Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістр
(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ СИМУЛЯЦІЇ "ГРА ЖИТТЯ" КОНВЕЯ

RESEARCH AND SOFTWARE IMPLEMENTATION OF THE CONWAY'S GAME OF LIFE SIMULATION

Виконав(ла): студент(ка) 2 курсу, групи КНТ-214м
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Системи штучного інтелекту

КОЛОКОЛ Я.О.
(ПРІЗВИЩЕ та ініціали)

Керівник Федорончак Т.В.
(ПРІЗВИЩЕ та ініціали)

Рецензент Бабенко Н.В.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Системи штучного інтелекту
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____ Сергій СУББОТІН
"____" _____2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

КОЛОКОЛА Ярослава Олексійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація симуляції "Гра Життя" Конвея. Research and Software Implementation of the Conway's Game of Life Simulation

керівник проєкту (роботи) к.т.н., доцент, ФЕДОРОНЧАК Тетяна Василівна
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від " 30 " вересня 2025 року № 447
2. Строк подання студентом проєкту (роботи) 8 грудня 2025 року
3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Матеріали і методи. 3. Розробка симуляції. 4. Експлуатація симуляції. 5. Дослідження патернів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів плакатів) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ФЕДОРОНЧАК Т.В., доцент		
Нормоконтроль	БЄЛОВА А.В., асистент		

7. Дата видачі завдання “30” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Матеріали і методи.	4-5 тижні	Розділ 2
4	Розробка симуляції.	6-7 тижні	Розділ 3
5	Експлуатація симуляції.	8 тиждень	Розділ 4
6	Дослідження патернів.	9 тиждень	Розділ 5
7	Оформлення пояснювальної записки та документів до неї.	10 тиждень	Додатки
8	Нормоконтроль та рецензування.	11 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка) _____ Ярослав КОЛОКОЛ
(підпис) (І’мя ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Тетяна ФЕДОРОНЧАК
(підпис) (І’мя ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра: 140 с., 6 табл., 47 рис., 2 дод., 24 джерел.

2D СИМУЛЯЦІЯ, ІГРОВИЙ РУШІЙ, СЕРЕДОВИЩЕ РОЗРОБКИ, API, C++, GODOT, SANDBOX, UI.

Об’єкт дослідження – процес моделювання динамічних систем за допомогою клітинних автоматів, що базуються на простих правилах взаємодії між елементами системи.

Предмет дослідження – методи та інструменти програмування, які використовуються для створення симуляцій.

Мета роботи – розробка комп’ютерної симуляції “Гра Життя” Конвея, що дозволяє дослідити закономірності розвитку систем із простими правилами взаємодії та продемонструвати принципи самоорганізації у дискретному середовищі.

Матеріали, методи та технічні засоби: об’єктно-орієнтоване програмування, мова програмування C++, інтегроване середовище розробки Visual Studio, meta-build система CMake, ігровий рушій Godot, персональний комп’ютер з процесором AMD Ryzen 7 5700U під управлінням операційної системи Microsoft Windows 10, ОЗУ 8ГБ.

Результати. Розроблено комп’ютерну симуляцію “Гра Життя” Конвея, яка дозволяє розглядати процес створення та знищення життя, створювати свої патерни та аналізувати їх поведінку.

Висновки. Було проведено дослідження методів програмної розробки комп’ютерної симуляції з використанням ігрового рушія Godot та мови програмування C++.

Галузь використання – галузь розваг, освіти та досліджень у сфері комп’ютерних ігор та симуляцій.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 140 pages, 6 tables, 47 figures, 2 appendixes, 24 sources.

2D SIMULATION, GAME ENGINE, DEVELOPMENT ENVIRONMENT, API, C++, GODOT, SANDBOX, UI.

The object of research is the process of modeling dynamic systems using cellular automata, which are based on simple rules of interaction between system elements.

The subject of research is programming methods and tools used to create simulations.

The purpose of the work is to develop a computer simulation of Conway's "Game of Life", which allows to investigate the patterns of development of systems with simple rules of interaction and demonstrate the principles of self-organization in a discrete environment.

Materials, methods and tools: object-oriented programming, C++ programming language, Visual Studio integrated development environment, CMake meta-build system, Godot game engine, personal computer with AMD Ryzen 7 5700U processor running Microsoft Windows 10 operating system, 8GB RAM.

Results. A computer simulation of Conway's "Game of Life" has been developed, which allows us to consider the process of creation and destruction of life, create our own patterns, and analyze their behavior.

Conclusions: A study was conducted on computer simulation software development methods using the Godot game engine and the C++ programming language.

The field of use is the field of entertainment, education, and research in the field of computer games and simulations.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
FPS	– Frames Per Second;
IDE	– Integrated Development Environment;
RAM	– Random Access Memory;
UI	– User Interface;
ЕКА	– Елементарний клітинний автомат;
КА	– Клітинний автомат;
ПЗ	– Програмне забезпечення.

ВСТУП

Сучасні інформаційні технології відкривають широкі можливості для моделювання складних систем і процесів за допомогою комп’ютерних симуляцій. Такі симуляції дозволяють відтворювати, досліджувати та аналізувати явища, які складно або неможливо спостерігати безпосередньо у реальному світі. Вони використовуються у найрізноманітніших галузях — від фізики, біології та економіки до соціології та штучного інтелекту.

Створення комп’ютерних симуляцій дає змогу виявляти закономірності у поведінці систем, прогнозувати їх розвиток, а також перевіряти гіпотези без необхідності проведення дорогих або тривалих експериментів. Зокрема, важливе місце серед таких моделей займають клітинні автомати — математичні структури, що дозволяють досліджувати еволюцію систем із великою кількістю простих елементів, які взаємодіють між собою за визначеними правилами.

Актуальність теми полягає у тому, що створення програмних реалізацій подібних моделей має як наукову, так і практичну цінність. З одного боку, це дає можливість вивчати фундаментальні принципи розвитку систем, а з іншого — слугує навчальним інструментом для демонстрації дії простих алгоритмів, що призводять до непередбачуваних і цікавих результатів. Таким чином, дослідження, присвячене реалізації та аналізу клітинних автоматів, зокрема моделей типу “Гра Життя”, є своєчасним і важливим у контексті розвитку сучасних підходів до комп’ютерного моделювання та штучного інтелекту.

Для досягнення поставленої мети у роботі визначаються наступні завдання:

- проаналізувати теоретичні основи клітинних автоматів та їх застосування у моделюванні складних систем;
- ознайомитися з прикладами існуючих моделей та підходів до побудови симуляцій;
- розробити алгоритм функціонування клітинного автомата відповідно до обраних правил;
- реалізувати ПЗ із можливістю візуалізації процесу еволюції патернів;

- провести тестування симуляції та дослідити характерні закономірності поведінки патернів;

- проаналізувати отримані результати та визначити потенційні напрями подальшого вдосконалення симуляції.

Методика дослідження буде включати аналіз літературних джерел, розробку програмного забезпечення за допомогою сучасного ігрового рушія, його тестування та оцінку ефективності.

У даній роботі буде розкрито принципи побудови клітинних автоматів, особливості їх поведінки та практичні аспекти реалізації симуляційних систем. Також буде продемонстровано, як із простих локальних правил можуть виникати складні структури та емерджентні властивості систем. Отримані результати можуть бути використані для навчальних, наукових і демонстраційних цілей, а також як основа для подальших досліджень у сфері моделювання динамічних процесів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Перший розділ буде присвячено детальному аналізу клітинних автоматів, їх видів та призначення. Буде розглянуто основні причини реалізації саме “Гри Життя” Конвея. Також будуть поставлені основні завдання.

1.1 Основні поняття і визначення

Клітинний автомат — це дискретна модель обчислень, що вивчається в теорії автоматів. Клітинні автомати також називають клітинними просторами, теселяційними автоматами, однорідними структурами, клітинними структурами, теселяційними структурами та ітераційними масивами. Клітинні автомати знайшли застосування в різних галузях, включаючи фізику, теоретичну біологію та моделювання мікроструктур [1].

Клітинний автомат складається з регулярної сітки комірок, кожна з яких знаходиться в одному з кінцевої кількості станів, таких як увімкнено та вимкнено (на відміну від зв’язаної решітки відображення). Сітка може мати будь-яку кінцеву кількість вимірів. Для кожної комірки визначається набір комірок, який називається її околицею, відносно заданої комірки. Початковий стан (час t = 0) вибирається шляхом призначення стану кожній комірці. Нове покоління створюється (просуваючи t на 1) відповідно до деякого фіксованого правила (зазвичай, математичної функції), яке визначає новий стан кожної комірки з точки зору поточного стану комірки та станів комірок у її околиці. Як правило, правило оновлення стану комірок однакове для кожної комірки та не змінюється з часом, і застосовується до всієї сітки одночасно, хоча відомі винятки, такі як стохастичний клітинний автомат та асинхронний клітинний автомат [1].

Основні класифікації клітинних автоматів пронумеровані від однієї до чотирьох. Це, по порядку, автомати, в яких шаблони зазвичай стабілізуються до однорідності, автомати, в яких шаблони еволюціонують у переважно стабільні або коливальні структури, автомати, в яких шаблони еволюціонують, здавалося б,

хаотично, та автомати, в яких шаблони стають надзвичайно складними та можуть тривати протягом тривалого часу зі стабільними локальними структурами. Вважається, що цей останній клас є обчислювально універсальним або здатним імітувати машину Тюрінга. Спеціальні типи клітинних автоматів є оборотними, де лише одна конфігурація веде безпосередньо до наступної, та тоталістичними, в яких майбутнє значення окремих комірок залежить лише від загального значення групи сусідніх комірок. Клітинні автомати можуть імітувати різноманітні реальні системи, включаючи біологічні та хімічні [1].

Один із способів моделювання двовимірного клітинного автомата — це використання нескінченного аркуша міліметрового паперу разом із набором правил, яких повинні дотримуватися клітинки. Кожен квадрат називається “клітиною”, і кожна клітинка має два можливі стани: чорний або білий. Околиця клітинки — це найближчі, зазвичай суміжні, клітинки. Два найпоширеніші типи околиць — це околиця фон Неймана та околиця Мура (рис. 1.1) [1].

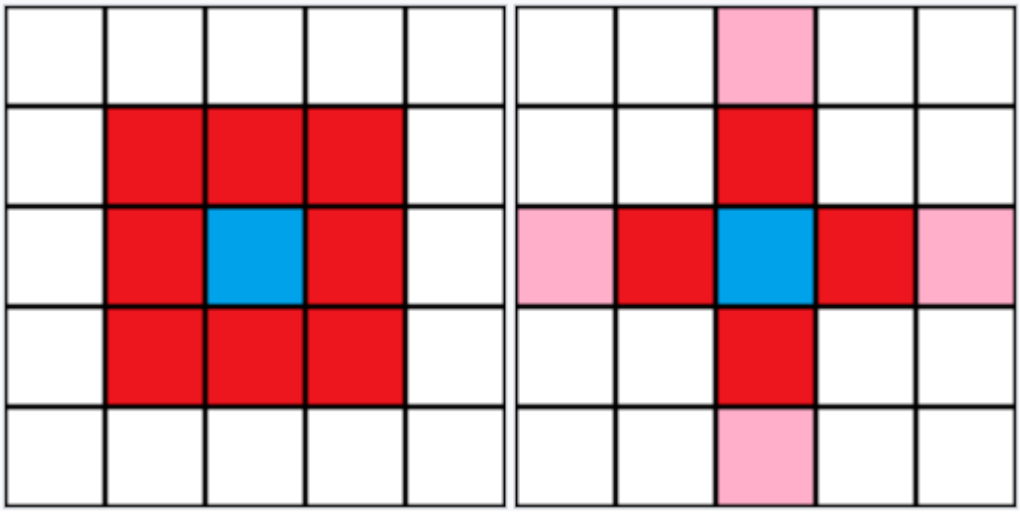

Рисунок 1.1 – Околиця фон Неймана(з права) та околиця Мура(зліва) [1]

Зазвичай вважається, що кожна клітинка у всесвіті починається в одному й тому ж стані, за винятком скінченної кількості клітинок в інших станах. Призначення значень станів називається конфігурацією. У більш загальному випадку іноді вважається, що всесвіт починається вкритим періодичним патерном,

і лише скінченна кількість клітинок порушує цей патерн. Останнє припущення є поширеним в одновимірних клітинних автоматах [1].

Клітинні автомати часто моделюються на скінченній сітці, а не на нескінченній. У двох вимірах всесвіт був би прямокутником, а не нескінченною площиною. Очевидна проблема зі скінченними сітками полягає в тому, як обробляти комірки на ребрах. Те, як вони обробляються, впливатиме на значення всіх комірок у сітці. Один з можливих методів — дозволити значенням у цих комірках залишатися постійними. Інший метод — по-різному визначати околиці для цих комірок. Можна сказати, що вони мають менше сусідів, але тоді також доведеться визначити нові правила для комірок, розташованих на ребрах. Ці комірки зазвичай обробляються періодичними граничними умовами, що призводить до тороїдального розташування: коли одна з них виходить зверху, інша потрапляє у відповідну позицію знизу, а коли одна зливається зліва, інша потрапляє праворуч. Це по суті імітує нескінченне періодичне мозаїчне розкладання, і в галузі диференціальних рівнянь з частинними похідними це іноді називають періодичними граничними умовами. Це можна уявити як склеювання лівого та правого країв прямокутника для утворення трубки, а потім склеювання верхнього та нижнього краю трубки для утворення тора (форми пончика). Всесвіти інших вимірів обробляються аналогічно. Це вирішує граничні задачі з околицями, але ще однією перевагою є те, що це легко програмується за допомогою модульних арифметичних функцій [1].

1.2 Огляд існуючих клітинних автоматів

Даний підрозділ розглядає найпопулярніші клітинні автомати, які використовуються у світі. За структурою простору клітинні автомати поділяються на:

- одновимірні (1D);
- двовимірні (2D);
- багатовимірні (3D та вище).

Окрім того, за типом оновлення клітин автомати можуть бути синхронними (всі клітини оновлюються одночасно) або асинхронними (оновлення відбувається послідовно чи випадково). За природою переходів вони поділяються на детерміновані та стохастичні.

1.2.1 Елементарний клітинний автомат

Елементарний клітинний автомат — це одновимірний клітинний автомат, де існує два можливих стани, а правило визначення стану комірки в наступному поколінні залежить лише від поточного стану комірки та двох її безпосередніх сусідів. Існує елементарний клітинний автомат, який здатний до універсальних обчислень і є однією з найпростіших можливих моделей обчислень [2].

Правила, що визначають зміни стану на кожному кроці часу, стосуються положення самої комірки та її безпосередніх сусідів. Отже, якщо ми знаходимося на кроці часу t і хочемо знати новий стан комірки в позиції n у момент часу t + 1, нам потрібно подивитися на стан комірок у позиціях n - 1, n та n + 1 у момент часу t. Оскільки для кожної комірки існує лише два можливі стани, існує лише вісім можливих способів конфігурації трійки n - 1, n та n + 1. Якщо ми визначимо, що може статися в кожній з цих восьми конфігурацій, ми можемо вичерпно визначити, як поводитиметься клітинний автомат. Стани часто відображаються візуально за допомогою кольорів, наприклад, білий для одного стану та чорний для іншого. Системи правил можна легко відобразити, показавши ці вісім трійок та стан, до якого вони призводять. Наприклад, у наступній системі правил на рисунку 1.2 ми бачимо з правила, що якщо n - 1, n та n + 1 чорні в момент t, то n буде білим в момент t + 1. Друге зліва правило говорить, що якщо n - 1 та n чорні, але n + 1 біле, то в момент t + 1 n буде чорним і так далі [3].

Рисунок 1.2 – Правило генерації [3]

Якщо ми почнемо з однієї чорної клітинки в центрі рядка білих клітинок та ітеративно застосуємо цю систему правил приблизно протягом 15 кроків, ми отримаємо цікавий результат (рис. 1.3) [3].

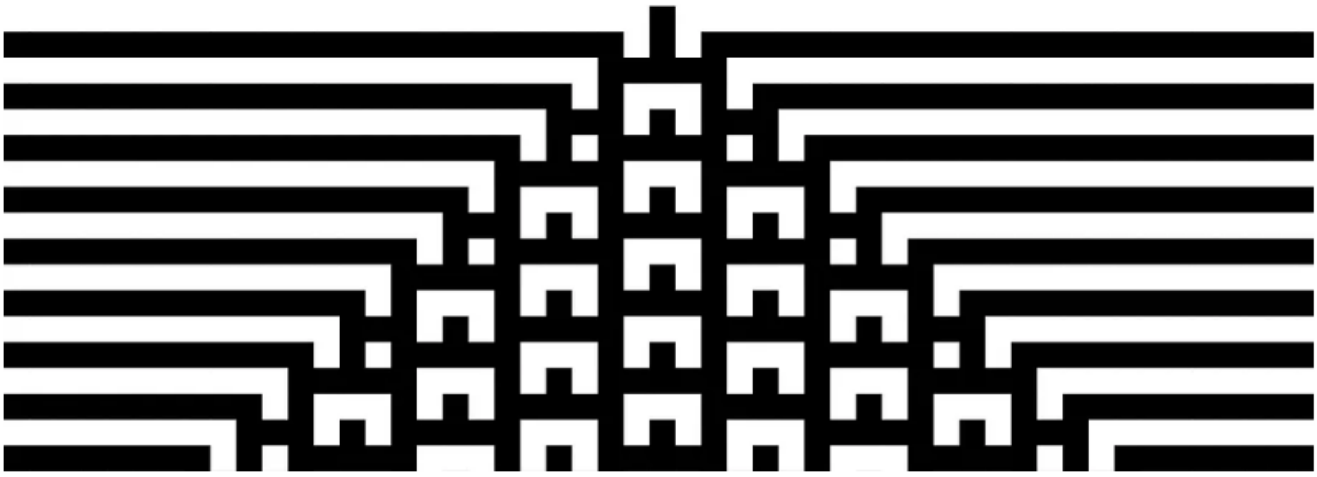

Рисунок 1.3 – Результат симуляції [3]

Існує два основні підходи, які часто використовуються під час соніфікації ЕКА: один полягає у генерації параметрів, які можна використовувати для керування, наприклад, синтезатором. Інший підхід полягає у відображенні станів комірок у музичні події (MIDI-нота, семпл ударних тощо) [3].

Існує багато способів відображення цих подій. Наприклад, ми можемо взяти сітку з восьми клітинок і відтворювати їх послідовно зліва направо, де чорна клітинка означає подію (наприклад, удар по барабану), а біла клітинка – паузу. Ця сітка з восьми клітинок генеруватиме один такт ритму, з якого ми можемо отримати другий такт і так далі. Цей підхід цікавіший, якщо у нас є КА з багатьма станами (тобто не ЕКА), де ми можемо відобразити кожен стан на різну висоту тону [3].

Основні переваги симуляції:

- простота реалізації (ЕКА має дуже прості правила, тому його легко запрограмувати навіть початківцю);

- наглядність моделі (результати можна легко візуалізувати у вигляді послідовності рядків, що дає змогу інтуїтивно розуміти поведінку системи);

- дослідження складних явищ із простих правил (навіть прості правила можуть породжувати складні, хаотичні або навіть обчислювально універсальні структури);

- мала обчислювальна складність (потрібно небагато пам’яті та обчислювальних ресурсів, симуляції можуть працювати швидко навіть на слабкому обладнанні);

- використання в моделюванні (ЕКА застосовують для вивчення самоорганізації, росту структур, поширення інформації, фізичних процесів, тощо).

Основні недоліки симуляції:

- обмежена реалістичність (через простоту правил і дискретність простору та часу такі автомати не завжди точно описують реальні фізичні або біологічні процеси.);

- чутливість до початкових умов (невелика зміна стартового стану може кардинально змінити результат, що ускладнює прогнозування.);

- відсутність аналітичних рішень (поведінку деяких правил важко або неможливо описати математично — потрібен лише експеримент);

- одновимірність (ЕКА є 1D моделлю, тому для складніших систем потрібні інші моделі);

- межові ефекти (поведінка на краях решітки може впливати на результат і спотворювати закономірності).

1.2.2 Мураха Ленгтона

Мураха Ленгтона – це клітинний автомат, створений Крісом Ленгтоном у 1986 році. Він досліджує ідею життя, тобто штучного життя або життя, яким воно “може бути”. Це як уявний експеримент, що демонструє, що складна поведінка у всесвіті не завжди є результатом складних систем. Іноді неймовірно прості системи, правила чи поведінка, масштабовані, можуть створювати захопливо складну емерджентну поведінку [4].

Оригінальна модель Ленгтона працює приблизно так: ми починаємо з сітки комірок та агента або курсора, якого називають “мурахою”. Кожна клітина може мати один із двох можливих станів: білий та чорний. Мураха повернута до одного з чотирьох напрямків світу. Модель працює поетапно (рис. 1.4) [4].

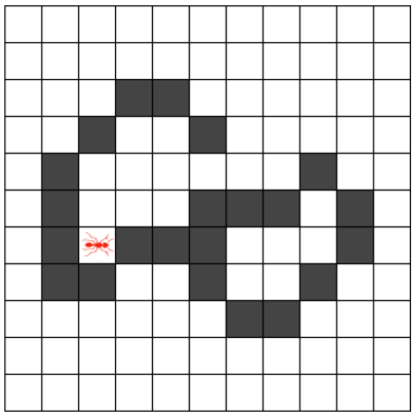

Рисунок 1.4 – Мураха [4]

Ось що відбувається на кожному кроці:

- якщо клітинка, на якій знаходиться мураха, біла, мураха робить чверть оберту праворуч; якщо клітинка чорна, мураха робить чверть оберту ліворуч;

- стан клітинки змінюється (якщо клітинка біла, вона стає чорною, а якщо чорна, то стає білою);

- мураха рухається вперед у будь-якому напрямку, в якому вона звернена до сусідньої клітини [4].

Під час повторних ітерацій правила, мураха проходить три етапи:

- спочатку він поводиться дещо систематично, здійснюючи зміни, здавалося б, симетрично (це триває кілька сотень кроків);

- потім він “ламається”, якщо можна так сказати (він поводиться хаотично, змінюючи стан комірок з одного стану в інший псевдовипадковим чином протягом приблизно наступних 10000 ітерацій);

- це створює умову для періодичного циклу (104 кроки), який самовідтворюється та триватиме до нескінченності [4].

Основні переваги симуляції:

- простота правил (мураха Ленгтона керується дуже простими правилами, але породжує надзвичайно складні та непередбачувані структури.);

- наглядність і легкість візуалізації (симуляція легко відображається у вигляді двовимірної решітки, тому її просто візуалізувати й спостерігати розвиток системи у часі);

- демонстрація принципів самоорганізації (після хаотичної фази мураха створює стабільну “дорогу” (highway), що добре ілюструє, як порядок може виникати з хаосу);

- низькі обчислювальні вимоги (для запуску симуляції не потрібно потужного обладнання);

- освітня цінність (використовується для навчання понять клітинних автоматів, системної динаміки, складності та штучного інтелекту).

Основні недоліки симуляції:

- обмежене застосування в реальних моделях (попри цікаву поведінку, мураха Ленгтона має здебільшого демонстраційний характер і не описує реальні фізичні чи біологічні процеси);

- чутливість до початкових умов (невелика зміна початкового стану може суттєво змінити результат симуляції);

- відсутність аналітичного опису (поведінку мурахи важко описати математично, тож передбачити кінцевий результат можна лише через моделювання);

- повільний розвиток складних структур (щоб мураха почала створювати впорядковану “дорогу”, потрібно багато кроків симуляції);

- межові ефекти (якщо поле має обмежені розміри, мураха може “впиратися” в межі, що порушує природний розвиток системи).

1.2.3 “Гра Життя” Конвея

“Гра Життя” Конвея — це клітинний автомат, розроблений британським математиком Джоном Хортоном Конвеєм у 1970 році. Це гра з нульовим гравцем,

що означає, що її еволюція визначається початковим станом і не потребує подальших дій. Гравець взаємодіє з клітинним автоматом, створюючи початкову конфігурацію та спостерігаючи за її еволюцією. Вона є повною за Тюрінгом і може імітувати універсальний конструктор (рис. 1.5) [5].

Рисунок 1.5 – “Гра Життя” Конвея [5]

Джон Хортон Конвей створив “Гру Життя”, бо хотів дізнатися, чи зможе він створити уявного робота з клітин, який зміг би збільшуватися в розмірах. Він поєднав безліч математичних ідей, щоб створити правила гри. Це одна з перших “симуляційних ігор”, яка відображає речі, що відбуваються в реальному житті. Вона важлива, тому що при використанні в складній математиці, вона може розглядати багато речей, таких як фізика, біологія, економіка та філософія [6].

Всесвіт гри — це нескінченна двовимірна ортогональна сітка з квадратних комірок, кожна з яких знаходиться в одному з двох можливих станів: живій або мертвій (або заселеній та незаселеній відповідно). Кожна клітина взаємодіє зі своїми вісьмома сусідами, які є клітинами, що розташовані горизонтально, вертикально або діагонально [5].

На кожному кроці часу відбуваються такі переходи:

- будь-яка жива клітина з менш ніж двома живими сусідами гине, ніби через недонаселення;

- будь-яка жива клітина з двома або трьома живими сусідами живе до наступного покоління;

- будь-яка жива клітина з більш ніж трьома живими сусідами гине, ніби від перенаселення;

- будь-яка мертва клітина з рівно трьома живими сусідами стає живою клітиною, ніби шляхом розмноження [5].

Початковий шаблон становить зародок системи. Перше покоління створюється шляхом одночасного застосування вищезазначених правил до кожної клітини в зародку, живої чи мертвої; народження та смерть відбуваються одночасно, а дискретний момент, у який це відбувається, іноді називають тіком. Кожне покоління є чистою функцією попереднього. Правила продовжують застосовуватися багаторазово для створення наступних поколінь (рис. 1.6) [5].

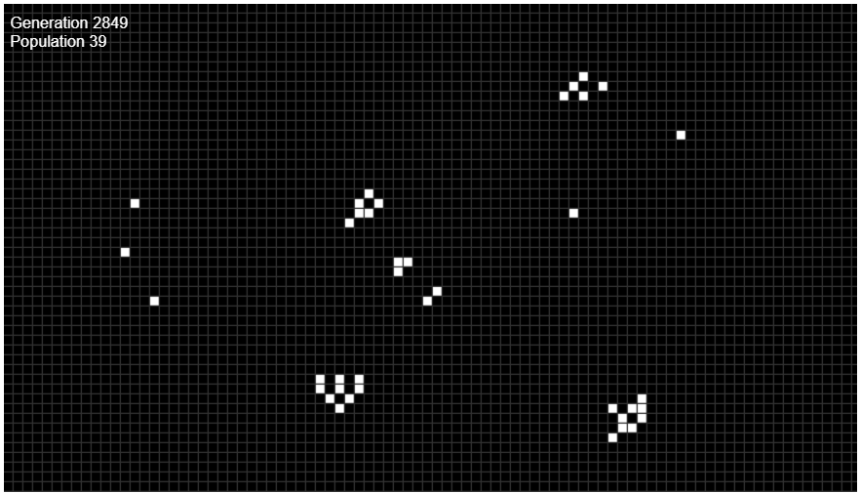

Рисунок 1.6 – Створення поколінь [5]

Вчені відкрили еволюцію фігур у “Грі Життя” Конвея завдяки систематичному дослідженню її правил. Починаючи з початку 1970-х років, творіння Джона Конвея стало віртуальною лабораторією, де дослідники експериментували, щоб розкрити захопливі та непередбачувані закономірності, що виникають із, здавалося б, простих взаємодій клітин [6].

Однак цього було недостатньо для вчених. Їхньою метою залишалося знайти ще більше різних закономірностей. Вони виявили закономірності, які можуть

створювати нові живі клітини. На рисунку 1.7 можна побачити 2 фігури, що рухаються у 2 напрямках, і коли вони стикаються одна з одною, вони утворюють планери, що рухаються лише в одному напрямку [6].

