного забезпечення	немає	Можливість генерувати ландшафт в режимі реального часу
2	Потреба в рушії для симуляції фізичних явищ та процесів	Навчальні заклади, дослідницькі лабораторії	немає	Диверсифікація генерованих структур шляхом втручання елементу випадковості

Стартап має декілька можливостей для монетизації бізнесу. По-перше це може бути сервіс або продукт. По-друге, можливо обрати іншу успішну 64 модель, що використовують інші успішні підприємства. Зазвичай засіб монетизації обирається під час стадії появи ідеї. Проаналізовані фактори потенційних загроз для майбутнього стартаппроекту. Результати наведені у таблиці 5.5.

Таблиця 5.5 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Невідомість	Клієнти надаватимуть перевагу відомим рішенням створення ландшафту	Вкладення коштів у рекламу та маркетинг
2	Консерватизм	Не бажання клієнтів переходити на автоматизовану технологію	Зобов’язання компанії самостійно інтегрувати свою систему

5.3 Розроблення ринкової стратегії проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів.

Таблиця 5.6 – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Компанії розробники ігрового програмного забезпечення	Середня	Є	Велика	Складно
2	Навчальні заклади, дослідницькі лабораторії	Висока	Є	Середня	Легко
Які цільові групи обрано: Під час аналізу потенційних груп споживачів було прийнято рішення що ми будемо працювати із компаніями та навчальними закладами різних розмірів.					

За результатами аналізу потенційних груп споживачів ми обрали цільові групи компаніями та навчальними закладами різних розмірів, які використовують засоби генерації та моделювання ландшафту.

Для роботи в обраному сегменті ринку необхідно сформувати базову стратегію розвитку.

Таблиця 5.7 – Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
	Створення легкого для використання рушія для процедурної генерації ландшафту	Диференційований маркетинг	Генерація ландшафту в режимі реального часу; диверсифікація результатів генерації за рахунок елементу ймовірності.	Стратегія спеціалізації

Результатом даного підрозділу є система рішень щодо ринкової поведінки компанії, вона визначає в якому напрямі буде працювати компанія на ринку

5.4 Розроблення маркетингової програми стартап-проекту

Під час розроблення маркетингової програми першим кроком є розробка маркетингової концепції товару, який отримає споживач. У таблиці 5.7 підсумовуємо результати аналізу конкурентоспроможності товару.

Таблиця 5.8 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Потреба у створенні рушія для генерації ландшафтів та природних структур	Створення відеогри чи симуляції, не затрачаючи час на розробку специфічного алгоритму	Можливість генерації в режимі реального часу; швидкість роботи, невеликий розмір та вимоги до апаратного забезпечення.

Наступним кроком є визначення цінових меж, якими необхідно керуватися при встановленні ціни на потенційний товар, це передбачає аналіз цін товарів конкурентів, та доходів споживачів продукту (табл. 5.8).

Таблиця 5.9 – Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	1000 грн \ міс.	340 грн \ міс.	Не важливо	2340-1000 грн.

Таблиця 5.10 – Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Робота на підприємствах	Робота з документами підприємства	1	Розробник - підприємства

Таблиця 5.11 – Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Розроблення відеоігр та фізичних явищ	Інтернет	1.Робота в режимі реального часу	1. Донести до споживача назву продукту 3. Переконати у необхідності продукту	Демонстрація роботи системи

Висновки до розділу

На даний момент, реалізацію рушія генерації ландшафту можна віднести до такої, що має можливості стати стартап – проектом, оскільки вона має комерційні можливості. Продукт може бути використаний будь – якою компанією, що займається розробкою відеоігр чи будь – яким навчальним закладом, що займається дослідженням процедурно згенерованих фізичних явищ. В даному розділі було визначено ключові переваги та недоліки розробки, порівняно з продуктами конкурентів побудовано прогноз розширення ринку збуту.

Було розроблено ринкову стратегію проекту до якої входять вибір цільових груп потенційних споживачів та визначення базової стратегії розвитку.

В кінці розділу було розроблено маркетингову програму стартап-проекту, а саме:

- визначення ключових переваг концепції потенційного товару;
- визначення меж встановлення ціни;
- формування системи збуту;
- створення концепції маркетингових комунікацій.

ВИСНОВОК

В роботі досліджувалась еволюція об’єктів за допомогою тривимірних клітинних автоматів.

Досягнення:

− вдосконалено методи дослідження еволюції тривимірних клітинних автоматів під впливом зовнішніх факторів та реалізовані в мобільному додатку;

− надана можливість визначення критеріїв присутності елементів регулярності в хаотичній структурі під час динамічного розвитку;

− запропоновані нові способи генерації тривимірних об’єктів та структур.

− проведено дослідження залежності часу стабілізації випадкового клітинного автомату від розмірності

Було виконано на практиці:

− розроблено програмне забезпечення, що реалізує основні функції по реалізації еволюції тривимірних автоматів;

− розроблено засоби задання функції впливу під час розвитку динамічного об’єкту;

− розроблено засоби перегляду стану та структури об’єкту під час еволюції.

Розроблений мобільний додаток має два способи реалізації редагування конфігурації клітинного автомату. За допомогою ручого редагування можна задавати початковий стан ітеріцій. Крім того, під час генерації є можливісь втрутитися в процес та відредагувати конфігурацію поточної ітерації. Для редагування ітерації розроблено інструменти зміни стану окремих клітин.

Для перегляду генерації розроблено два методи. Перший – загальний метод перегляду генерації тривимірного клітинного автомату. Другий метод –

перегляд специпічних ділянок автомату, не доступних за допомгогю загального методу.

Ще одна перевага розробленого додатку – можливість зберегти конфігурацію та ще й правило генерування автомату до локальноїї бази даних. Також є можливіть завантажити раніше збережену конфігурацію та продовжити роботу. Крім того, інтерфейс додатку доволі інтуїтивний та простий, що спрощує роботу з клінинним автоматом.

Подальший розвиток розробленого додатку дозволить полегшити процес задання кольорів клітин, збільшити швидкість обрахунку великих систематизованих автоматів, надати функціонал для роботи з автоматами інших типів та зручного розповсюдження збережених конфігурацій мережею Інтернет.

