**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра автоматизованих систем обробки інформації i управління**

|  |  |
| --- | --- |
| «На правах рукопису»  УДК 004.924:004.021:515.1 | «До захисту допущено»  В.о. завідувача кафедри  \_\_\_\_\_\_\_\_\_\_\_\_\_ І. П. Муха  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2017 р. |

**Магістерська дисертація**

**на здобуття ступеня магістра**

зі спеціальності 8.05010301 «Програмне забезпечення систем»

на тему: “Дослідження властивостей тривимірних клітинних автоматів”

Виконав: студент VI курсу, групи ІП-51м

|  |  |  |
| --- | --- | --- |
|  | Котолупов Олег Владиславович | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Науковий керівник | доцент, канд. техн. наук  Фіногенов О.Д. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Консультант із нормоконтролю | доцент, канд. техн. наук  Ліщук К.І. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Рецензент | доцент, канд. техн. наук  Яблонський П.М. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

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

Студент \_\_\_\_\_\_\_\_\_\_\_\_

Київ — 2017

**Національний технічний університет україни**

**«Київський політехнічний інститут  
імені Ігоря Сікорського»**

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації i управління

Рівень вищої освіти — другий (магістерський)

Спеціальність 8.05010301 «Програмне забезпечення систем»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

\_\_\_\_\_\_\_\_\_\_\_\_\_ І. П. Муха

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2017 р.

**ЗАВДАННЯ**

**на магістерську дисертацію студенту**

Котолупову Олегу Владиславовичу

1. Тема дисертації: «Дослідження властивостей тривимірних клітинних автоматів»,

науковий керівник дисертації Фіногенов Олексій Дмитрович, канд. техн. наук, доцент

2. Термін подання студентом дисертації: «25» січня 2017 р.

3. Об’єкт дослідження: тривимірні клітинні автомати із правилами, аналогічними грі “Життя”.

4. Предмет дослідження: пошук тривимірних клітинних автоматів 4 класу, що виказують ознаки переходу від хаосу до впорядкованості.

5. Перелік завдань, які потрібно розробити: розглянути існуючі програми для роботи із клітинними автоматами загалом та зокрема тривимірними клітинними автоматами, сформулювати вимоги до програми для дослідження властивостей тривимірних клітинних автоматів, розробити програму для надання необхідного для дослідження функціоналу, розглянути варіанти правил клітинних автоматів з метою пошуку автоматів 4 класу.

6. Орієнтовний перелік ілюстративного матеріалу: діаграма класів розробленої програми, ієрархія об’єктів сцени Unity, діаграма варіантів використання програми

7. Орієнтовний перелік публікацій: тези “Дослідження властивостей тривимірних клітинних автоматів”, стаття “Розробка програмного забезпечення для моделювання трьохвимірних клітинних апаратів”

8. Консультанти розділів дисертації:

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
| завдання видав | завдання прийняв |
|  |  |  |  |

9. Дата видачі завдання: «3» листопада 2016 р.

Календарний план

| № з/п | Назва етапів виконання магістерської дисертації | Термін виконання етапів магістерської дисертації | Примітка |
| --- | --- | --- | --- |
| 1 | Ґрунтовне ознайомлення з предметною областю | 15.12.2015 |  |
| 2 | Визначення структури магістерської дисертації; вивчення літератури | 01.03.2016 |  |
| 3 | Робота над першим розділом магістерської дисертації | 15.05.2016 |  |
| 4 | Проведення наукового дослідження; робота над другим розділом магістерської дисертації | 15.10.2016 |  |
| 5 | Проведення наукового дослідження; робота над статтею за результатами наукового дослідження | 15.12.2016 |  |
| 6 | Робота над третім розділом магістерської дисертації; підготовка статті за результатами наукового дослідження; розроблення програмного забезпечення | 01.03.2017 |  |
| 7 | Робота над четвертим розділом магістерської дисертації; завершення роботи над основною частиною магістерської дисертації | 15.05.2017 |  |
| 8 | Оформлення текстової і графічної частин магістерської дисертації | 25.05.2017 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Котолупов О.В. |  |
| Науковий керівник дисертації | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Фіногенов О.Д. | |

Реферат

Дисертацію виконано на 123 аркуші, вона містить 1 додаток на 27 листах та перелік посилань на використані джерела з 46 найменувань. У роботі наведено 51 рисунок та 3 таблиці.

**Актуальність теми.** Клітинні автомати є доволі розповсюдженою математичною моделлю, що може бути використана у багатьох розділах науки, наприклад, при моделюванні певних складних процесів. Найбільш дослідженими є двовимірні клітинні автомати, яскравими прикладами яких є гра “Життя” та мураха Ленгтона. Дані клітинні автомати відносяться до 4 класу. Це означає, що вони виявляють ознаки хаотичної поведінки, при цьому будучи здатними до формування стабільних структур.

Однак тривимірні клітинні автомати не є настільки дослідженими. Можлива сфера їх застосування включає, але не обмежується моделюванням поведінки газів та рідин у тривимірному просторі або квантових ефектів. Тим не менш, досліджень у цьому напряму доволі небагато. Це зумовлено зокрема відсутністю програмних засобів 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 доцільним є створення програмного засобу для роботи із тривимірними автоматами та дослідження їх властивостей за допомогою створеної програми.

**Зв’язок роботи з науковими програмами, планами, темами.** Дисертаційна робота виконувалась згідно з планом науково-дослідних робіт кафедри нарисної геометрії, інженерної та комп’ютерної графіки фізико-математичного факультету Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» в рамках пошукової теми «Геометричне моделювання структур методами фрактальної геометрії», державний реєстраційний номер 0113U007912

**Мета і задачі дослідження.** Метою дисертаційної роботи є розробка програмного засобу для роботи із тривимірними клітинними автоматами та подальше їх дослідження за допомогою створеного застосунку.

Для досягнення вказаної мети було розв’язано такі задачі:

* систематизувати існуючі правила клітинних автоматів та засоби для роботи з клітинними автоматами та визначити їх переваги й недоліки;
* визначити функціонал, необхідний для всебічного дослідження тривимірних клітинних автоматів і надання зручного способу їх відображення та взаємодії з ними;
* розробити програмний засіб для роботи з тривимірними клітинними автоматами, який реалізує визначений функціонал, необхідний для їх дослідження;
* провести експериментальні дослідження властивостей тривимірних автоматів за допомогою розробленого застосунку, зробити висновки щодо подальшого напрямку дослідження.

*Об’єктом дослідження* є тривимірні клітинні автомати із правилами, що базуються на грі “Життя”.

*Предметом дослідження* є пошук тривимірних клітинних автоматів 4 класу, які знаходяться на границі хаосу та впорядкованості.

**Методи дослідження.** Для розв’язання поставленої задачі використовувалися такі методи: методи теорії автоматів (для дослідження клітинних автоматів); методи теорії алгоритмів та програмування (для програмної реалізації розроблених алгоритмів); методи теорії ймовірності та математичної статистики (для проведення експериментів).

**Наукова новизна одержаних результатів** складається з наступних положень:

* уперше поставлено задачу дослідження тривимірних клітинних автоматів за ймовірнісними правилами переходу до наступного стану;
* удосконалено методи дослідження тривимірних клітинних автоматів шляхом створення програмного засобу, що надає увесь необхідний для дослідження функціонал та зручний інтерфейс для роботи з ним;
* розширено потенційні області використання тривимірних автоматів шляхом надання інструменту наслідування кольору клітин при їх зародження під час еволюції системи.

**Практичне значення одержаних результатів.** Визначено типові властивості ймовірнісних тривимірних автоматів. Розроблено програмний засіб для дослідження тривимірних клітинних автоматів. Запропоновані подальші шляхи дослідження тривимірних автоматів.

**Апробація результатів дисертації.** Основні положення й результати роботи представлено на конференціях «Системний аналіз та інформаційні технології» (2017 р.) та «СПГМ-2017»

**Публікації.** Результати дисертації викладено в 3 наукових працях, зокрема:

* у одній статті у науковому журналі, включеному до Переліку наукових фахових видань України з технічних наук;
* у двох публікаціях тез доповідей наукових конференцій (у соавторстві).

**Ключові слова:** клітинний автомат, еволюція, Unity, ітерація, тривимірне відображення

РЕФЕРАТ

Диссертация выполнена на 123 листах, она содержит 1 приложение на 27 листах и перечень ссылок на использованные источники из 46 наименований. В работе приведены 51 рисунок и 3 таблицы.

**Актуальность темы.** Клеточные автоматы являются довольно распространенной математической моделью, которая может быть использована во многих разделах науки, например, при моделировании определенных сложных процессов. Наиболее исследованными являются двумерные клеточные автоматы, яркими примерами которых являются игра "Жизнь" и муравей Ленгтона. Данные клеточные автоматы относятся к 4 классу. Это означает, что они проявляют признаки хаотического поведения, при этом будучи способными к формированию стабильных структур.

Однако трехмерные клеточные автоматы не столь исследованы. Возможная сфера их применения включает, но не ограничивается моделированием поведения газов и жидкостей в трехмерном пространстве или квантовых эффектов. Тем не менее, исследований в этом направлении довольно немного. Это обусловлено в частности отсутствием программных средств для проведения такого исследования. Отображение трехмерных автоматов, их задание и обзор процесса их эволюции требуют специального функционала, который не предоставляется средствами для работы с двумерными клеточными автоматами. Поэтому целесообразным является создание программного средства для работы с трехмерными автоматами и исследования их свойств с помощью созданной программы.

**Связь работы с научными программами, планами, темами.** Диссертационная работа выполнялась в соответствии с планом научно-исследовательских работ кафедры начертательной геометрии, инженерной и компьютерной графики физико-математического факультета Национального технического университета Украины «Киевский политехнический институт имени Игоря Сикорского» в рамках поисковой темы «Геометрическое моделирование структур методами фрактальной геометрии», государственный регистрационный номер 0113U007912

**Цель и задачи исследования.** Целью исследования является разработка программного средства для работы с трехмерными клеточными автоматами и дальнейшее их исследование с помощью созданного приложения.

Для достижения указанной цели были решены следующие задачи:

* систематизировать существующие правила клеточных автоматов и средства для работы с клеточными автоматами и определить их преимущества и недостатки;
* определить функционал, необходимый для всестороннего исследования трехмерных клеточных автоматов и предоставления удобного способа их отображения и взаимодействия с ними;
* разработать программное средство для работы с трехмерными клеточными автоматами, реализующее указанный функционал, необходимый для их исследования;
* провести экспериментальные исследования свойств трехмерных автоматов с помощью разработанного приложения, сделать выводы относительно дальнейшего направления исследования.

*Объектом исследования* являются трехмерные клеточные автоматы с правилами, основанными на игре "Жизнь".

*Предметом исследования* является поиск трехмерных клеточных автоматов 4 класса, которые находятся на границе хаоса и упорядоченности.

**Методы исследования.** Для решения поставленной задачи использовались следующие методы: методы теории автоматов (для исследования клеточных автоматов) методы теории алгоритмов и программирования (для программной реализации разработанных алгоритмов) методы теории вероятности и математической статистики (для проведения экспериментов).

**Научная новизна исследования** состоит из следующих положений:

* впервые поставлена ​​задача исследования трехмерных клеточных автоматов с вероятностными правилами перехода к следующему состоянию;
* усовершенствованы методы исследования трехмерных клеточных автоматов путем создания программного средства, предоставляющего весь необходимый для исследования функционал и удобный интерфейс для работы с ним;
* расширены потенциальные области применения трехмерных автоматов путем предоставления инструмента наследования цвета клеток при их зарождении в ходе эволюции системы.

**Практическое значение полученных результатов.** Определены типичные свойства вероятностных трехмерных автоматов. Разработано программное средство для исследования трехмерных клеточных автоматов. Предложены дальнейшие пути исследования трехмерных автоматов.

**Апробация результатов диссертации.** Основные положения и результаты работы представлены на конференции «Системный анализ и информационные технологии» (2017) и «СПГМ-2017»

**Публикации.** Результаты диссертации изложены в 3 научных трудах, в частности:

* в одной статье в научном журнале, включенном в Перечень научных профессиональных изданий Украины по техническим наукам;
* в двух публикации тезисов докладов научных конференций (в соавторстве).

**Ключевые слова:** клеточный автомат, эволюция, Unity, итерация, трехмерное отображение

ABSTRACT

The thesis is presented in 123 pages. It contains 1 appendix in 27 pages and bibliography of 46 references. 51 figures and 3 tables are given in the thesis.

**Topic relevance.** Cellular automata is quite common mathematical model that can be used in many branches of science, such as the simulation of complex processes. The most researched ones are two-dimensional cellular automata, bright examples of which are Game of Life and Langton’s Ant. These cellular automata belong to the class 4. This means that they show signs of chaotic behavior, although being able to form stable structures.

However, three-dimensional cellular automata are not as studied. Their field of application includes but is not limited to modeling the behavior of gases and liquids in three-dimensional space or quantum effects. However, there are only a few researches in this area. This is particularly due to the lack of software for such research. Display of three-dimensional automata, their setting and reviewing of their evolution need special functionality that is not provided with applications for two-dimensional cellular automata. Therefore, it is appropriate to create the software to work with three-dimensional cellular automata and study their properties using this software.

**Thesis connection to scientific programs, plans, and topics.** The thesis was carried out according to the plan of scientific research by the Descriptive geometry, engineering and computer drawing department, Faculty of Physics and Mathematics at National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute" as part of the topic «Geometric modeling of structures using fractal geometry methods», state registration number 0113U007912

**Research goal and objectives.** The aim of the thesis is the development of software tools for working with three-dimensional cellular automata and their further studies using the created application.

To achieve the aforementioned goals the following problems were solved:

* systematizing existing rules of cellular automata and tools for working with cellular automata and identify their strengths and weaknesses;
* determining the functionality required for a comprehensive study of three-dimensional cellular automata and provide a convenient way of displaying and interacting with them;
* developing a software tool to work with three-dimensional cellular automata, which implements forementioned functionality required for their research;
* conduct experimental studies of the properties of three-dimensional automata using developed application, draw conclusions on the further direction of research.

*The object of the research* are three-dimensional cellular automata with rules based on the Game of Life.

*The subject of the research* is to find three-dimensional cellular automata Class 4, located on the edge of chaos and order.

**Methods of research.** To solve this problem the following methods were used: methods of automata theory (for the research of cellular automata); methods of the theory of algorithms and programming (for software implementation of the developed algorithms); methods of probability theory and mathematical statistics (for the experiments).

**Scientific contribution** consists of the following:

* for the first time the task was posed to study three-dimensional cellular automata with probabilistic rules of the transition to the next state;
* improved methods of the research of three-dimensional cellular automata by creating a software tool that provides all the functionality required for such research and user-friendly interface to work with it;
* expanded the potential scope of the three-dimensional automata implementation by providing color inheritance for the spawning cells during the evolution of the system

**Practical value of obtained results.** Determined typical properties of three-dimensional stochastic automata. Developed software tool for the research of three-dimensional cellular automata. Proposed further possibilities to study three-dimensional automata.

**Approbation of the thesis results.** Key position and results presented at the conferences "System Analysis and Information Technologies" (2017) and "СПГМ-2017"

**Publications.** The results of the thesis outlined in 3 scientific papers including:

* in one article in a scientific journal, included in the list of scientific professional publications of Ukraine for Technical Sciences;
* in two publication abstracts of scientific conference (in co-authorship).

**Keywords:** cellular automata, evolution, Unity, iteration, three-dimensional display

ЗМІСТ

[Вступ 17](#__RefHeading___Toc4634_1396686415)

[1 Огляд існуючих рішень 19](#__RefHeading___Toc4644_1396686415)

[1.1 Загальний огляд клітинних автоматів 19](#__RefHeading___Toc4646_1396686415)

[1.1.1 Основні поняття клітинних автоматів 19](#__RefHeading___Toc4648_1396686415)

[1.1.2 Класифікація клітинних автоматів 21](#__RefHeading___Toc4650_1396686415)

[1.2 Одновимірні клітинні автомати 25](#__RefHeading___Toc4652_1396686415)

[1.2.1 Основні відомості та код Вольфрама 25](#__RefHeading___Toc4654_1396686415)

[1.2.2 Правило 30 27](#__RefHeading___Toc4656_1396686415)

[1.2.3 Правило 110 29](#__RefHeading___Toc4658_1396686415)

[1.3 Мураха Ленгтона 31](#__RefHeading___Toc4660_1396686415)

[1.3.1 Класичний варіант мурахи Ленгтона 31](#__RefHeading___Toc4662_1396686415)

[1.3.2 Розширення ідеї мурахи Ленгтона 33](#__RefHeading___Toc4664_1396686415)

[1.4 Гра “Життя” 36](#__RefHeading___Toc4666_1396686415)

[1.4.1 Основні правила гри “Життя” 36](#__RefHeading___Toc4668_1396686415)

[1.4.2 Натюрморти 37](#__RefHeading___Toc4670_1396686415)

[1.4.3 Осцилятори 40](#__RefHeading___Toc4672_1396686415)

[1.4.4 Космічні кораблі 41](#__RefHeading___Toc4674_1396686415)

[1.5 Програмні реалізації клітинних автоматів 44](#__RefHeading___Toc4676_1396686415)

[1.5.1 Реалізації двовимірних клітинних автоматів 44](#__RefHeading___Toc4678_1396686415)

[1.5.2 Реалізації тривимірних клітинних автоматів 52](#__RefHeading___Toc4680_1396686415)

[Висновки до розділу 1 53](#__RefHeading___Toc14975_3493313287)

[2 Опис функціональних можливостей розроблюваного застосунку 55](#__RefHeading___Toc4682_1396686415)

[2.1 Основна ідея роботи 55](#__RefHeading___Toc4684_1396686415)

[2.2 Відображення стану тривимірного автомату 57](#__RefHeading___Toc4686_1396686415)

[2.3 Розпізнавання переходу автомату до тривіального стану 62](#__RefHeading___Toc4688_1396686415)

[2.3.1 Випадок детермінованого клітинного автомату 62](#__RefHeading___Toc4690_1396686415)

[2.3.2 Випадок недетермінованих (ймовірнісних) клітинних автоматів 64](#__RefHeading___Toc4692_1396686415)

[2.4 Задання стану клітин автомату 69](#__RefHeading___Toc4694_1396686415)

[2.4.1 Безпосереднє задання стану окремих клітин 69](#__RefHeading___Toc4696_1396686415)

[2.4.2 Початкове задання конфігурації автомату. Завантаження та запис 70](#__RefHeading___Toc4698_1396686415)

[Висновок до розділу 2 72](#__RefHeading___Toc14977_3493313287)

[3 Опис програмної реалізації 73](#__RefHeading___Toc4700_1396686415)

[3.1 Движок Unity як базис для розроблюваної програми 73](#__RefHeading___Toc4702_1396686415)

[3.1.1 Основні факти та призначення движка Unity 73](#__RefHeading___Toc4704_1396686415)

[3.1.2 Функціональні можливості движку та особливості його побудови 74](#__RefHeading___Toc4706_1396686415)

[3.1.3 Написання скриптів для движка Unity 77](#__RefHeading___Toc4708_1396686415)

[3.2 Опис скриптів та їх взаємозв’язку 79](#__RefHeading___Toc4712_1396686415)

[3.2.1 Об’єктна структура програми 79](#__RefHeading___Toc4714_1396686415)

[3.2.2 Клас LifeGame 86](#__RefHeading___Toc4716_1396686415)

[3.2.3 Клас GameDisplay 87](#__RefHeading___Toc4718_1396686415)

[3.2.4 Клас Game3D 88](#__RefHeading___Toc4720_1396686415)

[3.2.5 Класи CameraMovement та CellSelection 89](#__RefHeading___Toc4722_1396686415)

[3.3 Огляд інтерфейсу програми 90](#__RefHeading___Toc4724_1396686415)

[Висновки до розділу 3 97](#__RefHeading___Toc14979_3493313287)

[4 Огляд результатів розробки програми 99](#__RefHeading___Toc4726_1396686415)

[4.1 Сценарії використання програми 99](#__RefHeading___Toc4728_1396686415)

[4.1.1 Дослідження еволюції конкретної конфігурації автомату 99](#__RefHeading___Toc4730_1396686415)

[4.1.2 Пошук автомату з необхідними властивостями 101](#__RefHeading___Toc4734_1396686415)

[4.1.3 Огляд еволюції певної ділянки автомату 102](#__RefHeading___Toc4736_1396686415)

[4.1.4 Дослідження поведінки автомату із наслідуванням кольору клітин. 103](#__RefHeading___Toc4738_1396686415)

[4.2 Результати виконання програми 104](#__RefHeading___Toc4740_1396686415)

[4.2.1 Приклади еволюції автоматів 104](#__RefHeading___Toc4742_1396686415)

[4.2.2 Аналіз статистичних даних еволюції автоматів 109](#__RefHeading___Toc4744_1396686415)

[4.3 Порівняння із аналогами та можливі варіанти подальшого розвитку 111](#__RefHeading___Toc4746_1396686415)

[4.3.1 Порівняння із іншими системами роботи з тривимірними клітинними автоматами 111](#__RefHeading___Toc4748_1396686415)

[4.3.2 Порівняння із системою для роботи з двовимірними клітинними автоматами Golly 113](#__RefHeading___Toc4750_1396686415)

[4.3.3 Можливі шляхи подальшого розвитку системи 114](#__RefHeading___Toc4752_1396686415)

[Висновки до розділу 4 116](#__RefHeading___Toc14981_3493313287)

[Висновок 118](#__RefHeading___Toc4754_1396686415)

[Список використаної літератури 120](#__RefHeading___Toc438139277)

[Додаток 1. Лістинг коду програми 124](#__RefHeading___Toc14250_1917348273)

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

КА — клітинні автомати

Unity – ігровий движок, що надає інтерфейс та набір бібліотек для швидкої розробки комп’ютерних ігор чи інших програм із застосуванням тривимірної графіки.

Префаб — серіалізований файл об’єкту Unity, що використовується для швидкого створення нового об’єкту із певними компонентами та їх характеристиками

Рендерер — різновид компонентів Unity, що відповідають за відображення об’єктів на сцені у двовимірному чи тривимірному форматі

Конфігурація — значення станів клітин автомату

# **Вступ**

Метою даної магістерської дисертації є розробка застосунку, що реалізовуватиме роботу з тривимірними клітинними автоматами задля вивчення їх властивостей та пошуку автоматів, що виказують ознаки хаотичної поведінки та містять конфігурації, що здатні зберігати свою цілісність протягом багатьох ітерацій.

Для досягнення вказаної мети було розв’язано такі задачі:

* систематизувати існуючі правила клітинних автоматів та засоби для роботи з клітинними автоматами та визначити їх переваги й недоліки;
* визначити функціонал, необхідний для всебічного дослідження тривимірних клітинних автоматів і надання зручного способу їх відображення та взаємодії з ними;
* розробити програмний засіб для роботи з тривимірними клітинними автоматами, який реалізує визначений функціонал, необхідний для їх дослідження;
* провести експериментальні дослідження властивостей тривимірних автоматів за допомогою розробленого застосунку, зробити висновки щодо подальшого напрямку дослідження.

Об’єктом досліджень є тривимірні клітинні автомати із правилами, що базуються на грі “Життя”. Сітка даного клітинного автомату є кубічною та скінченною. Стан клітини на наступному кроці при цьому обраховується на основі кількості сусідів даної клітини. Сусідами клітини виступають 26 клітин, чиї координати відрізняються від координати центральної клітини не більше, ніж на одиницю. Кожна клітина може приймати один з двох станів: “жива” та “мертва”.

Предметом дослідження є пошук тривимірних клітинних автоматів, які знаходяться на границі хаосу та впорядкованості, типовим прикладом яких серед двовимірних клітинних автоматів є гра “Життя”. Подібні автомати можуть містити стійкі структури на кшталт натюрмортів чи осциляторів з гри “Життя”, а також хаотичну реакцію на незначні зміни, що може відбуватись тисячі ітерацій.

Актуальність теми базується на малодослідженості характеристик тривимірних клітинних автоматів. Двовимірні клітинні автомати, зокрема гра “Життя”, вже досить добре вивчені; вони також активно застосовуються у різних галузях науки для моделювання певних процесів, а також вплинули на різні галузі, не пов’язані безпосередньо з клітинними автоматами. Тривимірні автомати не мають такого широкого застосування та не були детально вивчені, хоча можуть мати аналогічний потенціал.

Наукова новизна одержаних результатівскладається з наступних положень:

* уперше поставлено задачу дослідження тривимірних клітинних автоматів за ймовірнісними правилами переходу до наступного стану;
* удосконалено методи дослідження тривимірних клітинних автоматів шляхом створення програмного засобу, що надає увесь необхідний для дослідження функціонал та зручний інтерфейс для роботи з ним;
* розширено потенційні області використання тривимірних автоматів шляхом надання інструменту наслідування кольору клітин при їх зародження під час еволюції системи.

# Огляд існуючих рішень

## Загальний огляд клітинних автоматів

### Основні поняття клітинних автоматів

Клітинний автомат (КА) — дискретна математична модель, яка визначає сукупність та описується набором клітинок, що утворюють періодичну решітку, та заданими правилами переходу, що визначають стан клітини за теперішнім станом самої клітинки та тих її сусідів, що знаходяться від неї на певній відстані, яка не перевищує максимальну.[1]

Поняття клітинних автоматів доволі широке, тому можна знайти доволі багато різних визначень. Найпоширенішими є:

* уперше поставлено задачу дослідження тривимірних клітинних автоматів за ймовірнісними правилами переходу до наступного стану;
* математичний об'єкт з дискретним простором та часом;
* регулярна структура двійкових скінченних автоматів з однаковими правилами переходів, що виражені у вигляді булевих функцій від станів сусідніх автоматів;
* стилізовані, синтетичні світи, що визначені простими правилами, подібно правилам настільної гри;
* математична ідеалізація фізичної системи, в якій час та простір дискретні, а фізичні величини приймають скінченну множину значень.