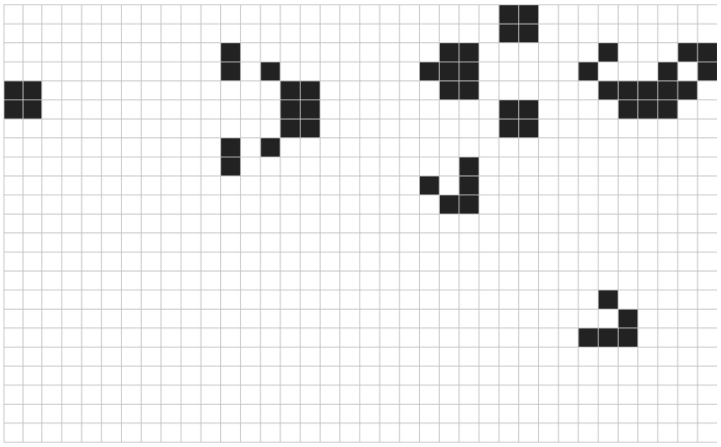

Рисунок 1.7 – Створення нових клітин [6]

Початковою метою Конвея було визначити цікавий та непередбачуваний клітинний автомат. Конвей експериментував з різними правилами, прагнучи знайти правила, які б дозволяли шаблонам очевидно зростати без обмежень, водночас ускладнюючи доведення того, що будь-який заданий шаблон може це робити. Більше того, деякі прості початкові шаблони повинні зростати та змінюватися протягом значного періоду часу, перш ніж зупинитися у статичній конфігурації або повторюваному циклі. Пізніше Конвей писав, що основною мотивацією для “Життя” було створення універсального клітинного автомата [6].

“Гра Життя” демонструє дивовижне виникнення складності з простоти. Незважаючи на основні правила, виникають складні закономірності, моделі поведінки та навіть симуляції реалістичних структур. Вона служить яскравою ілюстрацією того, як прості системи можуть породжувати складність, а взаємодія окремих компонентів може призвести до емерджентних властивостей. Крім того, вона підкреслює значення початкових умов і те, як вони формують еволюцію динамічних систем, відображаючи ширші концепції в галузях від математики та інформатики до біології та філософії [6].

Основні переваги симуляції:

- прості правила (базується на дуже простих локальних правилах, але здатна породжувати неймовірно складні, динамічні та навіть обчислювально універсальні структури);

- наглядність і легкість симуляції (модель легко реалізувати та візуалізувати, стан кожної клітинки добре видно на двовимірній сітці);

- демонстрація емерджентних властивостей (показує, як з простих взаємодій можуть виникати складні закономірності, самоорганізація та стійкі структури);

- освітня цінність (часто використовується для пояснення понять клітинних автоматів, складних систем, теорії хаосу, еволюції та штучного життя);

- активна дослідницька спільнота (існує безліч досліджень, патернів і модифікацій, що робить її зручною платформою для експериментів).

Основні недоліки симуляції:

- відсутність практичного моделювання реальності (попри складність поведінки, “Гра життя” не відображає жоден конкретний природний чи фізичний процес — вона радше абстрактна модель);

- чутливість до початкових умов (невелика зміна у стартовій конфігурації може повністю змінити результат симуляції);

- важкість передбачення (неможливо аналітично передбачити еволюцію системи);

- межові ефекти (на обмеженому полі клітинки “вмирають” на межах, тому часто використовують періодичні межі або безкінечні решітки для реалістичності);

- обмеження у швидкодії при великих полях (для великих сіток і тривалих симуляцій потрібні значні обчислювальні ресурси).

“Гра Життя” Конвея є найбільш цікавим КА, ніж інші аналоги тому, що має високу наочність і освітню цінність. Це дозволяє ефективно демонструвати принципи клітинних автоматів, самоорганізації, еволюції та складних систем. Вона є двовимірною моделлю, що краще відображає природні процеси та дозволяє створювати різноманітні патерни (глайдери, осцилятори, стабільні структури). Крім того, ця симуляція є Тюрінг-повною системою, що робить її універсальною платформою для дослідження обчислювальних процесів, симуляцій та

моделювання штучного життя. Завдяки великій кількості існуючих досліджень і відкритому потенціалу для модифікацій, її реалізація у дипломній роботі є як науково обґрунтованою, так і практично цікавою.

1.3 Постановка завдання до роботи

Метою роботи є програмна реалізація комп'ютерної симуляції "Гра Життя" Конвея. Симуляція має відповідати всім критеріям та правилам, яким відповідає обраний КА.

Задачі роботи:

- аналіз вимог до симуляції;
- вибір інструментів розробки;
- розробка архітектури ПЗ;
- програмна реалізація симуляції;
- тестування фінального проєкту;
- дослідження патернів.

1.4 Висновок

У цьому розділі було проведено аналіз предметної області, визначено основні поняття, пов'язані з предметною областю, об'єктом дослідження та розробки симуляції.

Було досліджено основні КА для реалізації комп'ютерної симуляції та проведено порівняльний аналіз цих КА, який показав причини реалізувати саме "Гру Життя" Конвея.

Було сформульовано і поставлено завдання до роботи.

2 МАТЕРІАЛИ І МЕТОДИ

Другий розділ буде присвячено вибору основних інструментів для реалізації симуляції, а саме: ігровий рушій, мова програмування та інтегроване середовище розробки.

2.1 Огляд існуючих ігрових рушіїв

Ігровий рушій — це програмний фреймворк, призначений переважно для розробки відеоігор, який може включати спеціалізовані програмні бібліотеки та пакети, такі як редактори рівнів. Термін “рушій” є прямим аналогом терміна “програмний рушій”, оскільки він використовується в різних секторах індустрії програмного забезпечення [7].

Розробники ігор можуть використовувати ігрові рушії для створення та публікації відеоігор на різних платформах, таких як настільні комп’ютери, мобільні пристрої, ігрові консолі та інші типи комп’ютерів. Основні функції, які зазвичай охоплює ігровий рушій, включають 2D- або 3D-рендерер, фізичний рушій, аудіо рушій, скрипти, анімацію, штучний інтелект, мережу, потокову передачу, управління пам’яттю, потоки, підтримку локалізації, графіка сцен та кінематику [7].

2.1.1 Unity

Unity — це кросплатформний ігровий рушій, що пропонує надійне середовище для створення інтерактивних 2D та 3D-досвідів. Він використовує компонентну архітектуру, де GameObjects — базові сутності в сценах де отримують функціональність завдяки таким компонентам, як Transform (для положення, обертання та масштабування), MeshRenderer (для візуальних елементів), Rigidbody (для фізики) та Script (для логіки). Розробники пишуть поведінку за допомогою скриптів C#, використовуючи середовище виконання Mono/.NET Unity. Редактор Unity дозволяє редагувати сцени в режимі реального часу, налагоджувати їх та

імпортувати ресурси, підтримуючи такі формати, як FBX, PNG та WAV. Рендеринг здійснюється за допомогою вбудованого URP (Universal Render Pipeline) або HDRP (High Definition Render Pipeline) з налаштуванням шейдерів за допомогою Shader Graph або HLSL. Фізика працює на базі NVIDIA PhysX, що забезпечує реалістичне моделювання зіткнень, сил та тканин [8].

Unity спрощує розробку ігор, використовуючи компонентну архітектуру, де скрипти визначають поведінку ігрових об’єктів у 2D- або 3D-середовищі. Редактор Unity пропонує візуальний інтерфейс для проєктування ігрових рівнів, де ресурси можна упорядковувати для створення сцен. Користувачі можуть перетягувати ресурси в редактор, такі як 3D-моделі, текстури та аудіофайли, а потім упорядковувати та налаштовувати їх для створення бажаного ігрового досвіду. C# є основною мовою сценаріїв, яка підтримується широкою екосистемою ресурсів, префабів та шаблонів для оптимізації розробки [8].

Основні переваги рушія:

- кросплатформність (Unity підтримує понад 25 платформ: Windows, macOS, Android, iOS, WebGL, Linux, консолі);

- зручне середовище розробки (інтуїтивний інтерфейс, візуальний редактор сцен, можливість налаштування об’єктів “drag and drop”);

- велика спільнота та документація (існує безліч туторіалів, форумів і прикладів, що значно спрощує навчання та пошук рішень);

- потужний фізичний та анімаційний рушій (вбудовані системи фізики, анімації та частинок дозволяють створювати реалістичні сцени без складних розрахунків);

- гнучкість і масштабованість (можна створювати як прості мобільні ігри, так і великі симуляції, візуалізації або інтерактивні навчальні програми).

Основні недоліки рушія:

- проблеми з продуктивністю у великих проєктах (для масштабних ігор або симуляцій Unity може споживати багато ресурсів, а оптимізація вимагає досвіду);

- обмеження безкоштовної версії (у комерційних проєктах із великим прибутком потрібна ліцензія, що може бути доволі дорого);

- менша візуальна якість (Unity потребує додаткових налаштувань або шейдерів для досягнення фотореалістичного рендерингу);

- складність у створенні власних інструментів (хоча Unity дозволяє розширювати функціонал, створення складних редакторських скриптів потребує глибокого знання API);

- великі розміри готових збірок (навіть проста гра може займати десятки мегабайт через структуру рушія та вбудовані бібліотеки).

2.1.2 Unreal

Unreal Engine – це широко використовуваний ігровий рушій, відомий своєю вражаючою універсальністю та потужністю. Epic Games створила рушій, щоб надати розробникам повний набір інструментів для створення візуально приголомшливих, захопливих 3D-ігор та віртуальних ігор. Незалежно від того, новачок ви чи досвідчений розробник, Unreal Engine пропонує велику документацію, навчальні посібники та онлайн-ресурси для підтримки вашого навчання. Завдяки таким помітним сценаріям використання, як Fortnite, Rocket League та Gears of War. Unreal Engine зарекомендував себе як провідний вибір для розробки ігор, а також допомагає в архітектурній візуалізації, віртуальній реальності та кіновиробництві [9].

Збірка Unreal Engine базується на модульній архітектурі, яка дозволяє розробникам підтримувати чистоту та роздільність коду. Це дозволяє розробникам легко оновлювати або додавати контент до ігрового коду, не порушуючи його роботу. Unreal Engine також має велику колекцію можливостей візуальних ефектів та інструментів рендерингу, а також пропонує широкий набір додаткових функцій, включаючи фізичні симуляції, інструменти анімації, аудіосистеми, мережеві функції та потужну систему візуальних сценаріїв під назвою Blueprints, доступну для тих, хто має обмежений досвід програмування [9].

Основні переваги рушія:

- фотореалістична графіка (Unreal Engine пропонує передові технології візуалізації, які забезпечують реалістичне освітлення, тіні та деталізацію сцени);

- візуальне програмування (дозволяє створювати логіку гри без глибоких знань коду);

- потужний фізичний рушій та система анімації (вбудовані інструменти забезпечують реалістичну фізику, частинки й складну анімацію персонажів);

- професійна екосистема (використовується не лише у геймдеві, а й у кіновиробництві, архітектурній візуалізації, автомобільному дизайні та наукових симуляціях);

- безкоштовна ліцензія до певного рівня прибутку (Unreal Engine безкоштовний для використання, а роялті сплачується лише після отримання значного прибутку).

Основні недоліки рушія:

- високі системні вимоги (для комфортної роботи потрібен потужний комп’ютер, що може бути проблемою для студентів або невеликих команд);

- крута крива навчання (Unreal Engine має багато інструментів і систем, тож початківцю важко швидко опанувати весь функціонал);

- великий розмір проєктів і рушія (навіть базові проєкти можуть займати десятки гігабайт, а сам рушій потребує багато дискового простору);

- довший час компіляції (зміни у коді або матеріалах можуть компілюватися значно довше);

- вищі вимоги до оптимізації (незважаючи на високу якість графіки, досягнення стабільної продуктивності потребує ретельного налаштування).

2.1.3 Godot

Godot — це кросплатформний, безкоштовний ігровий рушій з відкритим вихідним кодом, випущений за ліцензією MIT. Спочатку його розробили в Буенос-Айресі аргентинські розробники програмного забезпечення Хуан Лінієцкі та Аріель Манзур для кількох компаній у Латинській Америці до публічного випуску в 2014

році. Середовище розробки працює на багатьох платформах і може експортувати на ще кілька. Воно призначене для створення як 2D, так і 3D ігор (рис. 2.1), орієнтованих на ПК, мобільні пристрої, веб-сайти, а також платформи віртуальної, доповненої та змішаної реальності, а також може використовуватися для розробки неігрового програмного забезпечення [10].

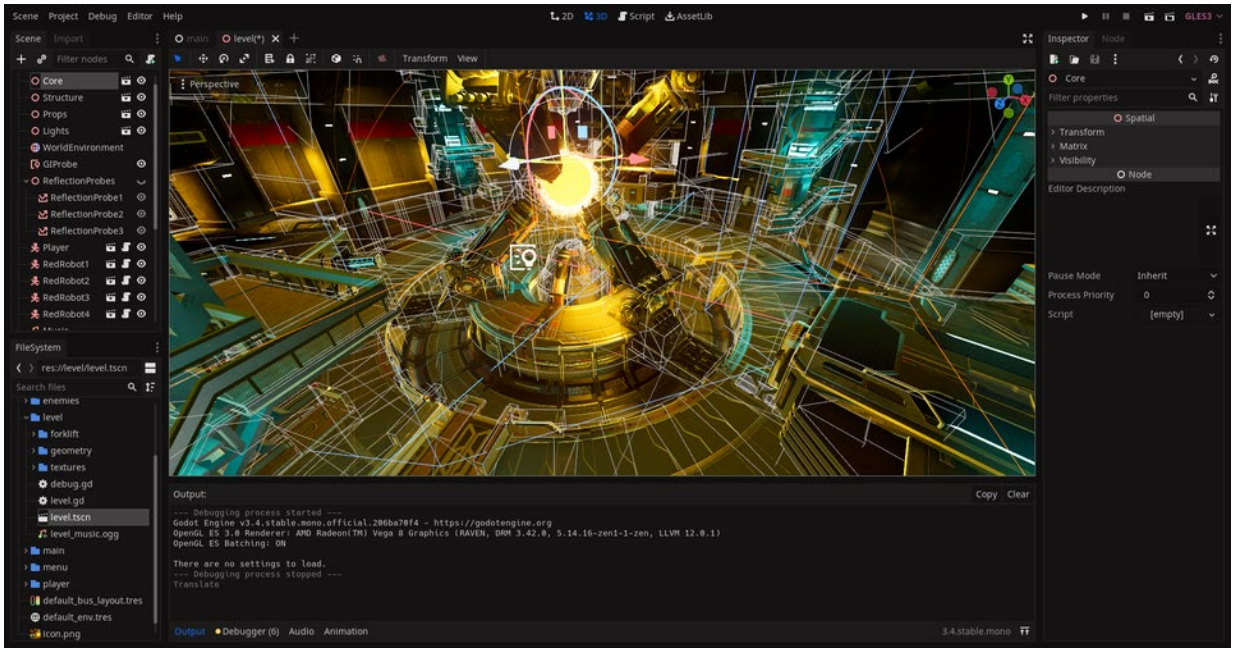

Рисунок 2.1 – Створення 3D гри в Godot [10]

Godot дозволяє розробникам відеоігор створювати як 3D, так і 2D ігри, використовуючи кілька мов програмування, таких як C++, C# та GDScript. Він використовує ієрархію вузлів для спрощення розробки. Класи можуть бути похідними від типу вузла для створення більш спеціалізованих типів вузлів, які успадковують поведінку. Вузли організовані всередині “сцени”, які є повторно використовуваними, екземплярними, успадковуваними та вкладеними групами вузлів. Вузли з’єднані сигналами, які можуть передавати об’єкти даних. Усі ігрові ресурси, включаючи скрипти та графічні ресурси, зберігаються як частина файлової системи комп’ютера (а не в базі даних). Це рішення для зберігання призначене для сприяння співпраці між командами розробників ігор, які використовують системи контролю версій програмного забезпечення [10].

Основні переваги рушія:

- відкритий вихідний код (Godot повністю безкоштовний і має відкриту ліцензію MIT);

- невелика вага та швидке завантаження (рушій дуже легкий, швидко встановлюється і не потребує потужного обладнання, що зручно для навчання та невеликих проєктів);

- інтуїтивна структура сцени (основна архітектура рушія побудована на вузлах, що забезпечує модульність і простоту побудови складних сцен);

- власна мова програмування (GDScript схожа на Python, тому її легко вивчити навіть початківцям);

- відсутність роялті або комерційних обмежень (ви повністю володієте своїм проєктом і не сплачуєте жодних відрахувань незалежно від прибутку).

Основні недоліки рушія:

- відсутність офіційної підтримки комерційного рівня (у випадку технічних проблем немає офіційної служби підтримки);

- обмежена підтримка мобільних та консолей (хоч експорт на мобільні платформи можливий, оптимізація потребує додаткових зусиль, а підтримка консолей офіційно обмежена);

- GDScript менш продуктивний (для складних або ресурсомістких проєктів може бути потрібне переписування логіки на C# або C++);

- менше навчальних матеріалів (хоч документація якісна, кількість туторіалів і відео все ще менша, ніж у інших рушіїв);

- менша база активів і плагінів (Godot Asset Library поки не така розвинута, як набір плагінів у інших рушіїв).

2.2 Порівняльний аналіз ігрових рушіїв

Ознайомившись з описаними вище ігровими рушіями, було проведено порівняльний аналіз (табл. 2.1).

Таблиця 2.1 – Порівняльна характеристика ігрових рушіїв

Характеристика / Ігровий рушій	Unity	Unreal	Godot
Якість графіки	Висока	Дуже висока	Середня
Вимоги до обладнання	Середні	Високі	Низькі
Розмір рушія та збірок	Середній	Дуже великий	Дуже компактний
Кросплатформність	Дуже широка	Дуже широка	Широка
Мова програмування	C#	C++	GDScript, C#, C++
Ліцензія та вартість	Безкоштовно до певного рівня прибутку	Безкоштовно, але 5% роялті після $1 млн прибутку	Повністю безкоштовний, open-source (MIT)
Підтримка 2D/3D	Потужна підтримка обох форматів	Орієнтований переважно на 3D	Дуже добра підтримка 2D, покращена 3D

Під час порівняння ігрових рушіїв для розробки симуляції, було визначено, що Godot є найкращим вибором. Godot — це повністю безкоштовний і відкритий рушій, що не потребує жодних ліцензійних платежів чи роялті. Це важливо для академічного середовища, де необхідна повна свобода використання програмного забезпечення та можливість модифікації його під потреби проєкту. Також рушій має зручну модульну архітектуру (Node-система), яка спрощує побудову симуляційних моделей. Завдяки цьому можна швидко створювати, тестувати й змінювати об’єкти без складного налаштування сцен чи взаємодій.

Таким чином, вибір Godot обґрунтований такими критеріями:

- відкритість і відсутність комерційних обмежень;
- відкритість коду;
- прогресивна спільнота;
- легкість освоєння та гнучкість у реалізації симуляцій;
- низькі системні вимоги та швидке тестування;
- можливість детального контролю за логікою моделі;
- академічна доцільність для навчальних і дослідницьких завдань.

2.3 Огляд існуючих мов програмування

Мови програмування бувають різні. І хоча обранний КА можна розробляти на будь-якій мові, в даній роботі йде прив’язка до ігрового рушія. Тому створюватись симуляція має в саме тій мові, на якій це можна зробити в Godot. В своїй основі, цей ігровий рушій дозволяє програмувати на трьох мовах програмування: GDScript, C# та C++.

2.3.1 GDScript

GDScript — це вбудована мова сценаріїв Godot Engine. На відміну від мов програмування загального призначення, таких як C++ або Java, GDScript оптимізовано для розробки ігор у Godot. Він легкий, інтуїтивно зрозумілий і дозволяє розробникам швидко керувати ігровими об’єктами, обробляти вхідні дані, керувати фізикою тощо. Його синтаксис зрозумілий та читабельний, що допомагає початківцям швидше розуміти концепції програмування [11].

GDScript – це серце скриптингу в Godot Engine. Його простий синтаксис, тісна інтеграція з Godot та зручність для початківців роблять його ідеальним для будь-кого, хто починає розробку ігор. Вивчивши основи змінних, функцій, потоку керування, сигналів та взаємодії вузлів, розробник зможе почати створювати інтерактивні ігри та втілювати свої творчі ідеї в життя [11].

Основні переваги мови:

- висока швидкість розробки (завдяки простому синтаксису, відсутності компіляції та миттєвому перезапуску сцен, код можна швидко тестувати й змінювати під час роботи);

- підтримка сигналів і сценової архітектури (мова природно працює з подієвою моделлю рушія: сигнали, вузли, сцени);

- простота та зрозумілість синтаксису (синтаксис GDScript дуже схожий на Python);

- глибока інтеграція з рушієм Godot (GDScript розроблена спеціально для Godot, тому забезпечує прямий доступ до всіх об’єктів, сцен і вузлів без потреби в додаткових бібліотеках або API);

- легка інтеграція з іншими мовами (за потреби можна комбінувати GDScript із C#, C++ або VisualScript у межах одного проєкту).

Основні недоліки мови:

- менше сторонніх бібліотек і пакетів (на відміну від інших мов, GDScript має обмежену екосистему сторонніх модулів);

- обмежене застосування поза Godot (мова створена виключно для рушія Godot, тому не використовується в інших середовищах чи платформах);

- молодість мови (мова ще розвивається, тому можуть бути незначні зміни у синтаксисі або недопрацьовані функції у різних версіях рушія);

- відсутність статичної типізації за замовчуванням (хоч статична типізація підтримується, вона не є обов’язковою, що може призвести до помилок під час виконання, якщо код не перевірений уважно);

- менше навчальних ресурсів (GDScript має порівняно невелику кількість туторіалів і книжок, тому навчання часто спирається на офіційну документацію та спільноту).

2.3.2 C#

Мова C# є найпопулярнішою мовою для платформи .NET, безкоштовного, кросплатформного середовища розробки з відкритим кодом. Програми на C# можуть працювати на багатьох різних пристроях, від пристроїв Інтернету речей (IoT) до хмари та будь-де між ними. Можна писати програми для телефонів, настільних комп’ютерів, ноутбуків та серверів [12].

C# — це кросплатформна мова загального призначення, яка робить розробників продуктивними під час написання високопродуктивного коду. З мільйонами розробників C# є найпопулярнішою мовою .NET. C# має широку підтримку в екосистемі та для всіх робочих навантажень .NET. Базуючись на

об’єктно-орієнтованих принципах, вона включає багато функцій з інших парадигм, не в останню чергу функціональне програмування [12].

Основні переваги мови:

- простий і зрозумілий синтаксис (C# поєднує в собі риси C++ і Java, має логічну структуру, строгі правила й зрозумілий синтаксис);

- об’єктно-орієнтована мова (усе в C# базується на принципах об’єктно-орієнтованого програмування, що полегшує розробку масштабних і структурованих проєктів);

- кросплатформність (завдяки .NET Core і сучасним версіям .NET, C# можна використовувати на Windows, macOS і Linux);

- безпечність типів і керування пам’яттю (C# має вбудований механізм “garbage collection” і строгий контроль типів, що мінімізує ризик помилок);

- велика спільнота та підтримка Microsoft (наявність офіційної документації, регулярних оновлень і численних навчальних матеріалів спрощує навчання та вирішення проблем).

Основні недоліки мови:

- залежність від .NET середовища (для виконання програм потрібна установка платформи .NET, що ускладнює розгортання на деяких системах);

- вищі вимоги до ресурсів (програми на C# часто споживають більше пам’яті, ніж еквівалентні реалізації на C/C++, особливо у великих застосунках);

- складність для початківців у великих проєктах (через багатий функціонал і об’ємний фреймворк новачкам може бути важко зрозуміти архітектуру великих застосунків);

- закрита екосистема Microsoft (хоча C# став відкритішим завдяки .NET Core, історично він пов’язаний з Microsoft, і багато рішень залишаються орієнтованими на Windows);

- менша гнучкість у низькорівневому програмуванні (C# не дає повного контролю над пам’яттю, тому не підходить для системного програмування або мікроконтролерів).

2.3.3 C++

C++ — це високорівнева мова програмування загального призначення, створена датським вченим-комп'ютерником Б'ярне Страуструпом. C++ зазвичай реалізується як компільована мова, і багато постачальників надають компілятори C++, включаючи Free Software Foundation, LLVM, Microsoft, Intel, Embarcadero, Oracle та IBM [13].

Мова C++ була розроблена з урахуванням системного програмування, вбудованого програмного забезпечення з обмеженими ресурсами та великих систем, з урахуванням продуктивності, ефективності та гнучкості використання як основних особливостей її розробки. C++ також виявився корисним у багатьох інших контекстах, ключовими перевагами якого є програмна інфраструктура та програми з обмеженими ресурсами, включаючи настільні програми, відеоігри, сервери та програми, критично важливі для продуктивності [13].

Основні переваги мови:

- висока продуктивність і швидкість виконання (C++ компілюється у нативний машинний код, що забезпечує максимально можливу швидкість роботи програм);

- повний контроль над пам'яттю (розробник сам керує виділенням і звільненням пам'яті, що дозволяє точно оптимізувати використання ресурсів);

- потужні можливості для оптимізації (розробник може контролювати навіть дрібні аспекти виконання, що дозволяє досягати максимальної ефективності);

- підтримка кількох парадигм програмування (C++ дозволяє комбінувати процедурний, об'єктно-орієнтований і шаблонний підходи, що робить його універсальною мовою);

- стандартна бібліотека STL (STL містить набір готових контейнерів і алгоритмів, які значно спрощують розробку).

Основні недоліки мови:

- складність мови (C++ має складний синтаксис і велику кількість особливостей, що створює високий поріг входу для новачків);

- ручне керування пам’яттю (помилки у виділенні або звільненні пам’яті можуть призвести до “витоків пам’яті”, аварій або нестабільності програми);

- довший цикл розробки (через компіляцію, складність синтаксису та потребу в оптимізації розробка займає більше часу);

- повільна компіляція великих проєктів (у великих кодових базах компіляція може займати значний час через шаблони та складну систему залежностей);

- висока складність налагодження (помилки, пов’язані з пам’яттю або шаблонами, часто важко відстежити й виправити).

2.4 Порівняльний аналіз мов програмування

Ознайомившись з описаними вище мовами програмування, було проведено порівняльний аналіз (табл. 2.2).

Таблиця 2.2 – Порівняльна характеристика мов програмування

Характеристика / Мова програмування	GDScript	C#	C++
Продуктивність	Середня	Висока	Дуже висока
Керування пам’яттю	Автоматичне	Автоматичне	Ручне
Складність синтаксису	Низька	Середня	Висока
Швидкість розробки	Висока	Середня	Низька
Можливості оптимізації	Обмежені	Середні	Максимальні
Підтримка парадигм	Об’єктно-орієнтована	Об’єктно-орієнтована	Багатопарадигмна
Портативність	Висока	Висока	Дуже висока

Порівнявши три мови програмування, було вирішено обрати саме C++ для розробити комп’ютерної симуляції. C++ забезпечує максимальну швидкість виконання, точний контроль над ресурсами, гнучкість у реалізації алгоритмів і стабільність роботи. Це робить його найоптимальнішим вибором для створення

високопродуктивних симуляційних систем, де ефективність та надійність мають ключове значення.

2.5 Огляд існуючих середовищ розробки

Інтегроване середовище розробки – це програмний застосунок, який допомагає програмістам ефективно розробляти програмний код. Воно підвищує продуктивність розробників, поєднуючи такі можливості, як редагування, збірка, тестування та пакування програмного забезпечення, у простій у використанні програмі. Так само, як автори використовують текстові редактори, а бухгалтери – електронні таблиці, розробники програмного забезпечення використовують IDE, щоб полегшити свою роботу [14].

Більшість інтегрованих середовищ розробки (IDE) містять функціональність, яка виходить за рамки редагування тексту. Вони забезпечують централізований інтерфейс для поширених інструментів розробника, що робить процес розробки програмного забезпечення набагато ефективнішим. Розробники можуть швидко почати програмувати нові програми, замість того, щоб вручну інтегрувати та налаштовувати різне програмне забезпечення. Їм також не потрібно вивчати всі інструменти, і вони можуть зосередитися лише на одній програмі [14].

2.5.1 Qt Creator

Qt Creator — це кросплатформне інтегроване середовище розробки на C++, JavaScript, Python та QML, яке спрощує розробку графічних застосунків. Воно є частиною SDK для фреймворку розробки графічних застосунків Qt та використовує Qt API, який інкапсулює виклики функцій графічного інтерфейсу хост-ОС. Воно включає візуальний налагоджувач та інтегрований WYSIWYG-конструктор макетів і форм графічного інтерфейсу. Редактор має такі функції, як підсвічування синтаксису та автозаповнення. Qt Creator використовує компілятор C++ з колекції компіляторів GNU в Linux. У Windows він може використовувати MinGW або

MSVC зі стандартною інсталяцією, а також може використовувати Microsoft Console Debugger під час компіляції з вихідного коду [15].