Список використаної літератури

1) *Hedlund, G. A. (1969). "Endomorphisms and automorphisms of the shift dynamical system".* Math. Systems Theory. *3 (4): 320–3751. doi:10.1007 / BF01691062.*

2) *Burton H. Voorhees (1996).* Computational analysis of one-dimensional cellular automata. *World Scientific. p. 8. ISBN 978-981-02-2221-5.*

3) *Weinberg, Steven (24 October 2002). "Is the Universe a Computer?".* The New York Review of Books. *Rea S. Hederman. Retrieved 12 October 2012.*

4) J. P. Crutchfield, "The Calculi of Emergence: Computation, Dynamics, and Induction", Physica D 75, 11–54, 1994.

5) Paul Chapman. Life universal computer. http://www.igblan.free-online.co.uk/igblan/ca/November 2002.

6) *Max Garzon (1995).* Models of massive parallelism: analysis of cellular automata and neural networks. *Springer. p. 149. ISBN 978-3-540-56149-1.*

7) *Sutner, Klaus (1991). "De Bruijn Graphs and Linear Cellular Automata"* Complex Systems. *5: 19–30.*

8) *Von Neumann, John; Burks, Arthur W. (1966).* Theory of Self-Reproducing Automata. *University of Illinois Press.*

9) *Stephen Wolfram*. (2002) A new kind of science. Vol. 5. Wolfram media Champaign.

10) Conway. “The game of life”. Scientific American 223.4 (1970), p. 4.

11) Max Garzon (1995). *Models of massive parallelism: analysis of cellular automata and neural networks*. Springer. p. 149. ISBN 978-3-540-56149-1.

12) Li, Wentian; Packard, Norman (1990). "The structure of the elementary cellular automata rule space" (PDF). *Complex Systems*. 4: 281–297. Retrieved 25 January 2013.

13) Nicolis (1974). "Dissipative Structures, Catastrophes, and Pattern Formation: A Bifurcation Analysis" (PDF). *PNAS*. 71 (7): 2748–2751. Retrieved 25 March 2017.

14) Coombs, Stephen (15 February 2009), *The Geometry and Pigmentation of Seashells*, pp. 3–4, retrieved 2 September 2012

15) Yves Bouligand (1986). *Disordered Systems and Biological Organization*. pp. 374–375.

16) Wolfram S. Cellular automation Fluids.// J.Stat.Phys. 1986. Vol. 45. PP. 471-526.

17) A. K. Dewdney, The hodgepodge machine makes waves, Scientific American, p. 104, August 1988.

18) J. P. Crutchfield, "The Calculi of Emergence: Computation, Dynamics, and Induction", Physica D 75, 11–54, 1994.

19) Minsky, M. "Cellular Vacuum". *International Journal of Theoretical Physics*. 21 (537–551): 1982.

ДОДАТОК А

ПРОГРАМНИЙ КОД

Rule.java

```
public class Rule {

    public static final String TAG = "Rule";
    public static final String ruleDelimeter = "/";
    public static final String numbersDelimeter = ",";

    public int [] keepAliveNeighboursNumber = {4,7};
    public int [] reviveNeighboursNumber = {5};
    private float darkerColorPercent = 0.5f ;

    // Getters
    public int getNeighboursAmount(Cube cube, CubeMap map){
        return neighbours(cube, map).size();
    }

    // Setters
    public void setKeepAliveNeighboursNumber (int [] i){

        this.keepAliveNeighboursNumber = i;

    }

    public void setReviveNeighboursNumber (int [] i){

        this.reviveNeighboursNumber = i;

    }

    public void setDarkerColorPercent (float percent){

        this.darkerColorPercent = percent;

    }

    public Rule(){}
    public Rule(int [] keepAliveNeighboursNumber, int [] reviveNeighboursNumber){

        if(
                keepAliveNeighboursNumber != null &&
                keepAliveNeighboursNumber.length != 0 &&
                reviveNeighboursNumber!= null &&
                reviveNeighboursNumber.length != 0){

            this.reviveNeighboursNumber = reviveNeighboursNumber;
            this.keepAliveNeighboursNumber = keepAliveNeighboursNumber;

        }

    }

    // Main methods

    //calculate the next iteration of the automata
    //and return the list of cubes to render on screen
    public CubeMap nextIterations(CubeMap map){

        ArrayList<Cube> nextIteration = map.toList();

        for(Iterator<Cube> iterator = nextIteration.iterator(); iterator.hasNext();){

            Cube cube = iterator.next();
            ArrayList<Cube> neighbours = neighbours(cube, map);

            countStatus(cube, neighbours);
            countColor(cube, neighbours);


        }

        map.clear();
        map.addAll(nextIteration);

        return map;

    }
```

```
private void countStatus(Cube cube, ArrayList<Cube> neighbours){

    int neighboursAmount = neighbours.size();

    if(cube.isAlive() && inKeepAlive(neighboursAmount)){

        cube.plusIteration();

    }else if (!cube.isAlive() && inRevive(neighboursAmount)){

        cube.setAlive(true);

    }else{

        cube.setAlive(false);
        cube.resetIterations();

    }

}

private void countColor(Cube cube, ArrayList<Cube> neighbours){

    if(!cube.isAlive()) return;

    String newColor = Settings.defaultCubeColor;

    if(Settings.enableColorDarkening){

        if(cube.getIterations() > 0){

            newColor = darkerColor(cube.getColor(), darkerColorPercent);

        }

    }

    if(Settings.enableColorInheritance){

        // TODO

    }


    cube.setColor(newColor);

}

public ArrayList<Cube> neighbours(Cube cube, CubeMap map){

    ArrayList<Cube> result = new ArrayList<>();

    for (int x = cube.getCoords()[0] - 1; x <= cube.getCoords()[0]+1; x++){
        for (int y = cube.getCoords()[1] - 1; y <= cube.getCoords()[1]+1; y++){
            for (int z = cube.getCoords()[2] - 1; z <= cube.getCoords()[2]+1; z++){

                Cube returnCube = map.getCubeAt(new int []{x, y, z});

                if(returnCube != null && returnCube.isAlive())
                    result.add(returnCube);

            }
        }
    }

    return result;

}


// UTILS

private boolean inKeepAlive(int amount){

    for(int i = 0; i < keepAliveNeighboursNumber.length; i++) {

        if(keepAliveNeighboursNumber[i] == amount)
        {
            return true;
        }
```

```
        }

        return false;

    }


    private boolean inRevive(int amount){

        for(int i = 0; i < reviveNeighboursNumber.length; i++) {
            if(reviveNeighboursNumber[i] == amount) {
                return true;
            }
        }

        return false;

    }

    // percent : 0 - 1 where 0 is black, 1 - original color
    private String darkerColor(String hexColor, float percent) {

        CellColor color = new CellColor(hexColor);

        color.RED = color.RED * percent < 0? 0: color.RED * percent;
        color.GREEN = color.GREEN * percent < 0? 0: color.GREEN * percent;
        color.BLUE = color.BLUE * percent < 0? 0: color.BLUE * percent;

        return String.format("#%02x%02x%02x", (int)(color.RED * 255), (int)(color.GREEN * 255), (int)(color.BLUE
* 255));

    }

    public static Rule fromString(String stringRule){

        if(!stringRule.contains(ruleDelimeter)) return null;

        Rule rule = new Rule();

        int [] intKeepAliveNumber = ArrayHelper.stringToIntArray(stringRule.split(ruleDelimeter)[0],
numbersDelimeter);
        int [] intReviveAliveNumber = ArrayHelper.stringToIntArray(stringRule.split(ruleDelimeter)[1],
numbersDelimeter);

        rule.setKeepAliveNeighboursNumber(intKeepAliveNumber);
        rule.setReviveNeighboursNumber(intReviveAliveNumber);

        return rule;

    }

    public String toString(){

        return ArrayHelper.intArrayToString(keepAliveNeighboursNumber, numbersDelimeter) +
                ruleDelimeter +
                ArrayHelper.intArrayToString(reviveNeighboursNumber, numbersDelimeter);

    }

}
```