Класичні КА в загальному випадку відповідають наступним критеріям:

* зміна значень всіх клітинок відбуваються одночасно після обчислення нового стану кожної клітинки решітки. Інакше порядок перебору клітин решітки при проходження ітеративного процесу суттєво впливав би на результат;
* решітка однорідна. Неможливо відрізнити жодні два місця на решітці по ландшафту. Однак на практиці решітка виявляється кінцевою множиною клітин (адже неможливо виділити необмежений об'єм даних). В результаті можуть мати місце крайові ефекти: клітини, що стоять на межах решітки будуть відрізнятися за кількістю сусідів. Щоб уникнути цього можна ввести періодичні крайові умови;
* взаємодії локальні. Лише околишні клітинки (як правило, сусідні) здатні вплинути на дану клітинку;
* множина станів клітинки кінцева. Ця умова потрібна, щоб для отримання нового значення стану клітини треба було виконати кінцеву кількість операцій (але це не заважає використовувати клітини для зберігання чисел із плаваючою комою для розв'язку прикладних задач).

Якщо з будь-якого початкового стану можна привести клітинного автомат в будь-яку задану конфігурацію шляхом варіювання значення загального вхідного параметра, такий КА називають повним.

У кожний момент часу кожен елемент КА приймає один стан зі скінченного набору станів. В залежності від цих станів в наступний момент часу набір елементів може прийняти новий стан. Якщо для елементів КА множини можливих станів відрізняються, такий клітинний автомат називається полігенним. Але на практиці використовуються комірки з еквівалентною множинами можливих станів алгебраїчною структурою – лінійні КА.[2]

Елементи можуть бути геометрично розташовані різноманітним чином. Розмірність простору може бути довільною, а число елементів – як безкінечним, так і скінченним. В останньому випадку виникає додаткова міра свободи в граничних умовах. Вони можуть бути різними, але на практиці використовуються постійні у часі (найчастіше – нульові) або періодичні граничних умовах. У динамічних КА геометрія може змінюватися з часом, а якщо геометрія різна на різних ділянках простору, такі кліткові КА називають неоднорідними.

Для КА визначається поняття сусідів – елементів, від яких залежить елемент КА. Можна назвати поняття сусідства ключовим для КА. При тому сусідство розуміється не в геометричному сенсі, а в інформаційному. Хоча зазвичай інформаційний сенс накладається на геометричний. Сусідство одиничних автоматів встановлюється постійним для кожного одиничного автомата решітки і визначається спеціальним вектором – індексом сусідства. Як правило, розглядаються d-мірні регулярні решітки, в цілочислові точки яких поміщені копії деякого автомата Мура. Стан елемента в наступний момент часу обчислюється із стану самого елементу і його сусідів. Сусідство у більшій мірі визначається геометрією КА. Для різних цілей можлива зміна числа вхідних станів елемента. Якщо для кожного елемента КА число входів і виходів однакове, такий КА називається збалансованим.

Для кожного клітинного автомату визначені певні локальні правила. Відповідно до локального правила змінюється стан елемента КА протягом часу. КА, в якому локальні правила різні для різних елементів, називається різнорідним. Локальне правило може бути недетермінованим, тобто змінюватися в часі або мати випадкову природу. Зазвичай, це правило встановлює залежність між поточним станом елементу та його сусідів та майбутнім станом цього елементу на наступному кроці.

Основний напрям дослідження клітинних автоматів — алгоритмічна розв'язність окремих задач. Також розглядаються питання побудови початкових станів, при яких клітинний автомат вирішуватиме задану задачу. Залишається відкритим, наприклад, питання про можливість побудови машини Тюринга у грі «Життя».

### 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.[3]

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.

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.

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».

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:

,

де f – функція переходів клітинки;

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

y i-1 – стан (i-1)-ої клітинки в даний момент часу;

y i – стан i-ої клітинки в даний момент часу;

y i +1 – стан (i +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:

.

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.

## Одновимірні клітинні автомати

### Основні відомості та код Вольфрама

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 28 = 256 можливих правил. Ці 256 правил кодуються відповідно коду Вольфрама - стандартною угодою, правилу, яке було запропоновано Вольфрамом. У деяких статтях ці 256 правил були досліджені та порівняні. Початкова умова для кожного автомату - одна центральна клітина - заповнена, решта - порожні. По осі Y відкладається дискретний час, а по осі X відкладаються стани клітин автомата.[4]

Код Вольфрама являє собою систему присвоєння імен, що часто використовується для одновимірних правил клітинного автомата, введену Стівеном Вольфрамом в роботі 1983 року та використану ним в своїй книзі A New Kind of Science.

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^S2n + 1 - 1, перетворений з S-кової в десяткову систему числення. Вона може бути розрахована наступним чином:

* виписати всі S2n + 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.

Код Вольфрама не визначає ні розмір (чи форму) околиці, ні число станів - вони передбачаються відомими з контексту. При використанні кодів самостійно без такого контексту, зазвичай передбачається, що вони відносяться до класу елементарних клітинних автоматів - одновимірних клітинних автоматів з двома станами клітин та з (суміжною) трьохклітинною околицею, які Вольфрам широко досліджує в своїй книзі. Відомі правила в цьому класі включають правило 30, правило 110 і правило 184. Правило 90 також цікаве тим, що воно створює трикутник Паскаля по модулю 2. Код такого типу суфіксом R, наприклад, як "Правило 37R", вказує на клітинний автомат другого порядку з такою ж структурою околиці.

У той час як в строгому сенсі цього слова кожен код Вольфрама в допустимих межах визначає різні правила, деякі з цих правил є ізоморфними і, як правило, вважаються еквівалентними. Наприклад, правило 110 ізоморфне правилам 124, 137 і 193, які можуть бути отримані з оригіналу відображенням вліво-вправо та іншою нумерацією станів. За угодою, кожен такий клас ізоморфізму представлений правилом з найменшим номером коду в ньому.

Недоліком коду Вольфрама та використання десяткової системи числення, зокрема, є те, що він ускладнює пошук таких ізоморфізмів порівняно з деякими альтернативними нотаціями. Незважаючи на це, він став стандартним де-факто способом позначення одновимірних клітинних автоматів.

### Правило 30

Правило 30 - одномірне двійкове правило для клітинного автомата, вперше описане Стівеном Вольфрамом в 1983 році. З чотирьох типів поведінки, описаних в його книзі "A New Kind of Science", Правило 30 володіє класом поведінки III, показуючи аперіодичну, хаотичну поведінку.

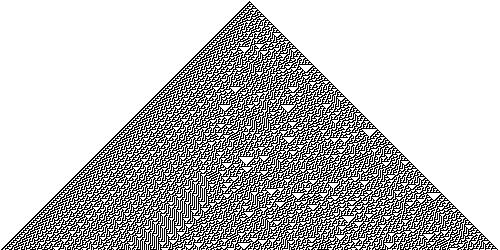
Правило являє інтерес, тому що воно породжує складні, багато в чому випадкові структури з простих, чітко визначених правил. Вольфрам вважає, що клітинні автомати в цілому і Правило 30 зокрема - ключ до розуміння того, як прості правила можуть породжувати складні структури і різну складну поведінку різних природних об'єктів. Наприклад, структуру, схожу на ту, що породжуються Правилом 30, можна знайти на раковині широко розповсюдженого тропічного молюска Conus textile. Також це правило було запропоновано для використання як шифратор послідовностей в криптографії.[1]

Якщо записати код правила в двійковому вигляді, то дзеркальне відображення коду правила, інверсія бітів коду і дзеркальне відображення з інверсією бітів мають в десятковій системі числення коди 120, 225 і 135 відповідно. Таким чином, правила під цими номерами є ізоморфними Правилу 30.

Правило 30 в бінарному вигляді можна записати наступним чином (тут три цифри зверху — це стан лівого сусіда, поточної клітини та правого сусіда відповідно):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Поточний стан | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
| Новий стан центральної клітини | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

На рисунку 1.1 відображено приклад еволюції автомату з одної початкової заповненої клітини (по горизонталі розміщені всі клітини одного покоління, по вертикалі — стан однієї клітини на кожному кроці):

Рисунок 1.1 — Еволюція одновимірного автомату Правило 30

Як можна бачити з рисунка 1.1, Правило 30 породжує послідовність, що здається в багатьох відносинах випадковою, при початкових умовах, які не можна назвати випадковими. Стівен Вольфрам запропонував використання вертикальних стовпчиків в еволюції клітинних автоматів при деякій заданій початковому стані як псевдовипадкову послідовність. Послідовності, отримані таким способом, задовольняють багатьом стандартним тестам на випадковість. Стівен Вольфрам використовує Правило 30 для генерації випадкових цілих чисел в пакеті Mathematica. Однак M. Sipper і M. Tomassini показали, що як ГПСЧ Правило 30 погано проходить тест на критерій згоди Пірсона (критерій χ²) в порівнянні з іншими псевдовипадковими послідовностями, які були отримані за допомогою інших клітинних автоматів.

Вольфрам припустив хаотичність еволюції за Правилом 30, грунтуючись, в основному, на її зовнішньому графічному вигляді. Однак пізніше було показано, що застосування Правила 30 задовольняє більш чітким визначенням хаосу, сформульованим Devaney і Knudson. Відповідно до критерію Devaney, Правило 30 демонструє ефект метелика - якщо задати 2 початкових стану, що розрізняються, наприклад, тільки одним бітом, то віддалені багатьма поколіннями нащадки цих 2 станів будуть абсолютно різні, періодичні зміни є щільними в просторі будь-яких змін топології Кантора. Також, Правило 30 має властивість перемішування. Відповідно до критерію Knudson, це показує чутливість до початкових умов і щільність орбіт процесу (будь-яка конфігурація в підсумку призводить до виникнення всіх можливих кінцевих конфігурацій клітин). Обидві ці характеристики хаотичної поведінки еволюції за Правилом 30 слідують з властивості Правила 30, яке легко перевірити: якщо дві конфігурації C і B розрізняються на одну клітку в позиції i, тоді після одного кроку нові конфігурації будуть відрізнятися в клітці i + 1.

### Правило 110

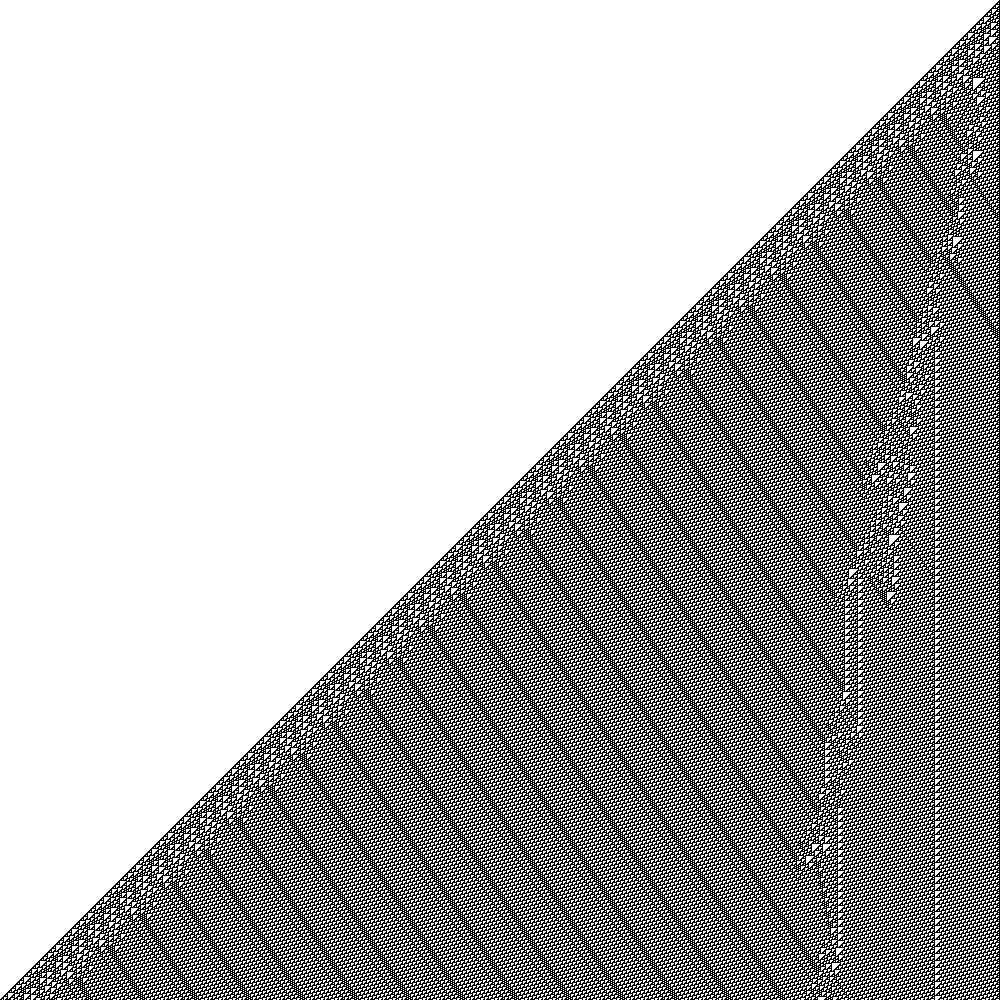
Правило 110 - елементарний одновимірний клітинний автомат з поведінкою, що знаходиться на межі хаосу і стабільності. В цьому відношенні Правило 110 ідентично грі «Життя». Відомо, що Правило 110 є Тьюринг-повним, що означає, що будь-яка обчислювальна процедура може бути реалізована за допомогою цього клітинного автомата.

Меттью Кук представив свій доказ цього факту на конференції Інституту Санта-Фе в 1998 році, але Вольфрам заборонив включати цей доказ в паперову версію матеріалів конференції, тому що не хотів, щоб воно було опубліковано до видання книги A New Kind of Science. У 2004 році доказ Кука було опубліковано в журналі Вольфрама «Complex Systems» (випуск 15, том 1), через 10 років після того як Кук вперше представив його.

Правило 110 в бінарному вигляді можна записати наступним чином (як і в минулому випадку, три цифри зверху — це стан лівого сусіда, поточної клітини та правого сусіда відповідно):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Поточний стан | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
| Новий стан центральної клітини | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

Наступним чином виглядає еволюція автомату з одної початкової заповненої клітини (по горизонталі розміщені всі клітини одного покоління, по вертикалі — стан однієї клітини на кожному кроці):

Рисунок 1.2 — Еволюція одновимірного автомату Правило 110

На рисунку 1.2 можна побачити, що нові клітини заповнюються лише ліворуч від початкової клітини. Це пояснюється тим, що згідно цьому правилу пуста клітина без правого сусіда ніколи не заповниться, а через те що клітини справа від початкової завжди пусті та завжди не мають правого сусіда, вони завжди залишатимуться пустими.

## Мураха Ленгтона

### Класичний варіант мурахи Ленгтона

Мураха Ленгтона - це двовимірний клітинний автомат з дуже простими правилами, винайдений Крісом Ленгтоном. Мураху можна також вважати двовимірною машиною Тьюринга з 2 символами і 4 станами.

Розглянемо нескінченну площину, розбиту на клітини, пофарбовані деяким чином в чорний і білий колір. Нехай в одній з клітин знаходиться «мураха», яка на кожному кроці може рухатися в одному з чотирьох напрямків в клітку, сусідню за стороною. Мураха рухається згідно з такими правилами:

* на чорному квадраті - повернути на 90 ° вліво, змінити колір квадрата на білий, зробити крок вперед на наступну клітину;
* на білому квадраті - повернути на 90 ° вправо, змінити колір квадрата на чорний, зробити крок вперед на наступну клітину.

Ці прості правила викликають досить складну поведінку: після деякого періоду досить випадкового руху мураха починає неодмінно будувати дорогу з 104 кроків, повторювану нескінченно, незалежно від початкової розмальовки поля. Це наводить на думку, що «магістральна» поведінка є стабільним аттрактором мурашки Ленгтон. Можливо, що «магістраль» є єдиним аттрактором при переміщенні мурашки.[2]

Мураха Ленгтона також може бути описаний як клітинний автомат, в якому майже все поле пофарбовано в чорно-білий колір, а клітина з «мурахою» має один з восьми різних кольорів, що кодують відповідно всі можливі комбінації чорного / білого кольору клітини і напрямки руху мурашки.

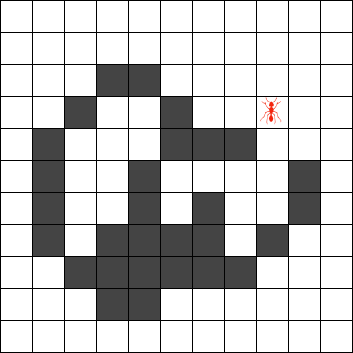
На рисунку 1.3 показано результат роботи клітинного автомату після декількох десятків ітерацій. В процесі роботи “мураха” здійснює рух по нетривіальній траєкторії, яка, як і стан клітин автомату, важко передбачається.

Рисунок 1.3 — Мураха Ленгтона

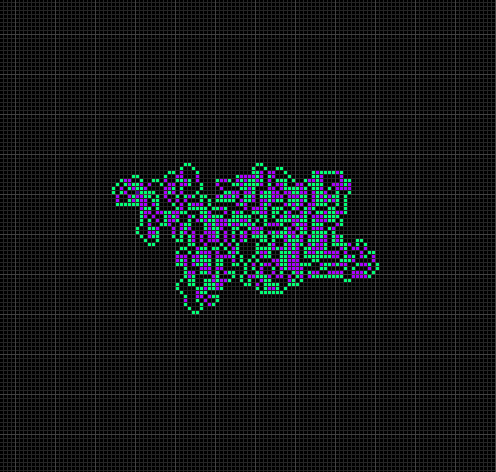
“Магістраль” - послідовність руху, до якої мураха переходить після певної кількості ітерацій — зображена на рисунку 1.4 (в нижній частині рисунку):

Рисунок 1.4 — Приклад магістралі

### Розширення ідеї мурахи Ленгтона

Існує просте розширення мурахи Ленгтона, в якому використовується більше двох кольорів клітин. Кольори змінюються циклічно. Для таких мурах існує також проста форма назви: для кожного наступного кольору використовується буква L або R (Л і П), в залежності від того, чи повертає мураха направо або наліво. Таким чином, мураха Ленгтона - це мураха RL.

Наведемо декілька прикладів. На рисунку 1.5 зображено результат роботи автомату з мурахою RLR, для якого характерна хаотична поведінка та досі не доведено, що мураха в цьому випадку створює “магістраль”:

Рисунок 1.5 — Еволюція автомату з правилами RLR

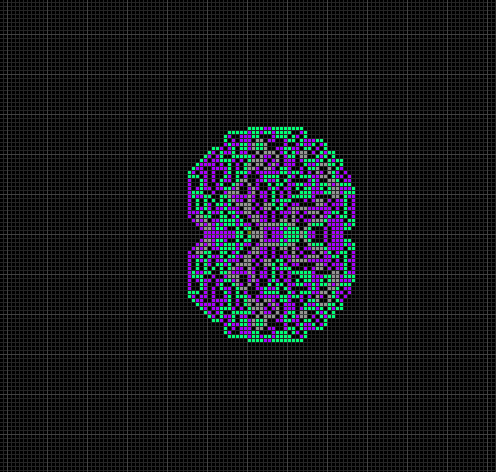
Деякі з цих узагальнених мурах Ленгтона малюють візерунки, які стають все більш симетричними. Один з простих прикладів - мураха RLLR. Одна достатня умова цього полягає в тому, що ім'я мурашки, що розглядається як циклічний список, складається з послідовних пар повторюваних букв LL або RR (циклічність списку означає, що остання буква може спаровуватися з першою).

Рисунок 1.6 — Еволюція автомату з правилами LLRR

На рисунку 1.6 можна чітко бачити симетричність структури. В даному випадку окрім чорного кольору також використані зелений та фіолетовий.

Деякі конфігурації мурахи здатні заповнювати певну площу одним кольором, збільшуючи цю площу з часом. Типовим прикладом цієї поведінки є мураха LRRRRRLLR — вона заповнює квадрат навколо себе одним кольором (в даному прикладі — фіолетовим).

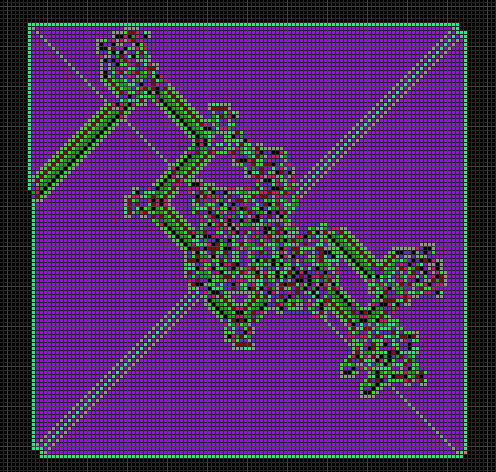
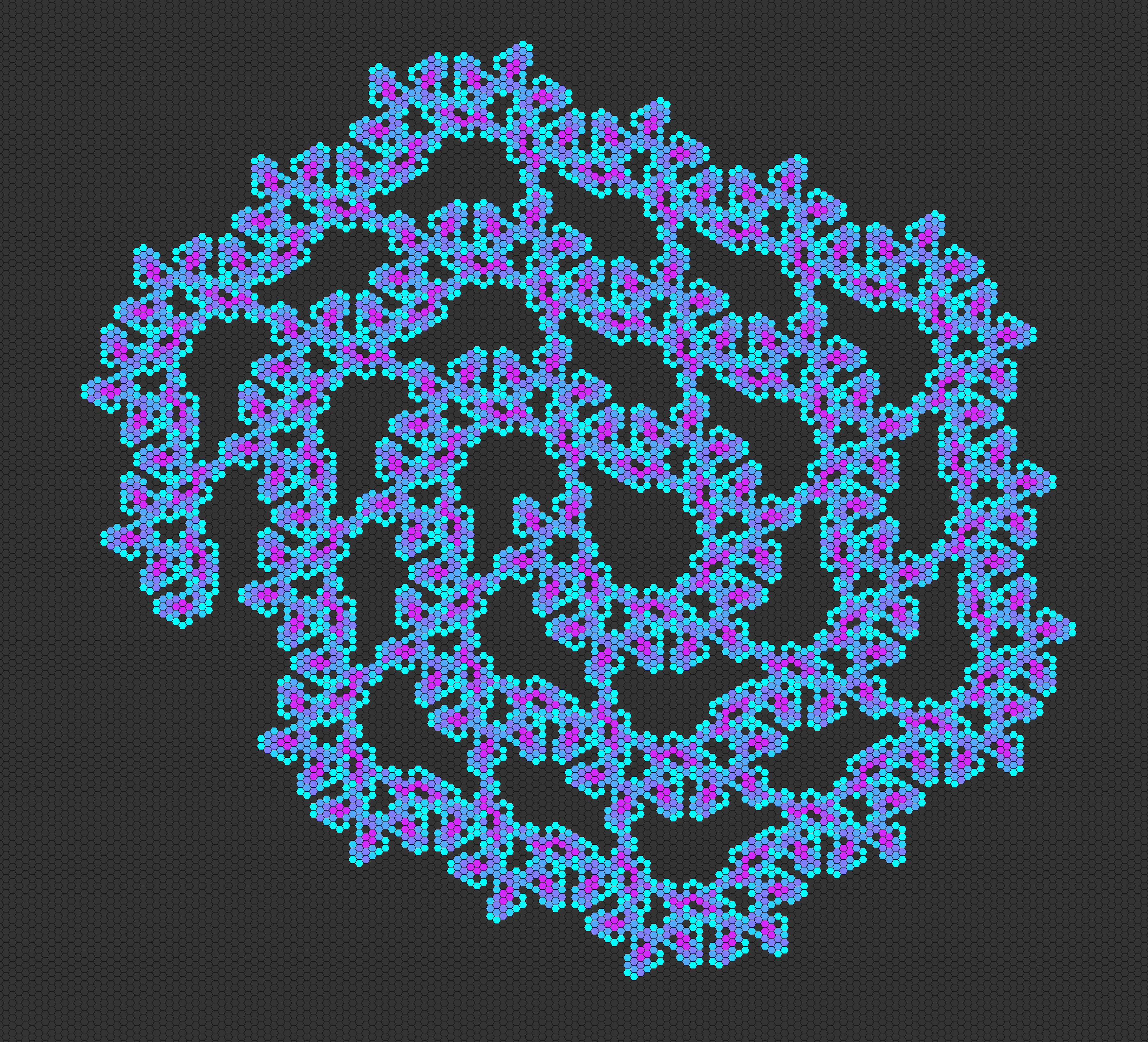


Рисунок 1.7 — Еволюція автомату з правилами LRRRRRLLR

Також іноді використовується гексагональна структура автомату. Для неї можливо більше ніж два повороти, тому процес їх задання складніший. Для подібних структур часто характерне поступове розширення повторюваного візерунку, як на рисунку 1.8.

Рисунок 1.8 — Еволюція автомату із гексагональною структурою

Таким чином, використовуючи різні конфігурації поля та мурахи можна отримати результат, котрий в багатьох випадках виказує ознаки клітинного автомату четвертого класу, з можливими повторюваними структурами та ознаками хаосу.

## Гра “Життя”

### Основні правила гри “Життя”

Гра «Життя» (англійською Conway's Game of Life) - клітинний автомат, винайдений англійським математиком Джоном Конвеєм в 1970 році.[5]

Джон Конвей зацікавився проблемою, запропонованою в 1940-х роках відомим математиком Джоном фон Нейманом, який намагався створити гіпотетичну машину, яка може відтворювати сама себе. 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.

### 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 1.9 — Псевдонатюрморт та “строгий” натюрморт

Ліворуч на рисунку 1.9 - 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 1.10 ліворуч зображено блок, праворуч — бі-блок.



Рисунок 1.10 — Блок та бі-блок

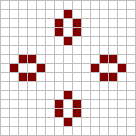
Другий за поширеністю натюрморт - вулик. Вулики часто виникають четвірками в конфігурації, що називають пасікою. Вулик та пасіку зображено на рисунку 1.11 ліворуч та праворуч відповідно.



Рисунок 1.11 — Вулик та пасіка

Існує велика кількість інших відомих натюрмортів, що відрізняються за розміром та частотою появи. Деякі з них зображені на рисунках 1.12 та 1.13.

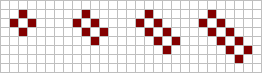
Рисунок 1.12 — Ящик, баржа та подовжені баржи



Рисунок 1.13 — Приклади інших натюрмортів гри “Життя”

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

### 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.[5]

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 1.14 - Блимавка

Серед інших осциляторів можна виділити крест – осцилятор з періодом 3.