Qt Creator забезпечує підтримку створення та запуску Qt-застосунків для робочих середовищ (Windows, Linux, FreeBSD та macOS), мобільних пристроїв (Android, BlackBerry, iOS, Maemo та MeeGo) та вбудованих пристроїв Linux. Налаштування збірки дозволяють користувачеві перемикатися між цільовими середовищами збірки, різними версіями Qt та конфігураціями збірки. Для цільових пристроїв та для мобільних пристроїв Qt Creator може створити інсталяційний пакет, встановити його на мобільний пристрій, підключений до комп'ютера розробника, та запустити його там [15].

Основні переваги середовища:

- кросплатформність (Qt Creator працює на Windows, Linux і macOS, а створені в ньому програми можна компілювати під різні операційні системи без суттєвих змін у коді);

- потужний дизайнер інтерфейсу (IDE містить візуальний редактор інтерфейсів, який дозволяє створювати вікна та елементи управління у drag-and-drop стилі, що суттєво прискорює розробку);

- вбудовані інструменти для налагодження та профілювання (IDE має власний debugger, аналізатор пам'яті, профілювальник продуктивності, а також інтеграцію з GDB, LLDB і Valgrind);

- модульність та налаштовуваність (IDE підтримує плагіни, дозволяє налаштовувати інтерфейс, схеми кольорів, клавіатурні скорочення та інші параметри під конкретного розробника);

- документація та підтримка спільноти (Qt має чітку, детальну документацію та активну спільноту, що допомагає швидко знаходити рішення для більшості типових проблем).

Основні недоліки середовища:

- високий поріг входу для початківців (незважаючи на зручний інтерфейс, повноцінне розуміння принципів роботи Qt потребує часу);

- складність налаштування середовища (для коректної роботи може знадобитися налаштування компілятора, дебагера та шляхів до бібліотек, особливо на Windows);

- ресурсоємність (Qt Creator споживає досить багато оперативної пам’яті, особливо під час роботи з великими проєктами або під час збірки);

- обмежена інтеграція з іншими фреймворками (IDE оптимізована під Qt, тому робота з іншими бібліотеками чи рушіями може бути менш зручною);

- менше розширень (кількість доступних плагінів та інструментів менша, ніж у великих IDE).

2.5.2 CLion

CLion — це комплексне та складне середовище розробки, розроблене для розробників на C та C++. Воно надає інструменти та функції, що спрощують процес програмування, полегшуючи написання, тестування, налагодження та підтримку коду. Завдяки інтелектуальним можливостям JetBrains, CLion підтримує розробника протягом усього процесу кодування, від автодоповнення та генерації коду до рефакторингу, налагодження та тестування [16].

CLion сумісний з платформами Windows, macOS та Linux, що робить його чудовим вибором для кросплатформної розробки. Крім того, він здатний обробляти проєкти різних типів та розмірів, від програм з одним файлом до масштабних проєктів [16].

Основні переваги середовища:

- професійне середовище для C/C++ (CLion створений спеціально для мов C і C++, тому має глибоку інтеграцію з їх особливостями);

- розумне автодоповнення коду (IDE забезпечує інтелектуальне автодоповнення, навігацію по коду, рефакторинг, швидкі підказки та аналіз помилок у реальному часі);

- підтримка CMake за замовчуванням (використовує CMake як основну систему складання, що робить роботу зі складними проєктами зручною та передбачуваною);

- кросплатформність (CLion працює на Windows, Linux і macOS, дозволяючи розробляти й тестувати один і той самий проєкт на різних системах);

- потужний аналізатор коду (CLion автоматично виявляє потенційні помилки, витоки пам’яті, некоректні типи, недосяжний код тощо).

Основні недоліки середовища:

- високі вимоги до системних ресурсів (IDE споживає багато оперативної пам’яті та процесорного часу, особливо на великих проєктах);

- складність для новачків (велика кількість налаштувань і функцій може спочатку відлякувати початківців у C++);

- повільне індексування великих проєктів (при першому відкритті IDE виконує повне індексування, що може тривати кілька хвилин для проєктів із тисячами файлів);

- менше оптимізації для GUI-проєктів (CLion більше орієнтований на логіку та алгоритмічну частину, ніж на візуальну розробку інтерфейсів);

- не така глибока інтеграція з деякими компіляторами Windows (хоч підтримка покращилася, іноді виникають дрібні несумісності або потреба в ручному налаштуванні середовища).

2.5.3 Visual Studio

Visual Studio — це потужний інструмент для розробників, який можна використовувати для завершення всього циклу розробки в одному місці. Це комплексне інтегроване середовище розробки, яке можна використовувати для написання, редагування, налагодження та збірки коду, а потім для розгортання програми. Visual Studio містить компілятори, інструменти автодоповнення коду, систему керування версіями, розширення та багато інших функцій для покращення кожного етапу процесу розробки програмного забезпечення [17].

Завдяки різноманітності функцій та мовної підтримки у Visual Studio, можна перейти від написання своєї першої програми “Hello World” до розробки та розгортання додатків. Наприклад, створювати, налагоджувати та тестувати додатки .NET та C++, редагувати сторінки ASP.NET у режимі вебдизайнера, розробляти кросплатформні мобільні та настільні додатки за допомогою .NET або створювати адаптивні вебінтерфейси на C# [17].

Основні переваги середовища:

- зручний і продуманий інтерфейс (має інтуїтивну структуру, вкладки, панелі, інтегрований файловий оглядач і редактор проєктів, що спрощує роботу навіть з великими проєктами);

- інтелектуальне автодоповнення коду (Visual Studio надає розумні підказки, автодоповнення, підсвічування помилок, контекстну інформацію про змінні й методи, що підвищує продуктивність програміста);

- потужний вбудований налагоджувач (дозволяє ставити точки зупину, переглядати значення змінних, відстежувати стек викликів, аналізувати пам’ять і виконання коду по кроках);

- зручність для навчання та командної розробки (завдяки великій кількості навчальних матеріалів, туторіалів і готових шаблонів Visual Studio підходить для студентів і викладачів);

- інтеграція з тестуванням і профілюванням (вбудовані інструменти для unit-тестування, профілювання продуктивності, аналізу пам’яті).

Основні недоліки середовища:

- високі системні вимоги (потребує багато оперативної пам’яті та дискового простору);

- великий час завантаження (через велику кількість модулів, запуск Visual Studio може тривати кілька десятків секунд, особливо на слабших ПК);

- обмежена кросплатформність (повноцінно Visual Studio працює лише на Windows);

- велика кількість опцій і меню (через безліч налаштувань новачкам може бути складно швидко зорієнтуватися в інтерфейсі IDE);

- орієнтованість на екосистему Microsoft (більшість можливостей оптимізовано під Windows і .NET, тому робота з іншими екосистемами потребує додаткових налаштувань або розширень).

2.6 Порівняльний аналіз середовищ розробки

Ознайомившись з описаними вище середовищами розробки, було проведено порівняльний аналіз (табл. 2.3).

Таблиця 2.3 – Порівняльна характеристика середовищ розробки

Характеристика / Середовище розробки	Qt Creator	CLion	Visual Studio
Інтерфейс користувача	Простий, легкий, орієнтований на Qt	Мінімалістичний, технічний, з глибокими налаштуваннями	Потужний, зручний, графічно насичений
Підтримувані мови	C++, QML, Python	C, C++, Python, Rust, Kotlin	C++, C#, Python, JavaScript, HTML, SQL
Кросплатформність	Windows, Linux, macOS	Windows, Linux, macOS	Windows
Продуктивність IDE	Висока, легка для системи	Середня, залежить від проєкту	Висока, але ресурсомістка
Ціна	Безкоштовна	Платна	Безкоштовна
Підтримка тестування	Базова	Google Test, Catch2, Boost.Test	MSTest, Google Test, Catch2, інтегровані фреймворки

Порівнявши три середовища розробки, було вирішено обрати саме Visual Studio для розробити комп’ютерної симуляції. Visual Studio має один із найкращих налагоджувачів серед усіх IDE, що дозволяє ефективно знаходити і виправляти помилки, відстежувати змінні, контролювати роботу з пам’яттю та виконання коду. Це особливо важливо при створенні симуляцій, де правильна логіка й оптимізація обчислень відіграють ключову роль. Також Visual Studio пропонує інтелектуальне

автодоповнення коду (IntelliSense), інтегровану систему тестування, контроль версій через Git і зручне профілювання продуктивності, що робить розробку більш ефективною та контрольованою.

2.7 Висновок

У цьому розділі були проаналізовані основні інструменти розробки та зроблені відповідні порівняння задля того, щоб отримати найкращі варіанти для розробки симуляції “Гра Життя” Конвея.

Було проведено огляд ігрових рушіїв, та встановлено, що найбільші переваги має саме Godot, тому що цей ігровий рушій легкий та має гарну інтеграцію з багатьма мовами програмування.

Було проведено порівняльний аналіз мов програмування та встановлено, що найкращим варіантом є мова програмування C++, тому що вона ефективна в розробці симуляції, має гарну інтеграцію з ігровими рушіями, а також має повний контроль над ресурсами.

Також в якості інтегрованого середовищ розробки був обраний Visual Studio.

3 РОЗРОБКА СИМУЛЯЦІЇ

Третій розділ буде присвячений розробці архітектури та коду симуляції “Гра Життя” Конвея. Будуть розглянуті основні класи, які слугуватимуть ядром ПЗ.

3.1 Структура проєкту

Структура проєкту гри — це місце, де знаходяться речі та як вони називаються. У широкому сенсі вона також може включати рішення щодо використання хмарних сервісів, контролю версій та контролю доступу. Кожна команда та кожен проєкт різні, і неможливо передбачити всі наслідки рішень на початку. Краще мати прості правила, яких люди можуть легко дотримуватися, та коригувати їх з часом, коли виникають певні больові точки [18].

Основні принципи структури проєкту:

- зробити речі легшими для пошуку;
- спрощувати тестування, оновлення та розгортання;
- спрощувати переміщення асетів через робочий простір;
- зробити так, щоб людям було легко працювати разом;
- передавати інформацію про логічну структуру проєкту;
- зробити роботу з проєктом приємною [18].

Структура C++ gamedev проєкту залежить від його масштабу та складності, але загалом повинна включати окремі папки для вихідного коду, ресурсів (графіка, звуки), документації, збірки та сторонніх бібліотек. Ключові елементи можуть бути розділені на логічні групи. У спільноті розробників рекомендовано розбивати код на менші, повторно використовувані модулі для полегшення розробки та тестування.

У даній дипломній роботі структура проєкту матиме майже такий самий шаблон, як в багатьох C++ проєктах на репозиторіях в GitHub (рис. 3.1).

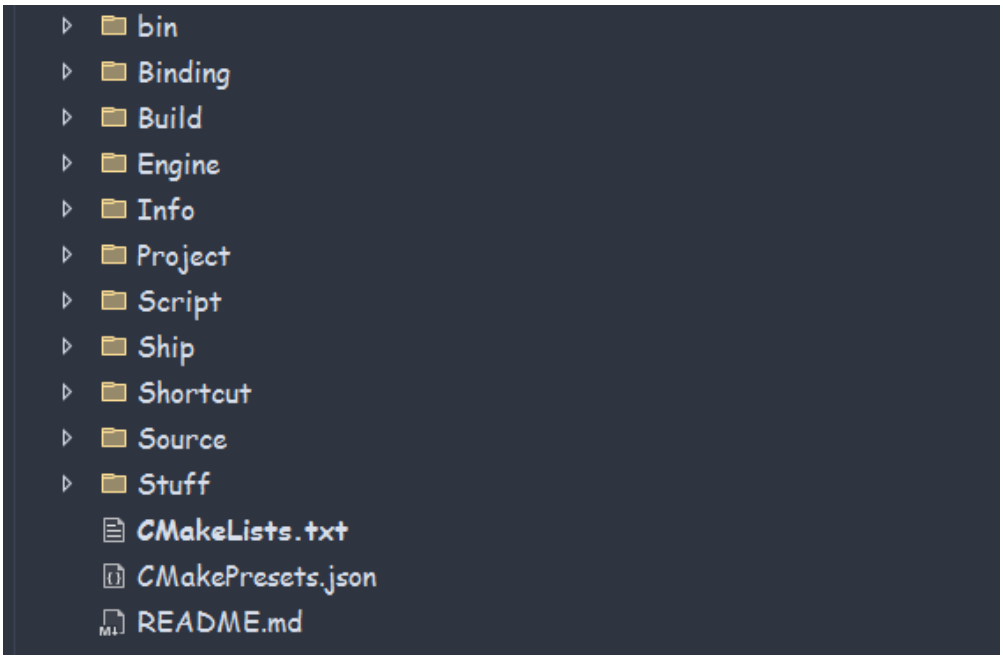

Рисунок 3.1 – Структура проєкту

Директорія Bin – призначена для зберігання бінарних файлів. Такі директорії ще зазвичай мають динамічні бібліотеки, статичні бібліотеки, файли бази даних, дебаг файли (в залежності від типу збірки проєкту). За бажанням, ця директорія може мати декілька версій одного проєкту.

Директорія Build – призначена для зберігання проміжних файлів проєкту. Такі файли не є критично важливими і слугують лише для прискорення розробки проєкту. Цю директорію можна сміливо очистити в будь-який момент самостійно, але таким зазвичай займається система збірки або мета-система збірки проєкту.

Директорія Engine – призначена для зберігання файлів ігрового рушія. Це можуть бути файли розробки (якщо код відкритий) або бінарні файли. Повністю зібраний Godot має один звичайний бінарний файл та бінарний файл з логом у консолі.

Директорія Info – призначена для зберігання корисної інформації. Така директорія може мати документацію до ігрового рушія, корисні посилання на джерела, нотатки, схеми, таблиці та будь-яка інша інформація, яка може принести користь у розробці проєкту.

Директорія Project – призначена для зберігання основних файлів проєкту. Ця директорія має багато асетів, які використовуються ігровим рушієм: спрайти, музика, сцени, моделі, текстури, шейдери, шрифти та текстові файли. У випадку ігрового рушія Godot, директорія Project також має мати файл з розширенням godot, що слугує маяком для рушія і позначає директорію як початкову точку проєкту. Без цього, неможливо відкрити проєкт у Godot.

Директорія Script – призначена для зберігання допоміжних скриптів. Це не обов'язково мають бути скрипти ігрового рушія. Директорія може мати будь-які скрипти: shell, bat, python, cmake, тощо. Скрипти слугують для автоматизації рутинних робіт, таких як:

- створення повторюваних файлів;
- очищення директорій;
- модифікація специфічних файлів;
- пошук будь-яких компонентів.

Директорія Ship – призначена для зберігання фінальних файлів проєкту (тобто на релізі проєкту). Ці файли мають бути відправлені замовнику проєкту або його покупцю. Проєкт не має мати жодних багів, несправностей чи колізій.

Директорія Source – призначена для зберігання основних файлів з кодом. Ці файли слугують основою для всього проєкту. Мова програмування може бути будь-яка. У випадку з Godot – це C++, тому директорія має файли hpp та cpp.

Директорія Stuff – призначена для зберігання будь-яких інших файлів. Це може бути:

- старі або непотрібні файли проєкту;
- збірка, яку потрібно тимчасово зберегти;
- файли інших проєктів.

3.2 Робочий процес

Весь робочий процес складається з декількох етапів, кожен з яких потребує особистого налаштування та уваги.

3.2.1 GDExtension

GDExtension — це технологія, специфічна для Godot, яка дозволяє рушію взаємодіяти з нативними спільними бібліотеками під час виконання. Розробник може використовувати її для запуску нативного коду без його компіляції разом із рушієм [19].

Існує три основні методи, за допомогою яких це досягається:

- набір функцій C, які Godot та GDExtension можуть використовувати для взаємодії (gdextension_interface.h);

- список функцій C, що надаються з API Godot (extension_api.json);

- формат файлу, який зчитується Godot для завантаження GDExtension (gdextension) [19].

Більшість розробників створюють GDExtensions з деякими існуючими мовними зв'язками, такими як godot-cpp (для C++) або з одними, створеними спільнотою (рис. 3.2). Прив'язки C++ для GDExtension побудовані на основі API C GDExtension та забезпечують зручніший спосіб "розширення" вузлів та інших вбудованих класів у Godot за допомогою C++. Ця нова система дозволяє розширювати Godot майже до того ж рівня, що й статично пов'язані модулі та бібліотеки C++ [19].

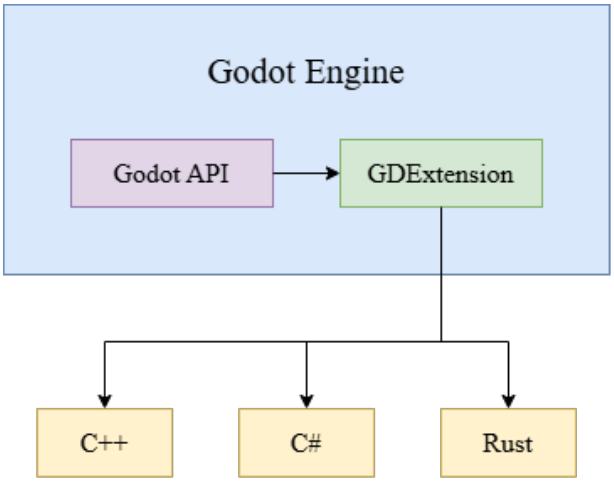

Рисунок 3.2 – Структура GDExtension

Саме GDExtension під C++ і буде використовуватися для створення симуляції “Гра Життя” Конвея. Ця технологія слугуватиме ключем та надійним мостом для створення C++ класів, з якими потім буде взаємодіяти ігровий рушій. Спочатку пишеться код, який розбитий на декілька cpp файлів, а потім система збірки збирає ці файли у одну динамічну бібліотеку. Дані про динамічну бібліотеку обов’язково вносяться у файл gdextension. Godot зчитує цей файл та завантажує відповідну бібліотеку. Такий процес навіть надає можливість розробнику компілювати та завантажувати код не виходячи з редактора ігрового рушія, що робить розробку проєкта набагато приємнішим процесом.

Таким чином GDExtension дозволяє нам використовувати C++ у розробці симуляції, що робить її більш оптимізованою для роботи та тестування.

3.2.2 CMake

CMake — це кросплатформне сімейство інструментів з відкритим вихідним кодом, призначене для збирання, тестування та пакування програмного забезпечення. CMake надає розробнику контроль над процесом компіляції програмного забезпечення за допомогою простих незалежних файлів конфігурації. На відміну від багатьох кросплатформних систем, CMake розроблено для використання разом із рідним середовищем збірки [20].

CMake є фактично стандартною мета-системою збірки програмного забезпечення, оскільки вона може:

- розширюватися за необхідності для підтримки нових функцій;

- створити власне середовище збірки, яке компілюватиме вихідний код, створюватиме бібліотеки, генеруватиме обгортки та збиратиме виконувані файли в довільних комбінаціях;

- підтримка кількох збірок з одного дерева вихідного коду;

- підтримка статичних та динамічних збірок бібліотек;

- створити файл кешу, призначений для використання з графічним редактором;

- підтримка складних ієрархій каталогів та програм, що залежать від кількох бібліотек;

- обробляти ситуації, коли виконувані файли необхідно зібрати для генерації коду, який потім компілюється та зв’язується в кінцеву програму [20].

Хоча Godot в основі використовує систему збірки SCons, він також дає можливість використовувати CMake. Спочатку створюється текстовий файл CMakeLists.txt, який вказує основні налаштування проєкту:

- назва, опис, посилання;

- шляхи для згенерованих файлів;

- архітектура;

- операційна система;

- тип збірки (дебаг, реліз);

- файли з кодом.

Потім виконуються CMake команди: генерація та збірка. Збірку вже виконує повноцінна система збірки, тому що CMake лише вказує, що треба зробити. Після компіляції та лінковки проєкту, його можна запустити у ігровому рушії або з консолі та перевірити результати. Весь процес можна розділити на три етапи (рис. 3.3).

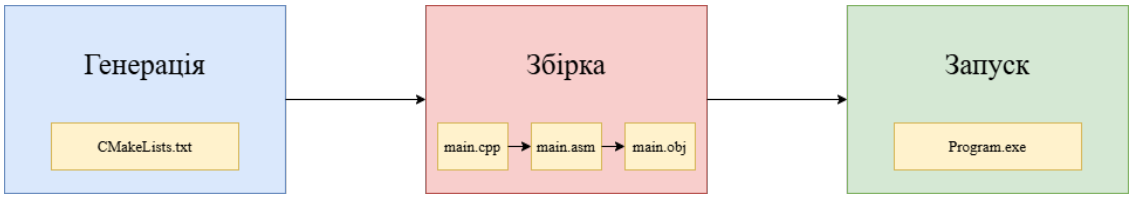

Рисунок 3.3 – Процес розробки

В цілому, розробка з таким процес відбувається досить швидко. Від зміни коду до відображення результату мінімум може зайняти від 5-7 секунд.

3.2.3 IntelliSense

Автодоповнення коду — це функція автодоповнення в багатьох інтегрованих середовищах розробки, яка пришвидшує процес написання коду

застосунків, виправляючи поширені помилки та пропонуючи рядки коду. Зазвичай це відбувається через спливаючі вікна під час введення тексту, запити параметрів функцій та підказки щодо синтаксичних помилок. Автодоповнення коду та пов'язані з ним інструменти служать для документації та усунення неоднозначностей для імен змінних, функцій та методів за допомогою статичного аналізу [21].

Ця можливість присутня в багатьох середовищах програмування. Visual Studio не є виключенням в цьому питанні і має цю технологію під назвою IntelliSense. Visual Studio навіть дає можливість налаштувати IntelliSense під свій смак. Так як процес розробки даної роботи не має специфічних потреб щодо автодоповнення, налаштування були залишені стандартними. У Visual Studio IntelliSense автоматично налаштовується під кожну генерацію CMake, тому необов'язково треба робити додаткових налаштувань. Але під час створення кожного cpp файлу, потрібно виконувати генерацію CMake, щоб IntelliSense правильно виконував автодоповнення коду, інакше можуть виникнути проблеми при розробці (рис. 3.4).

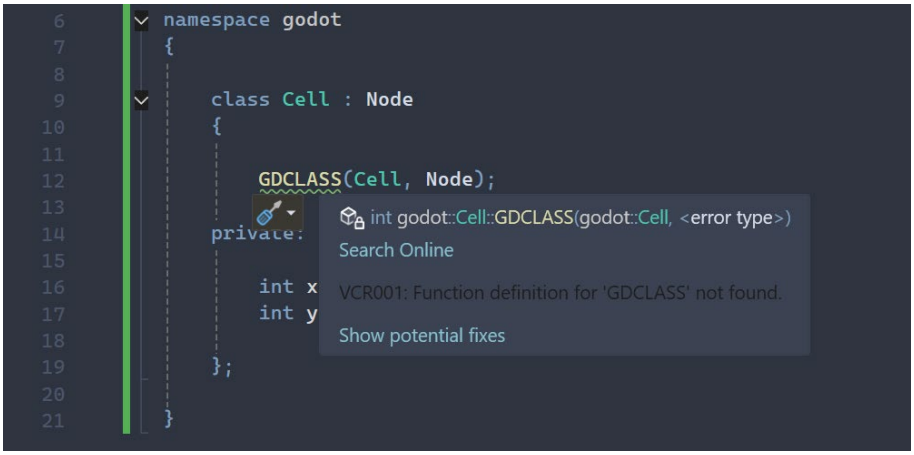

Рисунок 3.4 – Помилка в IntelliSense

Загалом під час розробки симуляції, IntelliSense давав такі можливості:

- доповнення коду (пропонує ключові слова, змінні та методи під час введення тексту, які можна вибрати для завершення коду кількома натисканнями клавіш);

- інформація про параметри (відображає невелике вікно, що показує параметри, типи та перевантаження для функції або конструктора, допомагаючи розробнику правильно їх використовувати);

- швидка інформація (надає вбудовану інформацію, таку як тип та визначення змінної або функції, при наведенні на неї курсора миші);

- помилки з хвилястими лініями (виділяє потенційні помилки або проблеми у вашому коді кольоровим підкресленням, допомагаючи розробнику виявляти та виправляти помилки на ранній стадії);

- синтаксичні поради (відображає синтаксис оператора під час його введення, особливо корисно для складних операторів);

- відфільтровані списки (може фільтрувати список автодоповнення, щоб відображати лише найважливіші елементи на основі введених символів);

- панель навігації (дворівневе випадаюче меню у верхній частині редактора, яке відображає визначення верхнього рівня у поточному файлі та допомагає швидко переміщатися між ними).

3.3 Архітектура симуляції

Вся архітектура розбивається на декілька C++ класів. Завдяки ігровому рушію Godot, розробку та кількість класів можна зменшити (непотрібно створювати свій велосипед), тому що Godot забезпечує всім необхідним і майже не потребує будь-яких доповнень. Всі інші елементи, це CMake скріпти, Godot робочі файли та шейдери.

3.3.1 Simulation

Клас Simulation – це перший C++ клас, з якого все починається. Він слугує стартовим місцем для всіх інших процесів. Вся обробка фізики, рендера, логіки, подій та фрейму починається з нього. Також цей клас реалізує основні функції

ініціалізації та деініціалізації. Після закінченння симуляції, клас виконує виклик виходу з програми.

Однак, слід зазначити, що даний клас не реалізує головний цикл всієї симуляції. Цим займається ігровий рушій Godot. Клас Simulation інтегрується у цикл, де потім він слугує головним ігровим об’єктом (рис. 3.5).

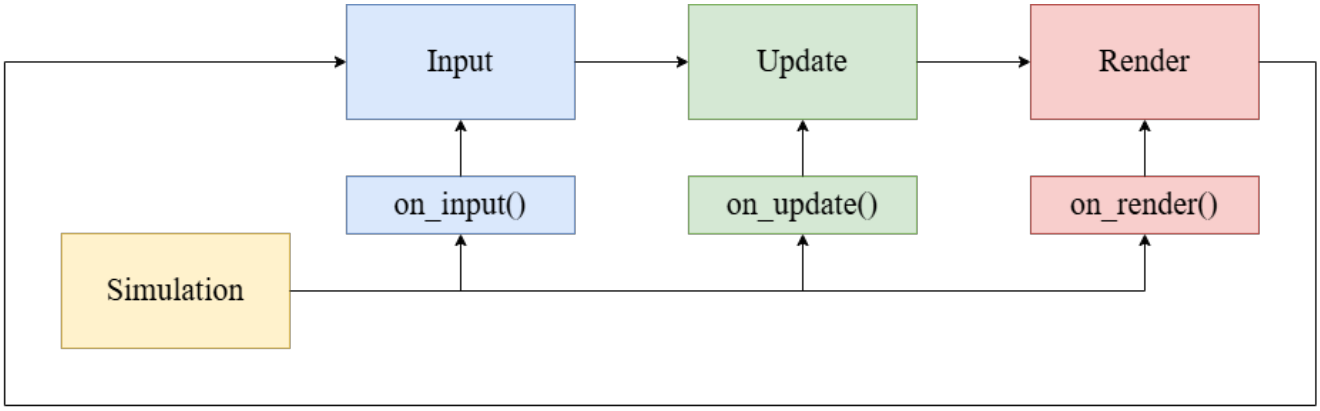

Рисунок 3.5 – Інтеграція Simulation

Сам ігровий цикл виконується безперервно під час гри. На кожному повороті циклу він обробляє введені користувачем дані без блокування, оновлює стан гри та рендерить гру. Він відстежує плин часу, щоб контролювати темп гри, її швидкість [22].

3.3.2 Player

Клас Player – це клас, який реалізує базові потреби користувача. Клас обробляє введення з клавіатури, миші та забезпечує плавне пересування віртуального гравця по двовимірному простору симуляції. Механізм руху реалізовано через оновлення позиції вузла відповідно до швидкості та напрямку, визначених користувачем. Переміщення відбувається за рахунок двох Godot функцій: “set_velocity” та “move_and_slide”. Вони реалізуються класом CharacterBody2D, який Player додає як дитину в ієрархії. Обробка переміщення обов’язково повинна бути в функції обробки фізики “_physics_process”.

Також клас має камеру. Камера дозволяє збільшувати або зменшувати зображення, що дає можливість користувачеві збільшувати або зменшувати фрагмент поля для детального перегляду чи навпаки — огляду більшої частини сітки. Обробку цих процесів можна створити і в звичайній функції обробки логіки “_process” або обробки подій “_input”. В цілому клас Player має досить просту структуру, часто яку використовують в демо-проєктах (рис. 3.6).