GameInstance.java

```
public class GameInstance implements ApplicationListener{

    private String TAG = "GameInstance";

    private ModelRenderBuilder modelRenderBuilder;
    private GridRenderBuilder gridRenderBuilder;
    private Automata automata;
    private Environment environment;
    private RendererController rendererController;
    private FPSCounter fps = new FPSCounter();

    private int actionWithCube = 0;
    private boolean gridIsVisible = false;
    private boolean isViewMode = true;

    private AutomataModel testModel;

    @Override
    public void create() {
```

```
        rendererController = LINKER.rendererController;
        environment = new Environment();
        automata = new Automata();

        testModel = AutomataModel.fromCoordsArray(Settings.testSimpleCube, Settings.automataRadius);

        automata.setModel(testModel);
        automata.setRule(new LifeRule());

        modelRenderBuilder = automata.getModelRenderBuilder();
        environment.addBuilder(modelRenderBuilder);

        modelRenderBuilder.setOnTouchListener(new ModelRenderBuilder.OnTouchListener() {
            @Override
            public void onTouch() {

                if(LINKER.activityListener!= null){
                    //LINKER.activityListener.logText("nb: " + String.valueOf(rule.getNeighboursAmount(new
RenderCube(modelRenderBuilder.getTouchResult().touchedCubeCenter, null, false),
modelRenderBuilder.getRenderMap())));
                }

            }
        });


    }

    @Override
    public void render() {

        final int command = rendererController.readCommand();

        switch (command){

            case -1:{

                break;

            }

            //MAIN CONTROLS
            case RendererController.START:{

                automata.start();
                break;

            }
            case RendererController.PAUSE:{

                automata.pause();
                break;

            }
            case RendererController.RESET:{

                testModel = AutomataModel.fromCoordsArray(Settings.testSimpleCube, Settings.automataRadius);
                automata.setModel(testModel);
                LINKER.renderer.resetCam();
                break;

            }
            case RendererController.NEXT:{

                automata.next();
                break;


            }

            case RendererController.LOAD_MODEL: {

                automata.setModel(rendererController.storage.currentModel);
                LINKER.renderer.resetCam();
                break;
            }

            case RendererController.LOAD_RULE:{

                automata.setRule(rendererController.storage.rule);
```

```
            }

            case RendererController.UPDATE_RADIUS:{

                automata.setRadius(Settings.automataRadius);
                if(gridRenderBuilder!= null) gridRenderBuilder.updateGridRadius(Settings.automataRadius);

            }

            //when the figure is touched
            case RendererController.FIGURE_TOUCHED:{

                if(isViewMode) return;

                //adding / painting / deleting a cube
                if(modelRenderBuilder.isTouched()){

                    String color = Integer.toHexString(rendererController.currentColor);

                    if(color.length()>=6){ color = "#" + color.substring(2); }

                    if(actionWithCube == RendererController.ADD_CUBE){


if(!modelRenderBuilder.cubeInCurrentOpenedLayer(modelRenderBuilder.getTouchResult().newCubeCenter)) break;

                        Cube cube = new Cube(color, modelRenderBuilder.getTouchResult().newCubeCenter);
                        cube.setAlive(true);

                        automata.addNewCube(cube);

                    }
                    if(actionWithCube == RendererController.REMOVE_CUBE){


if(!modelRenderBuilder.cubeInCurrentOpenedLayer(modelRenderBuilder.getTouchResult().touchedCubeCenter)) break;

                        automata.removeCube(new Cube(modelRenderBuilder.getTouchResult().touchedCubeCenter));

                    }
                    if(actionWithCube == RendererController.PAINT_CUBE){


if(!modelRenderBuilder.cubeInCurrentOpenedLayer(modelRenderBuilder.getTouchResult().touchedCubeCenter)) break;

                        automata.paintCube(new Cube(color,
modelRenderBuilder.getTouchResult().touchedCubeCenter));

                    }

                }

                break;

            }

            case RendererController.STRETCH:{

                modelRenderBuilder.stretch();
                break;

            }
            case RendererController.SQUEEZE:{

                modelRenderBuilder.squeeze();
                break;

            }
            case RendererController.EDIT_MODE:{

                modelRenderBuilder.squeeze();
                modelRenderBuilder.setViewMode(false);
                this.isViewMode = false;

                if(gridRenderBuilder == null){
                    gridRenderBuilder = new GridRenderBuilder(automata.getAutomataRadius());
                }

                gridRenderBuilder.build();
                gridRenderBuilder.bindAttributesData();
```

```
            gridRenderBuilder.reset();
            environment.addGrid(gridRenderBuilder);

            break;

        }
        case RendererController.VIEW_MODE:{

            this.isViewMode = true;
            modelRenderBuilder.setViewMode(true);
            gridIsVisible = false;
            environment.removeGrids();
            LINKER.renderer.resetAdditionalStride();
            break;
        }
        case RendererController.LAYER_UP:{

            modelRenderBuilder.layerUp();
            gridRenderBuilder.translateGrid((int)Settings.renderCubeSize);

            break;
        }

        case RendererController.LAYER_DOWN:{

            modelRenderBuilder.layerDown();
            gridRenderBuilder.translateGrid(-(int)Settings.renderCubeSize);

            break;
        }

        case RendererController.SHOW_GRID:{

            gridIsVisible = true;
            modelRenderBuilder.setSliced(true);
            break;
        }
        case RendererController.HIDE_GRID:{

            gridIsVisible = false;
            modelRenderBuilder.setSliced(false);
            break;
        }
        case RendererController.PAINT_CUBE:{

            actionWithCube = RendererController.PAINT_CUBE;
            break;

        }
        case RendererController.ADD_CUBE:{

            actionWithCube = RendererController.ADD_CUBE;
            break;

        }
        case RendererController.REMOVE_CUBE:{

            actionWithCube = RendererController.REMOVE_CUBE;
            break;

        }


    }

    LINKER.renderer.setFigureUniforms();
    LINKER.renderer.setFigureScaleFactor();

    automata.execute();
    modelRenderBuilder.draw();

    if(gridIsVisible){

        LINKER.renderer.setGridUniforms();
        LINKER.renderer.setGridScaleFactor();

        gridRenderBuilder.draw();
    }

    LINKER.activityListener.logFps(fps.frames());
```

```
    }

    public AutomataModel getCurrentModel(){

        if(automata == null) return null;

        return automata.getModel();

    }

    public Rule getCurrentRule(){

        if(automata == null) return null;

        return automata.getRule();

    }


}
```