Рисунок 1.15 - Крест

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



Рисунок 1.16 - Фумарола

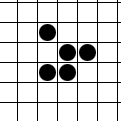
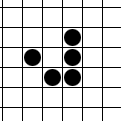
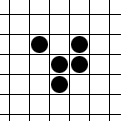
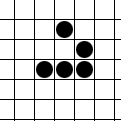
Знайдено й велику кількість інших осциляторів (наприклад, галактика Кока з періодом 8 чи пентадекатлон з періодом 15). Зокрема, для створення нового осцилятору достатньо об’єднати два вже відомих осциляторів в одну конфігурацію. Таким чином, можна зробити висновок, що на нескінченному полі кількість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

### 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 1.17.

Рисунок 1.17 - 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.

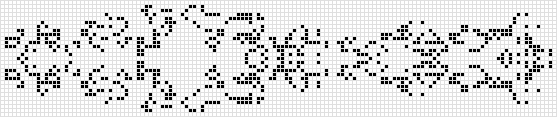
Швидкість космічного корабля визначається відношенням відстані зміщення до періоду. Часто швидкість виражається через c. Так, швидкість планера в «Життя» дорівнює c / 4, так як він переміщується на одну клітину по діагоналі за чотири покоління. Найпростіший ортогональний космічний корабель, ВКК, рухається 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 = с \* max (| x |, | y |) / n.

Найбільш поширеними різновидами космчних кораблів, окрім планерів, є легкі, середні та важкі космічні кораблі, зображені на рисунку 1.18 (починаючи з лівого краю).

Рисунок 1.18 — 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 3с / 7. Він зображений на рисунку 1.19.

Рисунок 1.19 - “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 1.20 — 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.

## Програмні реалізації клітинних автоматів

### Реалізації двовимірних клітинних автоматів

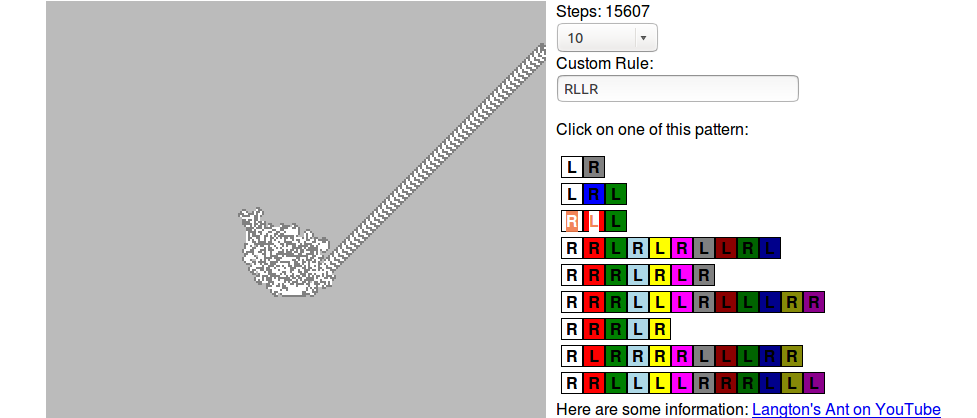
Існує велика кількість програмних реалізацій різних клітинних автоматів. Найбільш поширені реалізації таких автоматів, як мураха Ленгтона та гра “Життя”. Особливо поширені реалізації цих автоматів у вигляді веб-сторінки на мові JavaScript. Зокрема, прикладом реалізації мурахи Ленгтона може слугувати реалізація Фабріса Вайнберга. Вона становить з себе веб-сторінку, на якій розміщено вихідний код програми та власне її виконання у браузері. Програма дозволяє як запустити класичний варіант мурахи, так і розширені варіанти з більшою кількістю станів. При цьому користувачеві надається можливість або обрати один з заздалегідь вказаних варіантів, що мають певні цікаві особливості, або задати правило власноруч, дозволяючи таким чином реалізувати будь-який варіант мурахи Ленгтона на квадратному полі. Реалізація Вайнберга дозволяє також обрати швидкість, з якою мураха рухається автоматом, розглядаючи більш незначні кроки чи загальну картину. На рисунках 1.21-1.23 зображено декілька скріншотів даної програми.

Рисунок 1.21 - Приклад виконання програми із стандартними правилами

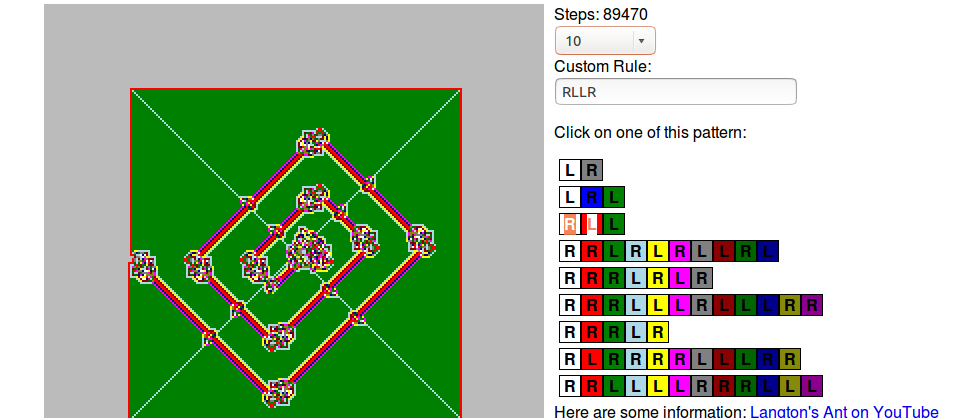
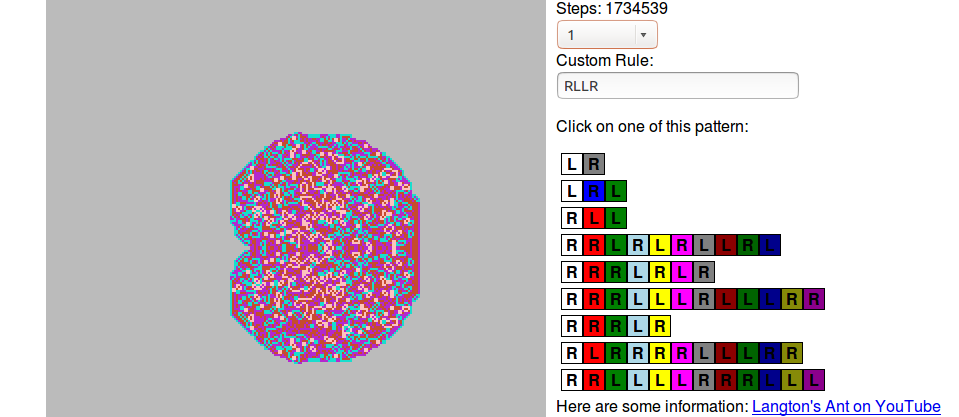
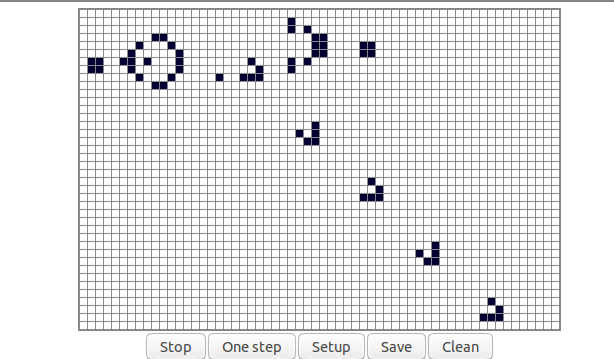
Рисунок 1.22 - Приклад виконання програми з використанням одного з заданих наборів розширених правил (RLRRRRLLLRR)

Рисунок 1.23 - Приклад виконання програми із власним набором розширених правил (RLLR, з симетричним візерунком)

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. Дана реалізація використовує інтерфейс російською мовою.



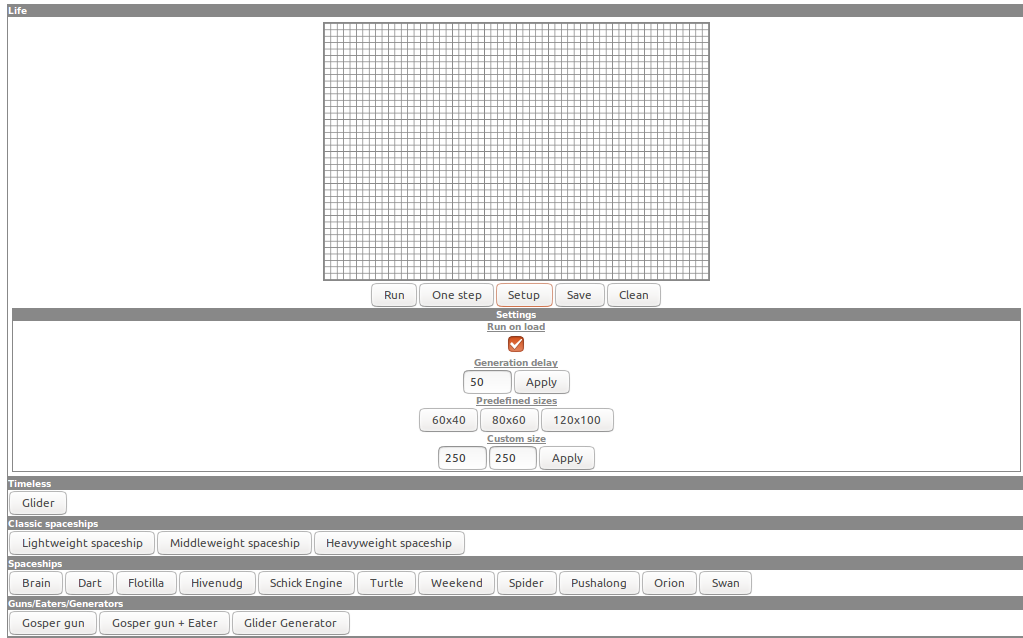
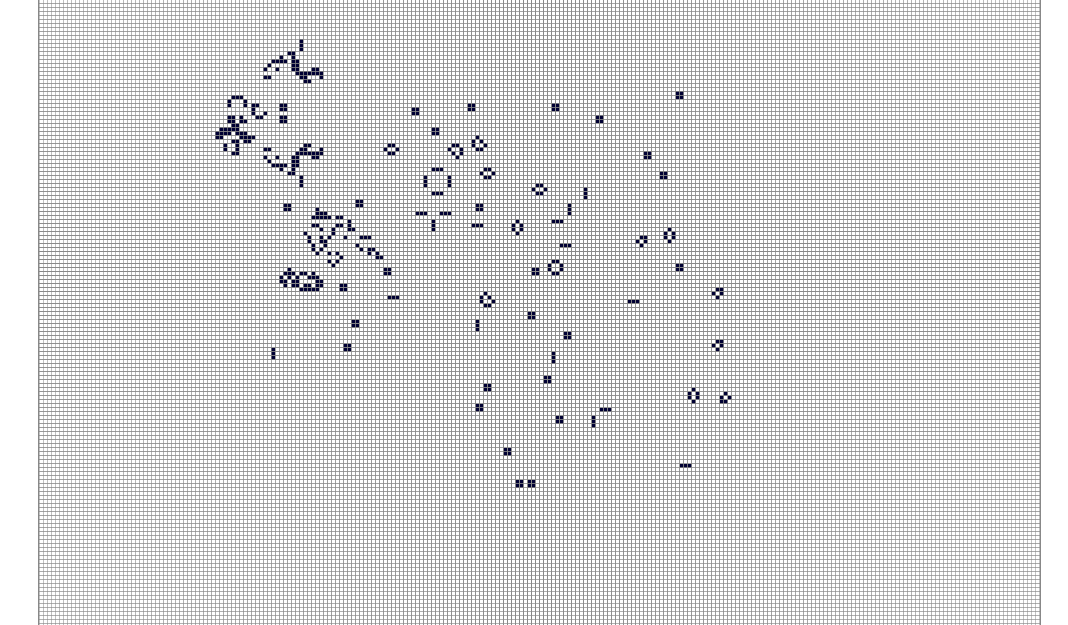
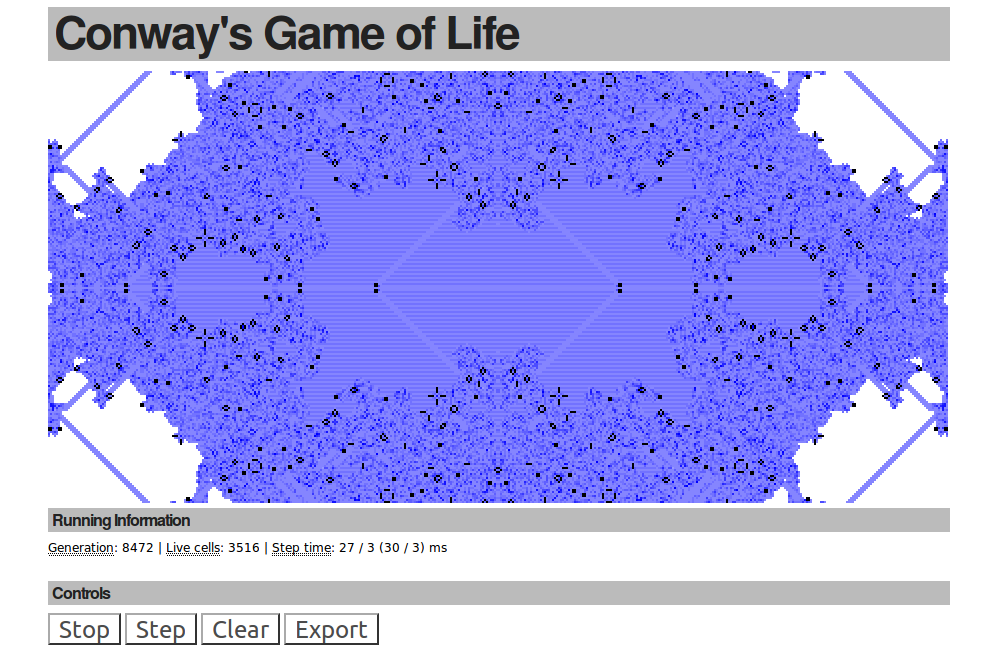
Рисунок 1.24 - Зовнішній вигляд інтерфейсу програми

Рисунок 1.25 - Результат виконання одної з доступних конфігурацій — планерної рушниці Госпера

Рисунок 1.26 - Результат виконання застосунку з випадковими початковими умовами

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 1.27 зображено кінцевий результат еволюції одного з шаблонів, наданих в даній реалізації. Блакитним та синім кольорами позначені клітини, що зараз є пустими, проте були заповнені в попередніх поколіннях.

Рисунок 1.27 - Кінцевий результат еволюції одного з шаблонів програми

Однак існуючі реалізації гри “Життя” не обмежуються онлайн-вейсіями на JavaScript. Звісно, подібні програми недостатньо швидкі та обмежені ресурсами браузеру порівняно з реалізаціями на інших платформах та мовах програмування. Також, ймовірно, що вищезазначені варіанти не використовують найшвидші алгоритми обрахунку наступної ітерації, тому навіть не зважаючи на обмеження браузеру такі програми будуть повільнішими за більш розвинені аналоги. Найбільш популярним мультиплатформенним застосунком для роботи з клітинними автоматами, зокрема грою “Життя”, є Golly — програмою, що симулює гру “Життя” та інші двовимірні клітинні автомати зі зручним інтерфейсом та швидким виконанням. Для цього програма застосовує алгоритм розрахунку великих площ клітинних автоматів під назвою Hashlife. Цей алгоритм дозволяє використовувати певну надмірність інформації в еволюції структурованих систем, наприклад, космічних кораблів чи осциляторів. Він надає можливість програмі запам’ятовувати результат еволюції певних конфігурацій, заносячи його в хеш-таблицю, та відразу розташовувати його замість багатократного обрахунку одних і тих самих елементів у великій кількості.

Наприклад, якщо в даній конфігурації автомату використовується декілька тисяч легких космічних кораблів, алгоритму необхідно буде прорахувати їх переміщення лише один раз, зберегти конфігурацію у хеш-таблицю, а далі, коли буде потрібно знову обрахувати рух цього космічного корабля, алгоритм звернеться до хеш-таблиці і відразу розмістить результат еволюції цієї конфігурації для всіх існуючих кораблів цього типу. Це дозволяє пришвидшити обрахування складних, проте систематизованих конфігурацій у тисячі разів, дозволяючи створювати гігантські структури та обраховувати їх у реальному часі. Алгоритм навіть дозволяє обрахунок різних частин автомату з різними швидкостями, максимізуючи швидкість роботи без втрати коректності виконання.

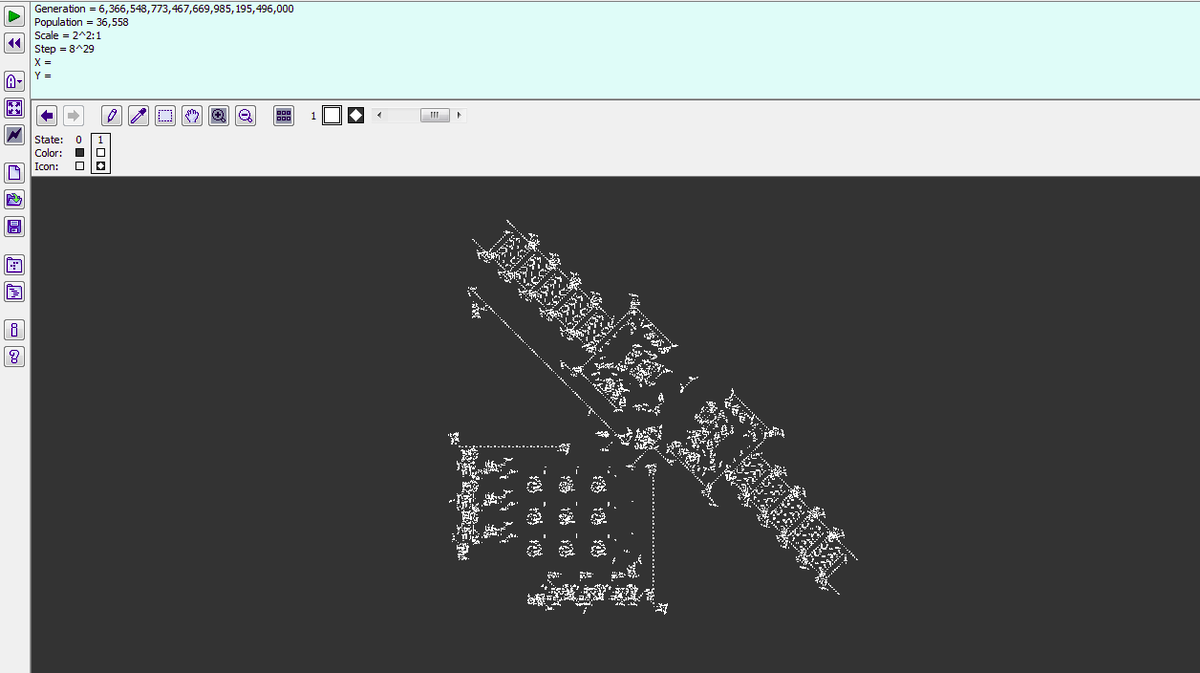
За допомогою використання алгоритму Hashlife програма Golly здатна обраховувати складні структури у реальному часі, дозволяючи створювати дуже великі космічні кораблі та дивитись на процес їх руху. Зокрема, разом з програмою постачається набір конфігурацій, як для гри “Життя”, так і для інших клітинних автоматів. Серед них можна побачити типових представників таких класів структур, як осцилятори, космічні кораблі, рущниці та ще складніші структури. Ці структури дозволяють детально розглянути ефективність роботи описаного вище алгоритму на практиці: навіть при великій швидкості переходу до наступної ітерації програма здатна обраховувати результат в реальному часі. На рисунку 1.28 наведено приклад однієї зі складних структур, що здатні оброблятися алгоритмом Hashlife значно швидше, ніж класичним методом.

Рисунок 1.28 — Еволюція надскладної структури у Golly

Функціонал програми дозволяє створювати власні конфігурації, призупиняти та змінювати швидкість переходу на нову ітерацію, зберігаючи при цьому швидкодію алгоритму Hashlife. Golly дозволяє масштабувати та розглядати як певні окремі елементи конфігурації, так і загальну структуру. При створенні власної конфігурації застосунок дозволяє копіювати вже створені структури, таким чином полегшуючи роботу з великими структурами. Створення нових космічних кораблів (наприклад, з новим кутом руху чи новою швидкістю, для якої раніше не було знайдено космічного корабля) зазвичай проводиться дослідниками саме в цьому застосунку, тому його можна назвати стандартом для гри “Життя”, хоча й існують інші альтернативи. На рисунку 1.29 зображено інтерфейс програми:

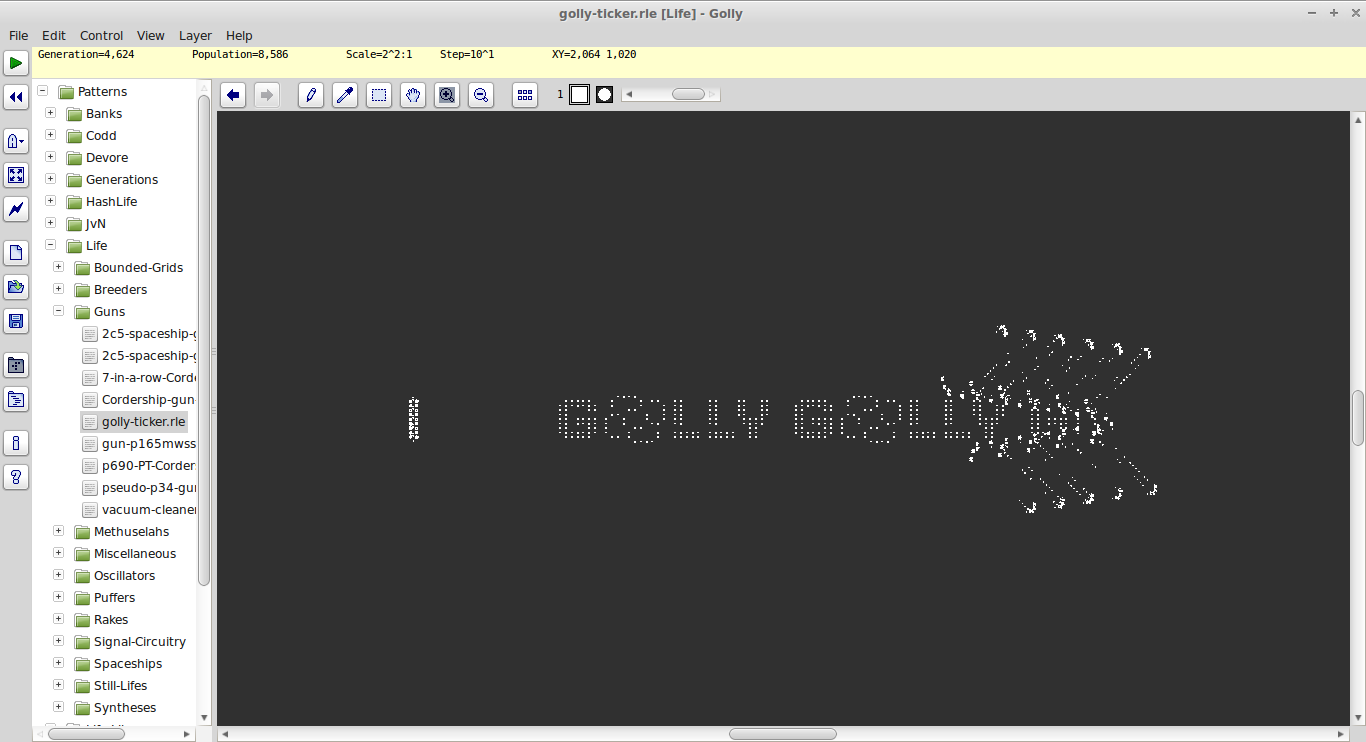
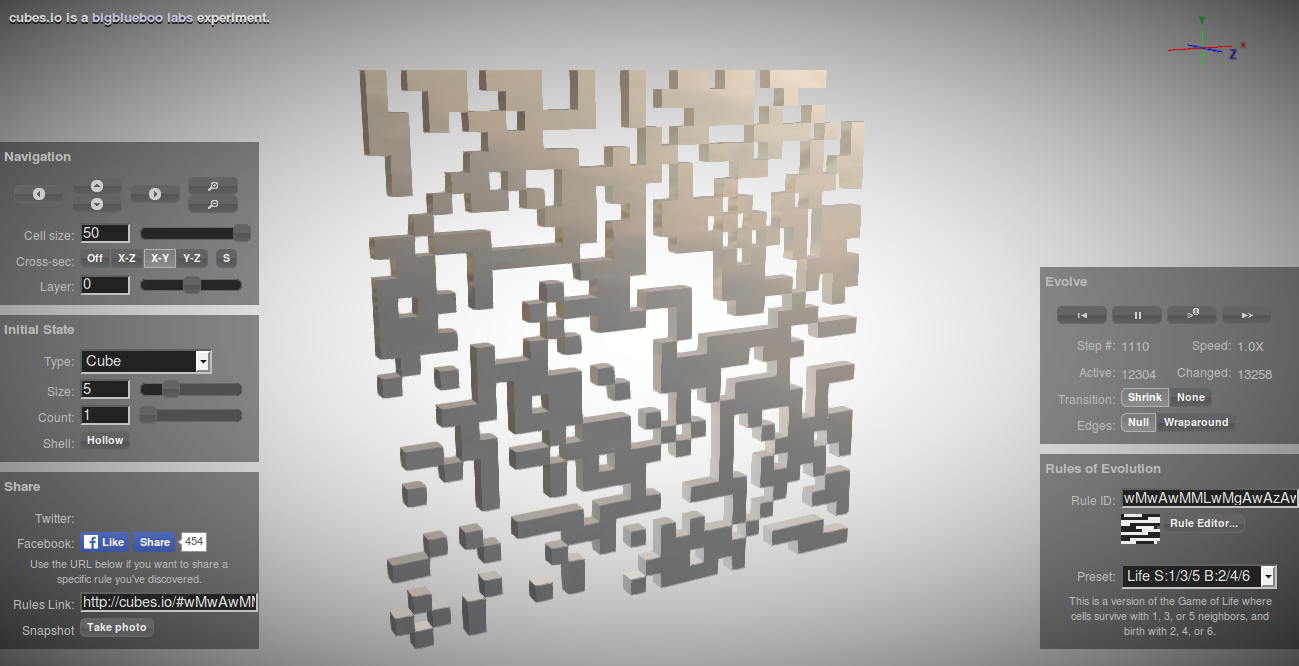


Рисунок 1.29 — Інтерфейс Golly

### Реалізації тривимірних клітинних автоматів

Тривимірні клітинні автомати є менш розповсюдженими порівняно з двовимірними. Це пов’язано з більш високою складністю в аналізі таких автоматів, пошуку правил, за якими автомат виказує ознаки хаосу та створення стабільних структур. Крім того, відображення процесу роботи ускладнене тим, що на двовимірному екрані важко відобразити еволюцію тривимірної структури. Тому завдання візуалізації тривимірних клітинних автоматів й досі залишається актуальним.