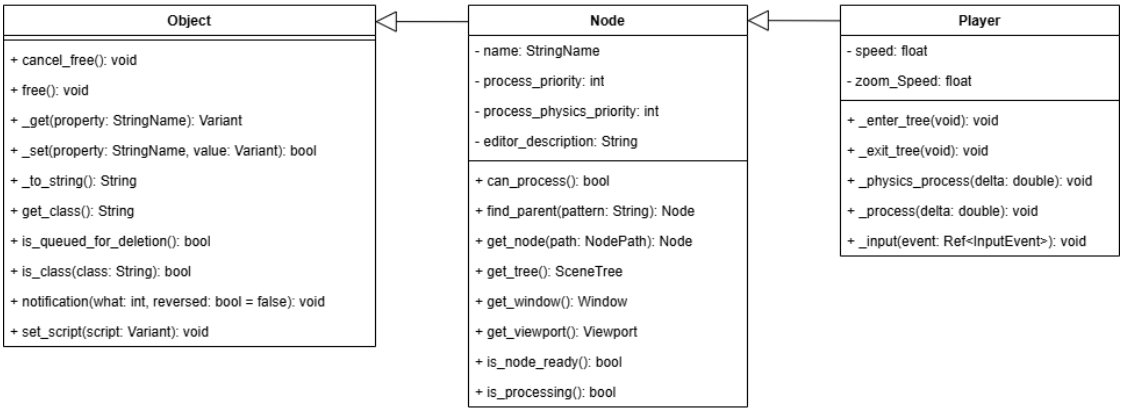

Рисунок 3.6 – Реалізація класу Player

Таким чином, клас Player відіграє роль інтерфейсного елемента, що забезпечує зручну навігацію користувача по полі клітинок та взаємодію з масштабом візуалізації, що є важливим для комфортної роботи із симуляцією. Після завершення симуляції, цей клас знищюється головним класом Simulation. Godot автоматично очищює виділену пам’ять, тому розробнику не обов’язково це робити в ручну.

3.3.3 Playground

Клас Playground – це клас, який реалізує життя клітин. Він відповідає за представлення, зберігання та оновлення стану клітинок у симуляції “Гра Життя” Конвея. Є центральним компонентом логіки моделі та реалізує всі операції, пов’язані з еволюцією ігрового поля. Саме цей клас дозволяє користувачу пересуватися по ігровому полю (рис. 3.7).

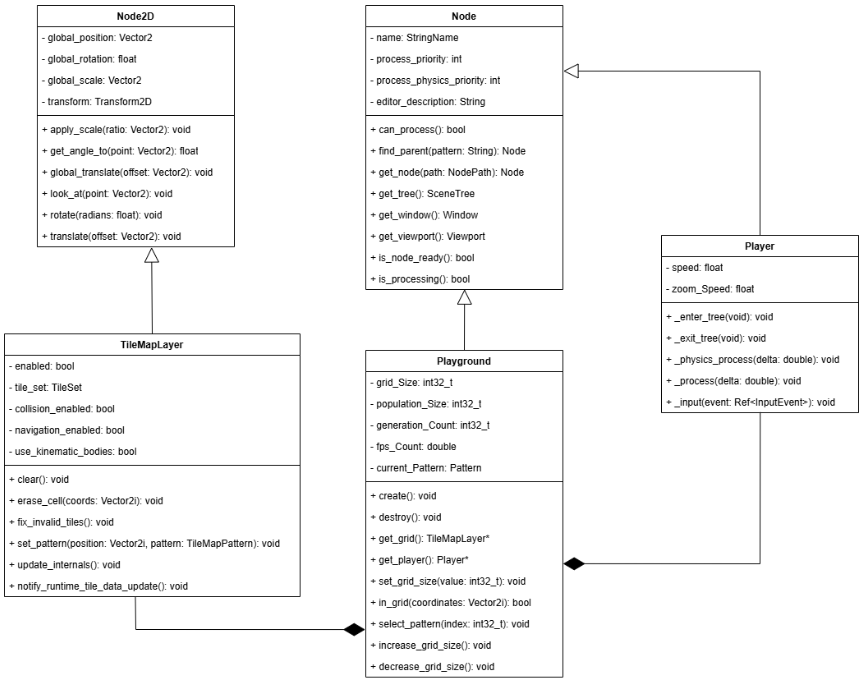

Рисунок 3.7 – Реалізація класу Playground

Основні функціональні можливості класу охоплюють:

- зберігання стану клітин;
- обчислення наступного покоління;
- ініціалізація та очищення поля;
- зміна стану окремих клітин;
- тісна взаємодія з ігровим рушієм;
- оптимізація обчислень.

Клас містить двовимірний масив або векторну структуру, яка відображає поточний стан кожної клітинки (жива/мертва). Ця структура даних забезпечує швидкий доступ до стану елементів та ефективне оновлення. Він реалізує правила “Гри Життя” Конвея, що визначають виникнення, виживання або смерть клітинок залежно від кількості їхніх живих сусідів. Обчислення здійснюється окремо від основного масиву станів, а саме у тимчасовий набір клітин, щоб уникнути побічних змін у процесі оновлення. Для того, щоб зменшити витрати на обчислення, у тимчасовий контейнер зберігаються лише живі клітини, де на кожному кроці симуляції відбувається перевірка стану цієї клітини, а також перевірка її сусідів.

Також клас надає методи для встановлення або перемикання стану обраної клітини, що використовується під час взаємодії користувача з полем (наприклад, у режимі редагування початкового покоління). Якщо перемикання лише однієї клітини не достатньо, то клас надає варіант перемикання станів цілим патерном. Можна обрати будь-який із заздалегіть розробленого списку.

Таким чином, клас Playground виконує роль основного обчислювального модуля симуляції, відповідаючи за зберігання структури поля та реалізацію правил еволюції клітинок, що є фундаментальною частиною “Гри Життя”.

3.3.4 Interface

Клас Interface забезпечує графічну та логічну взаємодію користувача з симуляцією “Гра Життя” Конвея. Основним призначенням класу є відображення елементів користувацького інтерфейсу, обробка подій та передача необхідних команд іншим компонентам системи, таким як Simulation та Playground. В своїй основі він використовує Godot UI класи (рис. 3.8).

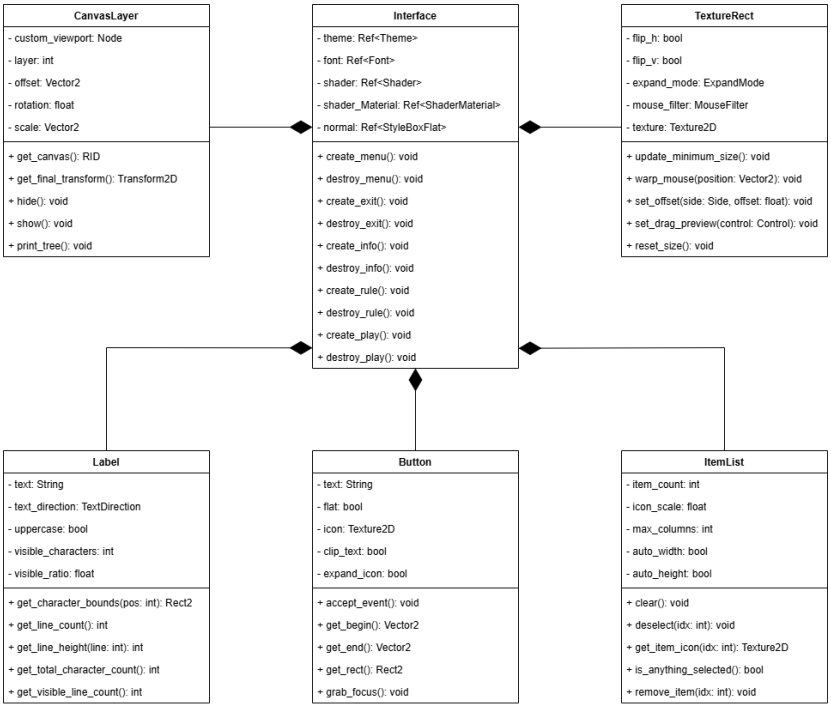

Рисунок 3.8 – Реалізація класу Interface

Функціональні можливості класу включають:

- відображення елементів UI;
- обробка подій користувача;
- передавання команд до логічних модулів;
- налаштування параметрів симуляції;
- відображення інформаційних повідомлень;
- інтеграція з Godot UI-системою.

Interface отримує події введення (натискання кнопок, вибір пунктів меню, зміна параметрів) і визначає відповідні дії. Наприклад, запуск або зупинення симуляції, перехід до наступного покоління, очищення поля, зміну швидкості роботи тощо. Також він містить механізм виводу службової інформації: поточного покоління, кількості живих клітинок, FPS, статусу симуляції та інших даних, що покращують зручність користування.

Таким чином, клас Interface виконує роль посередника між користувачем та основними компонентами симуляції, забезпечуючи інтуїтивне керування, візуалізацію стану системи та узгоджену взаємодію різних частин застосунку.

3.3.5 Locator

Клас Locator реалізує шаблон проєктування “Service Locator” та слугує централізованим механізмом доступу до глобальних сервісів симуляції “Гра Життя” Конвея. Основним призначенням класу є спрощення доступу до часто використовуваних об’єктів, усунення жорстких залежностей між компонентами та забезпечення зручного керування службами під час роботи застосунку.

Клас сервісу визначає абстрактний інтерфейс до набору операцій. Конкретний постачальник сервісу реалізує цей інтерфейс. Окремий локатор сервісу забезпечує доступ до сервісу, знаходячи відповідного постачальника, приховуючи як конкретний тип постачальника, так і процес, який використовується для його пошуку [23].

Завдяки цьому класу модулі на кшталт Player, Playground та Interface можуть взаємодіяти через спільний доступний контейнер сервісів, а не напряму. Це знижує зв'язність системи, спрощує розширення та полегшує підтримку коду (рис. 3.9). Також, Locator може забезпечувати перевірку наявності сервісу перед його використанням і кидати винятки, або повертати значення за замовчуванням у випадку відсутності зареєстрованого об'єкта. Це підвищує надійність програми.

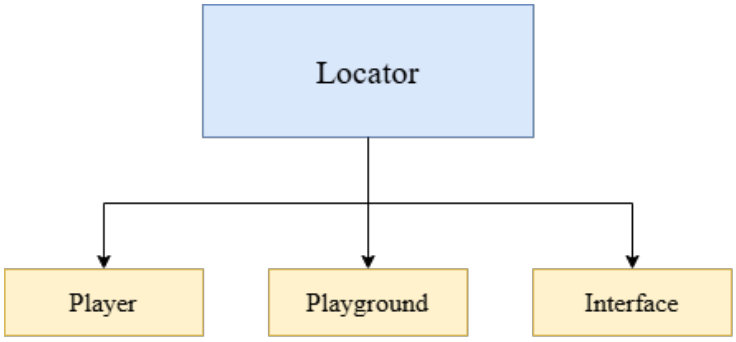

Рисунок 3.9 – Принцип роботи Locator

Таким чином, клас Locator забезпечує єдиний точковий доступ до ключових сервісів системи та слугує інструментом організації архітектури застосунку відповідно до шаблону Service Locator. Його використання зменшує кількість залежностей між компонентами, підвищує модульність та полегшує розширення функціональності симуляції.

3.4 Шейдери

Шейдери – це спеціальний вид програм, що працюють на графічних процесорах (GPU). Спочатку вони використовувалися для затінення 3D-сцен, але сьогодні можуть робити набагато більше. Розробник може використовувати їх для керування тим, як движок малює геометрію та пікселі на екрані, що дозволяє досягати різних ефектів [24].

Сучасні рендерингові механізми, такі як Godot, малюють все за допомогою шейдерів: відеокарти можуть виконувати тисячі інструкцій паралельно, що

призводить до неймовірної швидкості рендерингу. Однак, через свою паралельну природу, шейдери не обробляють інформацію так, як це робить типова програма. Код шейдера виконується на кожній вершині або пікселі окремо. Розробник не може зберігати дані між кадрами. Як результат, під час роботи з шейдерами, потрібно кодувати та мислити інакше, ніж в інших мовах програмування [24].

Коли відбувається рендер клітини на полі, клас Playground вже автоматично використовує стандартний шейдер. Таким чином не потрібно писати весь “Render Pipeline” з нуля. Однак для досягнення деяких спецефектів, було розроблено власний вершинний та піксельний шейдер. Ці шейдери “прикрашають” симуляцію та дають їй ефект старого телевізора, бо “Гру Життя” Джон Конвей придумав ще задовго до створення сучасних моніторів.

3.5 Висновок

У цьому розділі було створено структуру проєкту. Визначені основні директорії, де мають зберігатися файли різного призначення. В цілому файлова ієрархія прозора та зрозуміла для навігації.

Було обрано робочий процес, який найбільше підходить під створення комп’ютерної симуляції. CMake дійсно спрощює роботу, має гарну інтеграцію з Visual Studio та чудово працює разом з IntelliSense.

Були розроблені основні C++ класи, кожен з яких має свою чітку роль у проєкті:

- Simulation (точка входу);
- Player (переміщення, зміна фокусу);
- Playground (створення та знищення клітин);
- Interface (відображення інтерфейсу);
- Locator (глобальний доступ).

Використовуючи ігровий рушій Godot, вдалося знизити кількість класів до необхідного мінімума та значно пришвидшити розробку проєкту.

4 ЕКСПЛУАТАЦІЯ СИМУЛЯЦІЇ

Четвертий розділ буде присвячений тестуванню та експлуатації симуляції “Гра Життя” Конвея. Будуть розглянуті основний функціонал та інтерфейс.

4.1 Запуск симуляції

Фінальна збірка проєкту має всього чотири файли (рис. 4.1). Щоб запустити гру, достатньо два рази натиснути на “Life.exe”. В залежності від потреб, запускати гру можна з терміналу вводячи необхідні аргументи. Файл “Life.console.exe” дозволяє запустити гру з відкритим терміналом.

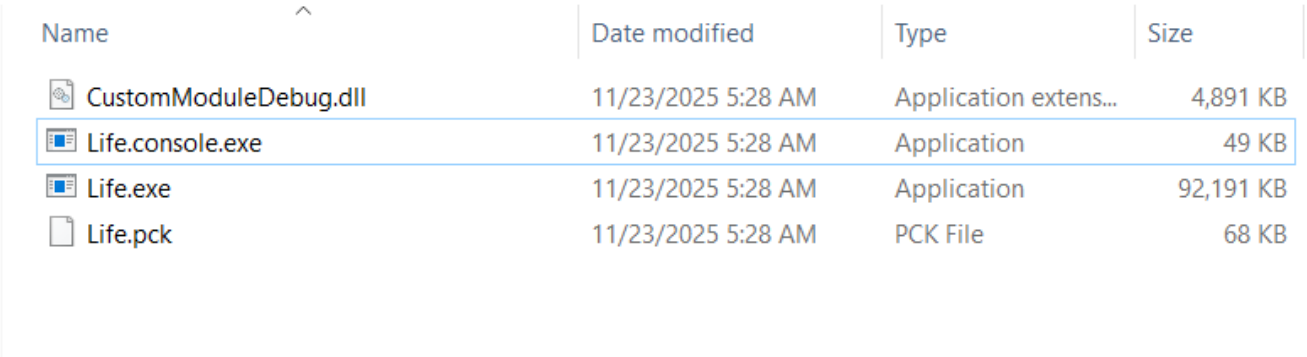

Рисунок 4.1 – Фінальна збірка проєкту

Файл “Life.pck” слугує сховищем для всіх матеріалів гри: сцени, текстури, аудіо, шейдери, тощо. Також важливо зазначити, що користувачу в жодному разі не можна переміщувати, змінювати чи видаляти файл “CustomModuleDebug.dll”. Саме ця динамічна бібліотека реалізує всю симуляцію. Видалення цього файлу призведе до відкриття порожньої сцени.

4.2 Системні вимоги

Мінімальні системні вимоги для роботи симуляції:

- операційна система: Windows 10;

- процесор: Intel Core i3;
- оперативна пам’ять: 4 Гб;
- відеопам’ять: 256 Мб;
- місце на жорсткому диску: 10 Мб.

Рекомендовані системні вимоги для роботи симуляції:

- операційна система: Windows 11;
- процесор: Intel Core i5;
- оперативна пам’ять: 8 Гб;
- відеопам’ять: 512 Мб;
- місце на жорсткому диску: 10 Мб.

4.3 Функціонал симуляції

Комп’ютерна симуляція призначена для розваг, досліджень, демонстрацій, тощо.

Симуляція має наступний функціонал:

- швидкий запуск;
- зручне головне меню;
- зручний та адаптований під різні монітори інтерфейс;
- налаштування сітки;
- ручна та атвоматична генерація клітин;
- можливість вручну проводити симуляцію;
- вибір патернів зі списку.

4.4 Застосування симуляції

Коли користувач запускає комп’ютерну симуляцію, то першим, що він побачить буде головне меню (рис. 4.2). Параметри шейдеру, який робить ефект старого телевізора, були налаштовані таким чином, щоб всі написи були дещо

спотворенні, проте і читабельні одночасно. Також слід зазначити, що весь інтерфейсу динамічно змінюється під різні розміри моніторів.

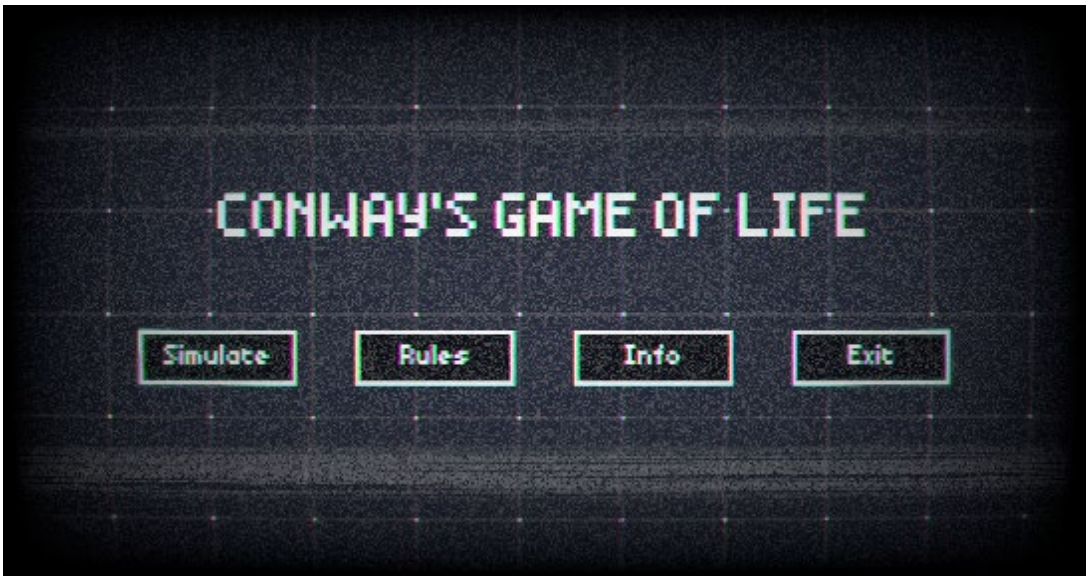

Рисунок 4.2 – Головне меню

Всі кнопки мають свій стиль під кожен випадок. У звичайному стані, кнопка - темна. Коли користувач наводить курсор миші, кнопка змінюється на світлу версію. А коли натискає на неї, то вона знову змінює свій стиль на інший (рис. 4.3).

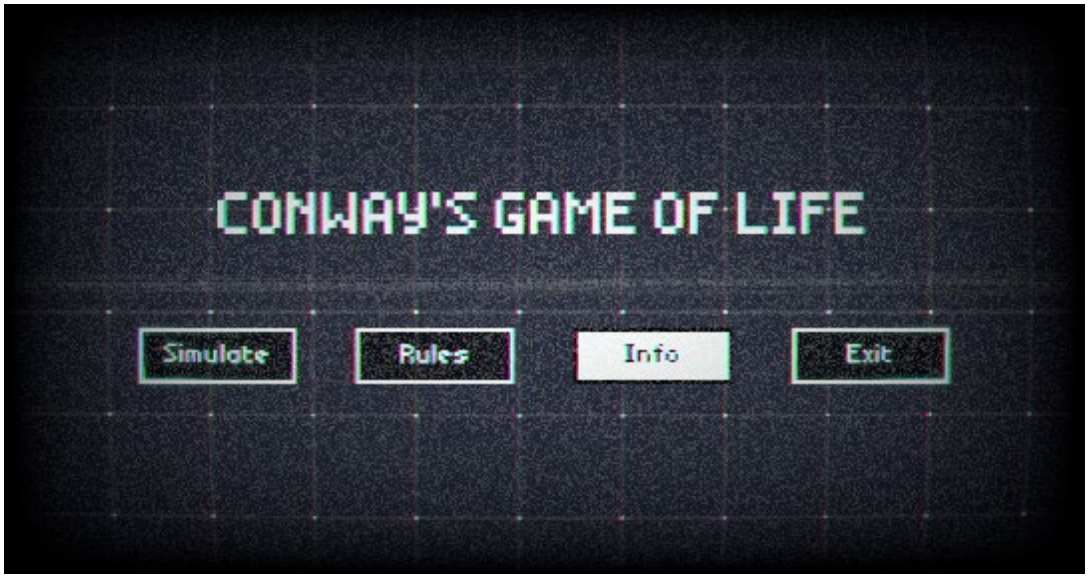

Рисунок 4.3 – Анімація головної кнопки

Кнопка “Exit” відкриває меню виходу з гри і пропонує користувачу вийти або залишитись (рис. 4.4). При натисканні “No”, користувача повертає до головного меню. При натисканні “Yes”, відбудеться повноцінний вихід з гри.

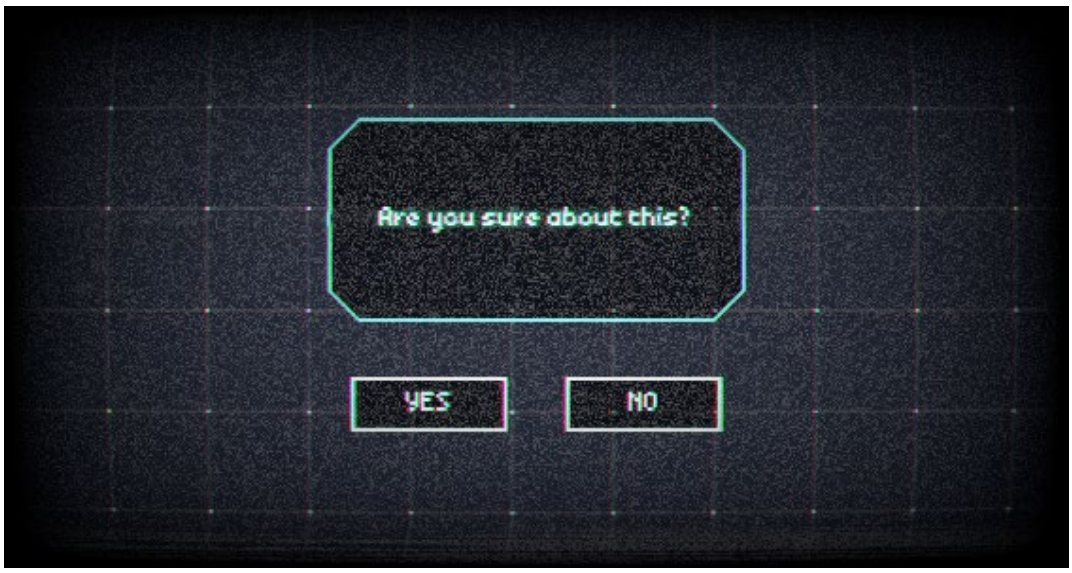

Рисунок 4.4 – Вихід з гри

Кнопка “Info” відкриває меню з основною інформацією про розробку проєкту (рис. 4.5), а саме: ім’я розробника, ігровий рушій, мова програмування та мета-система збірки.

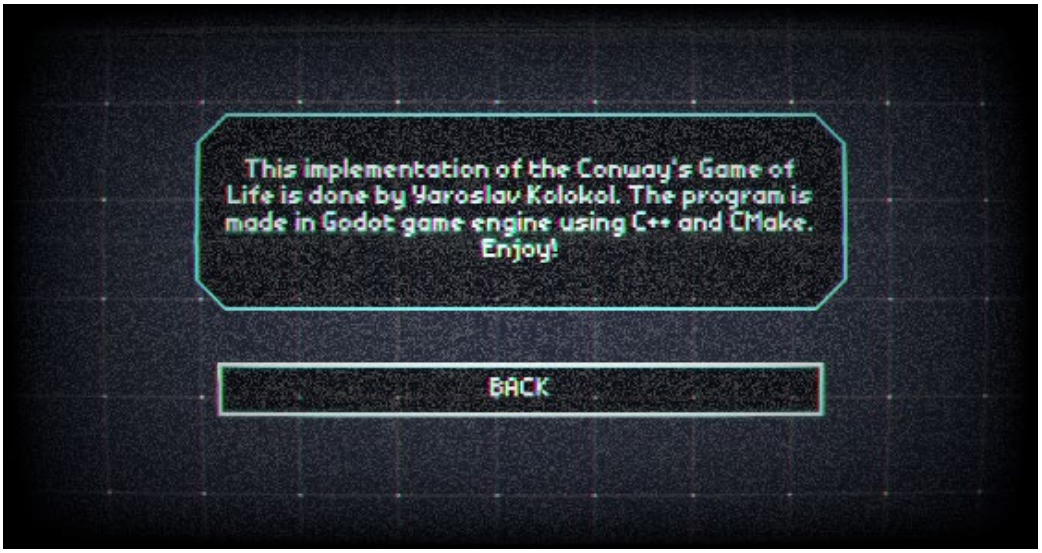

Рисунок 4.5 – Інформація про розробку

Кнопка “Rules” відкриває меню з інформацією, яка пояснює правила “Гри Життя” Конвея (рис. 4.6). Кнопка “Back” повертає користувача назад до головного меню.

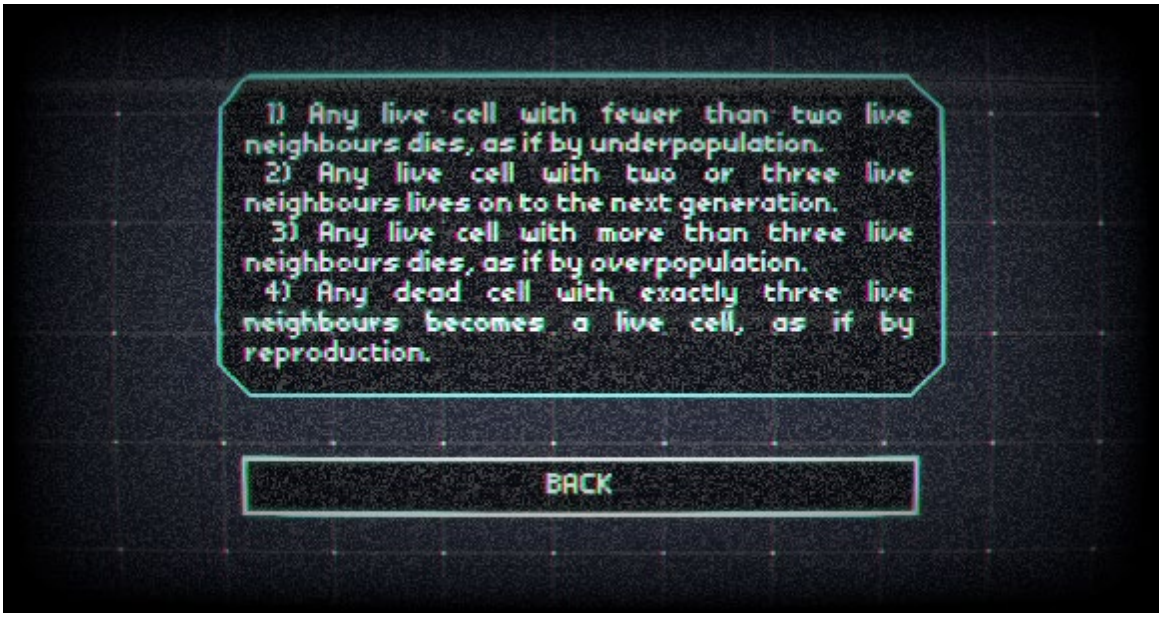

Рисунок 4.6 – Інформація про правила гри

Кнопка “Simulate” відкриває основну гру, де і буде відбуватися вся симуляція “Гри Життя” Конвея (рис. 4.7).