GraphicsRenderer.java

```
package com.cellular.automata.cellularautomata;

import android.graphics.Bitmap;
import android.opengl.GLES20;
import android.opengl.GLSurfaceView;
import android.opengl.Matrix;

import com.cellular.automata.cellularautomata.interfaces.MainView;
import com.cellular.automata.cellularautomata.interfaces.ApplicationListener;
import com.cellular.automata.cellularautomata.interfaces.EnvironmentListener;
import com.cellular.automata.cellularautomata.interfaces.ScreenshotListener;
import com.cellular.automata.cellularautomata.shaders.FigureShader;
import com.cellular.automata.cellularautomata.shaders.GridShader;
import com.cellular.automata.cellularautomata.data.CellColor;
import com.cellular.automata.cellularautomata.utils.CubeCenter;
import com.cellular.automata.cellularautomata.utils.ObjectSelectHelper;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.ArrayList;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import static android.opengl.GLES20.glReadPixels;
import static android.opengl.Matrix.invertM;
import static android.opengl.Matrix.multiplyMM;
import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.rotateM;
import static android.opengl.Matrix.translateM;

public class GraphicsRenderer implements GLSurfaceView.Renderer {

    private MainView activityListener;
    private ApplicationListener applicationListener;
    private EnvironmentListener environmentListener;
    private ScreenshotListener screenshotListener;

    //shaders
    private FigureShader figureShader;
    private GridShader gridShader;

    private volatile float xAngle = -45f;
    private volatile float yAngle = 10f;

    private float lightXAngle = -90f;
    private float lightYAngle = 10f;

    //moving figure left-right, up-down
    private float strideX = 0f;
    private float strideY = 0f;
    private float additionalLayerStrideY = 0f;

    //screen
    private int width;
    private int height;

    //initial scale
```

```
    private float scaleFactor = 1f;

    //other
    private boolean screenshot = false;
    private boolean figureVisible = true;
    private boolean gridVisible = false;

    //mvp matrices
    private float[] viewMatrix = new float[16];
    private float[] modelMatrix = new float[16];
    private float[] figureModelMatrix = new float[16];
    private float[] gridModelMatrix = new float[16];
    private float[] projectionMatrix = new float[16];
    private float[] viewProjectionMatrix = new float[16];
    private float[] MVPMatrix = new float[16];
    //matrix that undo the effects of view and projection matrix
    private final float[] invertedViewProjectionMatrix = new float[16];

    //light matrices
    private final float[] lightPosInModelSpace = new float[]{0.0f, 0.0f, 0.0f, 1.0f};

    private float[] frontLightModelMatrix = new float[16];
    private final float[] frontLightPosInWorldSpace = new float[4];
    private final float[] frontLightPosInEyeSpace = new float[4];

    private float[] backLightModelMatrix = new float[16];
    private final float[] backLightPosInWorldSpace = new float[4];
    private final float[] backLightPosInEyeSpace = new float[4];

    private float[] rightLightModelMatrix = new float[16];
    private final float[] rightLightPosInWorldSpace = new float[4];
    private final float[] rightLightPosInEyeSpace = new float[4];

    private float[] topLightModelMatrix = new float[16];
    private final float[] topLightPosInWorldSpace = new float[4];
    private final float[] topLightPosInEyeSpace = new float[4];

    private float[] leftLightModelMatrix = new float[16];
    private final float[] leftLightPosInWorldSpace = new float[4];
    private final float[] leftLightPosInEyeSpace = new float[4];

    private float[] bottomLightModelMatrix = new float[16];
    private final float[] bottomLightPosInWorldSpace = new float[4];
    private final float[] bottomLightPosInEyeSpace = new float[4];

    public FigureShader getShader(){return figureShader;}
    public GridShader getGridShader(){return gridShader;}

    public void setFigureScaleFactor(float scaleFactor) {this.scaleFactor = scaleFactor; }
    public float getScaleFactor() {return scaleFactor;}

    public void setStride(float strideX, float strideY){this.strideX = strideX; this.strideY = strideY; }
    public float getStrideX(){return strideX;}
    public float getStrideY(){return strideY;}
    public void setXAngle(float xAngle) { this.xAngle = xAngle; }
    public void setYAngle(float yAngle) { this.yAngle = yAngle > 360? yAngle - 360: yAngle; }
    public float getXAngle() {return this.xAngle; }
    public float getYAngle() { return this.yAngle; }
    public void translateFigureVertical(float distance){ additionalLayerStrideY += distance;}
    public void resetAdditionalStride(){additionalLayerStrideY = 0f;}
    public void resetCam(){

        xAngle = -45f;
        yAngle = 10f;

        strideX = 0f;
        strideY = 0f;
        resetAdditionalStride();

        scaleFactor = 1;

    }

    public void screenshot(ScreenshotListener listener){this.screenshotListener = listener; this.screenshot =
true;}


    public void handleTouch(float x, float y){

        if(environmentListener!=null ){
            environmentListener.onScreenTouched(x, y);
```

```
        }
    }

    public void setEnvironmentListener(EnvironmentListener environmentListener) {
        this.environmentListener = environmentListener;
    }

    public void setActicvityListener(MainView activityListener){
        this.activityListener = activityListener;
    }


    GraphicsRenderer(ApplicationListener applicationListener){

        this.applicationListener = applicationListener;

    }

    public ObjectSelectHelper.TouchResult getTouchedResult(float normalizedX, float normalizedY,
ArrayList<CubeCenter> cellCentersList){

        return ObjectSelectHelper.getTouchResult(cellCentersList, normalizedX, normalizedY,
invertedViewProjectionMatrix, modelMatrix, scaleFactor, strideX, strideY, (float)height/width);

    }



    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {

        CellColor bgColor =  new CellColor(Settings.backgroundColor);
        GLES20.glClearColor(bgColor.RED, bgColor.GREEN, bgColor.BLUE, 0.0f);

        // Remove faces that are the back side to the screen
        GLES20.glEnable(GLES20.GL_CULL_FACE);

        // Enable depth testing to remove drawing objects that are behind other objects
        GLES20.glEnable(GLES20.GL_DEPTH_TEST);


        // Position the eye in front of the origin.
        final float eyeX = 0.0f;
        final float eyeY = 0.0f;
        final float eyeZ = -0.5f;