Яскравим прикладом якісної реалізації цього завдання є cubes.io – WebGL-реалізація тривимірного клітинного автомату для браузерів. Дана реалізація не надає настільки широкого функціоналу, як Golly, зокрема, в ній взагалі відсутня можливість задавати власну початкову конфігурацію чи змінювати стан клітин в процесі роботи, а також програма працює лише з автоматами, клітини в яких мають лише два можливих стани. Проте дана реалізація має певні плюси: вона має зручний інтерфейс, що дозволяє регулювати форму початкової конфігурації, правила переходу до наступного стану, а також змінювати формат відображення. Як можна побачити з рисунка 1.30, програма дозволяє як розглядати структуру з різних сторін в трьох вимірах, так і дивитись на окремі шари у двох з трьох координат.

Рисунок 1.30 — Зовнішній вигляд cubes.io

Таким чином, разом з можливістю призупинення та покрокового перегляду, застосунок надає можливість детально розглянути процес еволюції обраної конфігурації при даних правилах. Також програма динамічно обчислює освітлення та віддзеркалення зображення на поверхні клітин, що дозволяє чіткіше бачити окремі клітини та таким чином допомагає у візуалізації клітинного автомату. Навігацію камери реалізовано за допомогою миші та клавіатури, що дозволяє зручно повертати камеру на необхідний кут.

Загалом візуалізація тривимірного клітинного автомату на веб-сайті cubes.io є досить гарною репрезентацією процесів, що відбуваються в автоматах, що розглядаються, проте не дає достатнього контролю над їх еволюцією чи можливості керувати її та направляти в потрібному напрямку. Такі висновки дозволяють визначити ціль даної роботи як реалізацію системи, що дозволятиме в тому ж ступені коректно та зручно відображати тривимірні клітинні автомати, проте надасть більший контроль над еволюцією системи.

## Висновки до розділу 1

В даному розділі було розглянуто поняття клітинних автоматів, їх класифікацію та способи застосування. Зокрема були детально розглянуті найбільш поширені двовимірні та тривимірні автомати. Серед двовимірних автоматів було розглянуто гру “Життя” та мураху Ленгтона, при цьому відзначено їх здатність до непередбачуваної еволюції та створення стабільних структур. Також було детально розглянуто існуючі програмні реалізації цих автоматів, тобто засоби для роботи із ними. В результаті було встановлено, що більшість існуючих програмних засобів для клітинних автоматів не пристосовані для роботи із тривимірними автоматами, а існуючі програми для роботи з ними не надають достатнього функціоналу для проведення повноцінного дослідження. Тому встановлено необхідність розробки нового застосунку, що реалізує увесь необхідний функціонал та надасть зручний інтерфейс для роботи з ним.

# Опис функціональних можливостей розроблюваного застосунку

## 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.

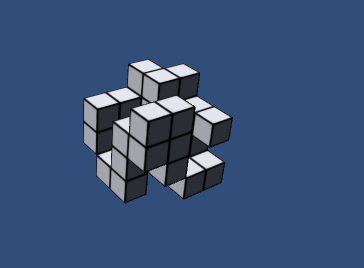
Застосунок повинен надавати можливість зберігати поточний стан клітинного автомату та завантажувати його в разі необхідності. Подібний функціонал дозволить зберігати знайдені стабільні структури чи конфігурації, здатні до довготривалої еволюції, що значно перевищує складністю початковий стан.

Розглянемо детальніше, яким саме чином планується вирішити вищезазначені проблеми.

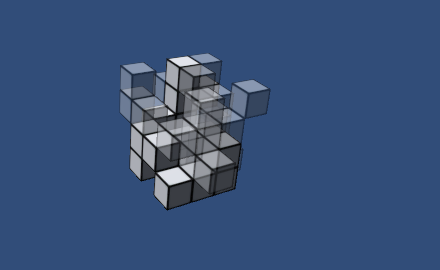
## Відображення стану тривимірного автомату

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:

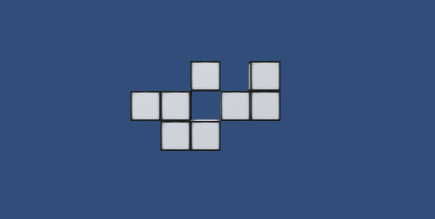
Рисунок 2.1 — Приклад тривимірного відображення клітинного автомату

Окрім вибору власне способу відображення окремої клітини, виникає необхідність у вирішенні й інших питань. Наприклад, для створення ефекту об’єму клітин, який необхідний для зручного розгляду стану тривимірного автомату, необхідно застосовувати освітлення. Наявність джерела світла у системі відображення забезпечує перспективу моделей клітин, дозволяючи таким чином розрізнити, на якому рівні у тривимірній структурі знаходяться клітини, що розглядаються. Також необхідний репрезентативний спосіб зобразити появу та зникнення клітини. При зміні стану до користувача необхідно чітко донести інформацію про те, як саме змінюється стан автомату. Для цього можна використати певну анімацію моделей клітин, що з’являються чи, навпаки, зникають при переході до наступної ітерації. Серед можливих анімацій варто зазначити поступову зміну прозорості від повністю невидимого до звичайного стану при появі та навпаки при зникненні клітини, а також зменшення та, відповідно, збільшення клітини з часом. Нижче можна побачити приклад анімації зміну стану клітин методом змінення їх прозорості. Рисунок 2.2 показує стан автомату в процесі переходу.

Рисунок 2.2 — Приклад відображення процесу переходу до нового стану

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.

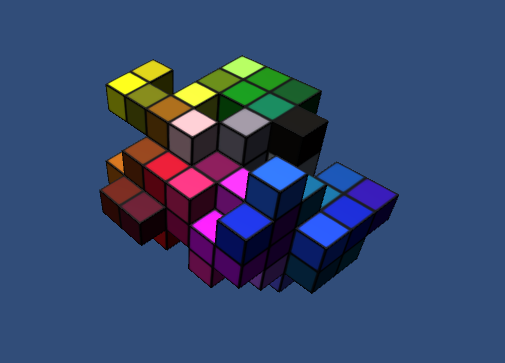
Проте навіть такі засоби можуть виявитись недостатніми для візуалізації внутрішньої структури автомату. При великій кількості заповнених клітин зовнішні клітини можуть закрити огляд внутрішніх, при цьому зміна кута огляду не допоможе. Тому важливим функціоналом, який потребує застосунок, є можливість 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 2.3.

Рисунок 2.3 — Приклад відображення окремого шару автомату

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 2.4.

Рисунок 2.4 — Приклад структури із кольоровим маркуванням клітин

## Розпізнавання переходу автомату до тривіального стану

### Випадок детермінованого клітинного автомату

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”.

### Випадок недетермінованих (ймовірнісних) клітинних автоматів

Вищезазначений підхід працює для детермінованих клітинних автоматів. Принциповою особливістю таких автоматів є те, що вони з певного стану завжди переходять в один конкретний стан. Проте ймовірнісні клітинні автомати не мають такої властивості. Через наявність елементу випадковості, певні стани з певною ймовірністю можуть перейти у два чи більше різних наступних стана. Через це навіть перехід до стану, що вже траплявся в процесі еволюції автомату, не гарантує, що еволюція клітинного автомату стала тривіальною. Загалом, у подібних умовах тривіальний стан описується значно складніше, адже певні структури, що випадково змінюються у часі, можна суб’єктивно також назвати тривіальними, якщо існує лише невелика кількість станів, в які дана конфігурація може перейти з певним розподілом ймовірностей.

Розглянемо декілька можливих варіантів еволюції ймовірнісних клітинних автоматів. Найпростішим варіантом розвитку є поступове зменшення кількості заповнених клітин та “смерть” клітинного автомату, при якій всі клітини становляться пустими. Зазвичай подібний сценарій характерний для клітинних автоматів, в яких для зародження клітини потребується занадто велика кількість сусідів, як і для збереження життя вже існуючої. У цьому випадку якщо більшість клітин початкової конфігурації й має достатню для життя кількість сусідів, клітини на краю заповненої області їх не матимуть, що викличе каскадну реакцію гибелі клітин, а через низьку швидкість відродження кількість клітин буде знижуватись до нуля.

Зворотнім варіантом є заповнення усього поля. Відповідно, якщо смерть клітини наступає лише при дуже великій кількості сусідей, може трапитись ситуація, коли навіть на границях автомату клітини не гинуть. Повністю автомат заповниться лише у тому випадку, якщо максимальна кількість сусідів, при якому клітина все ще не гине від перенаселення, перевищує можливу кількість сусідів (у випадку тривимірного автомату — 26). Однак автомат все одно може бути заповнений губчатою структурою, в якій заповнені клітини перемежовуються пустими, формуючи необхідну кількість сусідів для того, щоб ні одна клітина не гинула, але й жодна пуста клітина не може заповнитись.

Окрім цих двох тривіальних варіантів існують й складніші типи еволюції. Певні правила клітинних автоматів здатні формувати стабільні структури в процесі еволюції. Перехід до тривіального стану може відбуватись шляхом зупинення будь-яких змін, що не входять до складу стабільних структур. Для цього не обов’язково необхідне заповнення всього автомату, а також при цьому не відбувається його остаточна смерть. Однак стабільні структури в ймовірнісних автоматах мають певні особливості. Якщо ми розглядаємо варіант ймовірнісних клітинних автоматів, при якому зародження клітини можливе в певному діапазоні кількостей сусідів, то структура може залишатись стабільною, зокрема, якщо ймовірнісний фактор ніяк не впливає на еволюцію системи. Це може відбутись, якщо всі пусті клітини, що є сусідами даної системи, мають меншу кількість сусідів, ніж необхідно для їх заповнення, а всі заповнені клітини мають кількість сусідів, яка є більшою, ніж верхня межа загибелі від самотності, та меншою, ніж нижня межа загибелі від перенаселення. В певних випадках такий діапазон кількості сусідів взагалі відсутній, і можливий варіант, що клітина одночасно має шанс загинути від самотності та перенаселення. Тоді існування стабільних структур взагалі неможливо — будь-яка структура через певний час випадково втрачає певні клітини і, як результат, стабільність.

На рисунку 2.5 зображено стабільну структуру, що виникає при тій самій конфігурації, що й у минулому прикладі. Структура становить з себе куб 2х2х2 клітин.

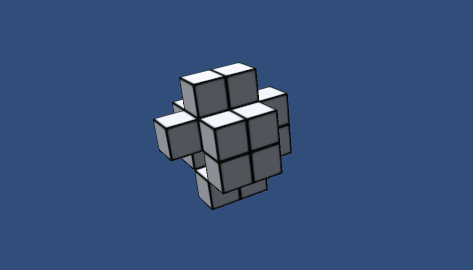
Рисунок 2.5 — Стабільна структура кубічної форми

Циклічні ж стабільні структури створити набагато складніше: так як зародження клітини залежить від випадкової кількості сусідів, то на певному кроці циклу клітина може створитись чи не створитись, що значно ускладнює формування таких структур. Значно ймовірніша поява структур, де певна клітина може випадковим чином заповнюватись чи видалятись, при цьому це ніяк не впливає на стабільність усієї системи. Це може статись, якщо поява цієї клітини не збільшує кількість сусідів інших клітин таким чином, щоб вони покинули “стабільну” зону між загибеллю від перенаселення та самотності. Таким чином, в ймовірнісних клітинних автоматах поява “змінних” стабільних структур зазвичай відбувається, коли поява окремих клітин не є обов’язковою для стабільності системи.

Найбільший інтерес представляють правила, за яких автомат здатен до активного розповсюдження, що не призводить до швидкого переходу у тривіальний стан. Зазвичай це автомати 3 та 4 класу. Саме такі клітинні автомати можуть викликати проблеми при підрахунку часу еволюції конфігурації. Описана вище проблема порівняння поточної ітерації з усіма попередніми стає ще більш суттєвою, адже навіть одне повторення не гарантує зациклення, а відсутність повторення ще не гарантує відсутність зациклення. Наприклад, якщо в процесі еволюції конфігурації з’явилась стабільна структура з випадково зароджуючимися клітинами, не впливаючими на стабільність структури, то вона може завадити коректному детектуванню зациклення і, відповідно, переходу в тривіальний стан.

Існують також варіанти напівстабільних структур, що здатні значний час залишатись стабільними, проте мають ймовірність вийти з цього стану. Наприклад, структура має декілька клітин, що з певною ймовірністю зароджуються та гинуть, не впливаючи на стабільність структури. Проте, якщо всі клітини випадково виникнуть на одній ітерації, в сумі їх буде достатньо, щоб сусідня клітина загинула від перенаселення, а структура через це втратила стабільність. Такі структури можуть викликати ситуації, коли тривіальний на перший погляд стан може через декілька десятків чи навіть сотен ітерацій вийти з подібного зациклення та продовжити активну еволюцію. Після цього еволюція може або продовжитись протягом значного часу, або завершитись одним з описаних раніше способів (повністю спустіти, заповнити все поле чи перейти до повністю стабільного, або навіть іншого напівстабільного стану).

Наведемо приклад такої напівстабільної структури. Нехай маємо такі правила клітинного автомату: зародження клітини при 6-8 сусідах, загибель від самотності при 4-6 та від перенаселення при 8-10 сусідах. Зображена на рисунку 2.6 конфігурація є напівстабільною при таких правилах. Структура є майже симетричною, але для повної симетричності не вистачає однієї клітини в лівій нижній частині зображення. Дана клітина зародиться на наступній ітерації з ймовірністю 1/3. Проте найлівіша та нижня ліва клітини також мають ймовірність 1/3 загинути від самотності. Якщо цього не трапиться, то структура стабілізується. Якщо ж хоча б одна з клітин зникне до того, як з’явиться нова клітина, структура дестабілізується і швидко перейде до пустого стану.

Рисунок 2.6 — Приклад напівстабільної структури

Цей приклад можна екстраполювати на менші ймовірності дестабілізації. Наприклад, якщо ймовірність виходу зі стабільного стану складає лише 10%, може пройти багато ітерацій, перед тим як система дестабілізується. Таким чином в напівстабільних системах відсутня гарантія того, що збереження структури протягом певної кількості ходів веде до її збереження на подальших ітераціях.

На основі розглянутих сценаріїв можна зробити висновок, що для ймовірнісних автоматів недоцільно використовувати той самий підхід до визначення часу еволюції, що й для детермінованих автоматів. Більшість структур, що є стабільними протягом нескінченно великих проміжків часу, взагалі не змінюються з часом. Формування стабільних структур з клітинами, що випадково змінюють свій стан, не порушуючи при цьому стабільність структури, є менш ймовірним. Виникнення ж зациклених структур у випадку ймовірнісного клітинного автомату є ще менш ймовірним. Так як подібний функціонал не є основною метою розробки програми, адже він лише слугує для зручного визначення часу еволюції системи, а ймовірність виникнення ситуацій, в яких з’являються складніші стабільні структури, є досить незначною, програма підтримуватиме лише визначення переходу до тривіального стану, що виражений повною відсутністю змін протягом значного часу. Надання складнішої системи визначення тривіального стану суттєво знизить продуктивність програми, що не є доцільним для обробки незначного проценту ситуацій другорядними підсистемами.

## Задання стану клітин автомату

### Безпосереднє задання стану окремих клітин

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.

### Початкове задання конфігурації автомату. Завантаження та запис

Окрім можливості задати значення окремих клітин, необхідним є також функціонал з задання початкової конфігурації автомату. Так як при заданні початкової конфігурації може виникнути потреба в нетривіальному заповненні великої кількості клітин, то необхідне рішення, що дозволить зменшити необхідність подібного підходу к мінімуму.

Першим способом вирішення цієї проблеми є випадкове задання початкової конфігурації. В багатьох випадках для перевірки властивостей певного набору правил тривимірного автомату найбільш доцільним є саме використання випадкового початкового стану, адже воно дозволить швидко розглянути еволюцію автомату на різних початкових умовах, що не є виродженими випадками. Доцільним є заповнення лише певної частини автомату замість усього поля, адже таким чином можна побачити, наскільки розширюється займана площа у процесі еволюції. Якщо ж все поле буде заповнено випадковими станами клітин, простору для розширення еволюція не матиме, що може нівелювати цінність експерименту. Також можливо надати користувачеві функціонал з визначення щільності заповнення початкової ділянки поля, що можна визначати через ймовірність заповнення окремої клітини.

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 — достатньо лише завантажити *.exe* файл та запустити його. 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.

## Висновок до розділу 2

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 відображення тривимірного автомату, задання стану автомату та підрахунок часу еволюції автомату.

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.

# Опис програмної реалізації

## Движок Unity як базис для розроблюваної програми

### Основні факти та призначення движка Unity

Unity – це мультиплатформовий движок для розробки відеоігор та інших застосунків, розроблюваний компанією Unity Technologies. Перша версія движку стала доступна у червні 2005 року. На даний момент найактуальнішою версією є 5.6.0f13. Движок підтримує такі платформи, як Windows, Mac OS, Linux, PlayStation 3/4, Xbox 360/One, Android, iOS та інші, включаючи можливість створення браузерної версії з використанням окремого плагіну чи JavaScript-реалізації OpenGL – WebGL. Останні версії движку розпочали підтримку систем VR.

Середа розробки Unity доступна на Windows та Mac OS. Движок підтримує скрипти на мовах програмування JavaScript та C#. Движок має безплатну версію, що дозволяє розробляти ігри та застосунки як для власного користування, так і для комерційного використання, за умови що прибуток від даного застосунку не перевищуватиме $100.000 на рік. Повна версія Unity має додатковий функціонал та ширшу підтримку з боку розробників, а також надає можливість розробляти продукти, прибуток від яких перевищуватиме $100.100.

Unity найширше використовується для розробки відеоігор. Він є популярним вибором для незалежних розробників через існування безплатної версії. Також він є популярним засобом для розробки мобільних додатків (процент мобільних ігор, написаних з використанням движку Unity, перевищує 50%) та браузерних ігор, зокрема, ігор для соціальних мереж, адже це один з небагатьох движків, що підтримують таку функцію. Проте, движок може використовуватись й для інших цілей. Будь-який застосунок, що потребує активної роботи з графікою чи VR-системами та розрахований на декілька платформ, доцільно створювати за допомогою движку Unity. Його безкоштовність, зручність в використанні, можливість роботи з тривимірною графікою та освітленням робить його ідеальним вибором для розробки системи моделювання еволюції тривимірних клітинних автоматів.

### Функціональні можливості движку та особливості його побудови

Движок Unity надає повноцінний функціонал для роботи з тривимірною графікою, освітленням, фізичними системами, звуком, інтерфейсом, анімацією, а також для контролю над управлінням та мережевою взаємодією. Цей движок надає інтерфейс розробника, що дозволяє створювати та розміщувати тривимірні чи двовимірні об’єкти за допомогою миші, налаштовувати їх властивості, присутні в цих об’єктах компоненти та їх параметри. Він також надає велику кількість різноманітного інструментарію для роботи із анімацією, графікою, фізикою та інтерфейсом користувача. При цьому функціонал Unity розширюваний — у середі розробки присутній магазин розширень, які дозволяють додати інструментарій для роботи у інших областях чи просто надають певний функціонал чи набір моделей, текстур, шейдерів тощо.

Для створення застосунку движок надає можливість по формуванню так званих сцен. Сцени містять певний набір об’єктів, кожен з яких має свою позицію, поворот, масштабування та набір різноманітних компонентів. Unity надає можливість створювати об’єкти заздалегідь та розміщувати їх за допомогою зручного інтерфейсу, або генерувати їх у процесі роботи застосунку через код. На кожній сцені існує принаймні одна камера. Камера відповідає за формування зображення з розташованих на сцені тривимірних моделей чи двовимірних спрайтів. Рухом камери також можна керувати за допомогою програмного коду.

Серед можливих компонентів варто виділити рендерери (компоненти, що відповідають за зображення об’єкту), джерела звуку, колайдери (компоненти, що відповідають за фізичні границі об’єкту для обрахунку фізики або детектування подій, пов’язаних з цією областю), системи часток тощо. Найпоширенішим типом компонентів є скрипти. Скрипти є спеціальними класами, що містять методи, які виконуються для даного об’єкту в певний час його життєвого циклу. Більшість функціоналу щодо керування станом об’єктів, їх позицією, властивостями та компонентами відбувається саме у скриптах. Будь-яку поведінку об’єкта, що неможливо задати іншими компонентами, реалізують за допомогою скриптів.

Робота з графікою у движку Unity поділяється на двовимірну та тривимірну. При двовимірному режимі використовується ортогональна камера, що завжди направлена в одну сторону. При тривимірному режимі використовується перспективна камера, яку можна повертати вільно під час роботи програми. Самі ж засоби відображення за тим ж принципом поділяються на спрайти та меші. Спрайт становить з себе двовимірну прямокутну картинку. Спрайти найчастіше використовуються у двовимірному режимі. Меш — це тривимірна модель, яка зазвичай обтягнута певними текстурами. Її можна використовувати як у двовимірному, так і у тривимірному варіанті. Обидві структури можна модифікувати шляхом надання їм необхідного матеріалу. Матеріал становить реалізацію шейдеру — спеціальної програми, що відповідає за обробку відображення об’єкта. Матеріал задає конкретні значення параметрів шейдеру, наприклад, текстуру, прозорість, яскравість тощо. Саме за допомогою шейдерів та матеріалів можливо змінювати характер відображення об’єкту, наприклад, бліки від світла, ефект дзеркала, нерівності поверхні і тому подібне.

Освітлення об’єктів, що також оброблюється шейдерами, можна задавати джерелами світла — точковими та направленими. Інформація про присутні джерела світла може використовуватись для створення візуальних ефектів на об’єктах. Крім того, Unity надає можливість використовувати й інші засоби відображення, наприклад, систему часток. Це візуальний ефект, який формує серію невеликих об’єктів із загалом випадковими характеристиками, які слідують певній ймовірнісній поведінці. Системи часток можуть використовуватись для створення ефектів диму, іскор, дощу, скляних уламок тощо.

Тісно пов’язана з графікою й анімація об’єктів. Вона дозволяє трансформувати об’єкт з часом чи навіть створювати складні системи для контролю над відображенням та характеристиками об’єкту в залежності від умов. Найпростішим варіантом анімації є покадрова анімація спрайтів, що просто замінює зображення з часом. Інші види анімації можуть змінювати певні параметри об’єкту чи його частин, наприклад, позицію, поворот, колір чи інші характеристики. Більш складна анімація тривимірних об’єктів реалізується шляхом задання скелету моделі та накладання необхідних рухів на неї. Unity має інструментарій для роботи з подібною анімацією, зокрема, з антропоморфними моделями. Окрім цього, движок надає можливість зручно контролювати перехід від одної анімації до іншої. Для цього реалізовані так звані аніматори — автомати, в яких станами є конкретні анімації. Аніматори дозволяють переходити у новий стан при виконанні певної умови, робити плавний перехід до нової анімації, змішувати їх у пропорції. За допомогою аніматорів та скриптів, що контролюють їх параметри, можна реалізувати складні анімовані моделі, наприклад, персонажів гри.

Фізична модель движку включає в себе колайдери, що здатні відслідковувати зіткнення з іншими об’єктами, фізичні матеріали, що відповідають за такі властивості, як тертя та пружність, а також так звані жорсткі тіла, на яких діє гравітація та до яких можна прикласти певну силу, задаючи фізично коректний рух об’єкту.

З версії Unity 5 движок містить зручний засіб формування інтерфейсу користувача. Він дозволяє розробляти гнучкий інтерфейс, який можна використовувати при будь-якому розширенні екрана. Для цього існує декілька засобів, що допомагають визначити формат відображення інтерфейсу незалежно від розширення. Зокрема, в Unity існує поняття якорю — точки чи області, відносно якої необхідно розраховувати позицію елементів інтерфейсу. Якоря дозволяють розтягувати інтерфейс, розташовувати його на фіксованій відстані від краю чи іншим чином контролювати позицію елементу. Крім того, у движку міститься система подій, що використовується для обробки натискання кнопок інтерфейсу, зміни значень в певних полях тощо. У випадку таких елементів до них можна додати функцію-слухача, яка виконуватиметься при виникненні цієї події. Це надає зручний спосіб прив’язати певний реалізований функціонал до елементів інтерфейсу.

Управління застосунком керується шляхом надання універсального інтерфейсу, що дозволяє отримати дані про дії користувача незалежно від платформи та засобу введення, що він використовує. Окрім можливості відслідкувати натиск та відпускання окремих кнопок, інтерфейс також дозволяє працювати із вісями, що здатні задавати ступінь натиску певної кнопки. За допомогою вісей реалізується, зокрема, обробка коліщатка миші чи рух джойстику. Конкретні кнопки, що відповідають за певні дії, допускається налаштовувати перед запуском програми, а за допомогою скриптів — і під час її виконання.

Мережева взаємодія контролюється рядом спеціальних класів, що дозволяють сформувати коректні мережеві запити на відправлення чи отримання інформації. Дана система дозволяє відносно легко організувати взаємодію з певним сервером. Однак Unity містить й потужніші інструменти для мережевої взаємодії, наприклад, автоматична синхронізація об’єктів з обробкою їх стану на сервері для реалізації режиму роботи декількох користувачів.

Таким чином, Unity надає широкий спектр можливостей у різних областях, формуючи комплексну систему для написання програм із використанням графіки.