Рисунок 4.7 – Основна гра

Коли починається основна гра, розроблений ПЗ повністю надає доступ користувачу і саме він вирішує з якими початковими даними починати симуляцію та з яким темпом. Сама гра в жодному разі не диктує якісь умови та надає повний доступ. З гри можна вийти в будь-який момент і повернутися до головного меню, якщо натиснути на кнопку “Back”.

Зверху в основній грі розташовані чотири написи, кожен з яких має свою відповідну роль:

- fps (відображає кількість кадрів за секунду);
- grid (відображає інформацію про сітку);
- generation (відображає номер поточного покоління);
- population (відображає кількість живих клітин).

Коли користувач нажимає на кнопку “+ Grid”, сітка збільшує свій розмір на одну одиницю по висоті та ширині. А якщо користувач нажимає на кнопку “- Grid”, то сітка зменшує свій розмір на одну одиницю по висоті та ширині (рис. 4.8). Також сітку можна переміщати, якщо натискати клавіши W, A, S, D і масштабувати використовуючи клавіши Q та E.

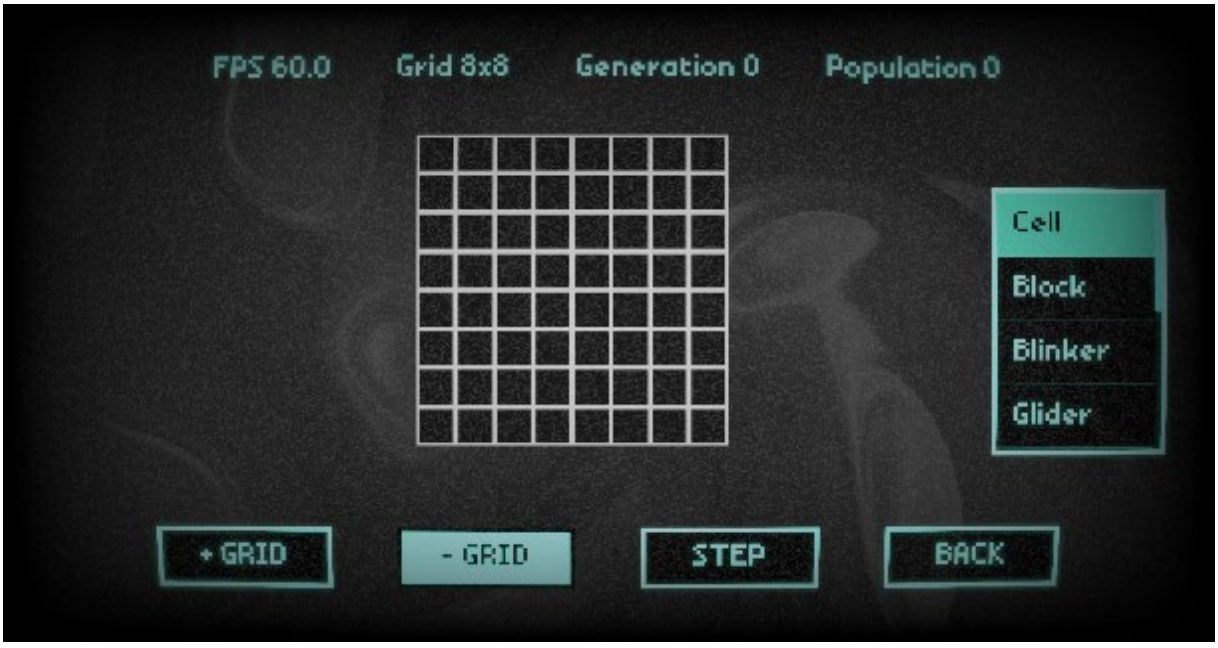

Рисунок 4.8 – Сітка для симуляцій

Основна суть полягає в тому що користувач може власноруч створювати та видаляти цілі патерни на сітці (одна клітина також вважається патерном). Для цього на правій стороні гри користувачу надається список із найвідоміших патернів у всьому світі. Якщо натиснути на патерн із списку і навести курсор на сітку, то можна побачити попередній перегляд результату (рис. 4.9). Після натискання середньої кнопки миші по сітці, патерн встановиться і буде готовий до симуляції. Знищувати клітини можна так само цілими патернами, тільки треба вже навести на будь-яку зафарбовану область та натиснути на праву кнопку миші.

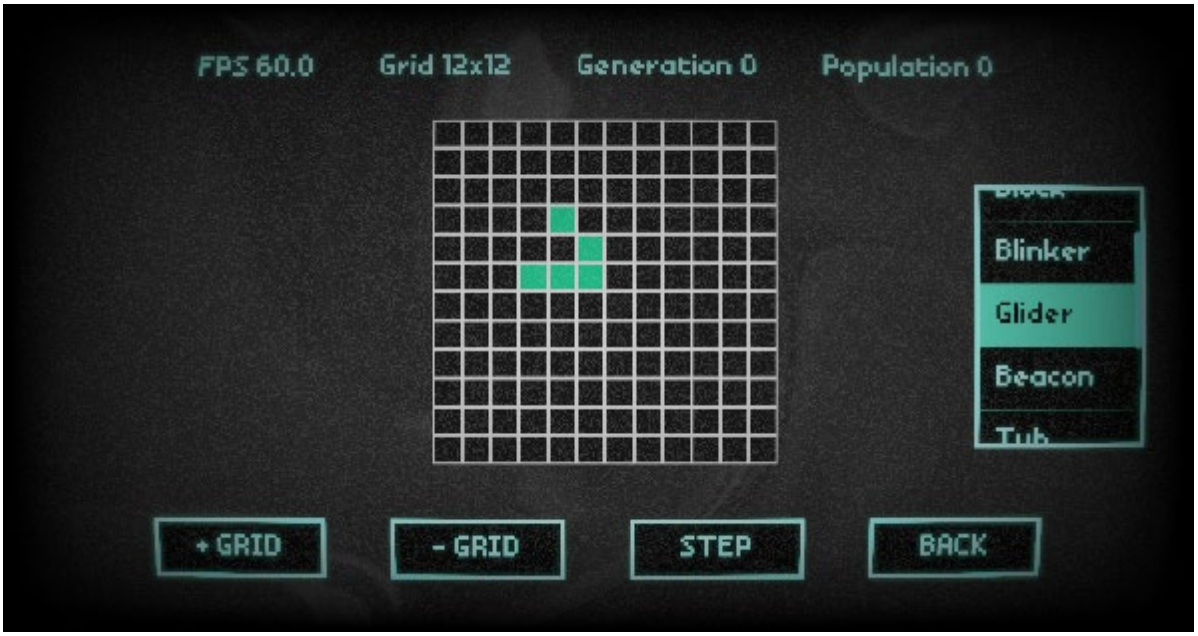

Рисунок 4.9 – Встановлення патерну

Сам процес симуляція відбувається досить просто. Щоб виконати один крок симуляції, достатньо нажати на кнопку “Step” або на пробіл. Після цього, відбудється обробка за всіма правилами “Гри Життя” Конвея (рис. 4.10):

- будь-яка жива клітина з менш ніж двома живими сусідами гине, ніби через недонаселення;

- будь-яка жива клітина з двома або трьома живими сусідами живе до наступного покоління;

- будь-яка жива клітина з більш ніж трьома живими сусідами гине, ніби від перенаселення;

- будь-яка мертва клітина з рівно трьома живими сусідами стає живою клітиною, ніби шляхом розмноження.

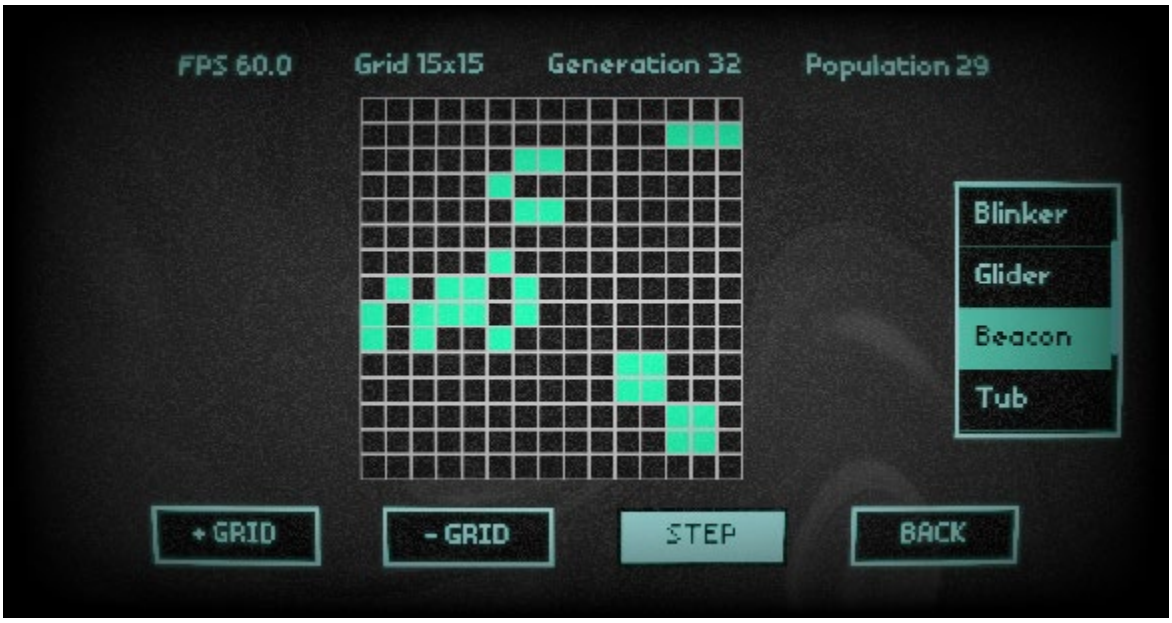

Рисунок 4.10 – Процес симуляції

Таким чином, можна проводити симуляції будь-яких масштабів, але все залежить від системних можливостей користувача. Чим більша сітка та чим більше клітин присутні на сітці, тим більша йде нагрузка на апартну частину користувача. З велики масштабами FPS починає падати, що приводить до негативного досвіду.

4.5 Висновок

У цьому розділі було показано інструкцію щодо запуску комп'ютерної симуляції та відображено основний функціонал.

Були описані мінімальні та рекомендовані системні вимоги для обладнання користувача.

Було детально оглянуто кожний аспект реалізованої “Гри Життя” Конвея із відображенням основних дій за допомогою скріншотів.

5 ДОСЛІДЖЕННЯ ПАТЕРНІВ

П’ятий розділ буде присвячений дослідженню патернів симуляції “Гра Життя” Конвея. Будуть розглянуті патерни із неоднаковою поведінкою в різних умовах симуляції за допомогою трьох експериментів.

5.1 Огляд існуючих патернів

Патерни у “Грі Життя” Конвея — це стани і конфігурації клітин на сітці, які поводяться певним чином під час еволюції за правилами гри. Вони діляться на такі типи:

- константи (ці патерни залишаються абсолютно незмінними від одного покоління до наступного);
- осцилятори (ці патерни повертаються до свого початкового стану після фіксованої кількості поколінь, повторюючи цикл);
- кораблі (ці патерни рухаються по сітці, змінюючи своє положення з часом, зберігаючи при цьому свою форму);
- гармати (ці патерни неодноразово створюють кораблі та інші патерни).

5.1.1 Block

Block — це стійкий патерн в “Грі життя” Конвея, що означає стабільну конфігурацію з чотирьох клітин, яка не змінюється з часом (рис. 5.1).

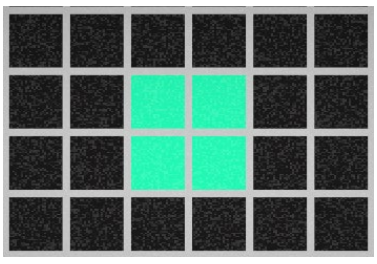

Рисунок 5.1 – Block патерн

Це дуже поширений патерн, утворений квадратом живих клітин 2x2, і він примітний тим, що є єдиним патерном, де всі живі клітини мають трьох сусідів. Його невеликий розмір та стабільність роблять його фундаментальним будівельним блоком для побудови складніших патернів. Він також використовується як каталізатор у реакціях, і ним можна маніпулювати за допомогою інших патернів, таких як Glider.

5.1.2 Blinker

Blinker — це простий 3-клітинний патерн у “Грі життя” Конвея, який коливається, або блимає, між вертикальним і горизонтальним станом кожні два покоління (рис. 5.2).

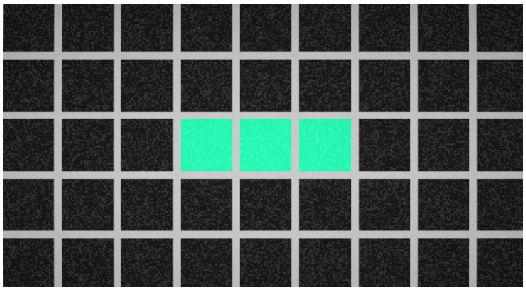

Рисунок 5.2 – Blinker патерн

Це найпоширеніший осцилятор, що складається з лінії трьох клітин, орієнтація яких змінюється з кожним поколінням, причому клітини вмирають і народжуються відповідно до правил гри.

5.1.3 Glider

Glider — це найменший і найпоширеніший патерн, який рухається по діагоналі по сітці, повторюючи свою форму кожні чотири покоління, зміщуючи при цьому одну клітинку вниз і одну клітинку праворуч (рис. 5.3).

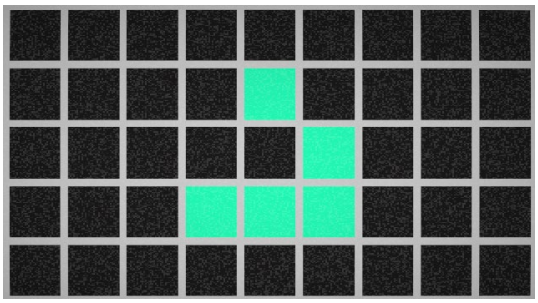

Рисунок 5.3 – Glider патерн

Його відкрив Річард Гай у 1970 році, коли вивчав R-pentomino, перший патерн для створення планерів. Планери можуть взаємодіяти з іншими патернами, і існує багато складних патернів, які їх використовують.

5.1.4 LWSS

LWSS, або легкий космічний корабель, — це патерн космічного корабля, який рухається по діагоналі по сітці, просуваючись на 4 клітинки за 14 поколінь, або на 2 клітинки за діагональне покоління (рис. 5.4).

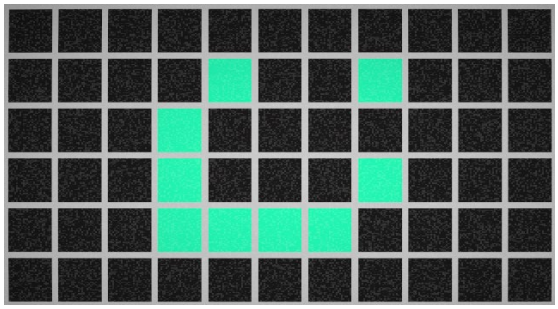

Рисунок 5.4 – LWSS патерн

Це найменший і найпоширеніший патерн рухомого космічного корабля, який можна створити з інших простих патернів, таких як планери. Патерн LWSS має унікальну хвостову іскру, яка може впливати на інші об’єкти. Також його можна синтезувати з інших патернів, таких як три планери або комбінація двох планерів та ще одного невеликого об’єкта.

5.1.5 Toad

Toad — це простий коливальний патерн із шести клітин, який циклічно перемикається між двома конфігураціями кожні два покоління (рис. 5.5).

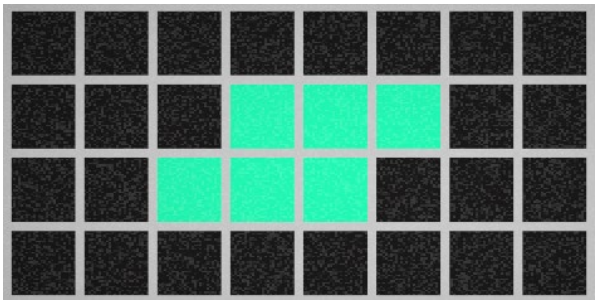

Рисунок 5.5 – Toad патерн

Це поширений приклад осцилятора, який періодично повторює свій стан. Toad також можна використовувати як компонент у більших, складніших патернах, таких як одноразовий поворотник для планерів або як частину “жаб’ячого перевертання”, який перевертає патерн до нової конфігурації.

5.1.6 Acorn

Acorn — це маленький семиклітинний патерн, відомий як один із найменших “мафусаїл” (рис. 5.6).

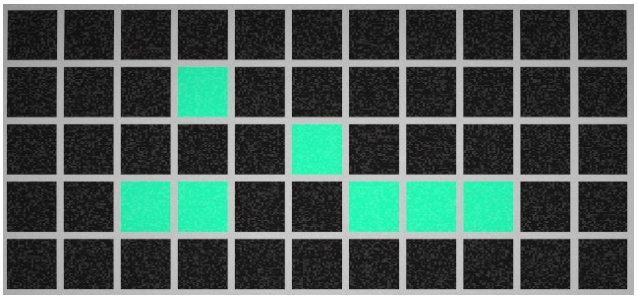

Рисунок 5.6 – Acorn патерн

Мафусаїли — це патерни, для стабілізації яких потрібен дуже тривалий час, і Acorn потрібно 5206 поколінь, щоб досягти стабільного стану з 633 клітин. Протягом своєї довгої еволюції він виробляє різні складні взаємодії, включаючи 13 планерів.

5.1.7 Pulsar

Pulsar — це великий, добре відомий осцилятор, який повертається до свого початкового стану після трьох поколінь (рис. 5.7).

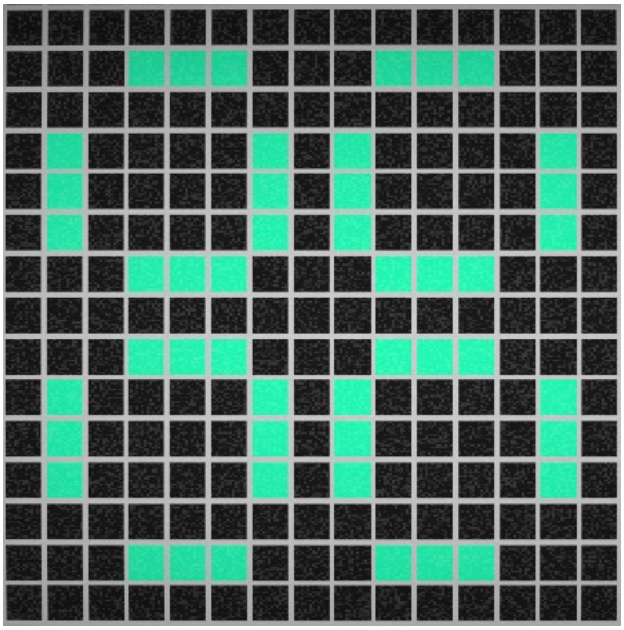

Рисунок 5.7 – Pulsar патерн

Це найпоширеніший осцилятор з періодом більше двох. Патерн створюється сіткою 13x13 комірок, які розширюються та стискаються, створюючи пульсуючий ефект, і складається з двох половин, які пульсують у протилежних напрямках.

5.2 Виживання патернів у випадковому середовищі

У першому експерименті було проведено симуляцію великої кількості випадкових початкових станів поля, що дозволило визначити, які саме структури

найчастіше з’являються та зберігаються протягом тривалого часу (табл. 5.1). Кожна випадкова конфігурація генерувалась як поле розміру 50 на 50 клітин із заданою початковою ймовірністю заповнення (p = 0.25). Загалом було проведено 100 симуляцій, тривалістю в 500 поколінь кожна.

Таблиця 5.1 – Частота появи патернів у випадкових ініціалізаціях

Характеристика / Патерн	Blinker	Toad	Beacon	Glider	LWSS
Частота виявлення (%)	82%	37%	19%	12%	3%
Середня тривалість існування (поколінь)	45	28	55	120	160

Аналіз показав, що попри початкову хаотичність, система швидко еволюціонує до стабільного набору повторюваних структур. Значна частина випадкових початкових станів зазвичай проходить через кілька фаз: від хаотичного “вибухового” зростання на перших кроках до формування більш впорядкованих регіонів, де присутні осцилятори, кораблі та стабільні стаціонарні об’єкти.

У більшості симуляцій спостерігався поділ поля на невеликі сектори, всередині яких залишались прості стабільні структури – наприклад, блоки та вулики, – а також типові короткоперіодичні осцилятори, такі як Blinker чи Toad. Набагато рідше в хаосі формувалися “кораблі”, але якщо вони виникали, то здебільшого продовжували існувати тривалий час, часом зіштовхуючись із іншими структурами та повністю перетворюючи локальну частину поля.

Ці спостереження узгоджуються з класичною поведінкою “Гри Життя”: система не зберігає хаос, а навпаки прагне до фрагментації на незалежні локальні структури. Таким чином, експеримент підтвердив відому властивість клітинного автомата – тенденцію до стабілізації з появою універсального набору елементарних фігур, які є фундаментальними “атомами” в подальшій еволюції.

5.3 Зміна кількості живих клітин у часі

Другий експеримент був присвячений дослідженню глобальної динаміки системи. Симуляція запускалася для кількох типів початкових конфігурацій: структурованих патернів (Pulsar, Beacon, LWSS), випадкових полів із різними густинами та комбінацій кількох патернів одночасно. Для кожної симуляції записувалася кількість живих клітин у кожному поколінні, що дозволило отримати числові характеристики процесу (табл. 5.2).

Таблиця 5.2 – Амплітуда змін кількості живих клітин

Характеристика / Патерн	Pulsar	Beacon	Випадкова ініціалізація
Мін. кількість клітин	12	6	200–250
Макс. кількість клітин	36	8	900–1200
Характер зміни	Чітка періодичність	Низька амплітуда	Хаотичний вибух

Було виявлено, що динаміка зміни кількості клітин дуже залежить від початкової впорядкованості та густини. Наприклад, Pulsar демонструє чітко виражені циклічні коливання з довжиною періоду 3. Це створює яскравий ритмічний малюнок, у якому кількість клітин змінюється за повторюваною схемою. У випадку Beacon коливання менш виражені та мають амплітуду, значно меншу за Pulsar, проте система також демонструє стабільну періодичність.

Цілком іншу картину дають випадкові ініціалізації: кількість клітин різко зростає в перших поколіннях через численні можливі варіанти взаємодії сусідніх клітин, проте через 20–50 кроків дерево розвитку поступово “виснажується”. Система втрачає значну частину клітин унаслідок конфліктів та перенаселення, і надалі кількість стабілізується на невеликому рівні.

У складних комбінаціях патернів спостерігалися цікаві феномени: кораблі, що рухаються через поле, змінювали форму локальних регіонів, порушуючи

стабільність осциляторів та утворюючи нові осцилюючі або стабільні фігури. Такі симуляції дають змогу побачити, як взаємодія окремих структур створює складні емерджентні ефекти, що важко передбачити без проведення експерименту.

Таким чином, другий експеримент показав, що еволюція кількості живих клітин у часі є важливим індикатором того, наскільки “енергійною” є конфігурація, а також дозволяє відрізнити хаотичні системи від систем із чіткою періодичною поведінкою.

5.4 Середня тривалість життя патернів

У третьому експерименті було визначено, скільки часу живуть певні класичні конфігурації до переходу в стаціонарний стан або до повного зникнення. Для кожного патерна було проведено 100 запусків із випадковим оточенням навколо (декілька випадкових клітин на відстані), що дозволило дослідити поведінку в більш реалістичних умовах, а не в ізольованій формі (табл. 5.3).

Таблиця 5.3 – Середня тривалість життя патернів у випадковому середовищі

Характеристика / Патерн	Block	Blinker	Toad	Pulsar	Glider	LWSS
Середня тривалість життя (поколінь)	85	120	70	40	95	110
Характер стійкості	Дуже стійкий, але чутливий до збурень	Найстійкіший осцилятор	Часто руйнується зовнішніми збуреннями	Висока чутливість	Залежить від зіткнень	Стійкіший за Glider

Середня тривалість життя стабільних фігур, таких як Block або Toad, фактично необмежена за умови відсутності зовнішніх збурень. Але навіть мінімальне втручання хаотичних елементів поблизу може запускати ланцюгові

реакції, які руйнують структуру. Так, Block, попри свою стабільність, може бути зруйнований уже через 10–15 поколінь, якщо в початковому оточенні є хоча б один активний осцилятор або фрагмент рухомої фігури.

Осцилятори демонстрували цікавіші результати. Blinker у середньому живе довше за інші осцилятори, оскільки його симетрична структура стійка до випадкових взаємодій. Pulsar виявився високочутливим до оточення: його великі "обертові" поля легко збурюються, через що структура або руйнується, або трансформується в менші осцилятори.

Рухомі патерни, такі як Glider та LWSS, показали найбільш варіативні результати. У деяких запусках кораблі рухались десятки поколінь, залишаючись повністю неушкодженими, тоді як в інших — зіштовхувалися з перешкодами вже на перших кроках. Це підтверджує відомий факт: рухомі структури значно залежні від середовища й особливо від наявності "сміття" — дрібних нестандартних конфігурацій хаотичного походження.

Загалом третій експеримент показав, що тривалість життя патернів не можна оцінювати лише виходячи з їхньої теоретичної стабільності. Навіть досконало відомі структури в реалістичному середовищі виявляють складну та непередбачувану поведінку.

5.5 Висновок

У цьому розділі було розглянуто декілька найпопулярніших патернів, кожен з яких має свою структуру та поведінку в різних умовах симуляції.

Були проведені три експерименти — масові запуски випадкових початкових конфігурацій, аналіз динаміки зміни кількості живих клітин та визначення середньої тривалості життя окремих патернів. Результати експериментів дозволяють отримати узагальнене уявлення про характер еволюції системи в "Грі Життя" Конвея та про поведінку різних структур.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи магістра було розроблено комп’ютерну симуляцію “Гра Життя” Конвея.

Було проведено аналіз предметної області, визначені ключові терміни, пов’язані з нею, а також з об’єктом дослідження й розробки. Розглянуті основні клітинні автомати. Визначені основні правила, за якими має відбуватися симуляція.

Проведено огляд та відбір інструментів розробки програмного забезпечення. Також був проведений порівняльний аналіз мов програмування, середовищ розробки та ігрових рушіїв. Обрані інструменти вдало підійшли для швидкої розробки симуляції.

Була розроблена проста структура проєкту та організован робочий процес, який дозволяє легко і швидко додавати або видаляти нові елементи системи. Створено основні класи, кожен з яких має своє призначення. Розроблено основне меню, яке забезпечує зручний інтерфейс для перемикання між вікнами. Реалізовано основний режим симуляції, де користувач може переміщатись по сітці та створювати або видаляти патерни, а потім проходитись по поколінням симуляції.

Описана невелика інструкція щодо запуску комп’ютерної симуляції. Також описано основний функціонал програми та проведено детальний огляд кожного аспекту її використання, включаючи зображення основних дій за допомогою скріншотів.