        // We are looking toward the distance
        final float lookX = 0.0f;
        final float lookY = 0.0f;
        final float lookZ = -5.0f;

        // Set our up vector. This is where our head would be pointing were we holding the camera.
        final float upX = 0.0f;
        final float upY = 1.0f;
        final float upZ = 0.0f;

        // Set the view matrix. This matrix can be said to represent the camera position.
        // NOTE: In OpenGL 1, a ModelView matrix is used, which is a combination of a model and
        // view matrix. In OpenGL 2, we can keep track of these matrices separately if we choose.
        Matrix.setLookAtM(viewMatrix, 0, eyeX, eyeY, eyeZ, lookX, lookY, lookZ, upX, upY, upZ);

        figureShader = new FigureShader(activityListener);
        gridShader = new GridShader(activityListener);


        if ( applicationListener == null) return;

        applicationListener.create();

    }

    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {

        // Set the OpenGL viewport to the same size as the surface.
        GLES20.glViewport(0, 0, width, height);

        // Create a new perspective projection matrix. The height will stay the same
        // while the width will vary as per aspect ratio.
        final float ratio = (float) width / height;
        final float left = -ratio;
```

```
    final float right = ratio;
    final float bottom = -1.0f;
    final float top = 1.0f;
    final float near = 3.0f;
    final float far = 100.0f;

    this.width = width;
    this.height = height;

    Matrix.orthoM(projectionMatrix, 0, left,right, bottom, top, near, far);

    if ( applicationListener == null) return;

}

@Override
public void onDrawFrame(GL10 gl) {

    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT | GLES20.GL_DEPTH_BUFFER_BIT);

    calculateModelMatrix();

    calculateLightMatrices(xAngle, yAngle);

    calculateInvertedMVPMatrix();

    if ( applicationListener == null) return;

    applicationListener.render();

    if(this.screenshot){takeScreenshot(); screenshot = false;}

}


private void calculateModelMatrix(){

    //manipulations with the cubes model matrix
    //push figure to the distance
    Matrix.setIdentityM(modelMatrix, 0);
    Matrix.translateM(modelMatrix, 0, 0f, 0f, -7.0f);

    //set user made stride
    Matrix.translateM(modelMatrix, 0, strideX/scaleFactor, -strideY/scaleFactor, 0.0f);

    //set user made rotation
    rotateM(modelMatrix, 0, yAngle, 1f, 0f, 0f);
    rotateM(modelMatrix, 0, xAngle, 0f, 1f, 0f);

    System.arraycopy(modelMatrix, 0, gridModelMatrix, 0, modelMatrix.length);

    //additional stride made when editing figure
    Matrix.translateM(gridModelMatrix, 0, 0, -additionalLayerStrideY, 0);




}


private void calculateLightMatrices(float xAngle, float yAngle){

    Matrix.setIdentityM(frontLightModelMatrix, 0);
    Matrix.translateM(frontLightModelMatrix, 0, 0.0f, 0.0f, -7.0f);

    //count all the light source position
    float lightDistance = Settings.lightDistance;

    rotateM(frontLightModelMatrix, 0, yAngle, 1f, 0f, 0f);
    rotateM(frontLightModelMatrix, 0, xAngle, 0f, 1f, 0f);

    System.arraycopy(frontLightModelMatrix ,0, backLightModelMatrix, 0, frontLightModelMatrix.length);
    System.arraycopy(frontLightModelMatrix ,0, rightLightModelMatrix, 0, frontLightModelMatrix.length);
    System.arraycopy(frontLightModelMatrix ,0, topLightModelMatrix, 0, frontLightModelMatrix.length);

    Matrix.translateM(backLightModelMatrix, 0, 0.0f, 0.0f, -lightDistance);
    Matrix.translateM(rightLightModelMatrix, 0, lightDistance, 0.0f, 0.0f);
    Matrix.translateM(topLightModelMatrix,  0, 0.0f, lightDistance, 0.0f);
```

```
        multiplyMV(frontLightPosInWorldSpace, 0, frontLightModelMatrix, 0, lightPosInModelSpace, 0);
        multiplyMV(frontLightPosInEyeSpace, 0, viewMatrix, 0, frontLightPosInWorldSpace, 0);

        multiplyMV(backLightPosInWorldSpace, 0, backLightModelMatrix, 0, lightPosInModelSpace, 0);
        multiplyMV(backLightPosInEyeSpace, 0, viewMatrix, 0, backLightPosInWorldSpace, 0);

        multiplyMV(rightLightPosInWorldSpace, 0, rightLightModelMatrix, 0, lightPosInModelSpace, 0);
        multiplyMV(rightLightPosInEyeSpace, 0, viewMatrix, 0, rightLightPosInWorldSpace, 0);

        multiplyMV(topLightPosInWorldSpace, 0, topLightModelMatrix, 0, lightPosInModelSpace, 0);
        multiplyMV(topLightPosInEyeSpace, 0, viewMatrix, 0, topLightPosInWorldSpace, 0);

    }

    private void calculateInvertedMVPMatrix(){

        multiplyMM(viewProjectionMatrix, 0, viewMatrix, 0, projectionMatrix, 0);
        invertM(invertedViewProjectionMatrix, 0, viewProjectionMatrix, 0);
        translateM(invertedViewProjectionMatrix, 0, 0f, 0f, -10f);

    }

    public void setFigureUniforms(){

        // Set our per-vertex lighting gridShader.
        figureShader.useProgram();
        figureShader.setUniforms(modelMatrix, viewMatrix, projectionMatrix,
                    frontLightPosInEyeSpace,
                    backLightPosInEyeSpace,
                    leftLightPosInEyeSpace,
                    rightLightPosInEyeSpace,
                    topLightPosInEyeSpace,
                    bottomLightPosInEyeSpace);

    }

    public void setGridUniforms(){

        gridShader.useProgram();
        gridShader.setUniforms(gridModelMatrix, viewMatrix, projectionMatrix);

    }

    public void setFigureScaleFactor(){

        figureShader.setScaleFactor(scaleFactor);

    }

    public void setGridScaleFactor(){

        gridShader.setScaleFactor(scaleFactor);

    }

    //UTILS
    private void takeScreenshot(){

        int screenshotSize = width * height;
        ByteBuffer bb = ByteBuffer.allocateDirect(screenshotSize * 4);
        bb.order(ByteOrder.nativeOrder());
        glReadPixels(0, 0, width, height, GLES20.GL_RGBA, GLES20.GL_UNSIGNED_BYTE, bb);
        int pixelsBuffer[] = new int[screenshotSize];
        bb.asIntBuffer().get(pixelsBuffer);
        bb = null;

        for (int i = 0; i < screenshotSize; ++i) {
            // The alpha and green channels' positions are preserved while the      red and blue are swapped
            pixelsBuffer[i] = ((pixelsBuffer[i] & 0xff00ff00)) |    ((pixelsBuffer[i] & 0x000000ff) << 16) |
((pixelsBuffer[i] & 0x00ff0000) >> 16);
        }

        Bitmap image = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
        image.setPixels(pixelsBuffer, screenshotSize-width, -width, 0, 0, width, height);

        if(screenshotListener != null){

            screenshotListener.onScreenShot(image);

        }
```

```
    }


}
```