### Написання скриптів для движка Unity

Як зазначалося раніше, майже уся логіка взаємодії з об’єктами на сцені задається шляхом створення скриптів. Кожен скрипт є класом, що успадковується від класу MonoBehaviour. Клас MonoBehaviour містить ряд атрибутів та методів, які є важливими для написання скриптів у Unity. Зокрема, атрибут *transform* містить компонент Transform даного об’єкту, і якому задається поточна позиція, поворот та масштаб об’єкту. Крім того, саме цей об’єкт містить посилання на вкладені об’єкти. Об’єктна структура цього движку передбачає можливість розташовувати об’єкти на інших об’єктах у ієрархічну структуру. При цьому позиція, поворот та масштаб батьківського об’єкту впливатимуть на вкладений об’єкт. Тому компоненти Transform надають можливість задавати як локальну, так і глобальну позицію чи інші параметри об’єкту.

Атрибут *gameObject* містить посилання на сам об’єкт. Це посилання корисне, наприклад, при створенні об’єкту чи передаванні його у якості аргументу. Метод *GetComponent* дозволяє знайти на поточному об’єкті необхідний компонент, щоб звернутись до нього, його методів чи атрибутів. Взаємодія із іншими елементами загалом може відбуватись двома способами: отриманням компоненту за допомогою методу *GetComponent* або отриманням посилання на інший об’єкт чи скрипт через публічні атрибути. Unity дозволяє задавати значення публічних атрибутів скрипту через середу розробки. Зокрема, можна зазначити посилання на інший об’єкт на даній сцені як значення певного атрибуту, таким чином отримавши доступ до іншого об’єкту.

Виконання конкретного програмного коду відбувається шляхом автоматичного виклику певних функцій. Ці функції формують життєвий цикл об’єкту в Unity. Розглянемо певні функції з циклу.

* *Awake –* найперший метод, що викликається в життєвому циклі. Цей метод викликається вперше при активації об’єкту на сцені. В ньому задається ініціалізація скрипту чи певні одноразові дії, які необхідно відразу виконати при появі об’єкту на сцені.
* *Start –* метод, аналогічний попередньому, але який викликається після *Awake,* проте до обробки першого кадру. Також використовується для ініціалізації та одноразових дій. Використання *Awake* для ініціалізації є кращою практикою.
* *Update –* метод, що викликається при кожній зміні кадру. При частоті кадрів 60 в секунду, він викликатиметься відповідно 60 раз кожну секунду. В даному методі зазвичай обробляють певну поведінку, що постійно впливає на об’єкт, наприклад, його переміщення чи параметри, що змінюються динамічно.
* *FixedUpdate –* метод, аналогічний *Update,* але такий, що викликається через фіксовані проміжки часу. Виклик методу *Update* відбувається через нерегулярні проміжки часу, адже це залежить від частоти кадрів, яка залежить від доступних програмі ресурсів. Для реалізації, наприклад, фізики надається перевага методу *FixedUpdate,* адже розробник може чітко передбачити, як часто цей метод викликатиметься, незалежно від продуктивності застосунку.
* *OnDestroy –* метод, що викликається при знищенні об’єкту. Він може використовуватись як деструктор, видаляючи зайві дані чи проводячи певні дії, які відповідають знищенню цього об’єкту (наприклад, створення візуального ефекту його знищення).
* *yield StartCoroutine –* популярний засіб запуску паралельного процесу для виконання певних дій. Може використовуватись з метою організації взаємодії з сервером чи проведення певних дій, які не повинні призупиняти роботу скрипту.

Існують й інші методи з життєвого циклу, які можна використати для реалізації певної логіки скрипту. Дані класи можуть звертатись до будь-яких об’єктів, у тому числі тих, що не є скриптами, формуючи таким чином обгортку для будь-якого функціоналу.

## Опис скриптів та їх взаємозв’язку

### Об’єктна структура програми

Об’єктна структура застосунку становить з себе множину взаємопов’язаних класів, що у сукупності реалізують усю логіку програми. Більшість класів становить з себе підклас класу MonoBehaviour та використовується у якості скриптів. Ці скрипти закріплюються за певними об’єктами на сцені чи на префабах у якості компонентів, їм задаються певні початкові значення за допомогою редактору Unity, а об’єкти, за якими вони закріплені, коректно налаштовуються та розміщуються на сцені.

Скрипти, що використовуються в даному застосунку, можна розділити на декілька груп. Основною групою є класи, що відповідають за власне обрахунок та відображення тривимірної структури клітинного автомату. До цієї групи, зокрема, відноситься клас, що не є підкласом MonoBehaviour, таким чином не будучи скриптом. Це клас LifeGame. У цьому класі знаходиться уся відділена від графічного зображення та інтерфейсу логіка обрахунку тривимірного клітинного автомату. Клас містить набір значень, що описує правила автомату, повністю зберігає поточний стан автомату, а також надає функціонал з ініціалізації, змінення розміру автомату, підрахунку наступної ітерації, підрахунку кількості сусідів тощо.

Даний клас агрегований класом GameDisplay. Це абстрактний клас, головною задачею якого є відображення структури автомату на екрані та загалом керування цим класом. Цей клас є породженим класом MonoBehaviour, але так як він є абстрактним, то безпосередньо як скрипт він не може використовуватись. Даний клас надає загальний функціонал для всіх форм відображення клітинного автомату, наприклад, інстанціювання екземпляру класу LifeGame на основі переданих до скрипта даних, запис та завантаження поточного стану автомату, керування процесом ітерації автомату та виклик абстрактних методів його відображення.

В програмі на даний момент використовується лише один породжений клас цього скрипта — клас Game3D, що відповідає за тривимірне відображення об’єкту, однак для тестування використовувався клас TestGame, що також є породженим класом GameDisplay. Цей клас відповідав за текстове відображення клітинного автомату та найпростіший функціонал та використовувався для тестування роботи клітинного автомату, а не його зручного відображення.

Клас Game3D є основним скриптом відображення клітинного автомату у якості тривимірної моделі. Він створює тривимірні моделі клітин та контролює їх зовнішній вигляд на основі поточного стану клітинного автомату. Цей скрипт реалізує методи відображення автомату цілком чи лише окремого шару, а також надає широкий спектр інструментів для контролю над зображенням зовнішніми скриптами. Наприклад, він надає інструментарій для змінення прозорості та кольору окремих клітин, взаємодії з автоматом через моделі кубів, що відражають окремі клітини, контролю над анімацією затухання чи появи клітини (як розширення функціоналу зі зміни прозорості) та випадкового задання початкового стану клітинного автомату. Крім того, скрипт містить й всі методи, що він успадкував від класу GameDisplay. Скрипт Game3D являє собою об’єкт, до якого звертаються інші скрипти та елементи інтерфейсу для додаткового керування процесом відображення, наприклад, для призупинення та відновлення ітерації автомату.

Наступною групою скриптів є скрипти, що відповідають за контроль відображення користувачем. Зокрема, за рух камери навколо тривимірної моделі автомату відповідає клас CameraMovement. Він також є підкласом MonoBehaviour і використовується у якості скрипту. Його основною метою є надання користувачу можливості повертати тривимірне зображення за допомогою миші, роздивляючись його структуру з різних сторін. Цей клас отримує інформацію про формат відображення від скрипту Game3D. У випадку відображення лише одного шару клітинного автомату CameraMovement виконує плавний поворот камери до перпендикулярного площині кута. Такий підхід дозволяє користувачеві зрозуміти, з якого саме кута він дивиться на даний шар.

Змінення стану та кольору клітин відбувається у класі CellSelection. Як і попередній клас, він є породженим класом MonoBehaviour і використовується у якості скрипту. Основною метою класу є відслідковування, на яку клітину користувач навів курсор, контроль над анімацією, що помічає цю клітину, підтримку коректної поведінки цієї анімації при переведенні на іншу клітину чи зміні шару, а також можливість обрати одну з заповнених клітин для вибору її кольору. Вибір кольору відбувається за допомогою спеціального плагіну Color Picker, що надає зручний інтерфейс у вигляді палітри та змінює колір обраного об’єкту при його виборі з палітри. Можливість зміни стану клітини чи її кольору, а також анімація поточної позиції курсору працюють лише при відображенні окремого шару автомату. Для зміни стану клітин в об’єкті LifeGame та контролю над відображенням автомату цей скрипт також звертається до скрипту Game3D.

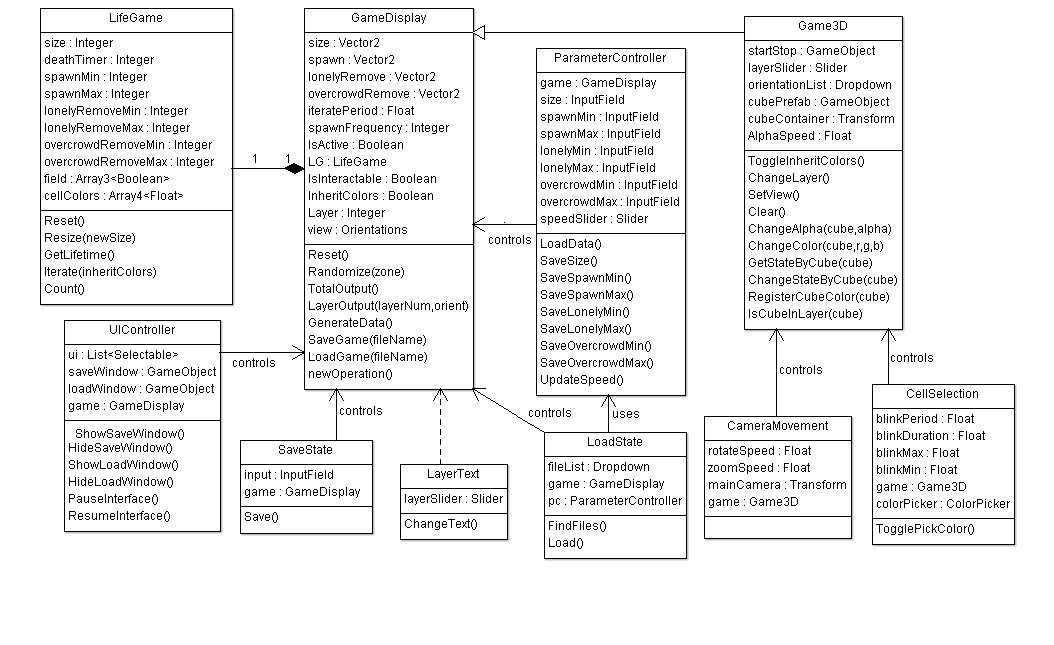
Останньою групою скриптів є такі, що відповідають за взаємодію між інтерфейсом користувача та відображеням клітинного автомату, а також за контроль над самим інтерфейсом. Основними класами цієї групи є класи UIController та ParameterController. Перший клас відповідає за відображення діалогових вікон запису та завантаження файлів. Він зберігає масив усіх контрольованих елементів інтерфейсу та вимикає можливість взаємодії з ними при виведенні одного з діалогових вікон, відновлюючи таку можливість після їх закриття.

Другий клас, ParameterController, відповідає за задання та виведення параметрів автомату. Він контролює взаємодію із полями введення значень розміру автомату, кількості сусідів, необхідної для зародження чи загибелі клітини тощо. При запуску програми чи завантаженні файлу в задачі даного класу входить вивести у текстові поля актуальні значення правил автомату та його розміру. Також цей клас ініціалізує коректну зміну розміру клітинного автомату при зміненні відповідного значення. Обидва даних класи є скриптами, тобто підкласами MonoBehaviour.

Класи SaveState та LoadState відповідають за запис та завантаження об’єктів класу LifeGame відповідно. Вони є проміжним класом для взаємодії між інтерфейсом користувача та класом GameDisplay, що виконує власне серіалізацію та десеріалізацію об’єкта. Клас LoadState також відповідає за виведення списку доступних файлів для завантаження.

Останній клас цієї групи, LayerText, що також є скриптом, контролює виведення номеру поточного шару, який відображається в даний момент, у відповідне поле над повзунком вибору шару. Усі класи даної групи взаємодіють з певними елементами інтерфейсу та, за винятком класу LayerText, із скриптом Game3D.

На рисунку 3.1 зображена повна діаграма класів UML.

Рисунок 3.1 — UML-діаграма класів програми

Для розуміння структури програми, окрім класової структури, необхідно також розглянути ієрархію об’єктів Unity та компоненти цих об’єктів. Але перед цим варто зазначити, які префаби використовуються в даній програмі. Єдиним префабом, що використовується в роботі, є префаб моделі клітини. Він містить компонент-рендерер, що відповідає за відображення тривимірної моделі (у даному випадку — куба), колайдер, що використовується для відстеження наведення на клітину курсору, а також скрипт ObjectColor з плагіну Color Picker, основною метою якого є задання обраного кольору об’єкту.

Кореневими об’єктами ієрархії є чотири об’єкти: EventSystem, ColorPicker, Canvas та GameMaster. Перший об’єкт є стандартним об’єктом для контролю над подіями, зокрема, подіями інтерфейсу користувача. Другий об’єкт, ColorPicker, формує елемент інтерфейсу, що відображає палітру вибору кольору. Цей об’єкт належить до плагіну Color Picker та власне реалізує його функціонал з визначення кольору окремих клітин.

Останні два об’єкти ж мають ширше призначення. Об’єкт GameMaster контролює еволюцію автомату та його відображення, а також містить ряд об’єктів-дітей, що виконують інші функції взаємодії з тривимірною моделлю автомату. Цей об’єкт містить скрипт Game3D, в який передаються базові значення параметрів автомату та посилання на певні елементи інтерфейсу, наприклад, кнопку старту/призупинення еволюції автомату та елементи інтерфейсу, що відповідають за задання шару та формату відображення автомату.

Дітьми GameMaster є два об’єкти. Перший з них — CubeContainer – є контейнером для всіх створюваних моделей кубів, що відражають поточний стан конкретних клітин. Він не має додактових компонентів та розташований в позиції об’єкта-батька. Другий об’єкт, Hinge, є об’єктом, що відповідає за керування відображенням та змінення стану автомату. Він також розташований в тій самій позиції, що й об’єкт батько, та містить два скрипти: CameraMovement та CellSelection. Обидва скрипти мають посилання на об’єкта-батька та задані параметри повернення камери та відображення курсору відповідно. Перший скрипт також має посилання на головну камеру. Головною камерою є об’єкт Main Camera, що є дитиною об’єкта Hinge. Він не містить додаткових компонентів, проте знаходиться на певній дистанції від об’єкта-батька. Камера повернута прямо на об’єкт Hinge. Скрипт CameraMovement здатен наближувати та віддаляти камеру від цього об’єкту, а також повертати цей об’єкт, таким чином надаючи можливість роздивитись тривимірну структуру з різних сторін. Ще одним об’єктом-дитиною Hinge є об’єкт Directional Light, що відповідає за направлене світло, яке надає зображенню об’ємність.

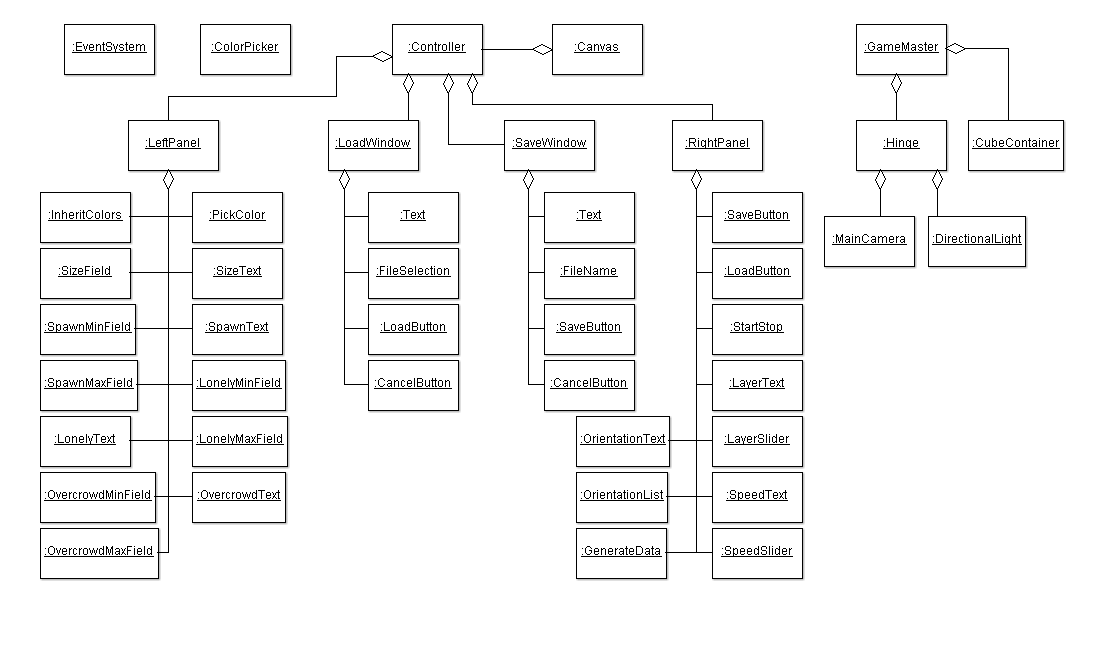
Об’єкт Canvas відповідає за зображення інтерфейсу. Цей об’єкт є стандартним об’єктом-контейнером для елементів інтерфейсу користувача та містить відповідні компоненти — скрипти Canvas, Canvas Scaler та Graphics Raycaster, що необхідні для коректної роботи інтерфейсу користувача. Об’єктом-дитиною Canvas є об’єкт Controller. Він містить компоненти-скрипти UIController та ParameterController, які містять посилання на необхідні елементи інтерфейсу та скрипт Game3D для контролю над ними. Також даний об’єкт містить чотири своїх об’єкти дитини: SaveWindow, LoadWindow, LeftPanel та PightPanel.

SaveWindow та LoadWindow є вимкненими на початку роботи програми. Ці об’єкти є діалоговими вікнами для запису та завантаження стану автомату відповідно. Вони містять відповідно скрипти SaveState та LoadState, за допомогою яких відбувається контроль над записом та завантаженням стану автомату. Дані скрипти містять посилання на скрипт Game3D, а у разі скрипту завантаження також й на скрипт ParameterController задля можливості оновити значення параметрів автомату у відповідних текстових полях. Діалогові вікна містять кнопки відміни, що звертаються до скрипту UIController для закриття вікна, та кнопки підтвердження, що окрім закриття вікна також звертаються до відповідного скрипта задля запису чи завантаження стану.

Об’єкти LeftPanel та RightPanel є лише контейнерами для елементів інтерфейсу, що знаходяться ліворуч та праворуч від тривимірної моделі автомату відповідно. Об’єкт LeftPanel вміщує такі елементи інтерфейсу, як перемикачі PickColor та InheritColors, що відповідають за можливість вибору кольору замість переключення стану та можливість наслідування кольору при підрахунку наступної ітерації; а також текстові поля та написи, що використовуються для задання параметрів автомату. При зміні значень у цих полях викликаються відповідно методу зі скрипту ParameterController, що відтворює зміни в об’єкті класу LifeGame. Перемикачі ж взаємодіють зі скриптами CellSelection та Game3D.

RightPanel вміщує такі елементи інтерфейсу, як кнопки запису та завантаження стану автомату, які звертаються до скрипту UIController для відкриття відповідних діалогових вікон; кнопку запуску/призупинення еволюції автомату, що звертається безпосередньо до скрипту Game3D; повзунок зміни поточного шару, який звертається до скриптів LayerText на відповідному написі та Game3D для власне змінення шару; повзунок зміни швидкості еволюції, що звертається безпосередньо до скрипту Game3D, та відповідний напис; а також список можливих форматів відображення, який при зміненні значення теж звертається до скрипту Game3D, з відповідним написом.

Увесь застосунок працює в одній сцені, без переходів на інші. На рисунку 3.2 зображена діаграма об’єктів сцени.

Рисунок 3.2 - UML-діаграма ієрархії об’єктів Unity

### Клас LifeGame

Як зазначалося раніше, клас LifeGame відповідає власне за обрахунок тривимірного клітинного автомату у відриві від способу його відображення. Це серіалізовуваний клас, що означає, що допускається його серіалізація та збереження у файл. Через це клас містить лише серіалізовувані атрибути — базові типи int, float та bool, а також їх масиви. До публічних атрибутів класу відносяться:

* *int size* – числове значення, яке відповідає довжині однієї сторони куба-поля автомату, вказаної в клітинах;
* *int spawnMin* – нижня границя діапазону кількості сусідів, при якому може зародитись клітина;
* *int spawnMax –* верхня границя того ж діапазону;
* *int lonelyRemoveMin* - нижня границя діапазону кількості сусідів, нижче якого клітина загине від самотності на наступній ітерації;
* *int lonelyRemoveMax* — верхня границя того ж діапазону;
* *int overcrowdRemoveMin* - нижня границя діапазону кількості сусідів, вище якого клітина загине від перенаселення на наступній ітерації;
* *int overcrowdRemoveMax* — верхня границя того ж діапазону;
* *bool[,,] field –* тривимірний масив булевих значень, що репрезентує клітинний автомат. Значення окремого поля відповідає стану клітини з відповідними координатами в сітці (*true* для заповненої та *false* для пустої). Масив завжди кубічний та має розмірність *size*;
* *float[,,,] cellColors –* чотиривимірний масив чисел з плаваючою точкою, що зберігає дані про колір кожної клітини. Перші три виміри масиву відповідають координатам клітини у сітці автомату та дорівнюють *size*, а останній завжди дорівнює трьом і відповідає трьом базовим кольорам (червоний, зелений, синій відповідно), значення яких зберігаються для даної клітини.

### Клас GameDisplay

Клас GameDisplay є базовим класом скриптів, що відповідають за відображення клітинного автомату. Він містить загальний інструментарій, необхідний всім подібним системам, та вказує декілька абстрактних методів для наслідування. До цих абстрактних методів, зокрема, відносяться *TotalOutput*, що відповідає за виведення стандартного, тривимірного відображення автомату, та метод *LayerOutput*, який відповідає за виведення конкретного шару при конкретному форматі виведення, які передаються до методу в якості аргументу. Обидва абстрактних методи не повертають значень.

Клас GameDisplay містить наступні публічні атрибути:

* *int size –* атрибут, що містить значення розміру клітинного автомату, що буде відображений цим класом;
* *Vector2 spawn –* атрибут, що містить базові мінімальне та максимальне значення кількості сусідів, за яким клітина зароджується;
* *Vector2 lonelyRemove –* атрибут, що містить базові мінімальне та максимальне значення кількості сусідів, нижче якої клітина гине від самотності;
* *Vector2 overcrowdRemove* – атрибут, що містить базові мінімальне та максимальне значення кількості сусідів, вище якої клітина гине від перенаселення;
* *float iteratePeriod –* кількість часу, що проходить між двома ітераціями;
* *int spawnFrequency –* відношення заповнених клітин до їх загальної кількості при заповненні поля випадковим чином;
* *int Layer –* атрибут, що заданий get- та set методами, як й усі наступні в даному класі. Він зберігає поточне значення шару, який відображується при пошаровому форматі, та викликає метод *LayerOutput* при присвоєнні цьому атрибуту нового значення (за умови того, що вибраний пошаровий формат відображення);
* *Orientations View –* атрибут типу Orientations (перелічення, що може приймати значення XYZ, XY, XZ та YZ), що позначає формат відображення автомату. Перший варіант відображає автомат у трьох вимірах, а при присвоєнні цього значення атрибуту викликається метод *TotalOutput,* а інші три варіанти відповідають відображенню одного шару в одній з трьох можливих площин та викликають метод *LayerOutput*;
* *bool IsActive –* атрибут, що здатен лише повертати значення, та показує, чи відбувається в даний момент еволюція автомату;
* *LifeGame LG* – атрибут, що повертає значення об’єкту, що містить інформацію про правила та поточний стан автомату;
* *bool IsInteractable* – атрибут, що показує, чи можна в даний момент взаємодіяти із відображенням автомату;
* *bool InheritColors –* атрибут, що показує, чи необхідно успадковувати колір при заповненні клітини.

### Клас Game3D

Породженим класом від абстрактного класу GameDisplay є клас Game3D. Цей клас становить з себе реалізацію GameDisplay, що використовує тривимірні моделі для відображення поточного стану автомату. Він також організує взаємодію з певними елементами інтерфейсу та надає функціонал для інших скриптів, що дозволяє керувати відображенням автомату. У класі містяться наступні публічні атрибути:

* *GameObject startStop* – посилання на кнопку інтерфейсу, що відповідає за запуск та призупинення еволюції автомату;
* *Slider layerSlider –* посилання на скрипт Slider, що контролює повзунок вибору поточного шару для відображення;
* *Dropdown orientationList –* посилання на скрипт Dropdown, що контролює меню вибору формату відображення автомату;
* *GameObject cubePrefab –* посилання на префаб об’єкту, що слугує відображенням окремої клітини;
* *Transform cubeContainer –* посилання на об’єкт-контейнер для усіх створюваних кубів;
* *float alphaBase –* базове значення швидкості анімації, що базується на зміні прозорості куба. Використовується для обрахунку атрибуту *AlphaSpeed;*
* *float AlphaSpeed –* атрибут, що повертає значення швидкості зміни прозорості в анімаціях появи чи зникнення кубів.

Також цей клас містить атрибути, що повертають значення атрибуту *size* локального об’єкту LifeGame та розмір тривимірної моделі з префабу *cubePrefab.*

### Класи CameraMovement та CellSelection

Класи CameraMovement та CellSelection відповідають за керування відображенням автомату та взаємодію з ним. Зокрема, CameraMovement надає можливість керувати камерою для розгляду структури автомату з будь-якої сторони. Цей клас підтримує відображення автомату цілком у тривимірній формі чи лише одного шару, в залежності від формату відображення. Керування камерою відбувається за допомогою миші: із затиснутою лівою кнопкою миші її рух повертає тривимірну модель автомату в обраному напрямку, а коліщатко миші здатне наближувати та віддаляти зображення. Даний клас містить наступні публічні атрибути:

* *float rotateSpeed –* визначає швидкість повернення камери під керівництвом користувача чи автоматично під час зміни формату відображення;
* *float zoomSpeed* – визначає швидкість руху камери при її наближенні чи віддаленні коліщатком миші;
* *Transform mainCamera –* посилання на головну камеру сцени, що використовується для змінення її позиції при наближенні чи віддаленні зображення;
* *Game3D game –* посилання на скрипт Game3D, який використовується для визначення формату відображення та іншої інформації.