Було проведено огляд відомих патернів, та проведено три експерименти з ними. Дослідження цих патернів продемонструвало їх характер та поведінку в різних умовах симуляції. Результати підтверджують, що “Гра Життя” Конвея є системою, здатною проявляти як хаотичну, так і впорядковану поведінку, але тривалі стабільні динамічні процеси зазвичай є винятком. Система схильна до самоорганізації та переходу від хаотичних конфігурацій до стійких патернів. При цьому складні довготривалі структури або рушії потребують спеціально побудованих початкових умов і майже ніколи не виникають випадково.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Cellular automaton [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Cellular_automaton

2. Elementary cellular automaton [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Elementary_cellular_automaton

3. Listening to Elementary Cellular Automata [Electronic resource]. - Access mode: https://medium.com/code-music-noise/listening-to-elementary-cellular-automata-661018229362

4. Langton’s Ant & Musical Patterning [Electronic resource]. - Access mode: https://medium.com/code-music-noise/langtons-ant-musical-patterning-i-6a11176d7fed

5. Conway's Game of Life [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

6. Conway’s Game of Life [Electronic resource]. - Access mode: https://medium.com/@Sash0k/conways-game-of-life-3555ecbf4d13

7. Game engine [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Game_engine

8. Technical Overview of Unity Game Engine [Electronic resource]. - Access mode: https://www.pubnub.com/guides/unity

9. What Is Unreal Engine [Electronic resource]. - Access mode: https://www.coursera.org/articles/what-is-unreal-engine

10. Godot (game engine) [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Godot_(game_engine)

11. GDScript Basics [Electronic resource]. - Access mode: https://www.polycode.tech/beginners-guide-to-godot-engine-scripting

12. A tour of the C# language [Electronic resource]. - Access mode: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview

13. C++ [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/C%2B%2B

14. What is an IDE [Electronic resource]. - Access mode: https://aws.amazon.com/what-is/ide/?nc1=h_ls

15. Qt Creator [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Qt_Creator

16. JetBrains' CLion [Electronic resource]. - Access mode: https://medium.com/@AlexanderObregon/jetbrains-clion-a-powerful-tool-for-modern-c-and-c-development-1b346f926dfe

17. What is Visual Studio [Electronic resource]. - Access mode: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022

18. How to structure a game project [Electronic resource]. - Access mode: https://www.code-spot.co.za/2020/11/13/how-to-structure-a-game-project

19. What is GDExtension [Electronic resource]. - Access mode: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html

20. Software Development with CMake [Electronic resource]. - Access mode: https://cmake.org/about

21. Code completion [Electronic resource]. - Access mode: https://en.wikipedia.org/wiki/Code_completion

22. Game Loop [Electronic resource]. - Access mode: https://gameprogrammingpatterns.com/game-loop.html

23. Service Locator [Electronic resource]. - Access mode: https://gameprogrammingpatterns.com/service-locator.html

24. Introduction to shaders [Electronic resource]. - Access mode: https://docs.godotengine.org/en/stable/tutorials/shaders/introduction_to_shaders.html

ДОДАТОК А

Текст програми

A.1 Файл CMakeLists.txt

```
# Minimum CMake version
cmake_minimum_required(VERSION 4.1.1)



# Add godot-cpp library
add_subdirectory("${CMAKE_SOURCE_DIR}/Binding"
"${CMAKE_BINARY_DIR}/Binding")



# Module data
project(
    "Custom Module"
    VERSION 1.0.0
    LANGUAGES CXX
    DESCRIPTION "This is custom Godot module made by Yaroslav Kolokol."
    HOMEPAGE_URL ""
)



# C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)


# Module source files
file(GLOB_RECURSE GD_SOURCE *.cpp)
list(FILTER GD_SOURCE INCLUDE REGEX "${CMAKE_SOURCE_DIR}/Source")
set(ENV{GD_MODULE_SOURCE} "${GD_SOURCE}")



# Debug vs Release
set(TEMP_NAME $ENV{GD_MODULE_NAME})
if("${CMAKE_BUILD_TYPE}" STREQUAL "Debug")
    set(ENV{GD_MODULE_NAME} "${TEMP_NAME}Debug")
else()
```

```
    set(ENV{GD_MODULE_NAME} "${TEMP_NAME}Release")
endif()



# Module
add_library(
    "$ENV{GD_MODULE_NAME}"
    SHARED
)


# Source
target_sources(
    "$ENV{GD_MODULE_NAME}"
    PUBLIC
    "$ENV{GD_MODULE_SOURCE}"
)


# Include
target_include_directories(
    "$ENV{GD_MODULE_NAME}"
    PUBLIC
    "$ENV{GD_MODULE_SOURCE_PATH}"
)


# Link godot-cpp
target_link_libraries(
    "$ENV{GD_MODULE_NAME}"
    PUBLIC
    "godot-cpp"
)
```

A.2 Файл Simulation.hpp

```
#ifndef SIMULATION_HPP
#define SIMULATION_HPP
```

```
#include "Core/Declaration.hpp"
#include "Global/Locator.hpp"

#include <godot_cpp/classes/node.hpp>


namespace godot
{

    class Simulation : public Node
    {

        GDCLASS(Simulation, Node);

    private:

        Node* interface{ nullptr };
        Node* playground{ nullptr };


        Node* player{ nullptr };
        Node* renderer{ nullptr };

    protected:

        static void _bind_methods();

    public:

        void _enter_tree() override;
        void _exit_tree() override;

        void exit();

    };

}

#define GD_SIMULATION_NAME "Simulation"
#define GD_SIMULATION GD_LOCATOR->get_data<Simulation>(GD_SIMULATION_NAME)
```

```
#endif
```

A.3 Файл Simulation.cpp

```
#include "Simulation.hpp"


#include "Memory/Memory.hpp"
#include "Renderer.hpp"
#include "Player.hpp"
#include "Interface.hpp"
#include "Playground.hpp"


#include <godot_cpp/classes/scene_tree.hpp>
#include <godot_cpp/classes/world_environment.hpp>
#include <godot_cpp/classes/environment.hpp>



namespace godot
{

    void Simulation::_bind_methods()
    {

        print_line("Binding methods in ", typeid(Simulation).name());



    }

    void Simulation::_enter_tree()
    {

        print_line("Starting simulation");


        set_name(GD_SIMULATION_NAME);
        GD_LOCATOR->set_data(GD_SIMULATION_NAME, this);
```

```
        interface = GD_NEW(Interface);
        interface->set_name(GD_INTERFACE_NAME);
        GD_LOCATOR->set_data(GD_INTERFACE_NAME, interface);


        playground = GD_NEW(Playground);
        playground->set_name(GD_PLAYGROUND_NAME);
        GD_LOCATOR->set_data(GD_PLAYGROUND_NAME, playground);



        add_child(interface);
        add_child(playground);


        GD_INTERFACE->create_menu();



    }

    void Simulation::_exit_tree()
    {







        print_line("Ending simulation");

    }

    void Simulation::exit()
    {

        get_tree()->quit();

    }

}
```

A.4 Файл Player.hpp

```
#ifndef PLAYER_HPP
#define PLAYER_HPP

#include "Core/Declaration.hpp"

#include <godot_cpp/classes/node.hpp>
#include <godot_cpp/classes/input_event.hpp>
#include <godot_cpp/classes/canvas_texture.hpp>
#include <godot_cpp/classes/shader_material.hpp>
#include <godot_cpp/classes/shader.hpp>
#include <godot_cpp/classes/sprite2d.hpp>


namespace godot
{

    class Player : public Node
    {

        GDCLASS(Player, Node);

    private:

        Ref<CanvasTexture> background_Data{};
        Ref<ShaderMaterial> shader_Material{};
        Ref<Shader> shader{};

        CharacterBody2D *body{ nullptr };
        CollisionShape2D* shape{ nullptr };
        Camera2D *camera{ nullptr };
        Sprite2D* background{ nullptr };

        float speed{ 500.0f };
        float zoom_speed{ 1.0f };

    protected:

        static void _bind_methods();

    public:
```

```
        void _enter_tree() override;
        void _exit_tree() override;

        void _physics_process(double delta) override;
        void _process(double delta) override;
        void _input(const Ref<InputEvent>& event) override;

    };

}


#endif
```

A.5 Файл Player.cpp

```
#include "Player.hpp"

#include "Memory/Memory.hpp"
#include "Input/Input.hpp"
#include "Resource/Resource_Loader.hpp"

#include <godot_cpp/classes/character_body2d.hpp>
#include <godot_cpp/classes/collision_shape2d.hpp>
#include <godot_cpp/classes/camera2d.hpp>
#include <godot_cpp/classes/input_event_key.hpp>


namespace godot
{

    void Player::_bind_methods()
    {

        print_line("Binding methods in ", typeid(Player).name());

    }

    void Player::_enter_tree()
    {
```

```
        background_Data.instantiate();
        shader = GD_RESOURCE_LOADER->load("res://Life/Background.gdshader");
        shader_Material.instantiate();
        shader_Material->set_shader(shader);
        shader_Material->set_shader_parameter("colour_1", Color{ 0.4, 0.4,
0.4, 1.0 });
        shader_Material->set_shader_parameter("colour_2", Color{ 0.0, 0.0,
0.0, 1.0 });
        shader_Material->set_shader_parameter("colour_3", Color{ 0.2, 0.2,
0.2, 1.0 });
        shader_Material->set_shader_parameter("spin_speed", 5);
        shader_Material->set_shader_parameter("pixel_filter", 2000);


        body = GD_NEW(CharacterBody2D);
        shape = GD_NEW(CollisionShape2D);
        camera = GD_NEW(Camera2D);
        background = GD_NEW(Sprite2D);

        body->set_name("Body");
        shape->set_name("Shape");
        camera->set_name("Camera");
        background->set_name("Background");

        add_child(body);
        body->add_child(shape);
        body->add_child(camera);
        body->add_child(background);


        camera->set_zoom(Vector2{ 3, 3 });
        background->set_scale(Vector2{ 1000, 1000 });
        background->set_texture(background_Data);
        background->set_material(shader_Material);

    }

    void Player::_exit_tree()
    {
```

```
}

void Player::_physics_process(double delta)
{

    body->set_velocity(GD_INPUT->get_vector("Left",    "Right",    "Up",
"Down") * speed);
    body->move_and_slide();

}

void Player::_process(double delta)
{




}

void Player::_input(const Ref<InputEvent>& event)
{

    Ref<InputEventKey> input_Event_Key = event;


    if (input_Event_Key.is_valid() == true)
    {

        if (input_Event_Key->is_pressed() == true)
        {

            if (input_Event_Key->get_keycode() == Key::KEY_Q)
            {

                camera->set_zoom(camera->get_zoom() * 1.1f);
                background->set_scale(background->get_scale() / 1.1f);

            }
            else if (input_Event_Key->get_keycode() == Key::KEY_E)
            {

                camera->set_zoom(camera->get_zoom() / 1.1f);
```

```
                    background->set_scale(background->get_scale() * 1.1f);

                }

            }

        }

    }

}
```

A.6 Файл Playground.hpp

```
#ifndef PLAYGROUND_HPP
#define PLAYGROUND_HPP


#include "Core/Declaration.hpp"
#include "Global/Locator.hpp"

#include <godot_cpp/classes/node.hpp>
#include <godot_cpp/classes/texture2d.hpp>
#include <godot_cpp/classes/sprite2d.hpp>
#include <godot_cpp/classes/tile_set_atlas_source.hpp>
#include <godot_cpp/classes/tile_set.hpp>
#include <godot_cpp/classes/input_event.hpp>
#include <godot_cpp/templates/hash_set.hpp>



namespace godot
{

    enum class Pattern
    {

        CELL,
        BLOCK,
        BLINKER,
        GLIDER,
```

```
    BEACON,
    TUB,
    R_PENTOMINO,
    ACORN,
    QUEEN_BEE_SHUTTLE

};

class Player;

class Playground : public Node
{

    GDCLASS(Playground, Node);

private:

    int32_t grid_Size{ 0 };
    int32_t population_Size{ 0 };
    int32_t generation_Count{ 0 };
    double fps_Count{ 0.0 };

    int32_t source_ID{ 0 };
    bool is_Playing{ false };

    Pattern current_Pattern{ Pattern::CELL };

    Ref<Texture2D> cell{};
    Ref<TileSetAtlasSource> source{};
    Ref<TileSet> set{};

    HashSet<Vector2i> generation1{};
    HashSet<Vector2i> generation2{};

protected:

    static void _bind_methods();

public:

    void _enter_tree() override;
    void _exit_tree() override;
```

```
        void _input(const Ref<InputEvent>& event) override;
        void _process(double delta) override;

        void create();
        void destroy();

        TileMapLayer* get_grid();
        TileMapLayer* get_pattern_grid();
        Player* get_player();

        int32_t get_grid_size() const;
        int32_t get_population_size() const;
        int32_t get_generation_count() const;
        double get_fps_count() const;

        int32_t get_source_id() const;
        bool get_is_playing();

        void set_grid_size(int32_t value);

        void increase_grid_size();
        void decrease_grid_size();
        bool in_grid(Vector2i coordinates);

        int32_t get_cell_status(Vector2i coordinates);
        int32_t get_cell_live_neighbors_amount(Vector2i coordinates);
        void step();

        void select_pattern(int32_t index);
        Vector<Vector2i> get_pattern_data(Vector2i coordinates);

        void create_cell(Vector2i coordinates);
        void destroy_cell(Vector2i coordinates);

    };

}

#define GD_PLAYGROUND_NAME "Playground"
#define GD_PLAYGROUND GD_LOCATOR->get_data<Playground>(GD_PLAYGROUND_NAME)
```

```
#endif
```

A.7 Файл Playground.cpp

```
#include "Playground.hpp"


#include "Memory/Memory.hpp"
#include "Engine/Engine.hpp"
#include "Resource/Resource_Loader.hpp"

#include "Player.hpp"

#include <godot_cpp/classes/tile_map_layer.hpp>
#include <godot_cpp/classes/tile_data.hpp>


#include <godot_cpp/classes/input_event_mouse_button.hpp>
#include <godot_cpp/classes/input_event_key.hpp>
#include <godot_cpp/classes/input_event_mouse_motion.hpp>


namespace godot
{

    void Playground::_bind_methods()
    {

        print_line("Binding methods in ", typeid(Playground).name());

    }

    void Playground::_enter_tree()
    {

        cell = GD_RESOURCE_LOADER->load("res://Life/Cell.png");


        source.instantiate();
        source->set_name("Cell Source");
```

```
    source->set_texture(cell);
    source->set_texture_region_size(Vector2i{ 16, 16 });
    source->create_tile(Vector2i{ 0, 0 });
    source->create_tile(Vector2i{ 1, 0 });
    source->create_tile(Vector2i{ 0, 1 });
    source->create_tile(Vector2i{ 1, 1 });


    set.instantiate();
    set->set_name("Cell Set");
    set->set_tile_size(Vector2i{ 16, 16 });
    set->set_tile_shape(TileSet::TILE_SHAPE_SQUARE);
    set->set_uv_clipping(true);
    set->add_custom_data_layer(0);
    set->set_custom_data_layer_name(0, "Status");
    set->set_custom_data_layer_type(0, Variant::INT);
    set->add_custom_data_layer(1);
    set->set_custom_data_layer_name(1, "Future");
    set->set_custom_data_layer_type(1, Variant::INT);
    source_ID = set->add_source(source);



}

void Playground::_exit_tree()
{



}

void Playground::_input(const Ref<InputEvent>& event)
{

    Ref<InputEventMouseButton> input_Event_Mouse_Button = event;
    Ref<InputEventKey> input_Event_Key = event;
    Ref<InputEventMouseMotion> input_Event_Mouse_Motion = event;


    if (input_Event_Mouse_Motion.is_valid() == true)
    {
```

```
                    Vector2       position       =       input_Event_Mouse_Motion-
>get_global_position();

                }

                if (input_Event_Mouse_Button.is_valid() == true && is_Playing ==
true)
                {

                    if (input_Event_Mouse_Button->is_pressed() == true)
                    {

                        if     (input_Event_Mouse_Button->get_button_index()     ==
MouseButton::MOUSE_BUTTON_MIDDLE)
                        {

                            TileMapLayer* grid = get_grid();
                            Vector2i    coordinates    =    grid->local_to_map(grid-
>get_local_mouse_position());
                            Vector<Vector2i> array = get_pattern_data(coordinates);

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

                                create_cell(array[i]);
                                if      (in_grid(array[i])      ==      true      &&
generation2.has(array[i]) == false)
                                {
                                    generation2.insert(array[i]);
                                }

                            }

                        }
                        else  if  (input_Event_Mouse_Button->get_button_index()  ==
MouseButton::MOUSE_BUTTON_RIGHT)
                        {

                            TileMapLayer* grid = get_grid();
                            Vector2i    coordinates    =    grid->local_to_map(grid-
>get_local_mouse_position());
```

```
                        Vector<Vector2i> array = get_pattern_data(coordinates);

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

                            destroy_cell(array[i]);
                            if      (in_grid(array[i])      ==      true      &&
generation2.has(array[i]) == true)
                            {
                                generation2.erase(array[i]);
                            }

                        }

                    }

                }

                if (input_Event_Key.is_valid() == true && is_Playing == true)
                {

                    if (input_Event_Key->is_pressed() == true)
                    {

                        if (input_Event_Key->get_keycode() == Key::KEY_SPACE)
                        {

                            step();

                        }

                    }

                }

            }

            void Playground::_process(double delta)
            {
```

```
            fps_Count = GD_ENGINE->get_frames_per_second();

            if (is_Playing == true)
            {

                TileMapLayer* grid = get_grid();
                TileMapLayer* pattern_Grid = get_pattern_grid();
                pattern_Grid->clear();

                Vector2i coordinates = pattern_Grid->local_to_map(pattern_Grid-
>get_local_mouse_position());


                Vector<Vector2i> array = get_pattern_data(coordinates);


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

                    if (in_grid(array[i]) == true)
                    {
                        pattern_Grid->set_cell(array[i], source_ID, Vector2i{ 1,
1 });
                    }

                }

            }

        }

        void Playground::create()
        {

            print_line("Creating the Playground");


            Player* player = GD_NEW(Player);
            player->set_name("Player");
            add_child(player);

            TileMapLayer* grid = GD_NEW(TileMapLayer);
```

```
    grid->set_name("Grid");
    grid->set_tile_set(set);
    add_child(grid);

    TileMapLayer* pattern_Grid = GD_NEW(TileMapLayer);
    pattern_Grid->set_name("Pattern Grid");
    pattern_Grid->set_tile_set(set);
    add_child(pattern_Grid);


    is_Playing = true;

}

void Playground::destroy()
{

    print_line("Destroying the Playground");


    Player* player = get_player();
    TileMapLayer* grid = get_grid();
    TileMapLayer* pattern_Grid = get_pattern_grid();


    remove_child(grid);
    GD_DELETE(grid);
    remove_child(pattern_Grid);
    GD_DELETE(pattern_Grid);
    remove_child(player);
    GD_DELETE(player);


    grid_Size = 0;
    population_Size = 0;
    generation_Count = 0;
    fps_Count = 0.0;
    is_Playing = false;
    current_Pattern = Pattern::CELL;

}
```

```
Player* Playground::get_player()
{
    return Object::cast_to<Player>(get_child(0));
}

TileMapLayer* Playground::get_grid()
{
    return Object::cast_to<TileMapLayer>(get_child(1));
}

TileMapLayer* Playground::get_pattern_grid()
{
    return Object::cast_to<TileMapLayer>(get_child(2));
}

int32_t Playground::get_grid_size() const
{
    return grid_Size;
}

int32_t Playground::get_population_size() const
{
    return population_Size;
}

int32_t Playground::get_generation_count() const
{
    return generation_Count;
}

double Playground::get_fps_count() const
{
    return fps_Count;
}

int32_t Playground::get_source_id() const
{
    return source_ID;
}

bool Playground::get_is_playing()
{
```

```
    return is_Playing;
}

void Playground::set_grid_size(int32_t value)
{

    TileMapLayer* grid = get_grid();

    grid->clear();
    grid_Size = value;

    for (int i{ 0 }; i < grid_Size; ++i)
    {

        for (int j{ 0 }; j < grid_Size; ++j)
        {

            grid->set_cell(Vector2i{ i, j }, source_ID, Vector2i{ 1, 0
});
            grid->get_cell_tile_data(Vector2i{      i,      j      })-
>set_custom_data("Status", 0);
            grid->get_cell_tile_data(Vector2i{      i,      j      })-
>set_custom_data("Future", 0);

        }

    }

}

void Playground::increase_grid_size()
{

    set_grid_size(get_grid_size() + 1);
    population_Size = 0;

}

void Playground::decrease_grid_size()
{

    set_grid_size(get_grid_size() - 1);
```

```
        population_Size = 0;

    }

    bool Playground::in_grid(Vector2i coordinates)
    {

        TileMapLayer* grid = get_grid();

        return grid->get_cell_source_id(coordinates) != -1;
    }

    int32_t Playground::get_cell_status(Vector2i coordinates)
    {

        TileMapLayer* grid = get_grid();
        int32_t status = 0;

        if (in_grid(coordinates) == true)
        {

            status = grid->get_cell_tile_data(coordinates)->get_custom_data("Status");
            int32_t live_Neighbors = get_cell_live_neighbors_amount(coordinates);


            if (status == 1 && live_Neighbors < 2)
            {
                status = 0;
            }
            else if (status == 1 && (live_Neighbors == 2 || live_Neighbors == 3))
            {
                status = 1;
            }
            else if (status == 1 && live_Neighbors > 3)
            {
                status = 0;
            }
            else if (status == 0 && live_Neighbors == 3)
            {
```

```
                status = 1;
            }

        }


        return status;
    }

    int32_t Playground::get_cell_live_neighbors_amount(Vector2i coordinates)
    {

        TileMapLayer* grid = get_grid();
        int32_t live_Neighbors = 0;
        TypedArray<Vector2i> neighbors
        {
            Vector2i{1, 0},
            Vector2i{-1, 0},
            Vector2i{0, 1},
            Vector2i{0, -1},
            Vector2i{1, 1},
            Vector2i{1, -1},
            Vector2i{-1, 1},
            Vector2i{-1, -1}
        };

        for (int l = 0; l < neighbors.size(); ++l)
        {

            Vector2i coordinates_Neighbor = coordinates + neighbors[l];

            if (in_grid(coordinates_Neighbor) == true)
            {

                int32_t           status_Neighbor           =           grid-
>get_cell_tile_data(coordinates_Neighbor)->get_custom_data("Status");

                if (status_Neighbor == 1)
                {
                    ++live_Neighbors;
                }
```

```
                }

            }

            return live_Neighbors;
        }

        void Playground::step()
        {

            TileMapLayer* grid = get_grid();

            generation_Count++;

            generation1.clear();
            generation1 = generation2;
            generation2.clear();

            TypedArray<Vector2i> neighbors
            {
                Vector2i{1, 0},
                Vector2i{-1, 0},
                Vector2i{0, 1},
                Vector2i{0, -1},
                Vector2i{1, 1},
                Vector2i{1, -1},
                Vector2i{-1, 1},
                Vector2i{-1, -1}
            };


            for (auto iterator = generation1.begin(); iterator !=
generation1.end(); ++iterator)
            {

                Vector2i coordinates = *iterator;
                //print_line("c = ", coordinates);
                //print_line("s = ", grid->get_cell_tile_data(coordinates)-
>get_custom_data("Status"));
                //print_line("f = ", get_cell_status(coordinates));
                //print_line("l                    =                    ",
get_cell_live_neighbors_amount(coordinates));
```

```
                    if       (get_cell_status(coordinates)       ==       1       &&
generation2.has(coordinates) == false)
                    {
                        generation2.insert(coordinates);
                    }
                    for (int l = 0; l < neighbors.size(); ++l)
                    {

                        Vector2i coordinates_Neighbor = coordinates + neighbors[l];
                        if    (get_cell_status(coordinates_Neighbor)    ==    1    &&
generation2.has(coordinates_Neighbor) == false)
                        {
                            generation2.insert(coordinates_Neighbor);
                        }

                    }

                    //print_line("g1 = ", coordinates);
                    //print_line("s1 = ", get_cell_status(coordinates));

                }

                for   (auto   iterator   =   generation1.begin();   iterator   !=
generation1.end(); ++iterator)
                {

                    Vector2i coordinates = *iterator;
                    destroy_cell(coordinates);
                    //print_line("g1 = ", coordinates);
                }

                for   (auto   iterator   =   generation2.begin();   iterator   !=
generation2.end(); ++iterator)
                {

                    Vector2i coordinates = *iterator;
                    create_cell(coordinates);
                    //print_line("g2 = ", coordinates);
                }

            }
```

```
void Playground::select_pattern(int32_t index)
{

    current_Pattern = static_cast<Pattern>(index);

}

Vector<Vector2i> Playground::get_pattern_data(Vector2i coordinates)
{

    Vector<Vector2i> data{};

    if (current_Pattern == Pattern::CELL)
    {

        data.push_back(coordinates + Vector2i{ 0, 0 });

    }
    else if (current_Pattern == Pattern::BLOCK)
    {

        data.push_back(coordinates + Vector2i{ 0, 0 });
        data.push_back(coordinates + Vector2i{ 1, 0 });
        data.push_back(coordinates + Vector2i{ 0, 1 });
        data.push_back(coordinates + Vector2i{ 1, 1 });

    }
    else if (current_Pattern == Pattern::BLINKER)
    {

        data.push_back(coordinates + Vector2i{ 0, 0 });
        data.push_back(coordinates + Vector2i{ 1, 0 });
        data.push_back(coordinates + Vector2i{ 2, 0 });

    }
    else if (current_Pattern == Pattern::GLIDER)
    {

        data.push_back(coordinates + Vector2i{ 0, 0 });
        data.push_back(coordinates + Vector2i{ 1, 1 });
        data.push_back(coordinates + Vector2i{ 1, 2 });
```

```
    data.push_back(coordinates + Vector2i{ 0, 2 });
    data.push_back(coordinates + Vector2i{ -1, 2 });

}
else if (current_Pattern == Pattern::BEACON)
{

    data.push_back(coordinates + Vector2i{ 0, 0 });
    data.push_back(coordinates + Vector2i{ 1, 0 });
    data.push_back(coordinates + Vector2i{ 0, 1 });
    data.push_back(coordinates + Vector2i{ 1, 1 });
    data.push_back(coordinates + Vector2i{ 2, 2 });
    data.push_back(coordinates + Vector2i{ 3, 2 });
    data.push_back(coordinates + Vector2i{ 2, 3 });
    data.push_back(coordinates + Vector2i{ 3, 3 });

}
else if (current_Pattern == Pattern::TUB)
{

    data.push_back(coordinates + Vector2i{ 0, 0 });
    data.push_back(coordinates + Vector2i{ 1, 1 });
    data.push_back(coordinates + Vector2i{ -1, 1 });
    data.push_back(coordinates + Vector2i{ 0, 2 });

}
else if (current_Pattern == Pattern::R_PENTOMINO)
{

    data.push_back(coordinates + Vector2i{ 0, 0 });
    data.push_back(coordinates + Vector2i{ 1, 0 });
    data.push_back(coordinates + Vector2i{ 0, 1 });
    data.push_back(coordinates + Vector2i{ 0, 2 });
    data.push_back(coordinates + Vector2i{ -1, 1 });

}
else if (current_Pattern == Pattern::ACORN)
{

    data.push_back(coordinates + Vector2i{ 0, 0 });
    data.push_back(coordinates + Vector2i{ 0, 2 });
    data.push_back(coordinates + Vector2i{ -1, 2 });
```

```
            data.push_back(coordinates + Vector2i{ 2, 1 });
            data.push_back(coordinates + Vector2i{ 3, 2 });
            data.push_back(coordinates + Vector2i{ 4, 2 });
            data.push_back(coordinates + Vector2i{ 5, 2 });

        }
        else if (current_Pattern == Pattern::QUEEN_BEE_SHUTTLE)
        {

            data.push_back(coordinates + Vector2i{ 0, 0 });
            data.push_back(coordinates + Vector2i{ 1, 0 });
            data.push_back(coordinates + Vector2i{ 0, 1 });
            data.push_back(coordinates + Vector2i{ 1, 1 });
            data.push_back(coordinates + Vector2i{ 5, 0 });
            data.push_back(coordinates + Vector2i{ 6, 1 });
            data.push_back(coordinates + Vector2i{ 7, 2 });
            data.push_back(coordinates + Vector2i{ 6, -1 });
            data.push_back(coordinates + Vector2i{ 7, -2 });
            data.push_back(coordinates + Vector2i{ 8, 0 });
            data.push_back(coordinates + Vector2i{ 8, 1 });
            data.push_back(coordinates + Vector2i{ 8, -1 });
            data.push_back(coordinates + Vector2i{ 9, 2 });
            data.push_back(coordinates + Vector2i{ 9, 3 });
            data.push_back(coordinates + Vector2i{ 9, -2 });
            data.push_back(coordinates + Vector2i{ 9, -3 });
            data.push_back(coordinates + Vector2i{ 20, 0 });
            data.push_back(coordinates + Vector2i{ 21, 0 });
            data.push_back(coordinates + Vector2i{ 20, 1 });
            data.push_back(coordinates + Vector2i{ 21, 1 });

        }

        return data;
    }

    void Playground::create_cell(Vector2i coordinates)
    {

        if (in_grid(coordinates) == true)
        {

            TileMapLayer* grid = get_grid();
```

```
                int32_t    status    =    grid->get_cell_tile_data(coordinates)-
>get_custom_data("Status");

                if (status == 0)
                {

                    grid->set_cell(coordinates, source_ID, Vector2i{ 0, 0 });
                    grid->get_cell_tile_data(coordinates)-
>set_custom_data("Status", 1);
                    ++population_Size;

                }

            }

        }

        void Playground::destroy_cell(Vector2i coordinates)
        {

            if (in_grid(coordinates) == true)
            {

                TileMapLayer* grid = get_grid();

                int32_t    status    =    grid->get_cell_tile_data(coordinates)-
>get_custom_data("Status");

                if (status == 1)
                {

                    grid->set_cell(coordinates, source_ID, Vector2i{ 1, 0 });
                    grid->get_cell_tile_data(coordinates)-
>set_custom_data("Status", 0);
                    --population_Size;

                }

            }

        }
```

```
}
```