RenderCubeMap.java

```
package com.cellular.automata.cellularautomata.data;

import android.util.Log;

import com.cellular.automata.cellularautomata.objects.RenderCube;
import com.cellular.automata.cellularautomata.utils.CubeCenter;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class RenderCubeMap {

    private String TAG = "RENDER_CUBE_MAP";

    private int automataRadius = 0;

    //the main array - represents the 3-D models
    private RenderCube[][][] map;
    //just a list of all the cubes
    private ArrayList<RenderCube> renderCubeList = new ArrayList<>();

    public RenderCubeMap(int automataRadius){

        this.automataRadius = automataRadius;
        this.map = new RenderCube[automataRadius * 2 - 1][automataRadius * 2 - 1][automataRadius * 2 - 1];

    }

    //GETTERS
    public ArrayList<RenderCube> getRenderCubeList() {
        return sort();
    }

    public ArrayList<CubeCenter> getCubeCenters(){

        ArrayList<CubeCenter> result = new ArrayList<>();
        for (RenderCube renderCube : renderCubeList){
            result.add(renderCube.center);
        }

        return result;

    }

    public ArrayList<CubeCenter> getLayer(int height){

        // can rewrite to make it faster

        ArrayList<CubeCenter> result = new ArrayList<>();

        for (RenderCube renderCube : renderCubeList){
            if((int)renderCube.center.y == height){

                result.add(renderCube.center);

            }

        }

        return result;

    }

    public int size(){
        return renderCubeList.size();
    }

    public RenderCube getCubeAt(int[] coords){

        if (!cubeExists(new CubeCenter(coords[0], coords[1], coords[2])))
            return null;
```

```
        int[] mapCoords = cubeCoordsToMapCoords(coords);
        return map[mapCoords[0]][mapCoords[1]][mapCoords[2]];

    }

    public RenderCube getCubeByCenter(CubeCenter center){

        if (!cubeExists(center))
            return null;

        int[] mapCoords = cubeCoordsToMapCoords(new int[]{(int)center.x, (int)center.y, (int)center.z});
        return map[mapCoords[0]][mapCoords[1]][mapCoords[2]];

    }

    public int height(){

        int low = 10000;
        int high = -10000;

        for (RenderCube renderCube : renderCubeList){

            if(renderCube.center.y < low)
                low = (int) renderCube.center.y;

            if(renderCube.center.y > high)
                high = (int) renderCube.center.y;

        }

        return high - low +1;

    }

    public CubeMap toCubeMap (){

        CubeMap cubeMap = new CubeMap(automataRadius);

        for(RenderCube cube: renderCubeList){

            Cube newCube = new Cube(cube.color.hexColor, new int[]{(int)cube.center.x, (int)cube.center.y,
(int)cube.center.z});
            newCube.setAlive(true);
            cubeMap.add(newCube);

        }

        return cubeMap;

    }

    // Converts CubeMap to RenderCubeMap
    public RenderCubeMap fromCubeMap(CubeMap map){

        return fromCubeList(map.toList(), map.getAutomataRadius());


    }

    public static RenderCubeMap fromCubeList(ArrayList<Cube> cubeList, int automataRadius){

        RenderCubeMap newMap = new RenderCubeMap(automataRadius);

        for (Cube cube: cubeList){

            if(cube.isAlive()){

                int [] coords = cube.getCoords();
                RenderCube renderCube = new RenderCube(new CubeCenter((float) coords[0], (float)coords[1],
(float)coords[2]), new CellColor(cube.getColor()), true);
                newMap.add(renderCube);

            }


        }

        return newMap;

    }
```

```
//MAIN METHODS

public void add(RenderCube renderCube){

    //checking renderCube stuff

    if(cubeExists(renderCube)) {
        Log.d(TAG, "renderCube already exists");
        return;
    }

    //actual adding the renderCube

    int [] coordsVector = cubeToMapCoords(renderCube);

    map[coordsVector[0]][coordsVector[1]][coordsVector[2]] = renderCube;

    renderCubeList.add(renderCube);

}

public void addAll(ArrayList<RenderCube> renderCubeList){

    for(RenderCube renderCube : renderCubeList){

        add(renderCube);

    }

}


public void remove(RenderCube renderCube){

    //checking renderCube stuff

    if(!cubeExists(renderCube)) {
        Log.d(TAG, "renderCube does not exist");
        return;
    }

    //actually removing renderCube

    int[] coords = cubeToMapCoords(renderCube);

    map[coords[0]][coords[1]][coords[2]] = null;

    Iterator<RenderCube> iterator = renderCubeList.iterator();

    while(iterator.hasNext()){

        if(iterator.next().center.equals(renderCube.center)){
            iterator.remove();
        }

    }

}


//ADDITIONAL STUFF
public boolean cubeExists(CubeCenter center){

    return cubeExists(new RenderCube(center, null, false));

}

public boolean cubeExists(RenderCube renderCube){

    int[] coords = cubeToMapCoords(renderCube);
    return cubeInBounds(renderCube) && map[coords[0]][coords[1]][coords[2]] != null;

}

private int[] cubeToMapCoords(RenderCube renderCube){

    return cubeCoordsToMapCoords(new int[]{
            (int) renderCube.center.x,
            (int) renderCube.center.y,
```

```
                (int) renderCube.center.z});

    }

    private int[] cubeCoordsToMapCoords(int[] coords){

        return new int[] {
                coords[0] + automataRadius -1,
                coords[1] + automataRadius -1,
                coords[2] + automataRadius -1
        };

    }


    public boolean cubeInBounds(RenderCube renderCube){

        int[] coordsVector = cubeToMapCoords(renderCube);

        int pos_x = Math.abs((int) renderCube.center.x);
        int pos_y = Math.abs((int) renderCube.center.y);
        int pos_z = Math.abs((int) renderCube.center.z);

        if(Math.abs(pos_x) > automataRadius -1 ||
                Math.abs(pos_y) > automataRadius -1 ||
                Math.abs(pos_z) > automataRadius -1){

            Log.d(TAG, "RenderCube center is out of automata bounds");
            Log.d(TAG, "Automata radius = " + automataRadius + " | RenderCube center: " +
                    String.valueOf(renderCube.center.x) + "," +
                    String.valueOf(renderCube.center.y) + "," +
                    String.valueOf(renderCube.center.x));
            return false;

        }

        return true;

    }

    public ArrayList<RenderCube> sort(){

        Collections.sort(renderCubeList);
        return renderCubeList;

    }

    public void clear(){

        this.map = new RenderCube[automataRadius * 2][automataRadius * 2][automataRadius * 2];
        this.renderCubeList.clear();

    }

}
```