Клас CellSelection, у свою чергу, відповідає за зміну стану клітин та їх кольору. Він підтримує два режими роботи: зміна стану та вибір клітини для зміни кольору. Власне вибір кольору відбувається за допомогою плагіну Color Picker. У першому випадку метод контролює відображення курсору та зміну стану клітини при натисканні лівої кнопки миші. У другому метод керує також відображенням обраної клітини та зміною цього виділення при натисканні на іншу клітину.

Клас CellSelection містить наступні публічні атрибути:

* *float blinkPeriod –* час між двома миготіннями курсору;
* *float blinkDuration –* тривалість одного миготіння курсору;
* *blinkMax –* максимальне значення прозорості при миготінні;
* *blinkMin –* мінімальне значення прозорості при миготінні;
* *Game3D game –* посилання на скрипт Game3D, що використовується для взаємодії з тривимірною моделлю автомату;
* *ColorPicker colorPicker –* посилання на скрипт ColorPicker, що відповідає за контроль над обиранням кольору клітини.

## Огляд інтерфейсу програми

Розроблений програмний застосунок використовує графічний інтерфейс користувача. Для реалізації інтерфейсу використовується вбудована в Unity система розробки інтерфейсу, що базується на розміщенні програмістом необхідних елементів безпосередньо на екрані та заданні інтерактивним елементам функцій, що мають викликатись при виникненні певних подій (наприклад, натискання кнопки).

Інтерфейс програми можна поділити на три групи. Першою групою є ліва панель. Вона містить елементи інтерфейсу, що відповідають за можливість змінювати правила переходу до наступної ітерації клітинного автомату, а також за можливість змінювати розмір поля автомату, переключатись між режимами зміни стану клітин та вибору клітини для зміни кольору, та нарешті за можливість успадковувати колір при переході до наступної ітерації.

Другою групою є власне відображення клітинного автомату. Автомат зображується як група тривимірних моделей кубів, за замовчуванням білого кольору, з якими можна взаємодіяти у форматі відображення окремого шару шляхом натискання лівої кнопки миші на окремі куби, або у тривимірному відображенні шляхом руху миші із затиснутою лівою кнопкою миші. У першому випадку натискання кнопки миші призведе до зміни стану клітини чи її вибору для зміни кольору, в залежності від режиму. У другому випадку відбувається повернення зображення у обраному напрямку навколо середини автомату, що моделюється тривимірним зображенням. Крім того, зображення можна віддалити та наблизити коліщатком миші незалежно від режиму чи формату відображення.

Нарешті, третьою панеллю є права панель, що містить ряд кнопок для запису та завантаження конфігурація автомату, запуску а призупинення еволюції автомату, а також повзунки для змінення поточного шару і швидкості еволюції та випадаючий список для вибору формату відображення. На рисунку 3.3 зображено загальний вигляд інтерфейсу програми.

Рисунок 3.3 — Загальний вигляд інтерфейсу

Розглянемо елементи інтерфейсу програми детальніше. У верхньому лівому куті екрану знаходяться два перемикача: Pick Color та Inherit Colors. Обидва перемикача змінюють свої значення при натисканні на них лівою кнопкою миші. Перший відповідає за переключення між двома режимами взаємодії із шаром автомату. Якщо перемикач виключений, то у форматі відображення окремого шару натискання лівої кнопки миші призведе до змінення стану клітини. Якщо ж перемикач Pick Color включений, то натискання лівої кнопки миші призведе до вибору відповідної клітини, у випадку якщо вона є заповненою. Вибрана клітина виділятиметься періодичною зміною прозорості.

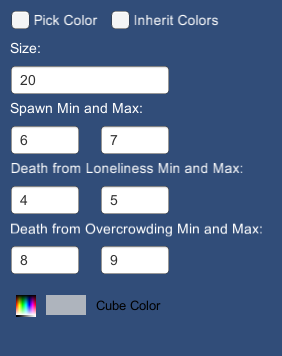
В свою чергу, перемикач Inherit Colors відповідає за використання чи ігнорування можливості успадковувати колір клітин при переході до наступної ітерації. Якщо перемикач виключений, при переході до наступної ітерації колір клітин, що вмирають, залишається незмінним, а нові клітини мають білий чи заздалегідь заданий колір. Якщо ж перемикач включений, клітини, що зароджуються, отримують колір, що приймає середнє значенні між кольорами усіх сусідніх клітин.

Наступним елементом інтерфейсу є поле для введення розміру автомату. Початковим значенням поля є розмір поточного автомату, зокрема довжина сторони поля кубічної форми у клітинах. Значення автоматично оновлюється при завантаженні нової конфігурації з файлу. При безпосередньому зміненні значення в полі автомат коректно змінює свій розмір, додаючи нові чи видаляючи зайві куби. Коректними значеннями поля є додатні цілі числа. Елемент знаходиться відразу під перемикачами.

Нижче знаходяться поля для введення кількості сусідів, необхідних для зародження клітини. Ліве поле використовується для задання мінімального значення, а праве — максимального. При обрахунку нової ітерації для кожної клітини обирається випадкове число з цього проміжку та перевіряється, чи має клітина дану кількість сусідів, в цьому разі клітина заповнюється. Коректними значеннями полів є додатні цілі значення, при цьому мінімальне значення має бути не більше максимального. Рівність значень забезпечить повну визначеність зародження клітин. Як і у випадку полю задання розміру автомату, значення автоматично оновлюється при завантаженні конфігурації з файлу та на початку роботи програму.

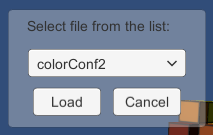
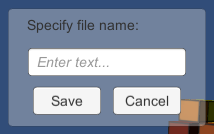
Ще нижче знаходяться поля задання кількості сусідів, нижче якої клітина гине від самотності, та кількості сусідів, вище якої вона гине від перенаселення. Принцип їх задання аналогічний заданню умови зародження клітини: ліве поле задає мінімальне значення, а праве — максимальне. Поля також повинні мати тільки цілі додатні значення, та мінімальне значення повинне не перевищувати максимальне. Значення також оновлюється при завантаженні файлу та задається автоматично при запуску програми.

Під цими полями знаходиться поле вибору кольору. Натискання на це поле відкриває палітру, на якій можна обрати певну точку, що відповідатиме обраному кольору. Аналогічно колір можна змінювати шляхом переміщення повзунків, що відповідають за основні кольори. Повзунки автоматично рухаються при виборі точки на палітрі. При закритті вікна обраний колір автоматично присвоюється виділеній клітині, але лише у режимі вибору кольору. Вибір кольору не матиме ніякого ефекту, якщо не виділена жодна клітина.

Рисунок 3.4 - Ліва панель інтерфейсу

У верхньому правому куті розташовані дві кнопки: Save Configuration та Load Configuration. Натискання цих кнопок відповідно відкриває вікна завантаження та запису поточної конфігурації.

Вікно запису містить поле для введення назви файлу та дві кнопки під ним. Ліва кнопка, Save, записує поточну конфігурацію в файл із назвою у полі введення. Поле повинне містити непустий рядок. Після запису вікно закривається. Права кнопка, Cancel, закриває вікно без збереження. Поки вікно відкрито, інші елементи інтерфейсу неактивні, а можливість взаємодії з моделлю автомату відсутня. Вікно завантаження аналогічне вікну збереження, має ті ж властивості та функціонал кнопок, проте замість поля введення назви файлу містить випадаючий список доступних для завантаження файлів. Для коректного завантаження файлу натиском кнопки Load необхідно обрати один з файлів. Список автоматично оновлюється при кожному відкритті вікна. Зображення вікон запису та завантаження розташовані на рисунку 3.5 (запис з лівої сторони, завантаження — з правої).

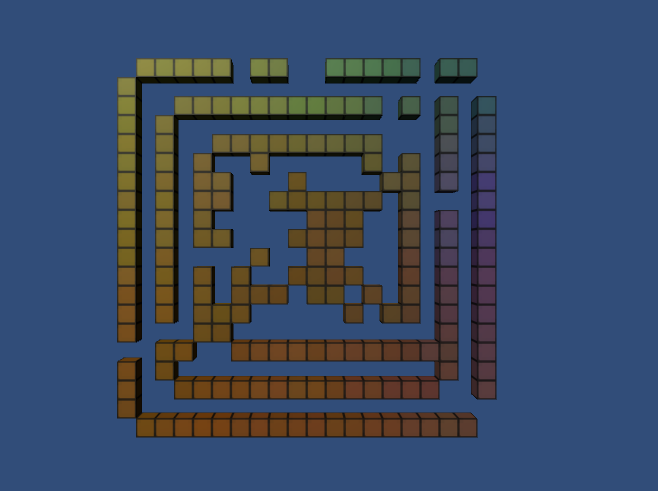
Рисунок 3.5 — Вікна запису (ліворуч) та завантаження (праворуч)

Під кнопками запису та завантаження знаходиться кнопка запуску та призупинення еволюції автомату. Ця кнопка міняє текст зі “Start” на “Stop” при запуску еволюції та назад при його зупинці. Змінювати стан автомату та колір клітин можна тільки якщо автомат зупинений. Також автомат самостійно зупиняється при відкритті вікон запису та завантаження.

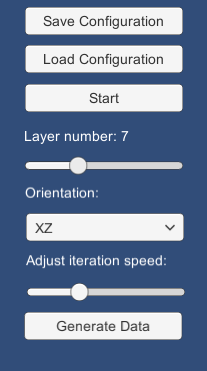
Ще нижче на правій панелі знаходяться список для вибору формату відображення та повзунок вибору поточного шару. Список містить чотири можливих формати відображення.

* Standard – стандартне тривимірне відображення, що допускає повернення моделі автомату, але не допускає її модифікацію.
* XY – відображення одного шару автомату з переднього боку. При цьому та інших форматах відображення окремого шару конкретний шар визначається повзунком. Формат не допускає повернення зображення, але дозволяє змінювати стан клітин та їх колір.
* XZ – відображення одного шару автомату з верхнього боку. Аналогічно попередньому формату.
* YZ – відображення одного шару з лівої сторони.

Приклад відображення одного шару зображено на рисунку 3.6.

Рисунок 3.6 — Приклад відображення окремого шару автомату

Нарешті, найнижчим на правій панелі є повзунок, що регулює швидкість ітерацій. Найбільшою можливою швидкістю є 10 ітерацій в секунду, найнижчою — одна ітерація в 3 секунди. Чим лівіше знаходиться повзунок, тим вище швидкість еволюції. При переході до нового стану клітини, що загинули, поступово стають прозорими, а клітини, що зародились, поступово набувають непрозорість. Час, за який це відбувається, залежить від швидкості переходу до нової ітерації та загалом дорівнює близько 30% від часу між двома ітераціями.

Рисунок 3.7 — Права панель інтерфейсу

Для усіх дій, окрім введення тексту у відповідні поля, необхідне використання лише миші з коліщатком. Виключенням є вихід з програми — він відбувається при натисканні кнопки Escape.

Таким чином, інтерфейс програми надає можливості по зручній роботі із застосунком, надаючи користувачеві спосіб використати будь-які функціональні можливості системи. Розташування інтерфейсу з двох боків екрану дозволяє мати зручний доступ до усіх функціональних можливостей, при цьому не заважаючи роботі безпосередньо з моделлю тривимірного автомату. Сама ж модель дозволяє відобразити клітинний автомат у зрозумілому користувачу форматі та надати зручний засіб змінення його стану.

## Висновки до розділу 3

У даному розділі було розглянуто програмну реалізацію застосунку для дослідження тривимірних автоматів. Зокрема, було описано класову та об’єктну структуру програми, зовнішній вигляд і функціональне призначення елементів інтерфейсу та засіб задання початкового стану автомату. Як можна бачити, застосунок надає увесь розглянутий у попередньому розділі функціонал, надаючи таким чином можливість використовувати його з метою дослідження тривимірних клітинних автоматів.

Крім того, було розглянуто ігровий движок Unity, що був використаний для створення цього застосунку. Окрім загальної інформації, було розглянуто функціональні можливості движку, типові рішення, які він надає для використання у створюваних проектах, а також типову структуру скриптів Unity, за допомогою яких й задається логіка програми.

# Огляд результатів розробки програми

## Сценарії використання програми

### Дослідження еволюції конкретної конфігурації 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 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). Якщо користувач хоче виключити елемент випадковості зі свого автомату, значення мінімуму та максимуму для кожного окремого параметру мають збігатись. Відповідне правило автомату самостійно оновиться при знятті виділення з текстового поля, в якому воно було задане.

Наступним кроком користувачу необхідно задати стан клітин початкової конфігурації. Для цього йому необхідно використовувати один з трьох 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 Save, що розпочне процес запису у файл, по завершенні якого вікно буде закрито. 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.

Нарешті, для запуску еволюції конфігурації користувач може натиснути кнопку “Start” (що змінить напис на ній на “Stop” до зупинки автомату). 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. Останнє також є зручним способом роздивитись структуру поточного стану автомату зсередини, а не тільки процес переходу до нового стану. Для цього рекомендується призупинити еволюцію автомату натисканням кнопки “Stop”. Швидкість еволюції можна регулювати відповідним повзунком 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 Game”, що викличе появу діалогового вікна. У ньому необхідно обрати у списку файл з відповідним даній конфігурації ім’ям та натиснути кнопку “Load”. 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.

### Пошук автомату з необхідними 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.

### Огляд еволюції певної ділянки автомату

Нехай в користувача у процесі дослідження певної конфігурації автомату виникла необхідність детально розглянути зміну стану клітин у певній ділянці. Проте через велику кількість клітин, що виникають у процесі ітерації, та складнощі огляду тривимірної структури автомату користувачу важко зрозуміти, які саме клітини знаходяться у цій ділянці. Тому користувач вирішує відмітити клітини цієї області певним кольором, щоб в процесі роботи програми чіткіше бачити клітини з цієї ділянки. Допустимо, що користувач вже створив заздалегідь файл конфігурації автомату, для якого необхідно проаналізувати певну ділянку (цей процес описаний іншим сценарієм). Для цього користувачу необхідно виконати наступну послідовність дій.

Спочатку необхідно відкрити файл конфігурації. Цей процес вже був описаний у першому розглянутому сценарії використання. Далі необхідно перейти в режим виділення клітини для зміни кольору. Для цього необхідно встановити галочку в полі “Pick Color”. Після цього користувач може вибирати клітину, для якої встановлюється колір. Сам процес вибору кольору відбувається за допомогою палітри. Натискання на символ палітри відкриє її, після чого користувач може обрати потрібний колір шляхом натискання на певну точку палітри чи пересуванням повзунків праворуч від палітри, що відповідають за красний, синій та зелений кольори. При натисканні на точку палітри вона автоматично закривається, а виділеній клітині присвоюється новий колір. Таким чином задається колір усіх необхідних клітин.

Далі користувачу варто зберегти конфігурацію, щоб уникнути необхідності задання кольору клітин ще раз. Процес запису конфігурації в файл був детально описаний у першому сценарії використання.

Після проведення цих операцій користувач може запустити еволюцію автомату. При цьому виділені клітини будуть зображуватися окремим кольором, тому користувачу буде зрозуміліше, які самі клітини знаходяться в тій зоні. При розгляді поточного стану можна використовувати ті ж самі методи, що й в інших сценаріях використання, на кшталт зміни формату відображення, поточного шару, повернення моделі та її наближення чи віддалення.

### Дослідження поведінки автомату із наслідуванням кольору клітин.

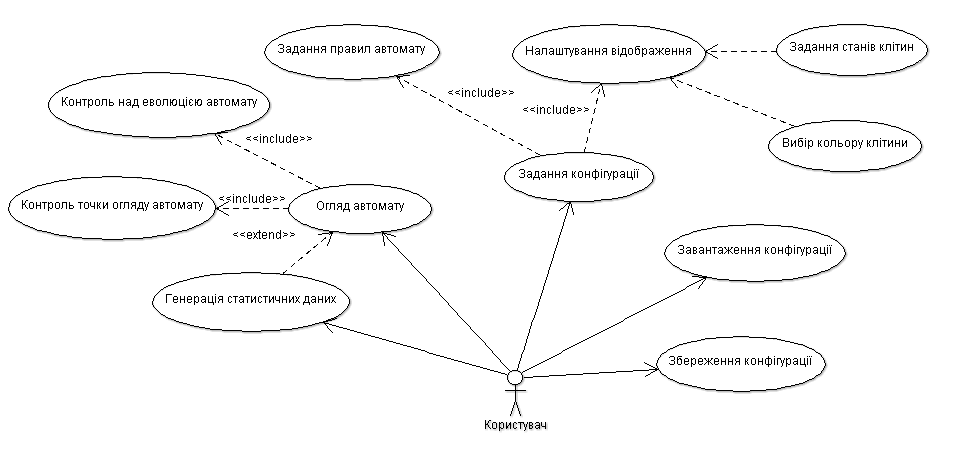
Розглянемо сценарій використання застосунку, за якого користувач бажає проаналізувати еволюцію певної конфігурації автомату, в якій задано колір початкових клітин, з використанням наслідування кольору. Загалом даний варіант схожий на дослідження звичайної конфігурації, однак містить певні додаткові етапи.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

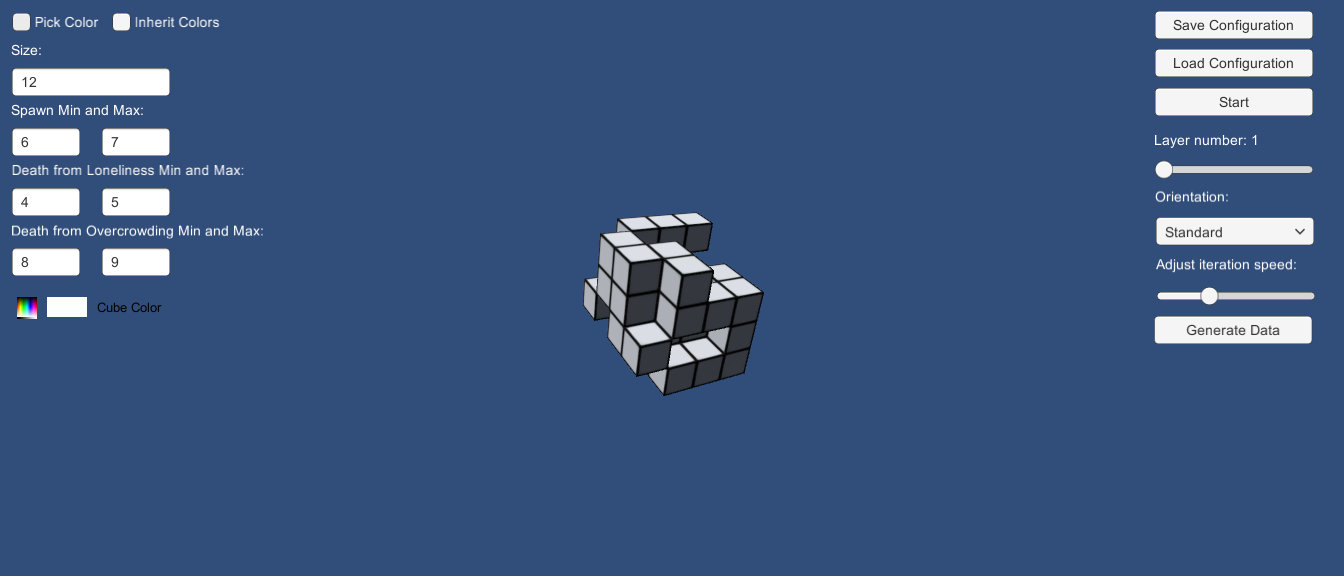
Еволюцію автомату можна призупинити з метою зміни кольорів клітин. Наприклад, користувач вирішив змінити колір клітин в певній області чи додати нову область іншого кольору. Для цього використовуються вже описані раніше засоби. У цьому разі після продовження еволюції результат зміниться, додаючи до автомату 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.

Рисунок 4.1 — UML-діаграма варіантів використання

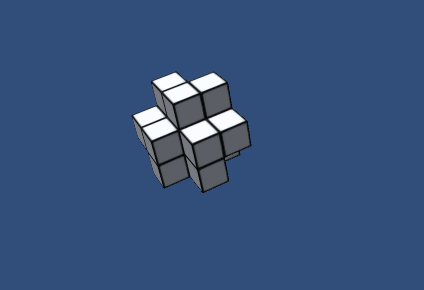
## Результати виконання програми

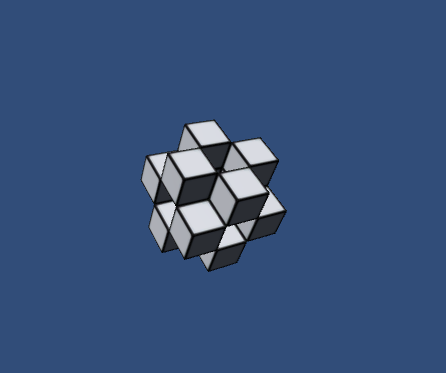
### Приклади еволюції автоматів

Наведемо приклади виконання програми. Спершу розглянемо еволюцію автомату без використання наслідування кольорів. Для цього впевнимося, що галочка в полі “Inherit Colors” не стоїть — це прискорить обрахунок еволюції через відсутність обрахунку середнього кольору клітин. Обраною конфігурацією є випадкове заповнення клітин автомату розміром 12 на 12 на 12 клітин, зародження клітини відбувається при 6 чи 7 сусідах, смерть від самотності — при менш ніж 4-5 клітин, а від перенаселення — при більш ніж 8-9 клітинах. Програма перед запуском еволюції системи виглядає як на рисунку 4.2.

Рисунок 4.2 — Початкова конфігурація

В результаті еволюції за вказаними правилами система перейшла у стабільний стан, зображений на рисунках 4.3-4.4

Рисунок 4.3 — Результат еволюції конфігурації

Рисунок 4.4 — Результат еволюції конфігурації (з іншого кута зору)

Також наведемо приклад еволюції системи з використанням наслідування кольору. Для цього використаємо ті ж самі правила автомату, що й у попередньому прикладі, але із конфігурацією, як на рисунку 4.5.

Рисунок 4.5 — Початкова конфігурація із заданим кольоровим маркуванням

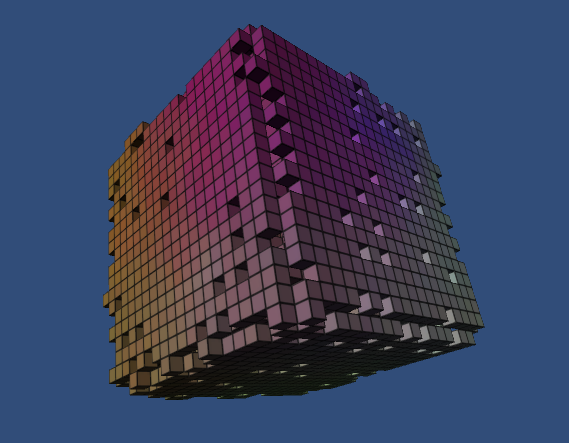
Дана початкова конфігурація в зазначених правилах здатна давати непередбачуваний результат: еволюція може закінчитись спустінням автомату, формуванням стабільних структур певного кольору чи розростанням на все поле автомату. В останньому випадку формуються тривимірні градієнти, при цьому передбачити, який градієнт сформується, до обрахунку еволюції виявляється складною задачею. Наведемо приклад однієї з найвдаліших еволюцій, результатом якої є куб з тривимірними градієнтами різних кольорів.

Рисунок 4.6 — Результат еволюції конфігурації з використанням наслідування кольору (загальний вигляд)

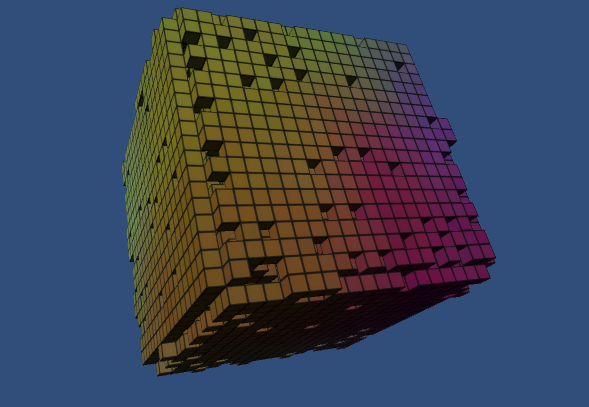
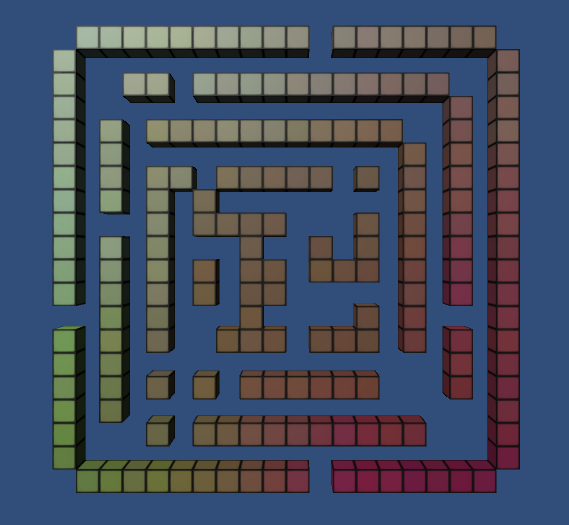
Рисунок 4.7 — Результат еволюції конфігурації з використанням наслідування кольору (загальний вигляд з іншого кута зору)

Рисунок 4.8 - Результат еволюції конфігурації з використанням наслідування кольору (відображення окремого шару автомату)

### Аналіз статистичних даних еволюції автоматів

Розглянемо результат багатократного обрахунку еволюції автоматів із заздалегідь заданими правилами та випадковою початковою конфігурацією. В даному випадку для кожної розмірності проведено 10 експериментів, для яких зазначено час еволюції у кількості ітерацій. Критерієм зупинки роботи автомату є перехід до тривіального стану. Дефіс означає, що час еволюції автомату перевищив 20 тисяч ітерацій, і обрахунок еволюції був перерваний для зменшення часу обрахунку у випадку наддовгих еволюцій.