A.8 Файл Interface.hpp

```
#ifndef INTERFACE_HPP
#define INTERFACE_HPP

#include "Core/Declaration.hpp"
#include "Global/Locator.hpp"

#include <godot_cpp/classes/color_rect.hpp>
#include <godot_cpp/classes/node.hpp>
#include <godot_cpp/classes/font.hpp>
#include <godot_cpp/classes/theme.hpp>
#include <godot_cpp/classes/shader_material.hpp>
#include <godot_cpp/classes/shader.hpp>
#include <godot_cpp/classes/texture2d.hpp>
#include <godot_cpp/classes/style_box.hpp>
#include <godot_cpp/classes/style_box_flat.hpp>


namespace godot
{

    class Interface : public Node
    {

        GDCLASS(Interface, Node);

    private:

        Ref<Theme> theme{};
        Ref<Theme> info_Theme{};
        Ref<Font> font{};
        Ref<ShaderMaterial> shader_Material{};
        Ref<Shader> shader{};
        Ref<Texture2D> background_Cell_Texture{};
        Ref<StyleBoxFlat> normal{};
        Ref<StyleBoxFlat> hover{};
        Ref<StyleBoxFlat> pressed{};
        Ref<StyleBoxFlat> info_Normal{};
```

```
    protected:

        static void _bind_methods();

    public:

        CanvasLayer* get_canvas_layer();

        void create_menu();
        void destroy_menu();

        void create_exit();
        void destroy_exit();

        void create_info();
        void destroy_info();

        void create_rule();
        void destroy_rule();

        void create_play();
        void destroy_play();

        void _enter_tree() override;
        void _exit_tree() override;

        void _process(double delta) override;

    };

}

#define GD_INTERFACE_NAME "Interface"
#define GD_INTERFACE GD_LOCATOR->get_data<Interface>(GD_INTERFACE_NAME)

#define GD_INTERFACE_CANVAS_LAYER_NAME "Canvas"



#endif
```

A.9 Файл Interface.cpp

```
#include "Interface.hpp"


#include "Memory/Memory.hpp"
#include "Engine/Engine.hpp"
#include "Resource/Resource_Loader.hpp"

#include "Simulation.hpp"
#include "Playground.hpp"

#include "Tile_Map_Cell.hpp"

#include <godot_cpp/classes/label.hpp>
#include <godot_cpp/classes/label_settings.hpp>
#include <godot_cpp/classes/button.hpp>
#include <godot_cpp/classes/canvas_layer.hpp>

#include <godot_cpp/classes/margin_container.hpp>
#include <godot_cpp/classes/h_box_container.hpp>
#include <godot_cpp/classes/v_box_container.hpp>
#include <godot_cpp/classes/grid_container.hpp>
#include <godot_cpp/classes/center_container.hpp>
#include <godot_cpp/classes/panel_container.hpp>
#include <godot_cpp/classes/texture_rect.hpp>
#include <godot_cpp/classes/item_list.hpp>









namespace godot
{

    void Interface::_bind_methods()
    {

        print_line("Binding methods in ", typeid(Interface).name());
```

```
}

CanvasLayer* Interface::get_canvas_layer()
{

    return Object::cast_to<CanvasLayer>(get_child(0));
}

void Interface::create_menu()
{

    print_line("Creating menu");


    shader_Material->set_shader_parameter("roll", true);
    shader_Material->set_shader_parameter("roll_size", 20.0);
    shader_Material->set_shader_parameter("roll_variation", 0.566);
    shader_Material->set_shader_parameter("static_noise_intensity",
0.1);
    shader_Material->set_shader_parameter("aberration", 0.01);
    shader_Material->set_shader_parameter("brightness", 2);


    CanvasLayer* canvas = GD_NEW(CanvasLayer);
    canvas->set_name(GD_INTERFACE_CANVAS_LAYER_NAME);
    add_child(canvas);


    CenterContainer* container0 = GD_NEW(CenterContainer);
    container0->set_name("Menu Container 0");
    container0->set_anchors_preset(Control::PRESET_FULL_RECT);
    canvas->add_child(container0);

    MarginContainer* container1 = GD_NEW(MarginContainer);
    container1->set_name("Menu Container 1");
    container1->set_anchors_preset(Control::PRESET_FULL_RECT);
    canvas->add_child(container1);

    PanelContainer* container2 = GD_NEW(PanelContainer);
    container2->set_name("Menu Container 2");
    container2->set_anchors_preset(Control::PRESET_FULL_RECT);
```

```
                container2->set_mouse_filter(Control::MOUSE_FILTER_IGNORE);
                container2->set_material(shader_Material);
                canvas->add_child(container2);



                GridContainer* grid = GD_NEW(GridContainer);
                grid->set_name("Grid");
                grid->set_columns(20);
                grid->add_theme_constant_override("h_separation", 0);
                grid->add_theme_constant_override("v_separation", 0);
                container0->add_child(grid);

                TextureRect * background_Cell = GD_NEW(TextureRect);
                background_Cell->set_name("Background Cell");
                background_Cell->set_texture(background_Cell_Texture);
                background_Cell->set_custom_minimum_size(Vector2{ 130, 130 });
                grid->add_child(background_Cell);
                for (int i = 0; i < 200; ++i)
                {
                    grid->add_child(background_Cell-
>duplicate(Node::DUPLICATE_SIGNALS | Node::DUPLICATE_GROUPS));
                }



                VBoxContainer* v_Box = GD_NEW(VBoxContainer);
                v_Box->set_name("Vertical Box");
                v_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
                v_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
                v_Box->add_theme_constant_override("separation", 100);
                container1->add_child(v_Box);



                Label* title = GD_NEW(Label);
                title->set_name("Title");
                title->set_text("CONWAY'S GAME OF LIFE");
                title->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
                title->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
                float k = 0.8f;
                title->set_modulate(Color{0.0f + k, 1.0f + k, 0.8f + k, 1.0f});
                title->set_theme(theme);
```

```
title->add_theme_font_size_override("font_size", 70);
v_Box->add_child(title);


HBoxContainer* h_Box = GD_NEW(HBoxContainer);
h_Box->set_name("Horizontal Box");
h_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
h_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
h_Box->add_theme_constant_override("separation", 75);
v_Box->add_child(h_Box);



Button* play = GD_NEW(Button);
play->set_name("Play");
play->set_text("Simulate");
play->set_focus_mode(Control::FOCUS_NONE);
play->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
play->set_theme(theme);
play->set_custom_minimum_size(Vector2{ 200, 70 });
h_Box->add_child(play);

Button* rule = GD_NEW(Button);
rule->set_name("Rule");
rule->set_text("Rules");
rule->set_focus_mode(Control::FOCUS_NONE);
rule->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
rule->set_theme(theme);
rule->set_custom_minimum_size(Vector2{ 200, 70 });
h_Box->add_child(rule);

Button* info = GD_NEW(Button);
info->set_name("Info");
info->set_text("Info");
info->set_focus_mode(Control::FOCUS_NONE);
info->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
info->set_theme(theme);
info->set_custom_minimum_size(Vector2{ 200, 70 });
h_Box->add_child(info);

Button* exit = GD_NEW(Button);
exit->set_name("Exit");
```

```
            exit->set_text("Exit");
            exit->set_focus_mode(Control::FOCUS_NONE);
            exit->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
            exit->set_theme(theme);
            exit->set_custom_minimum_size(Vector2{ 200, 70 });
            h_Box->add_child(exit);


            exit->connect("pressed",                          callable_mp(this,
&Interface::destroy_menu), CONNECT_DEFERRED);
            exit->connect("pressed", callable_mp(this, &Interface::create_exit),
CONNECT_DEFERRED);

            info->connect("pressed",                          callable_mp(this,
&Interface::destroy_menu), CONNECT_DEFERRED);
            info->connect("pressed", callable_mp(this, &Interface::create_info),
CONNECT_DEFERRED);

            rule->connect("pressed",                          callable_mp(this,
&Interface::destroy_rule), CONNECT_DEFERRED);
            rule->connect("pressed", callable_mp(this, &Interface::create_rule),
CONNECT_DEFERRED);

            play->connect("pressed",                          callable_mp(this,
&Interface::destroy_menu), CONNECT_DEFERRED);
            play->connect("pressed", callable_mp(this, &Interface::create_play),
CONNECT_DEFERRED);
            play->connect("pressed",                callable_mp(GD_PLAYGROUND,
&Playground::create), CONNECT_DEFERRED);

        }

        void Interface::destroy_menu()
        {

            print_line("Destroying menu");

            CanvasLayer* canvas = get_canvas_layer();
            remove_child(canvas);
            GD_DELETE(canvas);

        }
```

```
void Interface::create_exit()
{

    print_line("Creating exit");


    CanvasLayer* canvas = GD_NEW(CanvasLayer);
    canvas->set_name(GD_INTERFACE_CANVAS_LAYER_NAME);
    add_child(canvas);


    CenterContainer* container0 = GD_NEW(CenterContainer);
    container0->set_name("Exit Container 0");
    container0->set_anchors_preset(Control::PRESET_FULL_RECT);
    canvas->add_child(container0);

    MarginContainer* container1 = GD_NEW(MarginContainer);
    container1->set_name("Exit Container 1");
    container1->set_anchors_preset(Control::PRESET_FULL_RECT);
    canvas->add_child(container1);

    PanelContainer* container2 = GD_NEW(PanelContainer);
    container2->set_name("Exit Container 2");
    container2->set_anchors_preset(Control::PRESET_FULL_RECT);
    container2->set_mouse_filter(Control::MOUSE_FILTER_IGNORE);
    container2->set_material(shader_Material);
    canvas->add_child(container2);


    GridContainer* grid = GD_NEW(GridContainer);
    grid->set_name("Grid");
    grid->set_columns(20);
    grid->add_theme_constant_override("h_separation", 0);
    grid->add_theme_constant_override("v_separation", 0);
    container0->add_child(grid);

    TextureRect* background_Cell = GD_NEW(TextureRect);
    background_Cell->set_name("Background Cell");
    background_Cell->set_texture(background_Cell_Texture);
    background_Cell->set_custom_minimum_size(Vector2{ 130, 130 });
    grid->add_child(background_Cell);
```

```
                for (int i = 0; i < 200; ++i)
                {
                    grid->add_child(background_Cell-
>duplicate(Node::DUPLICATE_SIGNALS | Node::DUPLICATE_GROUPS));
                }


                VBoxContainer* v_Box = GD_NEW(VBoxContainer);
                v_Box->set_name("Vertical Box");
                v_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
                v_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
                v_Box->add_theme_constant_override("separation", 100);
                container1->add_child(v_Box);


                Label* info = GD_NEW(Label);
                info->set_name("Info Text");
                info->set_text("Are you sure about this?");
                info->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
                info->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
                info->set_autowrap_mode(TextServer::AUTOWRAP_WORD);
                info->set_custom_minimum_size(Vector2{ 200, 200 });
                float k = 0.8f;
                info->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
                info->set_theme(info_Theme);
                v_Box->add_child(info);


                HBoxContainer* h_Box = GD_NEW(HBoxContainer);
                h_Box->set_name("Horizontal Box");
                h_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
                h_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
                h_Box->add_theme_constant_override("separation", 75);
                v_Box->add_child(h_Box);


                Button* yes = GD_NEW(Button);
                yes->set_name("Yes");
                yes->set_text("YES");
                yes->set_focus_mode(Control::FOCUS_NONE);
                yes->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
                yes->set_theme(theme);
```

```
            yes->set_custom_minimum_size(Vector2{ 200, 70 });
            h_Box->add_child(yes);


            Button* no = GD_NEW(Button);
            no->set_name("No");
            no->set_text("NO");
            no->set_focus_mode(Control::FOCUS_NONE);
            no->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
            no->set_theme(theme);
            no->set_custom_minimum_size(Vector2{ 200, 70 });
            h_Box->add_child(no);


            yes->connect("pressed",                  callable_mp(GD_SIMULATION,
&Simulation::exit), CONNECT_DEFERRED);
            no->connect("pressed", callable_mp(this, &Interface::destroy_exit),
CONNECT_DEFERRED);
            no->connect("pressed",  callable_mp(this,  &Interface::create_menu),
CONNECT_DEFERRED);

        }

        void Interface::destroy_exit()
        {

            print_line("Destroying exit");

            CanvasLayer* canvas = get_canvas_layer();
            remove_child(canvas);
            GD_DELETE(canvas);

        }

        void Interface::create_info()
        {

            print_line("Creating info");


            CanvasLayer* canvas = GD_NEW(CanvasLayer);
            canvas->set_name(GD_INTERFACE_CANVAS_LAYER_NAME);
```

```
        add_child(canvas);


        CenterContainer* container0 = GD_NEW(CenterContainer);
        container0->set_name("Info Container 0");
        container0->set_anchors_preset(Control::PRESET_FULL_RECT);
        canvas->add_child(container0);

        MarginContainer* container1 = GD_NEW(MarginContainer);
        container1->set_name("Info Container 1");
        container1->set_anchors_preset(Control::PRESET_FULL_RECT);
        canvas->add_child(container1);

        PanelContainer* container2 = GD_NEW(PanelContainer);
        container2->set_name("Info Container 2");
        container2->set_anchors_preset(Control::PRESET_FULL_RECT);
        container2->set_mouse_filter(Control::MOUSE_FILTER_IGNORE);
        container2->set_material(shader_Material);
        canvas->add_child(container2);


        GridContainer* grid = GD_NEW(GridContainer);
        grid->set_name("Grid");
        grid->set_columns(20);
        grid->add_theme_constant_override("h_separation", 0);
        grid->add_theme_constant_override("v_separation", 0);
        container0->add_child(grid);

        TextureRect* background_Cell = GD_NEW(TextureRect);
        background_Cell->set_name("Background Cell");
        background_Cell->set_texture(background_Cell_Texture);
        background_Cell->set_custom_minimum_size(Vector2{ 130, 130 });
        grid->add_child(background_Cell);
        for (int i = 0; i < 200; ++i)
        {
            grid->add_child(background_Cell-
>duplicate(Node::DUPLICATE_SIGNALS | Node::DUPLICATE_GROUPS));
        }


        VBoxContainer* v_Box = GD_NEW(VBoxContainer);
        v_Box->set_name("Vertical Box");
```

```
            v_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
            v_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
            v_Box->add_theme_constant_override("separation", 100);
            container1->add_child(v_Box);


            Label* info = GD_NEW(Label);
            info->set_name("Info Text");
            info->set_text("This implementation of the Conway's Game of Life is
done by Yaroslav Kolokol. The program is made in Godot game engine using C++ and
CMake. Enjoy!");
            info->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
            info->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
            info->set_autowrap_mode(TextServer::AUTOWRAP_WORD);
            info->set_custom_minimum_size(Vector2{ 800, 200 });
            float k = 0.8f;
            info->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
            info->set_theme(info_Theme);
            v_Box->add_child(info);


            Button* back = GD_NEW(Button);
            back->set_name("Back");
            back->set_text("BACK");
            back->set_focus_mode(Control::FOCUS_NONE);
            back->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
            back->set_theme(theme);
            back->set_custom_minimum_size(Vector2{ 200, 70 });
            v_Box->add_child(back);


            back->connect("pressed",                              callable_mp(this,
&Interface::destroy_info), CONNECT_DEFERRED);
            back->connect("pressed", callable_mp(this, &Interface::create_menu),
CONNECT_DEFERRED);

        }

        void Interface::destroy_info()
        {

            print_line("Destroying info");
```

```
    CanvasLayer* canvas = get_canvas_layer();
    remove_child(canvas);
    GD_DELETE(canvas);

}

void Interface::create_rule()
{

    print_line("Creating rule");


    CanvasLayer* canvas = GD_NEW(CanvasLayer);
    canvas->set_name(GD_INTERFACE_CANVAS_LAYER_NAME);
    add_child(canvas);


    CenterContainer* container0 = GD_NEW(CenterContainer);
    container0->set_name("Rule Container 0");
    container0->set_anchors_preset(Control::PRESET_FULL_RECT);
    canvas->add_child(container0);

    MarginContainer* container1 = GD_NEW(MarginContainer);
    container1->set_name("Rule Container 1");
    container1->set_anchors_preset(Control::PRESET_FULL_RECT);
    canvas->add_child(container1);

    PanelContainer* container2 = GD_NEW(PanelContainer);
    container2->set_name("Rule Container 2");
    container2->set_anchors_preset(Control::PRESET_FULL_RECT);
    container2->set_mouse_filter(Control::MOUSE_FILTER_IGNORE);
    container2->set_material(shader_Material);
    canvas->add_child(container2);


    GridContainer* grid = GD_NEW(GridContainer);
    grid->set_name("Grid");
    grid->set_columns(20);
    grid->add_theme_constant_override("h_separation", 0);
    grid->add_theme_constant_override("v_separation", 0);
    container0->add_child(grid);
```

```
                TextureRect* background_Cell = GD_NEW(TextureRect);
                background_Cell->set_name("Background Cell");
                background_Cell->set_texture(background_Cell_Texture);
                background_Cell->set_custom_minimum_size(Vector2{ 130, 130 });
                grid->add_child(background_Cell);
                for (int i = 0; i < 200; ++i)
                {
                    grid->add_child(background_Cell-
>duplicate(Node::DUPLICATE_SIGNALS | Node::DUPLICATE_GROUPS));
                }


                VBoxContainer* v_Box = GD_NEW(VBoxContainer);
                v_Box->set_name("Vertical Box");
                v_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
                v_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
                v_Box->add_theme_constant_override("separation", 100);
                container1->add_child(v_Box);


                Label* info = GD_NEW(Label);
                info->set_name("Rule Text");
                info->set_text("     1) Any live cell with fewer than two live
neighbours dies, as if by underpopulation.\n    2) Any live cell with two or three
live neighbours lives on to the next generation.\n     3) Any live cell with more
than three live neighbours dies, as if by overpopulation.\n    4) Any dead cell with
exactly three live neighbours becomes a live cell, as if by reproduction.");
                info->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_FILL);
                info->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
                info->set_autowrap_mode(TextServer::AUTOWRAP_WORD);
                info->set_custom_minimum_size(Vector2{ 800, 200 });
                float k = 0.8f;
                info->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
                info->set_theme(info_Theme);
                v_Box->add_child(info);


                Button* back = GD_NEW(Button);
                back->set_name("Back");
                back->set_text("BACK");
                back->set_focus_mode(Control::FOCUS_NONE);
```

```
            back->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
            back->set_theme(theme);
            back->set_custom_minimum_size(Vector2{ 200, 70 });
            v_Box->add_child(back);


            back->connect("pressed",                          callable_mp(this,
&Interface::destroy_rule), CONNECT_DEFERRED);
            back->connect("pressed", callable_mp(this, &Interface::create_menu),
CONNECT_DEFERRED);

        }

        void Interface::destroy_rule()
        {

            print_line("Destroying rule");

            CanvasLayer* canvas = get_canvas_layer();
            remove_child(canvas);
            GD_DELETE(canvas);

        }

        void Interface::create_play()
        {

            print_line("Creating play");


            shader_Material->set_shader_parameter("roll", false);
            shader_Material->set_shader_parameter("roll_size", 0);
            shader_Material->set_shader_parameter("roll_variation", 0);
            shader_Material->set_shader_parameter("static_noise_intensity",
0.04);
            shader_Material->set_shader_parameter("aberration", 0);
            shader_Material->set_shader_parameter("brightness", 1);


            float k = 0.8f;
```

```
CanvasLayer* canvas = GD_NEW(CanvasLayer);
canvas->set_name(GD_INTERFACE_CANVAS_LAYER_NAME);
add_child(canvas);


MarginContainer* container0 = GD_NEW(MarginContainer);
container0->set_name("Play Container 0");
container0->set_anchors_preset(Control::PRESET_FULL_RECT);
canvas->add_child(container0);


VBoxContainer* v_Box = GD_NEW(VBoxContainer);
v_Box->set_name("Vertical Box");
v_Box->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
v_Box->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
v_Box->add_theme_constant_override("separation", 500);
container0->add_child(v_Box);


HBoxContainer* h_Box0 = GD_NEW(HBoxContainer);
h_Box0->set_name("Horizontal Box0");
h_Box0->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
h_Box0->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
h_Box0->add_theme_constant_override("separation", 75);
v_Box->add_child(h_Box0);


HBoxContainer* h_Box1 = GD_NEW(HBoxContainer);
h_Box1->set_name("Horizontal Box1");
h_Box1->set_h_size_flags(Control::SIZE_SHRINK_CENTER);
h_Box1->set_v_size_flags(Control::SIZE_SHRINK_CENTER);
h_Box1->add_theme_constant_override("separation", 75);
v_Box->add_child(h_Box1);


Label* fps = GD_NEW(Label);
fps->set_name("FPS");
fps->set_text("FPS 0");
fps->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
fps->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
fps->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
fps->set_theme(theme);
```

```
		h_Box0->add_child(fps);


		Label* size = GD_NEW(Label);
		size->set_name("Grid Size");
		size->set_text("Grid 0x0");
		size->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
		size->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
		size->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
		size->set_theme(theme);
		h_Box0->add_child(size);


		Label* generation = GD_NEW(Label);
		generation->set_name("Generation");
		generation->set_text("Generation 0");
		generation->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
		generation->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
		generation->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f
});
		generation->set_theme(theme);
		h_Box0->add_child(generation);


		Label* population = GD_NEW(Label);
		population->set_name("Population");
		population->set_text("Population 0");
		population->set_horizontal_alignment(HORIZONTAL_ALIGNMENT_CENTER);
		population->set_vertical_alignment(VERTICAL_ALIGNMENT_CENTER);
		population->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f
});
		population->set_theme(theme);
		h_Box0->add_child(population);


		Button* grid_Up = GD_NEW(Button);
		grid_Up->set_name("+ Grid");
		grid_Up->set_text("+ GRID");
		grid_Up->set_focus_mode(Control::FOCUS_NONE);
		grid_Up->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
		grid_Up->set_theme(theme);
		grid_Up->set_custom_minimum_size(Vector2{ 200, 70 });
```

```
                h_Box1->add_child(grid_Up);


                Button* grid_Down = GD_NEW(Button);
                grid_Down->set_name("- Grid");
                grid_Down->set_text("- GRID");
                grid_Down->set_focus_mode(Control::FOCUS_NONE);
                grid_Down->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f
});
                grid_Down->set_theme(theme);
                grid_Down->set_custom_minimum_size(Vector2{ 200, 70 });
                h_Box1->add_child(grid_Down);


                Button* step = GD_NEW(Button);
                step->set_name("Step");
                step->set_text("STEP");
                step->set_focus_mode(Control::FOCUS_NONE);
                step->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
                step->set_theme(theme);
                step->set_custom_minimum_size(Vector2{ 200, 70 });
                h_Box1->add_child(step);


                Button* back = GD_NEW(Button);
                back->set_name("Back");
                back->set_text("BACK");
                back->set_focus_mode(Control::FOCUS_NONE);
                back->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
                back->set_theme(theme);
                back->set_custom_minimum_size(Vector2{ 200, 70 });
                h_Box1->add_child(back);


                MarginContainer* container1 = GD_NEW(MarginContainer);
                container1->set_name("Play Container 1");
                container1->set_h_grow_direction(Control::GROW_DIRECTION_BOTH);
                container1->set_v_grow_direction(Control::GROW_DIRECTION_BOTH);
                container1->set_anchors_preset(Control::PRESET_CENTER_RIGHT);
                container1->set_mouse_filter(Control::MOUSE_FILTER_IGNORE);
                container1->add_theme_constant_override("margin_top", 300);
                container1->add_theme_constant_override("margin_bottom", 300);
```

```
                container1->add_theme_constant_override("margin_right", 300);
                canvas->add_child(container1);


                ItemList* list = GD_NEW(ItemList);
                list->set_name("List");
                list->set_custom_minimum_size(Vector2{ 200, 300 });
                list->set_focus_mode(Control::FOCUS_NONE);
                list->set_modulate(Color{ 0.0f + k, 1.0f + k, 0.8f + k, 1.0f });
                list->set_theme(theme);
                list->add_item("Cell");
                list->add_item("Block");
                list->add_item("Blinker");
                list->add_item("Glider");
                list->add_item("Beacon");
                list->add_item("Tub");
                list->add_item("R-pentomino");
                list->add_item("Acorn");
                list->add_item("Queen Bee Shuttle");
                container1->add_child(list);
                list->select(0);


                PanelContainer* container2 = GD_NEW(PanelContainer);
                container2->set_name("Play Container 2");
                container2->set_anchors_preset(Control::PRESET_FULL_RECT);
                container2->set_mouse_filter(Control::MOUSE_FILTER_IGNORE);
                container2->set_material(shader_Material);
                canvas->add_child(container2);


                list->connect("item_selected",           callable_mp(GD_PLAYGROUND,
&Playground::select_pattern), CONNECT_DEFERRED);


                back->connect("pressed",                 callable_mp(GD_PLAYGROUND,
&Playground::destroy), CONNECT_DEFERRED);
                back->connect("pressed",                          callable_mp(this,
&Interface::destroy_play), CONNECT_DEFERRED);
                back->connect("pressed", callable_mp(this, &Interface::create_menu),
CONNECT_DEFERRED);
```

```
            grid_Up->connect("pressed",             callable_mp(GD_PLAYGROUND,
&Playground::increase_grid_size), CONNECT_DEFERRED);
            grid_Down->connect("pressed",           callable_mp(GD_PLAYGROUND,
&Playground::decrease_grid_size), CONNECT_DEFERRED);

            step->connect("pressed",                callable_mp(GD_PLAYGROUND,
&Playground::step), CONNECT_DEFERRED);

        }

        void Interface::destroy_play()
        {

            print_line("Destroying play");

            CanvasLayer* canvas = get_canvas_layer();
            remove_child(canvas);
            GD_DELETE(canvas);

        }

        void Interface::_enter_tree()
        {

            font = GD_RESOURCE_LOADER->load("res://Life/PixelFont.ttf");
            shader = GD_RESOURCE_LOADER->load("res://Life/CRT.gdshader");
            background_Cell_Texture          =          GD_RESOURCE_LOADER-
>load("res://Life/Background_Cell.png");

            shader_Material.instantiate();
            shader_Material->set_shader(shader);


            normal.instantiate();
            normal->set_bg_color(Color{ 0, 0, 0, 0.8 });
            normal->set_border_width_all(5);
            normal->set_border_color(Color{ 1, 1, 1, 1 });
            normal->set_corner_radius_all(0);
            normal->set_corner_detail(0);
```

```
                hover.instantiate();
                hover->set_bg_color(Color{ 1, 1, 1, 0.8 });
                hover->set_border_width_all(5);
                hover->set_border_color(Color{ 0, 0, 0, 1 });
                hover->set_corner_radius_all(0);
                hover->set_corner_detail(0);


                pressed.instantiate();
                pressed->set_bg_color(Color{ 0.7, 0.7, 0.7, 0.5 });
                pressed->set_border_width_all(0);
                pressed->set_corner_radius_all(0);
                pressed->set_corner_detail(0);


                theme.instantiate();
                theme->set_color("font_color", "Label", Color{ 1, 1, 1, 1 });
                theme->set_color("font_color", "Button", Color{ 1, 1, 1, 1 });
                theme->set_color("font_hover_color", "Button", Color{ 0, 0, 0, 1 });
                theme->set_color("font_pressed_color", "Button", Color{ 0, 0, 0, 1
});
                theme->set_font("font", "Button", font);
                theme->set_font("font", "Label", font);
                theme->set_font_size("font_size", "Button", 32);
                theme->set_font_size("font_size", "Label", 32);
                theme->set_stylebox("normal", "Button", normal);
                theme->set_stylebox("hover", "Button", hover);
                theme->set_stylebox("pressed", "Button", pressed);
                theme->set_font("font", "ItemList", font);
                theme->set_font_size("font_size", "ItemList", 32);
                theme->set_color("font_color", "ItemList", Color{ 1, 1, 1, 1 });
                theme->set_color("font_hovered_color", "ItemList", Color{ 0, 0, 0, 1
});
                theme->set_color("font_selected_color", "ItemList", Color{ 0, 0, 0,
1 });
                theme->set_color("font_hovered_selected_color", "ItemList", Color{
0, 0, 0, 1 });
                theme->set_constant("h_separation", "ItemList", 40);
                theme->set_constant("v_separation", "ItemList", 40);


                Ref<StyleBoxFlat> item_List_Hovered{};
```

```
item_List_Hovered.instantiate();
item_List_Hovered->set_bg_color(Color{ 1, 1, 1, 0.8 });
item_List_Hovered->set_border_width_all(5);
item_List_Hovered->set_border_color(Color{ 0, 0, 0, 1 });
item_List_Hovered->set_corner_radius_all(0);
item_List_Hovered->set_corner_detail(0);
theme->set_stylebox("hovered", "ItemList", item_List_Hovered);


Ref<StyleBoxFlat> item_List_Selected{};
item_List_Selected.instantiate();
item_List_Selected->set_bg_color(Color{ 0.7, 0.7, 0.7, 0.5 });
item_List_Selected->set_border_width_all(0);
item_List_Selected->set_corner_radius_all(0);
item_List_Selected->set_corner_detail(0);
theme->set_stylebox("selected", "ItemList", item_List_Selected);


Ref<StyleBoxFlat> item_List_Panel{};
item_List_Panel.instantiate();
item_List_Panel->set_bg_color(Color{ 0, 0, 0, 0.8 });
item_List_Panel->set_border_width_all(5);
item_List_Panel->set_border_color(Color{ 1, 1, 1, 1 });
item_List_Panel->set_corner_radius_all(0);
item_List_Panel->set_corner_detail(0);
theme->set_stylebox("panel", "ItemList", item_List_Panel);


info_Normal.instantiate();
info_Normal->set_bg_color(Color{ 0, 0, 0, 0.8 });
info_Normal->set_border_width_all(5);
info_Normal->set_border_color(Color{ 0.5, 0.5, 0.5, 1 });
info_Normal->set_anti_aliased(false);
info_Normal->set_corner_radius_all(40);
info_Normal->set_corner_detail(1);
info_Normal->set_expand_margin_all(30);


info_Theme.instantiate();
info_Theme->set_color("font_color", "Label", Color{ 1, 1, 1, 1 });
info_Theme->set_font("font", "Label", font);
info_Theme->set_font_size("font_size", "Label", 32);
```

```
            info_Theme->set_stylebox("normal", "Label", info_Normal);

        }

        void Interface::_exit_tree()
        {

        }
        void Interface::_process(double delta)
        {

            if (GD_PLAYGROUND->get_is_playing() == true)
            {

                CanvasLayer* canvas = get_canvas_layer();

                Label*   fps   =   Object::cast_to<Label>(canvas->get_child(0)-
>get_child(0)->get_child(0)->get_child(0));
                fps->set_text(String{ "FPS " + String::num_real(GD_PLAYGROUND-
>get_fps_count()) });

                Label*   grid   =   Object::cast_to<Label>(canvas->get_child(0)-
>get_child(0)->get_child(0)->get_child(1));
                grid->set_text(String{           "Grid           "           +
String::num_int64(GD_PLAYGROUND->get_grid_size())         +         "x"         +
String::num_int64(GD_PLAYGROUND->get_grid_size()) });

                Label* generation = Object::cast_to<Label>(canvas->get_child(0)-
>get_child(0)->get_child(0)->get_child(2));
                generation->set_text(String{       "Generation       "       +
String::num_int64(GD_PLAYGROUND->get_generation_count()) });

                Label* population = Object::cast_to<Label>(canvas->get_child(0)-
>get_child(0)->get_child(0)->get_child(3));
                population->set_text(String{       "Population       "       +
String::num_int64(GD_PLAYGROUND->get_population_size()) });

            }
        }

    }
```