ModelRendererBuilder

```
package com.cellular.automata.cellularautomata.objects;

import android.opengl.GLES20;
import android.util.Log;

import com.cellular.automata.cellularautomata.LINKER;
import com.cellular.automata.cellularautomata.GraphicsRenderer;
import com.cellular.automata.cellularautomata.animation.Animator;
import com.cellular.automata.cellularautomata.data.CubeDataHolder;
import com.cellular.automata.cellularautomata.data.RenderCubeMap;
import com.cellular.automata.cellularautomata.data.VertexArray;
import com.cellular.automata.cellularautomata.interfaces.CellSelectListener;
import com.cellular.automata.cellularautomata.data.CellColor;
import com.cellular.automata.cellularautomata.utils.CubeCenter;
import com.cellular.automata.cellularautomata.utils.ObjectSelectHelper;

import java.util.ArrayList;

import static android.opengl.GLES20.GL_ARRAY_BUFFER;
import static android.opengl.GLES20.glBindBuffer;
import static android.opengl.GLES20.glDeleteBuffers;
import static android.opengl.GLES20.glDrawArrays;
import static android.opengl.GLES20.glEnableVertexAttribArray;
```

```
import static android.opengl.GLES20.glGenBuffers;
import static android.opengl.GLES20.glVertexAttribPointer;
import static com.cellular.automata.cellularautomata.Constants.COLOR_COMPONENT_COUNT;
import static com.cellular.automata.cellularautomata.Constants.NORMAL_COMPONENT_COUNT;
import static com.cellular.automata.cellularautomata.Constants.POSITION_COMPONENT_COUNT;

public class ModelRenderBuilder {

    private String TAG = "RENDER_BUILDER";

    private VertexArray vertexPosArray;
    private VertexArray vertexColorArray;
    private VertexArray vertexNormalArray;

    private float[] vertexPositionData;
    private float[] vertexColorData;
    private float[] vertexNormalData;

    private int vertexDataOffset = 0;
    private int vertexColorDataOffset = 0;
    private int vertexNormalDataOffset = 0;

    private int vertexBufferPositionIdx = 0;
    private int vertexBufferColorIdx = 0;
    private int vertexBufferNormalIdx = 0;

    //the automata radius (from center to the bound)
    private int automataRadius = 0;
    private RenderCubeMap renderMap;

    private CellSelectListener selectListener;
    private boolean isTouched = false;
    private boolean isStretched = false;
    private boolean isSliced = false;
    private boolean viewMode = true;
    private int currentLayerOpened = 0;

    private ObjectSelectHelper.TouchResult touchResult;

    private OnTouchListener onTouchListener;
    private Animator animator;


    public interface OnTouchListener{

        void onTouch();

    }

    public ModelRenderBuilder(int automataRadius){

        this.renderMap = new RenderCubeMap(automataRadius);

    }


    //GETTERS

    public ObjectSelectHelper.TouchResult getTouchResult() {
        return touchResult;
    }

    public ArrayList<CubeCenter> getCellCentersList(){

        if(isSliced){

            return renderMap.getLayer(currentLayerOpened);

        }else{

            return renderMap.getCubeCenters();

        }

    }

    public RenderCubeMap getRenderMap() {

        return renderMap;

    }
```

```
//SETTERS

//this method can be called from the Application Manager
public void setSelectListener(CellSelectListener listener){

    if(listener != null) this.selectListener = listener;

}

public void setAutomataRadius(int radius){

    this.automataRadius = radius;

}


public void setOnTouchListener(OnTouchListener listener){

    this.onTouchListener = listener;

}

public void setRenderMap(RenderCubeMap map){

    this.renderMap = map;
    animator = new Animator(renderMap.size(), renderMap.height(), renderMap.getRenderCubeList());

}



//MAIN METHODS
public boolean isTouched(){

    if(isTouched){
        isTouched = false;
        return true;
    }
    return false;
}

public void handleTouch(ObjectSelectHelper.TouchResult touchResult){

    this.touchResult = touchResult;
    isTouched = true;
    if(onTouchListener != null) onTouchListener.onTouch();

}

public void setViewMode(boolean isViewMode){

    this.viewMode = isViewMode;
    isSliced = false;
    resetCurrentLayer();

}

public void setSliced(boolean isSliced){

    this.isSliced = isSliced;

}

public void layerUp(){

    this.currentLayerOpened += 1;

}

public void layerDown(){

    this.currentLayerOpened -= 1;

}



private void resetCurrentLayer(){
    this.currentLayerOpened = 0;
}
```

```
    public void stretch(){

        if(isStretched) return;

        animator = new Animator(renderMap.size(), renderMap.height(), renderMap.sort());
        isStretched = true;

    }

    public void squeeze(){

        if(!isStretched) return;
        isStretched = false;

    }

    //general function, is called first when want to add a cubed
    public void addNewCube(CubeCenter center, CellColor color){

        addNewCube(new RenderCube(center, color));

    }

    public void addNewCube(RenderCube renderCube){

        renderMap.add(renderCube);
        //rebuild figure
        build();
        bindAttributesData();

    }

    public void addAllCubes(ArrayList<RenderCube> renderCubes){