Таблиця 4.1 — Статистичні дані часу еволюції автомату №1



Вищезазначені дані приведені для автомату, в якому зароджується клітина при 6 чи 7 сусідах, гине від самотності при менш ніж 4-5 сусідах, а від перенаселення — при більш ніж 8-9 сусідах. Як можна бачити з даних, при досягненні розміру автомату значення 11 велика кількість експериментів тривали кількість ітерацій, що перевищує максимальну встановлену для експерименту. Це пов’язано з тим, що даний клітинний автомат є ймовірнісним, тому при певних умовах його еволюція може бути дуже тривалою без переходу до тривіального стану. При великій розмірності автомату його поведінка зазвичай протікає одним з трьох шляхів: повне спустіння автомату, формування стабільної структури або розповсюдження на все поле автомату. Перші два варіанти відповідають часу еволюції в декілька десятків ітерацій. Останній ж зазвичай приводить до довготривалої еволюції.

Час заповнення всього поля може становити тисячі ітерацій, однак не це дозволяє еволюції тривати екстремально довгий час. Після заповнення поля формується структура з вкладених один в одного концентричних кубів, розмежованих шаром пустих клітин в одну клітину. Це пов’язано із тим, що при восьми сусідах, що має кожна клітина у одному шарі таких концентричних кубів, клітина не гине, але й не зароджує клітини у сусідніх шарах. Однак всередині цієї структури відбувається випадкова еволюція, що може тривати десятки тисяч ітерацій та зупиниться лише у разі випадкового виникнення стану, в якому жодна клітина не може зародитись всередині структури.

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. Розглянемо автомати, правила яких є зміненими за одним параметром.

Таблиця 4.2 — Статистичні дані часу еволюції автомату №2



Даний автомат має ті ж правила, що й попередній, однак смерть від самотності настає при менш ніж 3-5 сусідів замість 4-5. Це призводить до меншого часу еволюції автомату. Подібний результат пов’язаний зокрема з тим, що при меншій можливості загибелі клітини від самотності виникає більший шанс переходу внутрішньої структури до тривіального стану. З цього можна зробити висновок, що смерть клітин від самотності є важливим елементом подовження часу еволюції. Розглянемо зворотній варіант — смерть від перенаселення відбуватиметься при 8-10 замість 8-9 клітин:

Таблиця 4.3 — Статистичні дані часу еволюції автомату №3

Дані експериментів показують, що ускладнення смерті клітин від перенаселення зменшує кількість еволюцій, зо завершуються швидким переходом до тривіального стану. Таким чином даний автомат є більш передбачуваним, втім він стабільніше формує структуру з концентричних кубів, чия еволюція триває десятки тисяч ітерацій. На основі цього можна зробити висновок, що на відміну від ускладнення загибелі від самотності, ускладнення загибелі від перенаселення покращує результати автомату, однак він все ще не відноситься до автоматів 4 класу.

Проведені експерименти над ймовірнісними тривимірними клітинними автоматами дозволяють зробити висновок, що серед них міститься доволі велика кількість автоматів 3 класу. Однак для знаходження автомату 4 класу більш перспективними вбачаються детерміновані автомати. Хоча відсутність в них елементу випадковості й зменшує їх здатність до хаотичного розвитку, можливість створення стабільних структур, чия поведінка є абсолютно передбачуваною, є тим критерієм, який може дозволити сформувати детермінований клітинний автомат 4 класу.

## Порівняння із аналогами та можливі варіанти подальшого розвитку

### Порівняння із іншими системами роботи з тривимірними клітинними автоматами

Огляд існуючих програм для роботи із клітинними автоматами виявив, що більшість таких застосунків не надають можливості працювати із тривимірними автоматами. Та ж незначна кількість програм, що підтримує таку можливість або безпосередньо розроблена для цього, мають обмежений функціонал, що не дозволяє використовувати їх для повноцінного дослідження тривимірних автоматів.

Розглянемо один з прикладів програм для роботи з тривимірними клітинними автоматами – JavaScript реалізацію для браузерів з веб-сайту “cubes.io”. Цей програмний застосунок є найяскравішим прикладом засобу моделювання роботи тривимірних автоматів. Він дозволяє зображувати такі автомати за допомогою тривимірних моделей із використанням шейдерів, освітлення, анімацій переходу до нового стану тощо. Однак описані засоби зображення тривимірних автоматів присутні й в даній реалізації подібної системи. “Cubes.io” застосовує дзеркальні шейдери на відміну від звичайного шейдеру із можливістю змінення кольору клітин. Ця можливість є ще однією суттєвою різницею двох систем - “cubes.io” не реалізує можливості змінити колір клітини та, відповідно, використати наслідування кольору при еволюції. Це суттєво обмежує варіації автоматів, які можна промоделювати в цьому застосунку.

Найбільш суттєвою різницею є неможливість змінювати стан клітин на початку еволюції чи у її процесі. “Cubes.io” дозволяє обирати одну із заздалегідь заданих початкових конфігурацій автомату, проте створити нову конфігурацію чи змінити існуючу не виявляється можливим. Це суттєво обмежує можливість використання застосунку для дослідження властивостей автоматів. Можливості контролю над відображенням автомату та процесом еволюції у двох програмах загалом ідентичні: обидві реалізації дозволяють повертати та наближувати модель, зображати окремий шар автомату, змінювати швидкість еволюції.

Ще одним недоліком системи є неможливість зберегти створену конфігурацію. Цінність програми суттєво знижується за відсутності зберегти певні результати для подальшого огляду чи розповсюдження або для багатократного використання однієї конфігурації без необхідності щоразу її задавати.

Головною перевагою “cubes.io” над розробленим застосунком є можливість задання будь-яких правил автомату. Розроблена програма дозволяє задавати варіації автомату гри “Життя” у трьох вимірах із додаванням елементу випадковості. Хоча “cubes.io” і не здатен працювати із ймовірнісними автоматами, він дозволяє реалізовувати майже будь-які правила автомату, що не обмежені лише зародженням та смертю клітин на основі кількості її сусідів. Через це кількість різних правил автоматів, що можливо задати у застосунку, перевищує мільярд.

Крім того, JavaScript-реалізація системи моделювання клітинних автоматів, дозволяє використовувати його прямо в браузері, без необхідності завантаження програми на комп’ютер. Це пришвидшує початок роботи із застосунком та покращує його потенціал для користувачів-любителів. Проте неможливість використання застосунку за відсутності підключення до мережі Інтернет може стати суттєвою перешкодою.

Таким чином, можна зробити висновок, що розроблений застосунок значно краще пристосований до використання для дослідження тривимірних клітинних автоматів, зокрема автоматів із правилами типу гри “Життя” та автоматами, що використовують фарбування клітин та наслідування кольору. Інструментарій, що надає розроблена програма, є зручнішим та ширшим за аналогічний у програмі “cubes.io”.

### Порівняння із системою для роботи з двовимірними клітинними автоматами Golly

Golly є найпопулярнішим застосунком для роботи із двовимірними клітинними автоматами. Хоча він не конкурує безпосередньо із розробленою програмою, адже Golly пристосований для роботи із двовимірними автоматами замість тривимірних, тим не менш, він має ряд функціональних можливостей, чия присутність є доцільною незалежно від розмірності автоматів, з якими працює застосунок. Розглянемо різницю між двома програмами.

Окрім очевидної різниці в розмірності автоматів, з якими програми працюють, найбільш суттєвою різницею є нездатність Golly працювати із кольоровими клітинами. Хоча застосунок і допускає роботу із багатьма станами клітин, проте для реалізації наслідування кольору чітко детермінованої кількості станів недостатньо. Колір клітини має бути її унікальною характеристикою, надання якої клітинам автомату не підтримується системою. Тим не менш, Golly дозволяє працювати з більшістю відомих двовимірних автоматів найрізноманітніших правил, такі як мураха Ленгтона, гра “Життя” та автомат фон Неймана, на відміну від розробленої програми.

Ще однією перевагою системи Golly є здатність працювати із дуже великими автоматами. Завдяки алгоритму Hashlife цей застосунок здатен оброблювати масиви однотипних елементів у грі “Життя” розміром в мільйони та навіть мільярди клітин. Подібну системи неможливо реалізувати для розробленого застосунку, адже алгоритм Hashlife базується на передбачуваності еволюції певних стабільних структур, яка відсутня при хаотичній еволюції ймовірнісного автомату, що швидко створює велике дерево можливих результатів еволюції.

Можливість регулювати швидкість еволюції та зручне масштабування є якостями, що притаманні як системі Golly, так і розробленому застосунку. Проте певний функціонал, необхідний при відображенні тривимірних автоматів, не має місце у цій програмі, адже при роботі із двовимірними автоматами немає необхідності у відображенні окремого шару автомату чи поверненні кута зображення. Можливість зберігати та завантажувати конфігурацію також присутня в обох реалізаціях.

Програма Golly є яскравим представником засобів для роботи із клітинними автоматами. Тим не менш, як можна бачити з порівняння, функціонал розробленого застосунку у багатьох аспектах відповідає цій програмі чи навіть покращує її. Проте певні недоліки, такі як значно менша швидкість обрахунку еволюції та неможливість працювати із двовимірними автоматами, все ж залишають необхідність у використанні Golly для багатьох дослідницьких задач.

### Можливі шляхи подальшого розвитку системи

Порівняльний аналіз розробленої програми із її аналогами показує певні недоліки системи, які можна виправити у наступній ітерації розробки застосунку. Серед функціоналу, що потребує впровадження чи доробки, відмітимо нижчезазначені можливості.

Ширші можливості з задання правил автомату. На даний момент можливості застосунку обмежені лише обрахуванням еволюцій тривимірних аналогів гри “Життя”. У подальшій розробці допускається додавання підтримки інших автоматів, таких як тривимірний варіант мурахи Легнтона чи інші, більш екзотичні правила автомату. Окрім необхідності надання математичної моделі для обрахунку цих автоматів, необхідним також буде створення зручного інтерфейсу для їх задання. Для цього може потребуватись створення повноцінного редактору правил автоматів, що необхідно винести у окреме вікно та надати ширший можливості з їх редагування.

Пришвидшення еволюції автомату. На даний момент при обрахунку еволюції автомату можна бачити доволі суттєве навантаження на пам’ять та процесор комп’ютеру. Вузькими місцями системи є графічне відображення великих автоматів та обробка надвеликих конфігурацій. Перша проблема може бути вирішена шляхом надання спеціального економного режиму простішого відображення, що погіршує якість зображення, проте дозволяє швидше обраховувати великі автомати без необхідності використання потужної машини. Друге питання може бути вирішено шляхом реалізації аналогу алгоритму Hashlife для тривимірних автоматів, проте подібна стратегія можлива лише для детермінованих автоматів. Тим не менш, навіть можливість пришвидшити час виконання лише певної групи автоматів може поліпшити зручність роботи із застосунком. Також допускається можливість паралельного обрахунку еволюції автомату, що має дозволити використання багатопроцесорних та розподілених систем з метою дуже швидкого обрахунку еволюції тривимірних автоматів.

Поліпшення інтерфейсу програми також є можливим шляхом подальшого розвитку програми. Зокрема, на даний момент не існує зручного способу задання кольору великих масивів клітин. Для цього можна додати такі функціональні можливості, як виділення декількох клітин одночасно, виділення цілих регіонів та градієнтне задання кольору клітин. Останній функціонал вбачається особливо перспективним, тому що від суттєво полегшить задання кольорових конфігурацій автомату. Крім того, корисним функціоналом може бути можливість самостійно обирати критерії випадкового заповнення автомату або копіювати певні ділянки поля для створення декількох копій однієї структури в одній конфігурації. Реалізація подібного функціоналу є особливо складним завданням з точки зору інтерфейсу, адже виділення області тривимірного простору важко зробити зручним у програмному застосунку.

Нарешті, доволі корисною може виявитись онлайн-база конфігурацій. Створення єдиної або децентралізованої бази конфігурацій може дозволити ділитись результатами дослідження із іншими користувачами системи, таким чином прискорюючи розповсюдження інформації про цікаві та корисні властивості тривимірних автоматів. Це також дозволить зекономити суттєву кількість часу шляхом знаходження необхідної конфігурації замість самостійного її створення. При цьому буде необхідний не лише функціонал із завантаження конфігурацій на сервер та з цього, але й можливість пошуку конфігурацій за назвою, правилами автомату чи автором.

Таким чином можна зробити висновок, що хоча застосунок і виконує задачу надання зручного механізму роботи із тривимірними автоматами, він може бути покращений та розширений додатковим функціоналом, що буде корисним для користувачів цієї системи.

## Висновки до розділу 4

В останньому розділі було розглянуто практичні результати роботи із створених застосунком. Серед іншого були розглянуті сценарії використання застосунку, що відображають його здатність бути використаним для дослідження властивостей тривимірних клітинних автоматів. Крім того, було показано результат еволюції та статистичні дані певних конфігурацій.

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.

Тим не менш, створений застосунок дозволяє продовжити дослідження у даному напрямку. Для досягнення кращих результатів може виникнути потреба у імплементації додаткового функціоналу у застосунку. Можливі варіанти подальшого його розвитку було розглянуто у даному розділі.

# Висновок

При виконанні даної магістерської дисертації було визначено, що тривимірні автомати є слабко вивченою групою клітинних автоматів. Проте існуючі програмні застосунки для роботи з тривимірними клітинними автоматами не дозволяють детально вивчити їх властивості та знаходити правила, за якими автомат належатиме до 4 класу, тобто його еволюція матиме хаотичний характер із можливістю формування стабільних структур. З метою задоволення потреб щодо інструментарію по дослідженню тривимірних клітинних автоматів було створено програмний застосунок, що моделює роботу тривимірних автоматів із правилами типу гри “Життя”. Ця програма надає зручний графічний інтерфейс для зображення поточного стану автомату та динамічного відображення процесу еволюції автомату, можливість задання стану окремих клітин а також їх колір, що може бути використано для реалізації наслідування кольору при еволюції автомату. Також застосунок надає можливість запису конфігурації у файл та завантаження з файлу, що дозволяє зберегти досліджувані конфігурації для повторного запуску їх еволюції.

Дослідження тривимірних автоматів за допомогою створеного застосунку дозволяє зробити висновок, що тривимірним ймовірнісним клітинним автоматам загалом не властива поява рухомих стабільних структур (“космічних кораблів”) як і осциляторів взагалі, проте вони здатні до створення стабільних структур, що не змінюються з часом (натюрмортів), не дивлячись на випадковий елемент. Крім того, випадковий характер еволюції автоматів надає автомату хаотичну поведінку, достатню для подовження еволюції випадкової конфігурації на десятки тисяч ітерацій.

Використання наслідування кольору клітин при еволюції автомату дозволяє моделювати складніші процеси, ніж при використанні простих тривимірних автоматів. Задання початкової кольорової конфігурації може привести до повного спустіння автомату, виникненню декількох стабільних структур випадкового кольору чи появі кольорових градієнтів із клітин, передбачити характер яких не виявляється можливим. Подальший пошук необхідної конфігурації дозволить формування подібних градієнтів, що можуть використовуватись для моделювання процесу змішування газів чи рідин або інших процесів, що відбуваються у тривимірному просторі, мають випадковий характер та пов’язані із взаємодією різнорідних сутностей. Окрім цього, подібні клітинні автомати можуть використовуватись у комп’ютерній графіці для створення зображень стохастичних фракталів.

Подальший розвиток розробленої програми 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. Daniel Dennett (1995), *Darwin's Dangerous Idea*, Penguin Books, London, ISBN 978-0-14-016734-4, ISBN 0-14-016734-X
2. Wolfram, Stephen (1983). "Statistical Mechanics of Cellular Automata". *Reviews of Modern Physics*. 55 (3): 601–644. Bibcode: 1983RvMP...55..601W. Doi:10.1103 / RevModPhys.55.601.
3. Pickover, Clifford A. (2009). *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics*. Sterling Publishing Company, Inc. p. 406. ISBN 978-1402757969.
4. John von Neumann, "The general and logical theory of automata," in L.A. Jeffress, ed., Cerebral Mechanisms in Behavior – The Hixon Symposium, John Wiley & Sons, New York, 1951, pp. 1–31.
5. Kemeny, John G. (1955). "Man viewed as a machine". *Sci. Amer*. 192: 58–67. doi:10.1038/scientificamerican0455-58.; *Sci. Amer.* 1955; 192:6 (errata).
6. Von Neumann, John; Burks, Arthur W. (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press.
7. Wiener, N.; Rosenblueth, A. (1946). "The mathematical formulation of the problem of conduction of impulses in a network of connected excitable elements, specifically in cardiac muscle". *Arch. Inst. Cardiol. México*. 16: 205.
8. Letichevskii, A. A.; Reshodko, L. V. (1974). "N. Wiener's theory of the activity of excitable media". *Cybernetics*. 8: 856–864. doi:10.1007/bf01068458.
9. Davidenko, J. M.; Pertsov, A. V.; Salomonsz, R.; Baxter, W.; Jalife, J. (1992). "Stationary and drifting spiral waves of excitation in isolated cardiac muscle". *Nature*. 355 (6358): 349–351. Bibcode:1992Natur.355..349D. Doi:10.1038 /355349a0. PMID 1731248.
10. Hedlund, G. A. (1969). "Endomorphisms and automorphisms of the shift dynamical system". *Math. Systems Theory*. 3 (4): 320–3751. doi:10.1007 / BF01691062.
11. Gardner, Martin (1970). "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"". *Scientific American* (223): 120–123.
12. Paul Chapman. Life universal computer. http://www.igblan.free-online.co.uk/igblan/ca/November 2002
13. Mitchell, Melanie (4 October 2002). "Is the Universe a Universal Computer?". *Science*. 298 (5591): 65–68. doi:10.1126/science.1075073.
14. Johnson, George (9 June 2002). "'A New Kind of Science': You Know That Space-Time Thing? Never Mind". *The New York Times*. The New York Times Company. Retrieved 22 January 2013.
15. "The Science of Everything". *The Economist*. 30 May 2002. Retrieved 22 January 2013.
16. Zenil, Hector (2010). "Compression-based investigation of the dynamical properties of cellular automata and other systems". *Complex Systems*.19 (1).
17. G. Cattaneo; E. Formenti; L. Margara (1998). "Topological chaos and CA". In M. Delorme; J. Mazoyer. *Cellular automata: a parallel model*. Springer. p. 239. ISBN 978-0-7923-5493-2.
18. Burton H. Voorhees (1996). *Computational analysis of one-dimensional cellular automata*. World Scientific. p. 8. ISBN 978-981-02-2221-5.
19. Max Garzon (1995). *Models of massive parallelism: analysis of cellular automata and neural networks*. Springer. p. 149. ISBN 978-3-540-56149-1.
20. Li, Wentian; Packard, Norman (1990). "The structure of the elementary cellular automata rule space" (PDF). *Complex Systems*. 4: 281–297. Retrieved 25 January 2013.
21. Nicolis (1974). "Dissipative Structures, Catastrophes, and Pattern Formation: A Bifurcation Analysis" (PDF). *PNAS*. 71 (7): 2748–2751. Retrieved 25 March 2017.
22. Richardson, D. (1972). "Tessellations with local transformations". *J. Computer System Sci*. 6 (5): 373–388. doi:10.1016/S0022-0000(72)80009-6.
23. Margenstern, Maurice (2007). *Cellular Automata in Hyperbolic Spaces – Tome I, Volume 1*. Archives contemporaines. p. 134. ISBN 978-2-84703-033-4.
24. Amoroso, Serafino; Patt, Yale N. (1972). "Decision Procedures for Surjectivity and Injectivity of Parallel Maps for Tessellation Structures". *J. Comput. Syst. Sci*. 6 (5): 448–464. doi:10.1016/s0022-0000(72)80013-8.
25. Sutner, Klaus (1991). "De Bruijn Graphs and Linear Cellular Automata" *Complex Systems*. 5: 19–30.
26. Kari, Jarkko (1990). "Reversibility of 2D cellular automata is undecidable". *Physica D*. 45: 379–385. Bibcode:1990PhyD...45..379K. doi:10.1016/0167-2789(90)90195-U.
27. Kari, Jarkko (1999). "On the circuit depth of structurally reversible cellular automata". *Fundamenta Informaticae*. 38: 93–107.
28. Durand-Lose, Jérôme (2001). "Representing reversible cellular automata with reversible block cellular automata". *Discrete Mathematics and Theoretical Computer Science*. AA: 145–154.
29. lachinski, Andrew (2001). *Cellular automata: a discrete universe*. World Scientific. pp. 44–45. ISBN 978-981-238-183-5.
30. Manneville, Paul (1992). "Collective behaviors in a family of high-dimensional cellular automata". *Physics Letters A*. 163 (4): 279–285. Bibcode:1992PhLA..163..279B. doi:10.1016/0375-9601(92)91013-H.
31. Pivato, M: "RealLife: The continuum limit of Larger than Life cellular automata", *Theoretical Computer Science*, 372 (1), March 2007, pp. 46–68
32. Giles, Jim (2002). "What Kind of Science is This?". *Nature* (417): 216–218.
33. Weinberg, Steven (24 October 2002). "Is the Universe a Computer?". *The New York Review of Books*. Rea S. Hederman. Retrieved 12 October 2012.
34. Wentian Li; Norman Packard; Chris G Langton (1990). "Transition phenomena in cellular automata rule space". *Physica D*. 45 (1–3): 77–94. Bibcode:1990PhyD...45...77L. doi:10.1016/0167-2789(90)90175-O.
35. Coombs, Stephen (15 February 2009), *The Geometry and Pigmentation of Seashells*, pp. 3–4, retrieved 2 September 2012
36. Yves Bouligand (1986). *Disordered Systems and Biological Organization*. pp. 374–375.
37. A. K. Dewdney, The hodgepodge machine makes waves, Scientific American, p. 104, August 1988.
38. Gerhardt, M.; Schuster, H. (1989). "A cellular automaton describing the formation of spatially ordered structures in chemical systems". *Physica D*. 36: 209–221. doi:10.1016/0167-2789(89)90081-x.
39. Muhtaroglu, Ali (August 1996). "4.1 Cellular Automaton Processor (CAP)". *Cellular Automaton Processor Based Systems for Genetic Sequence Comparison/Database Searching*. Cornell University. pp. 62–74.
40. Tomassini, M.; Sipper, M.; Perrenoud, M. (2000). "On the generation of high-quality random numbers by two-dimensional cellular automata". *IEEE Transactions on Computers*. 49 (10): 1146–1151. doi:10.1109/12.888056.
41. Chowdhury, D. Roy; Basu, S.; Gupta, I. Sen; Chaudhuri, P. Pal (June 1994). "Design of CAECC - cellular automata based error correcting code". *IEEE Transactions on Computers*. 43 (6): 759–764. doi:10.1109/12.286310.
42. J. P. Crutchfield, "The Calculi of Emergence: Computation, Dynamics, and Induction", Physica D 75, 11–54, 1994.
43. Minsky, M. "Cellular Vacuum". *International Journal of Theoretical Physics*. 21 (537–551): 1982.
44. K. Zuse, "The Computing Universe", Int. Jour. of Theo. Phy. 21, 589–600, 1982.
45. E. Fredkin, "Digital mechanics: an informational process based on reversible universal cellular automata", Physica D 45, 254–270, 1990.
46. F. Berto, G. Rossi, J. Tagliabue, The Mathematics of the Models of Reference, College Publications, 2010

# Додаток 1. Лістинг коду програми

LifeGame.cs:

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

using System;

using System.Runtime.Serialization;

using System.Reflection;

[Serializable]

public class LifeGame {

public const int RANDOM\_ELEMENT = 5;

public const float RANDOM\_MAGNITUDE = 250f;

public int size;

public int deathTimer;

public int spawnMin;

public int spawnMax;

public int lonelyRemoveMin;

public int lonelyRemoveMax;

public int overcrowdRemoveMin;

public int overcrowdRemoveMax;

public bool[,,] field;

public float[,,,] cellColors;

public LifeGame (int size, int spawnMin, int spawnMax, int lonelyRemoveMin, int lonelyRemoveMax, int overcrowdRemoveMin, int overcrowdRemoveMax) {

this.size = size;

this.spawnMin = spawnMin;

this.spawnMax = spawnMax;

this.lonelyRemoveMin = lonelyRemoveMin;

this.lonelyRemoveMax = lonelyRemoveMax;

this.overcrowdRemoveMin = overcrowdRemoveMin;

this.overcrowdRemoveMax = overcrowdRemoveMax;

Reset();

}

public void Reset() {

field = new bool[size, size, size];

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

for (int k = 0; k < size; k++) {

field[i, j, k] = false;

}

}

}

cellColors = new float[size, size, size, 3];

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

for (int k = 0; k < size; k++) {

cellColors[i, j, k, 0] = 1;

cellColors[i, j, k, 1] = 1;

cellColors[i, j, k, 2] = 1;

}

}

}

}

public void Resize(int newSize) {

bool[,,] newField = new bool[newSize, newSize, newSize];

float[,,,] newColors = new float[newSize, newSize, newSize, 3];

for (int i = 0; i < newSize; i++) {

for (int j = 0; j < newSize; j++) {

for (int k = 0; k < newSize; k++) {

newField[i, j, k] = i < size && j < size && k < size ? field[i, j, k] : false;

newColors[i, j, k, 0] = i < size && j < size && k < size ? cellColors[i, j, k, 0] : 1;

newColors[i, j, k, 1] = i < size && j < size && k < size ? cellColors[i, j, k, 1] : 1;

newColors[i, j, k, 2] = i < size && j < size && k < size ? cellColors[i, j, k, 2] : 1;

}

}

}

field = newField;

cellColors = newColors;

size = newSize;

}

public int GetLifetime() {

int count = 0;

int timer = 0;

while (timer < 5) {

count++;

if (!Iterate(false)) {

timer++;

} else if (timer != 0) {

timer = 0;

}

if (count > 20000) {

return -1;

}

}

return count;

}

public bool Iterate(bool inheritColors) {

bool change = false;

bool[,,] newField = new bool[size, size, size];

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

for (int k = 0; k < size; k++) {

newField[i, j, k] = field[i, j, k];

}

}

}

int neighbors;

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

for (int k = 0; k < size; k++) {

neighbors = GetNeighborsCount(i, j, k);

if (!field[i, j, k] && neighbors == UnityEngine.Random.Range(spawnMin, spawnMax+1)) {

newField[i, j, k] = true;

if (inheritColors) {

var avgColor = GetNeighborsAverageColor(i, j, k);

cellColors[i, j, k, 0] = avgColor.r

+ UnityEngine.Random.Range(-RANDOM\_ELEMENT, RANDOM\_ELEMENT) / RANDOM\_MAGNITUDE;

cellColors[i, j, k, 1] = avgColor.g

+ UnityEngine.Random.Range(-RANDOM\_ELEMENT, RANDOM\_ELEMENT) / RANDOM\_MAGNITUDE;

cellColors[i, j, k, 2] = avgColor.b

+ UnityEngine.Random.Range(-RANDOM\_ELEMENT, RANDOM\_ELEMENT) / RANDOM\_MAGNITUDE;

}

change = true;

}

if (field[i, j, k] && (neighbors < UnityEngine.Random.Range(lonelyRemoveMin, lonelyRemoveMax+1)

|| neighbors > UnityEngine.Random.Range(overcrowdRemoveMin, overcrowdRemoveMax+1))) {

newField[i, j, k] = false;

change = true;

}

}

}

}

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

for (int k = 0; k < size; k++) {

field[i, j, k] = newField[i, j, k];

}

}

}

return change;

}

public int Count() {

int res = 0;

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

for (int k = 0; k < size; k++) {

res += field[i, j, k] ? 1 : 0;

}

}

}

return res;

}

public bool Spawn(int x, int y, int z) {

if (field[x, y, z] == true) {

return false;

}

field[x, y ,z] = true;

return true;

}

public bool Remove(int x, int y, int z) {

if (field[x, y, z] == false) {

return false;

}

field[x, y, z] = false;

return true;

}

private Color GetNeighborsAverageColor(int x, int y, int z) {

var res = new Color(0, 0, 0);

var count = 0;

for (int i = Mathf.Max(x - 1, 0); i <= Mathf.Min(x + 1, size - 1); i++) {

for (int j = Mathf.Max(y - 1, 0); j <= Mathf.Min(y + 1, size - 1); j++) {

for (int k = Mathf.Max(z - 1, 0); k <= Mathf.Min(z + 1, size - 1); k++) {

if (field[i, j, k] && (i != x || j != y || k != z)) {

res.r = (res.r \* count + cellColors[i, j, k, 0]) / (count + 1);

res.g = (res.g \* count + cellColors[i, j, k, 1]) / (count + 1);

res.b = (res.b \* count + cellColors[i, j, k, 2]) / (count + 1);

count++;

}

}

}

}

return res;

}

private int GetNeighborsCount(int x, int y, int z) {

var res = 0;

for (int i = Mathf.Max(x - 1, 0); i <= Mathf.Min(x + 1, size - 1); i++) {

for (int j = Mathf.Max(y - 1, 0); j <= Mathf.Min(y + 1, size - 1); j++) {

for (int k = Mathf.Max(z - 1, 0); k <= Mathf.Min(z + 1, size - 1); k++) {

if (field[i, j, k] && (i != x || j != y || k != z)) {

res++;

}

}

}

}

return res;

}

}

**GameDisplay.cs:**

using UnityEngine;

using System.Collections;

using System.IO;

using System.Runtime.Serialization;

using System.Reflection;

using System;

using System.Runtime.Serialization.Formatters.Binary;

public abstract class GameDisplay : MonoBehaviour {

public Vector2 size;

public Vector2 spawn;

public Vector2 lonelyRemove;

public Vector2 overcrowdRemove;

public float iteratePeriod;

public int spawnFrequency;

public int testAmount;

public string outputFile;

public int Layer {

get {

return layer;

}

set {

layer = value;

if (view != Orientations.XYZ) {

LayerOutput(layer, view);

}

}

}

public Orientations View {

get {

return view;

}

set {

view = value;

if (view == Orientations.XYZ) {

TotalOutput();

} else {

LayerOutput(layer, view);

}

}

}

public bool IsActive {

get {

return isActive;

}

}

public LifeGame LG {

get {

return lg;

}

}

public bool IsInteractable { get; set; }

public bool InheritColors { get; set; }

protected LifeGame lg;

protected float timeToIterate;

protected bool isActive;

protected Vector3 selection;

private int layer;

private Orientations view;

protected virtual void Awake() {

lg = new LifeGame((int)size.y, (int)spawn.x, (int)spawn.y,

(int)lonelyRemove.x, (int)lonelyRemove.y, (int)overcrowdRemove.x, (int)overcrowdRemove.y);

isActive = false;

layer = 0;

view = Orientations.XYZ;

IsInteractable = true;

InheritColors = false;

}

// Update is called once per frame

protected virtual void Update () {

if (Input.GetKeyDown(KeyCode.Escape)) {

Application.Quit();

}

if (IsActive) timeToIterate += Time.deltaTime;

if (timeToIterate >= iteratePeriod) {

timeToIterate -= iteratePeriod;

lg.Iterate(InheritColors);

//Debug.Log(lg.Count());

if (View == Orientations.XYZ) {

TotalOutput();

} else {

LayerOutput(Layer, View);

}

}

}

public virtual void Reset() {

//lg.Reset();

}

public abstract void TotalOutput();

public abstract void LayerOutput(int layerNum, Orientations orient);

public virtual void ToggleActivity() {

isActive = !isActive;

}

public void Randomize(int zone) {

int zoneStart = (lg.size - zone) / 2;

int zoneFinish = zoneStart + zone;

for (int i = zoneStart; i < zoneFinish; i++) {

for (int j = zoneStart; j < zoneFinish; j++) {

for (int k = zoneStart; k < zoneFinish; k++) {

if (UnityEngine.Random.Range(0, 100) < spawnFrequency) {

lg.Spawn(i, j, k);

}

}

}

}

}

public void GenerateData() {

int endSize = lg.size;

if (endSize > 10) {

GenerateDataSingle(endSize);

} else {

for (int i = (int)size.x; i <= endSize; i++) {

GenerateDataSingle(i);

}

}

}

protected void GenerateDataSingle(int i) {

string res;

int lifetime;

lg.size = i;

res = lg.size + "," + lg.spawnMin + "," + lg.spawnMax + ","

+ lg.lonelyRemoveMin + "," + lg.lonelyRemoveMax + ","

+ lg.overcrowdRemoveMin + "," + lg.overcrowdRemoveMax + ",,";

for (int k = 0; k < testAmount; k++) {

lg.Reset();

Randomize((lg.size + 1) / 3);

lifetime = lg.GetLifetime();

res += "," + (lifetime == -1 ? "-" : lifetime.ToString());

}

File.AppendAllText(outputFile, res + "\r\n");

Debug.Log("Data for size " + lg.size + " saved to the file");

}

public void SaveGame(string fileName) {

var stream = Stream.Null;

try {

stream = File.Open(Application.persistentDataPath + "/" + fileName, FileMode.OpenOrCreate);

var formatter = new BinaryFormatter { Binder = new VersionDeserializationBinder() };

formatter.Serialize(stream, lg);

} catch (IOException e) {

Debug.Log("Exception while saving data to file:");

Debug.Log(e);

} finally {

stream.Dispose();

}

}

public void LoadGame(string fileName) {

using (Stream stream = File.Open(Application.persistentDataPath + "/" + fileName, FileMode.Open)) {

var formatter = new BinaryFormatter { Binder = new VersionDeserializationBinder() };

lg = (LifeGame)formatter.Deserialize(stream);

Reset();

if (View == Orientations.XYZ) {

TotalOutput();

} else {

LayerOutput(Layer, View);

}

}

}

public sealed class VersionDeserializationBinder : SerializationBinder {

public override Type BindToType(string assemblyName, string typeName) {

if (string.IsNullOrEmpty(assemblyName) || string.IsNullOrEmpty(typeName)) {

return null;

}

Type typeToDeserialize = null;

assemblyName = Assembly.GetExecutingAssembly().FullName;

// The following line of code returns the type.

typeToDeserialize = Type.GetType(typeName + ", " + assemblyName);

return typeToDeserialize;

}

}

}

public enum Orientations {

XYZ = 0,

XY = 1,

XZ = 2,

YZ = 3

}

**Game3D.cs:**

using UnityEngine;

using UnityEngine.UI;

using System.Collections.Generic;

using System.Linq;

public class Game3D : GameDisplay {

public const string TAG\_CUBE = "Cube";

public GameObject startStop;

public Slider layerSlider;

public Dropdown orientationList;

public GameObject cubePrefab;

public Transform cubeContainer;

public float alphaBase;

public float AlphaSpeed {

get {

return alphaBase / iteratePeriod;

}

}

public float cubeSize {

get {

return cubePrefab.GetComponent<MeshRenderer>().bounds.size.x;

}

}

public int fieldSize {

get {

return lg.size;

}

}

private MeshRenderer[,,] cubes;

private Dictionary<MeshRenderer, float> smooth;

// Use this for initialization

protected override void Awake () {

base.Awake();

Reset();

Randomize((lg.size + 1) / 3);

if (View == Orientations.XYZ) {

TotalOutput();

} else {

LayerOutput(Layer, View);

}

}

protected override void Update() {

base.Update();

smooth = smooth.Where(pair => pair.Key.material.color.a != pair.Value)

.ToDictionary(pair => pair.Key, pair => pair.Value);

foreach (var cube in smooth.Keys) {

var c = cube.material.color;

ChangeAlpha(cube, Mathf.Clamp(c.a + (c.a < smooth[cube] ? AlphaSpeed : -AlphaSpeed), 0, 1));

}

foreach (var pair in smooth.Where(pair => pair.Key.material.color.a == 0)) {

ChangeColor(pair.Key, 1, 1, 1);

}

}

public override void Reset() {

base.Reset();

if (cubes != null) {

for (int i = 0; i < cubes.GetLength(0); i++) {

for (int j = 0; j < cubes.GetLength(1); j++) {

for (int k = 0; k < cubes.GetLength(2); k++) {

Destroy(cubes[i, j, k].gameObject);

}

}

}

}

cubes = new MeshRenderer[fieldSize, fieldSize, fieldSize];

smooth = new Dictionary<MeshRenderer, float>();

var start = transform.position - Vector3.one \* (fieldSize - 0.5f) \* cubeSize / 2;

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

cubes[i, j, k] = ((GameObject)Instantiate(

cubePrefab,

start + new Vector3(i, j, k) \* cubeSize,

Quaternion.identity

)).GetComponent<MeshRenderer>();

cubes[i, j, k].transform.SetParent(cubeContainer);

cubes[i, j, k].GetComponent<ObjectColor>().Game = this;

ChangeAlpha(cubes[i, j, k], 0);

}

}

}

layerSlider.maxValue = fieldSize;

if (View == Orientations.XYZ) {

TotalOutput();

} else {

LayerOutput(Layer, View);

}

}

public override void ToggleActivity() {

base.ToggleActivity();

startStop.transform.GetChild(0).GetComponent<Text>().text = isActive ? "Stop": "Start";

}

public void ToggleInheritColors() {

InheritColors = !InheritColors;

}

public void ChangeLayer() {

if (View != Orientations.XYZ) {

Clear();

}

Layer = (int)layerSlider.value - 1;

applyColliders();

layerOffset();

}

public void SetView() {

Clear();

View = (Orientations)orientationList.value;

applyColliders();

layerOffset();

}

public override void TotalOutput() {

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

if (smooth.ContainsKey(cubes[i, j, k])) {

smooth[cubes[i, j, k]] = lg.field[i, j, k] ? 1 : 0;

} else {

smooth.Add(cubes[i, j, k], lg.field[i, j, k] ? 1 : 0);

}

ChangeColor

(

cubes[i, j, k],

lg.cellColors[i, j, k, 0],

lg.cellColors[i, j, k, 1],

lg.cellColors[i, j, k, 2]

);

}

}

}

}

public override void LayerOutput(int layerNum, Orientations orient) {

for (int i = 0; i < fieldSize; i++) {

for (int k = 0; k < fieldSize; k++) {

int x = layerNum;

int y = layerNum;

int z = layerNum;

switch (orient) {

case Orientations.XY:

x = i;

y = k;

break;

case Orientations.XZ:

x = i;

z = k;

break;

case Orientations.YZ:

y = i;

z = k;

break;

}

if (smooth.ContainsKey(cubes[x, y, z])) {

smooth[cubes[x, y, z]] = lg.field[x, y, z] ? 1 : 0;

} else {

smooth.Add(cubes[x, y, z], lg.field[x, y, z] ? 1 : 0);

}

ChangeColor

(

cubes[x, y, z],

lg.cellColors[x, y, z, 0],

lg.cellColors[x, y, z, 1],

lg.cellColors[x, y, z, 2]

);

}

}

}

public void Clear() {

cubeContainer.localPosition = Vector3.zero;

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

if (smooth.ContainsKey(cubes[i, j, k])) {

smooth[cubes[i, j, k]] = 0;

} else {

smooth.Add(cubes[i, j, k], 0);

}

}

}

}

}

public static void ChangeAlpha(MeshRenderer cube, float alpha) {

Color c = cube.material.color;

c.a = alpha;

cube.material.color = c;

}

public static void ChangeColor(MeshRenderer cube, float r, float g, float b) {

Color c = cube.material.color;

c.r = r;

c.g = g;

c.b = b;

cube.material.color = c;

}

public bool GetStateByCube(Transform cube) {

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

if (cube == cubes[i, j, k].transform) {

return lg.field[i, j, k];

}

}

}

}

return false;

}

public void ChangeStateByCube(Transform cube) {

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

if (cube == cubes[i, j, k].transform) {

lg.field[i, j, k] = !lg.field[i, j, k];

}

}

}

}

}

public void RegisterCubeColor(MeshRenderer cube) {

var c = cube.material.color;

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

if (cube == cubes[i, j, k]) {

lg.cellColors[i, j, k, 0] = c.r;

lg.cellColors[i, j, k, 1] = c.g;

lg.cellColors[i, j, k, 2] = c.b;

}

}

}

}

}

public bool IsCubeInLayer(MeshRenderer cube) {

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

if (cube == cubes[i, j, k]) {

switch (View) {

case Orientations.XYZ:

return true;

case Orientations.XY:

return k == Layer;

case Orientations.XZ:

return j == Layer;

case Orientations.YZ:

return i == Layer;

}

}

}

}

}

return false;

}

private void layerOffset() {

var offset = -(Layer - lg.size / 2) \* cubeSize;

switch (View) {

case Orientations.XY:

cubeContainer.localPosition = new Vector3(0, 0, offset);

break;

case Orientations.XZ:

cubeContainer.localPosition = new Vector3(0, offset, 0);

break;

case Orientations.YZ:

cubeContainer.localPosition = new Vector3(offset, 0, 0);

break;

}

}

private void applyColliders() {

for (int i = 0; i < fieldSize; i++) {

for (int j = 0; j < fieldSize; j++) {

for (int k = 0; k < fieldSize; k++) {

var status = true;

switch (View) {

case Orientations.XY:

status = k == Layer;

break;

case Orientations.XZ:

status = j == Layer;

break;

case Orientations.YZ:

status = i == Layer;

break;

}

cubes[i, j, k].GetComponent<BoxCollider>().enabled = status;

}

}

}

}

}

**CameraMovement.cs:**

using UnityEngine;

using System.Collections;

using UnityEngine.EventSystems;

using System.Collections.Generic;

public class CameraMovement : MonoBehaviour {

public float rotateSpeed;

public float zoomSpeed;

public Transform mainCamera;

public Game3D game;

private float x;

private float y;

private bool isRotate;

private const float MIN\_DIST\_RATE = 1.2f;

private const float MAX\_DIST\_RATE = 5f;

// Use this for initialization

void Start () {

x = 0;

y = 0;

isRotate = false;

}

// Update is called once per frame

void Update() {

var edge = game.cubeSize \* game.fieldSize \* (Mathf.Sqrt(3) / 2);

var dist = Mathf.Clamp

(

mainCamera.localPosition.z + Input.GetAxis("Mouse ScrollWheel") \* zoomSpeed,

-edge \* MAX\_DIST\_RATE,

-edge \* MIN\_DIST\_RATE

);

mainCamera.localPosition = new Vector3(mainCamera.localPosition.x, mainCamera.localPosition.y, dist);

switch (game.View) {

case Orientations.XYZ:

var hit = new RaycastHit();

var eventSystem = EventSystem.current;

var eventDataCurrentPosition = new PointerEventData(eventSystem) {

position = Input.mousePosition

};

var results = new List<RaycastResult>();

eventSystem.RaycastAll(eventDataCurrentPosition, results);

if (Input.GetKeyDown(KeyCode.Mouse0)

&& results.Count == 0

&& Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit)

&& hit.transform.CompareTag(Game3D.TAG\_CUBE)

&& game.IsInteractable) {

isRotate = true;

}

if (Input.GetKeyUp(KeyCode.Mouse0)) {

isRotate = false;

}

if (isRotate) {

x = Input.GetAxis("Mouse X") \* rotateSpeed;

y = -Input.GetAxis("Mouse Y") \* rotateSpeed;

transform.rotation \*= Quaternion.AngleAxis(x \* rotateSpeed, Vector3.up);

transform.rotation \*= Quaternion.AngleAxis(y \* rotateSpeed, Vector3.right);

}

break;

case Orientations.XY:

transform.RotateAround(

transform.position,

Vector3.Cross(transform.forward, Vector3.forward),

Mathf.Clamp(rotateSpeed, 0, Vector3.Angle(transform.forward, Vector3.forward))

);

transform.RotateAround(

transform.position,

Vector3.Cross(transform.up, Vector3.up),

Mathf.Clamp(rotateSpeed, 0, Vector3.Angle(transform.up, Vector3.up))

);

break;

case Orientations.XZ:

transform.RotateAround(

transform.position,

Vector3.Cross(transform.forward, Vector3.up),

Mathf.Clamp(rotateSpeed, 0, Vector3.Angle(transform.forward, Vector3.up))

);

transform.RotateAround(

transform.position,

Vector3.Cross(transform.up, Vector3.left),

Mathf.Clamp(rotateSpeed, 0, Vector3.Angle(transform.up, Vector3.left))

);

break;

case Orientations.YZ:

transform.RotateAround(

transform.position,

Vector3.Cross(transform.forward, Vector3.right),

Mathf.Clamp(rotateSpeed, 0, Vector3.Angle(transform.forward, Vector3.right))

);

transform.RotateAround(

transform.position,

Vector3.Cross(transform.up, Vector3.up),

Mathf.Clamp(rotateSpeed, 0, Vector3.Angle(transform.up, Vector3.up))

);

break;

}

}

}

**CellSelection.cs:**

using UnityEngine;

using System.Collections;

using UnityEngine.EventSystems;

using System.Collections.Generic;

public class CellSelection : MonoBehaviour {

public float blinkPeriod;

public float blinkDuration;

public float blinkMax;

public float blinkMin;

public Game3D game;

public ColorPicker colorPicker;

private float blink;

private float selection;

private float blinkTime;

private float selectionTime;

private MeshRenderer prevMesh;

private bool pickColor;

private MeshRenderer selectedMesh;

// Use this for initialization

void Start () {

blink = 0;

blinkTime = 0;

selection = 0;

selectionTime = 0;

prevMesh = null;

selectedMesh = null;

pickColor = false;

}

// Update is called once per frame

void Update () {

// current cursor position blinking

blinkTime += Time.deltaTime;

if (blinkTime < blinkDuration / 2) {

blink = blinkMin + 2 \* blinkMax \* blinkTime / blinkDuration;

} else if (blinkTime < blinkDuration) {

blink = blinkMin + 2 \* blinkMax \* (blinkDuration - blinkTime) / blinkDuration;

} else if (blinkTime > blinkPeriod) {

blinkTime = 0;

}

// selected cube blinking

selectionTime += Time.deltaTime;

if (selectionTime < blinkDuration / 2) {

selection = blinkMin + 2 \* blinkMax \* selectionTime / blinkDuration;

} else if (selectionTime < blinkDuration) {

selection = blinkMin + 2 \* blinkMax \* (blinkDuration - selectionTime) / blinkDuration;

} else {

selectionTime = 0;

}

// find out whether we hit the cube or not

var hit = new RaycastHit();

var eventSystem = EventSystem.current;

var eventDataCurrentPosition = new PointerEventData(eventSystem) {

position = Input.mousePosition

};

var results = new List<RaycastResult>();

eventSystem.RaycastAll(eventDataCurrentPosition, results);

if (results.Count == 0 // if we didn't hit UI elements

&& Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit) // and we hit an object

&& hit.transform.CompareTag(Game3D.TAG\_CUBE) // and it is a cube

&& game.View != Orientations.XYZ // and we don't use standard view

&& !game.IsActive && game.IsInteractable) { // and field is currently interactible and inactive

var curMesh = hit.collider.GetComponent<MeshRenderer>(); // find the mesh we hit

if (prevMesh != curMesh) { // if we moved to the new mesh

if (prevMesh != null && !isSelectedMesh(prevMesh)) {

// if we left the mesh and it wasn't selected mesh in color pick mode - set alpha to standard value

Game3D.ChangeAlpha(prevMesh, game.GetStateByCube(prevMesh.transform) ? 1 : 0);

}

if (Input.GetKey(KeyCode.Mouse0) && !pickColor) {

// if left mouse button is held and it's not color pick mode - change cube state

// (remember that this happens when we move to the new mesh)

game.ChangeStateByCube(hit.transform);

}

prevMesh = curMesh; // save current mesh for next Update

} else if (Input.GetKeyDown(KeyCode.Mouse0)) { // if left mouse button was pressed

if (pickColor) {

if (game.GetStateByCube(curMesh.transform)) {

if (isMeshInLayer(selectedMesh)) {

// if we selected new mesh and another one was previously selected, set its alpha to standard value

Game3D.ChangeAlpha(selectedMesh, 1);

}

if (selectedMesh == curMesh) {

// if we select already selected mesh - deselect it

colorPicker.receiver = null;

selectedMesh = null;

} else {

// otherwise set new selected mesh

colorPicker.receiver = curMesh.gameObject;

selectedMesh = curMesh;

}

}

} else {

game.ChangeStateByCube(hit.transform); // change cube state (as left mouse button was just pressed)

}

}

if (!isSelectedMesh(curMesh)) {

// control the blinking of current mesh unless it is selected in color pick mode

Game3D.ChangeAlpha(curMesh, game.GetStateByCube(hit.transform) ? 1 - blink : blink);

}

} else if (prevMesh != null && !isSelectedMesh(prevMesh)) {

// if we just moved away from the field - set alpha of last targeted mesh to standard value

// (unless it is selected in color pick mode)

Game3D.ChangeAlpha(prevMesh, game.GetStateByCube(prevMesh.transform) ? 1 : 0);

}

if (!isMeshInLayer(selectedMesh) || !game.GetStateByCube(selectedMesh.transform)) {

// if selected mesh was turned off - deselect it

colorPicker.receiver = null;

selectedMesh = null;

}

if (isMeshInLayer(selectedMesh) && pickColor && game.View != Orientations.XYZ && !game.IsActive) {

// if we're in pick mode, standard view and inactive - control the blinking of selected mesh

Game3D.ChangeAlpha(selectedMesh, 1 - selection);

}

}

public void TogglePickColor() {

pickColor = !pickColor;

// when deselecting pickColor - set selected mesh's alpha to standard value

if (!pickColor && selectedMesh != null) {

Game3D.ChangeAlpha(selectedMesh, 1);

}

}

private bool isSelectedMesh(MeshRenderer mesh) {

return pickColor && mesh == selectedMesh;

}

private bool isMeshInLayer(MeshRenderer mesh) {

if (mesh == null) {

return false;

}

return game.IsCubeInLayer(mesh);

}

}

**UIController.cs:**

using UnityEngine;

using UnityEngine.UI;

using System.Collections.Generic;

public class UIController : MonoBehaviour {

public List<Selectable> ui;

public GameObject saveWindow;

public GameObject loadWindow;

public GameDisplay game;

// Use this for initialization

void Start () {

}

// Update is called once per frame

void Update () {

}

public void ShowSaveWindow() {

saveWindow.SetActive(true);

PauseInterface();

}

public void HideSaveWindow() {

saveWindow.SetActive(false);

ResumeInterface();

}

public void ShowLoadWindow() {

loadWindow.SetActive(true);

PauseInterface();

}

public void HideLoadWindow() {

loadWindow.SetActive(false);

ResumeInterface();

}

private void PauseInterface() {

foreach (var item in ui) {

item.interactable = false;

}

if (game.IsActive) {

game.ToggleActivity();

}

game.IsInteractable = false;

}

private void ResumeInterface() {

foreach (var item in ui) {

item.interactable = true;

}

game.IsInteractable = true;

}

}

**ParameterController.cs:**

using UnityEngine;

using UnityEngine.UI;

using System.Collections;

using System;

public class ParameterController : MonoBehaviour {

public GameDisplay game;

public InputField size;

public InputField spawnMin;

public InputField spawnMax;

public InputField lonelyMin;

public InputField lonelyMax;

public InputField overcrowdMin;

public InputField overcrowdMax;

public Slider speedSlider;

// Use this for initialization

void Start () {

LoadData();

}

// Update is called once per frame

void Update () {

}

public void LoadData() {

size.text = game.LG.size.ToString();

spawnMin.text = game.LG.spawnMin.ToString();

spawnMax.text = game.LG.spawnMax.ToString();

lonelyMin.text = game.LG.lonelyRemoveMin.ToString();

lonelyMax.text = game.LG.lonelyRemoveMax.ToString();

overcrowdMin.text = game.LG.overcrowdRemoveMin.ToString();

overcrowdMax.text = game.LG.overcrowdRemoveMax.ToString();

}

public void SaveSize() {

try {

game.LG.Resize(int.Parse(size.text));

game.Reset();

} catch(Exception e) {

}

}

public void SaveSpawnMin() {

try {

game.LG.spawnMin = int.Parse(spawnMin.text);

} catch (Exception e) {

}

}

public void SaveSpawnMax() {

try {

game.LG.spawnMax = int.Parse(spawnMax.text);

} catch (Exception e) {

}

}

public void SaveLonelyMin() {

try {

game.LG.lonelyRemoveMin = int.Parse(lonelyMin.text);

} catch (Exception e) {

}

}

public void SaveLonelyMax() {

try {

game.LG.lonelyRemoveMax = int.Parse(lonelyMax.text);

} catch (Exception e) {

}

}

public void SaveOvercrowdMin() {

try {

game.LG.overcrowdRemoveMin = int.Parse(overcrowdMin.text);

} catch (Exception e) {

}

}

public void SaveOvercrowdMax() {

try {

game.LG.overcrowdRemoveMax = int.Parse(overcrowdMax.text);

} catch (Exception e) {

}

}

public void UpdateSpeed() {

game.iteratePeriod = speedSlider.value;

}

}