        renderMap.addAll(renderCubes);
        //rebuild figure
        build();
        bindAttributesData();

    }

    //general function is called first when we want to paint the cube
    public void paintCube(CubeCenter center, CellColor color){

        RenderCube renderCubeToRecolor = renderMap.getCubeByCenter(center);
        if(renderCubeToRecolor == null) return;

        renderCubeToRecolor.paintCube(color);
        build();
        bindAttributesData();

    }

    //general function is called first when we want to delete the cube
    public void deleteCube(CubeCenter center){

        renderMap.remove(new RenderCube(center, new CellColor ("#ffffff")));


    }



    //here all the vertex data should be generated
    public void build(){

        if(howManyCells() <= 0){
            Log.d(TAG, "no cells to build");
            return;
        }

        vertexColorDataOffset = 0;
        vertexDataOffset = 0;
        vertexNormalDataOffset = 0;

        vertexPositionData = new float[CubeDataHolder.getInstance().sizeInVertex * POSITION_COMPONENT_COUNT *
howManyCells()];
        vertexNormalData = new float[CubeDataHolder.getInstance().sizeInVertex * NORMAL_COMPONENT_COUNT *
howManyCells()];
```

```
        vertexColorData = new float[(vertexPositionData.length / POSITION_COMPONENT_COUNT) *
COLOR_COMPONENT_COUNT];

        for (RenderCube renderCube : renderMap.getRenderCubeList()){

            appendCell(renderCube);

        }

        vertexPosArray = new VertexArray(vertexPositionData);
        vertexColorArray = new VertexArray(vertexColorData);
        vertexNormalArray = new VertexArray(vertexNormalData);

        vertexPositionData = null;
        vertexNormalData = null;
        vertexColorData = null;

    }

    private void appendCell(RenderCube renderCube){

        for (float f: renderCube.getCubePositionData()){
            vertexPositionData[vertexDataOffset++] = f;
        }

        for (float f: renderCube.getCubeNormalData()){
            vertexNormalData[vertexNormalDataOffset++] = f;
        }

        for (float f: renderCube.getCubeColorData()){
            vertexColorData[vertexColorDataOffset++] = f;
        }

        renderCube.releaseCubeData();

    }

    //This method can be called only inside the GL context
    //binds data to the VBOs
    public void bindAttributesData(){

        if(howManyCells() <= 0){
            Log.d(TAG, "No data to bind");
            return;
        }

        glDeleteBuffers(3, new int[]{vertexBufferPositionIdx, vertexBufferColorIdx, vertexBufferNormalIdx}, 0);

        final int buffers[] = new int[3];
        glGenBuffers(3, buffers, 0);
        glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);

        vertexBufferPositionIdx = buffers[0];
        vertexBufferColorIdx = buffers[1];
        vertexBufferNormalIdx = buffers[2];

        vertexPosArray.bindBufferToVBO(vertexBufferPositionIdx);
        vertexColorArray.bindBufferToVBO(vertexBufferColorIdx);
        vertexNormalArray.bindBufferToVBO(vertexBufferNormalIdx);

    }

    public void draw(){

        GraphicsRenderer renderer = LINKER.renderer;

        if(howManyCells() <= 0){
            Log.d(TAG, "No cells to draw");
            return;
        }

        //draw figure
        glBindBuffer(GL_ARRAY_BUFFER, vertexBufferPositionIdx);
        glEnableVertexAttribArray(renderer.getShader().getPositionAttributeLocation());
        glVertexAttribPointer(renderer.getShader().getPositionAttributeLocation(), POSITION_COMPONENT_COUNT,
GLES20.GL_FLOAT, false, 0, 0);


        glBindBuffer(GL_ARRAY_BUFFER, vertexBufferColorIdx);
        glEnableVertexAttribArray(renderer.getShader().getColorAttributeLocation());
```

```
        glVertexAttribPointer(renderer.getShader().getColorAttributeLocation(), COLOR_COMPONENT_COUNT,
GLES20.GL_FLOAT, false, 0, 0);


        glBindBuffer(GL_ARRAY_BUFFER, vertexBufferNormalIdx);
        glEnableVertexAttribArray(renderer.getShader().getNormalAttributeLocation());
        glVertexAttribPointer(renderer.getShader().getNormalAttributeLocation(), NORMAL_COMPONENT_COUNT,
GLES20.GL_FLOAT, false, 0, 0);



        if(viewMode){

            if(isStretched){

                animator.drawFullStretchedFigure(renderer.getShader());

            }else{

                animator.drawClosedFigure(renderer.getShader());

            }

        }else{


            if(isSliced){

                animator.drawSlicedFigure(currentLayerOpened, renderer.getShader());

            }else{

                float[] resetScatterVector = {0.0f, 0.0f, 0.0f};
                renderer.getShader().setScatter(resetScatterVector);
                glDrawArrays(GLES20.GL_TRIANGLES, 0, CubeDataHolder.getInstance().sizeInVertex * howManyCells());

            }



        }


    }

    //additional stuff

    private int howManyCells(){

        return renderMap.size();

    }

    public boolean cubeExists(CubeCenter cubeCenter){

        return renderMap.cubeExists(cubeCenter);

    }

    public boolean cubeInCurrentOpenedLayer(CubeCenter center) {

        return !isSliced || (int) center.y == currentLayerOpened;

    }

}
```

ДОДАТОК Б
ГРАФІЧНІ МАТЕРІАЛИ