A.10 Файл Locator.hpp

```
#ifndef LOCATOR_HPP
#define LOCATOR_HPP


#include <godot_cpp/templates/a_hash_map.hpp>
#include <godot_cpp/variant/string.hpp>


namespace godot
{

    class Locator
    {

    private:

        AHashMap<String, void*> data{};

    public:

        static Locator* get();

        void set_data(String key, void* value);
        template<typename T> T* get_data(String key)
        {
            return static_cast<T*>(data[key]);
        }

    };

}

#define GD_LOCATOR godot::Locator::get()




#endif
```

A.11 Файл Locator.cpp

```
#include "Locator.hpp"

namespace godot
{

    Locator locator{};

    Locator* Locator::get()
    {
        return &locator;
    }

    void Locator::set_data(String key, void* value)
    {

        data[key] = value;

    }

}
```

ДОДАТОК Б

Слайди презентації

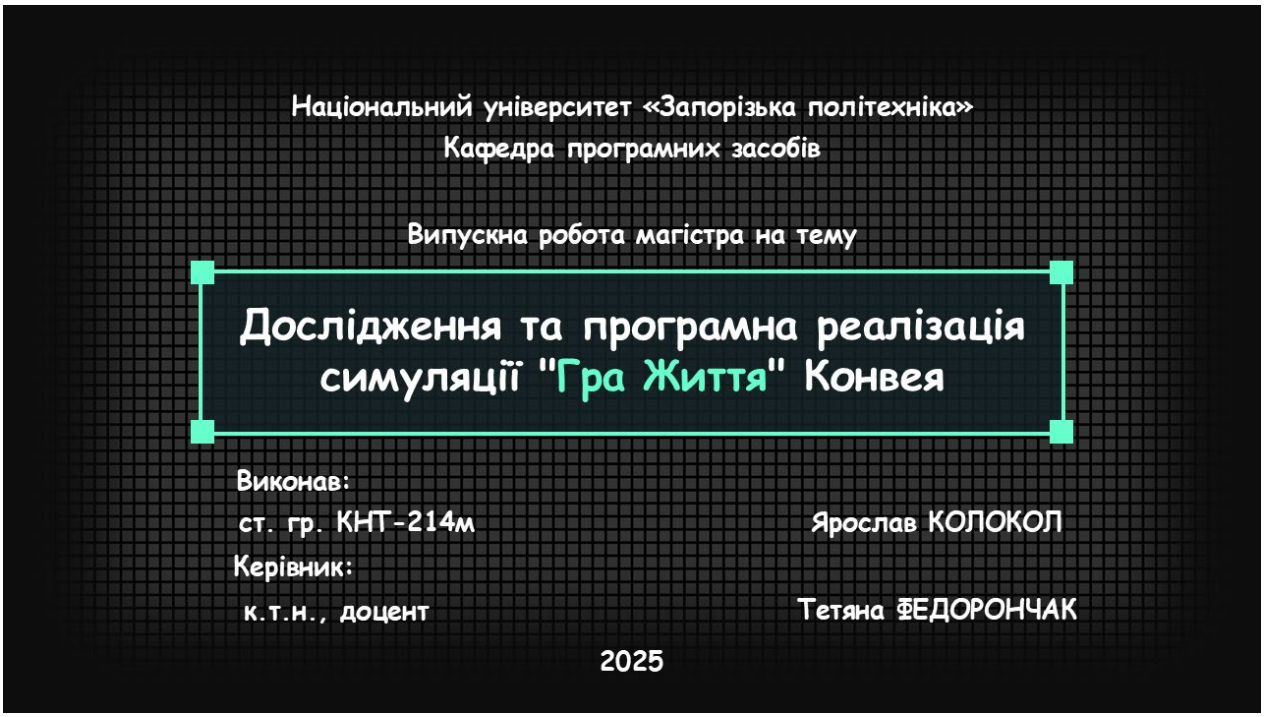

Рисунок Б.1 – Слайд №1

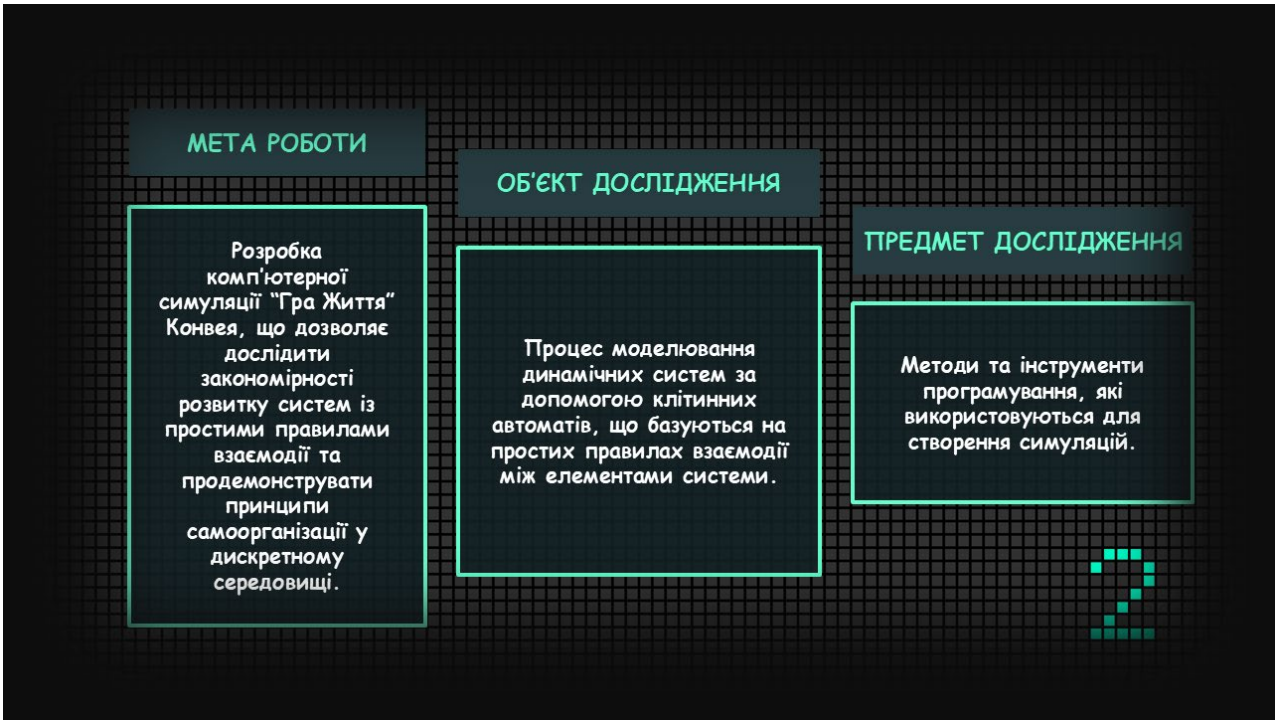

Рисунок Б.2 – Слайд №2

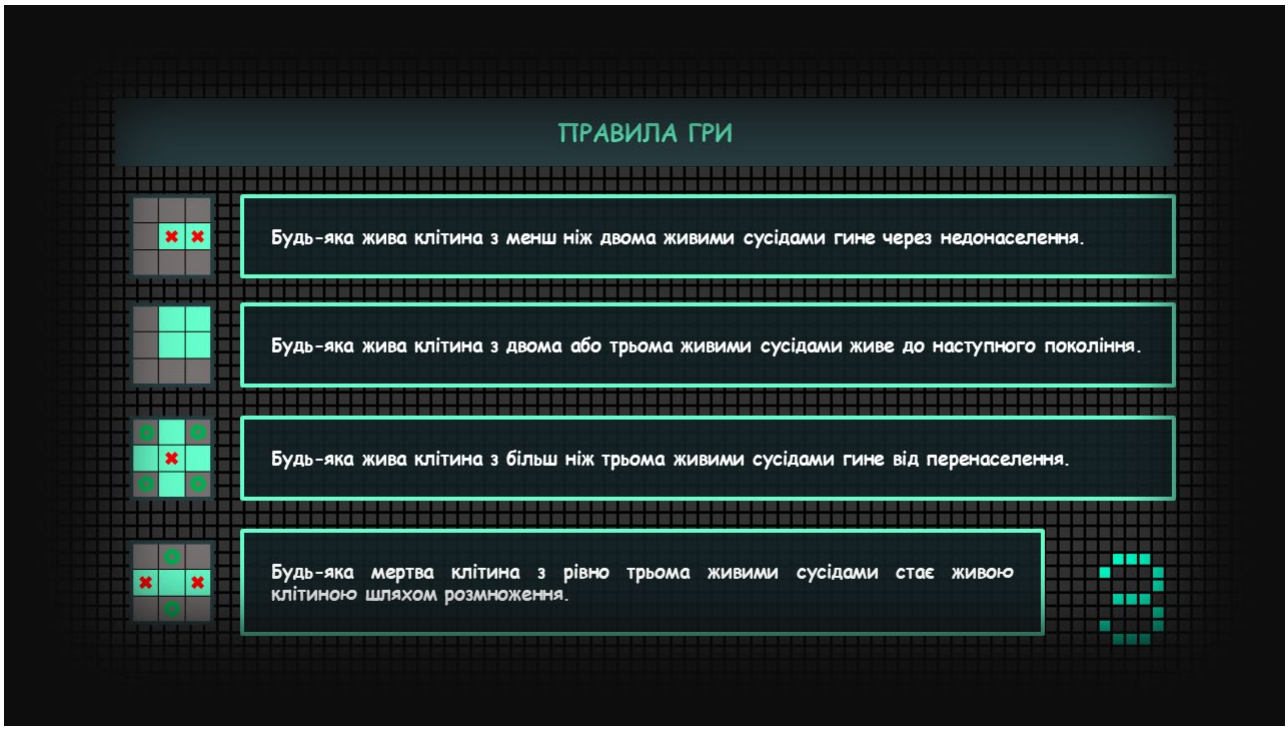

Рисунок Б.3 – Слайд №3

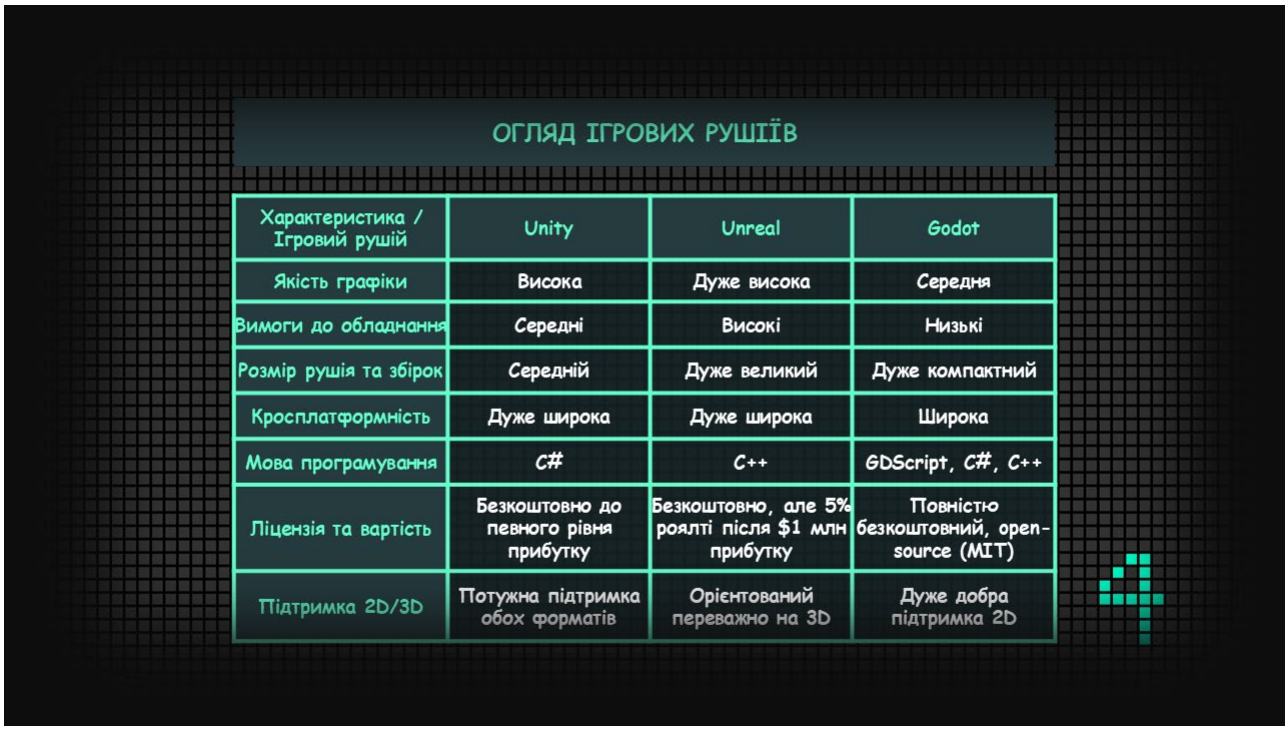

Рисунок Б.4 – Слайд №4

Рисунок Б.5 – Слайд №5

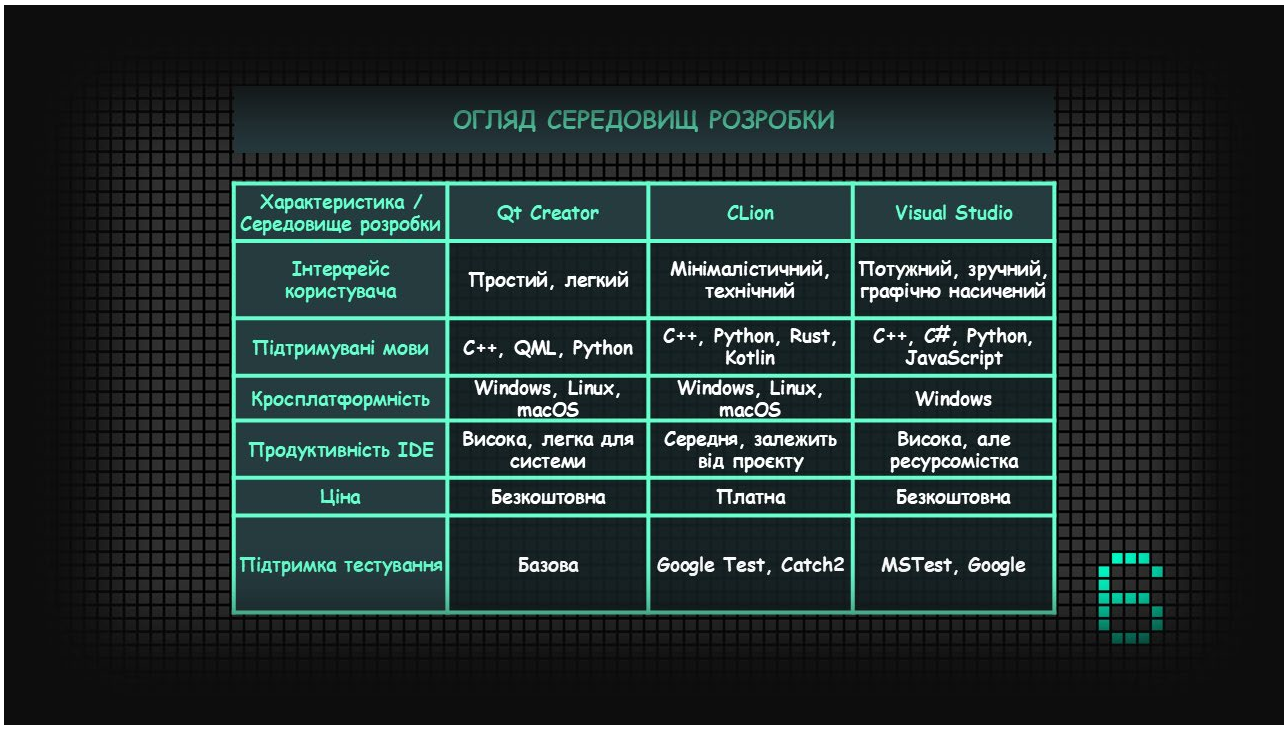

Рисунок Б.6 – Слайд №6

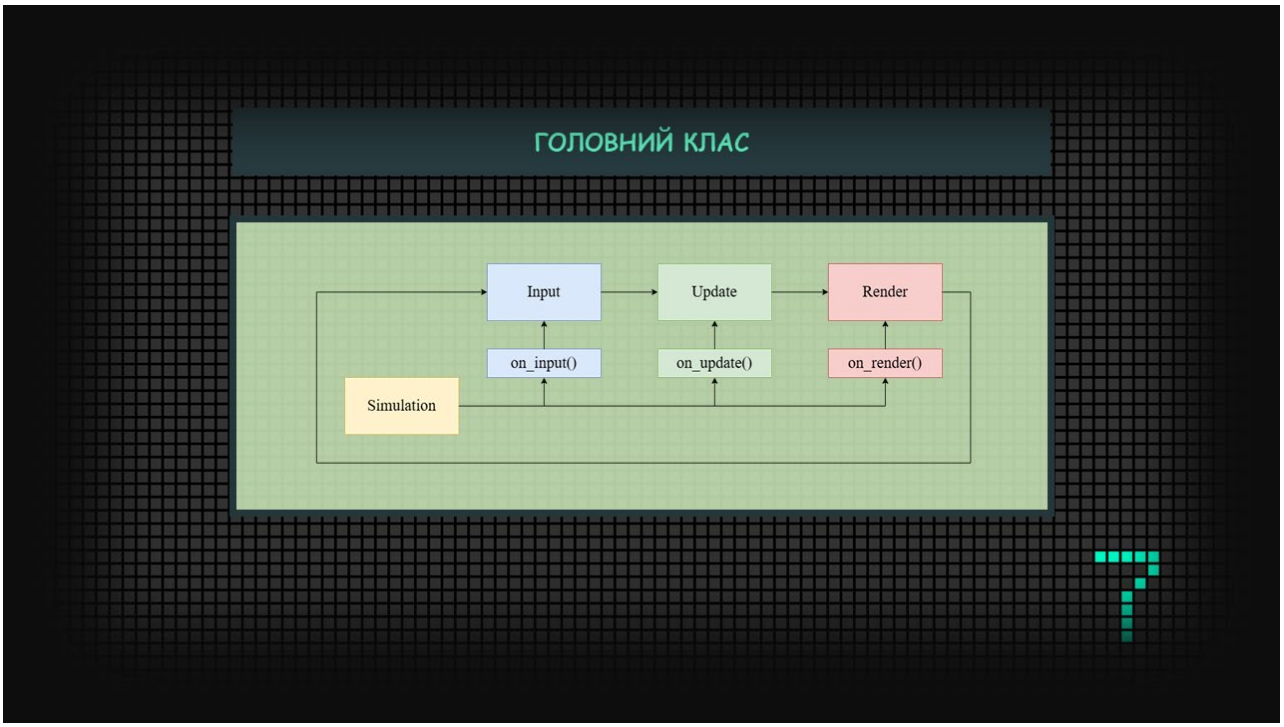

Рисунок Б.7 – Слайд №7

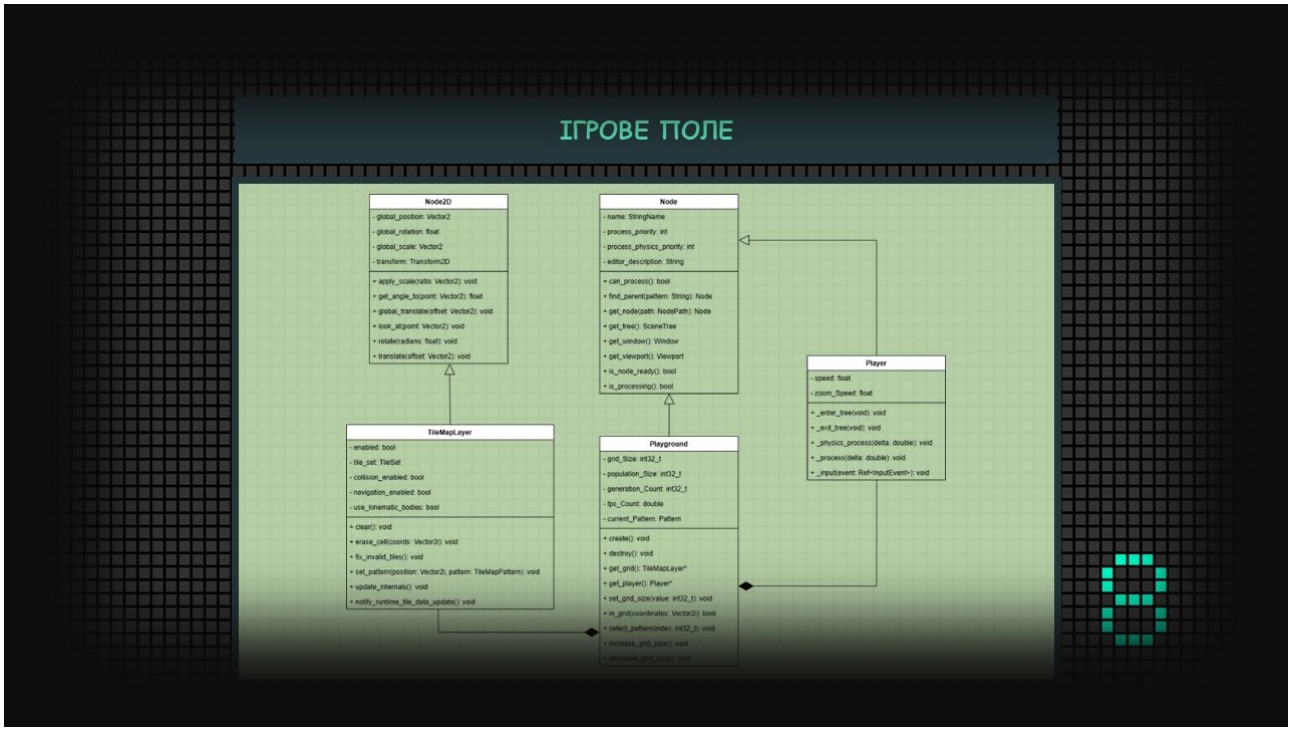

Рисунок Б.8 – Слайд №8

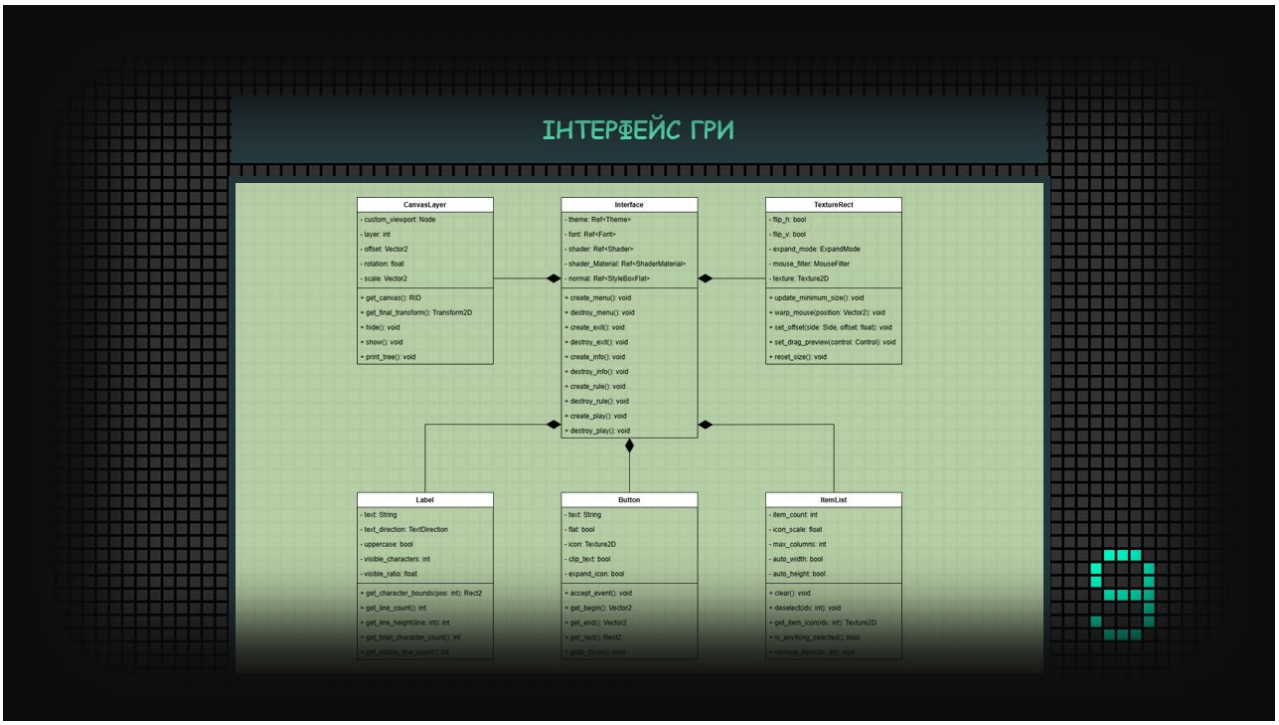

Рисунок Б.9 – Слайд №9

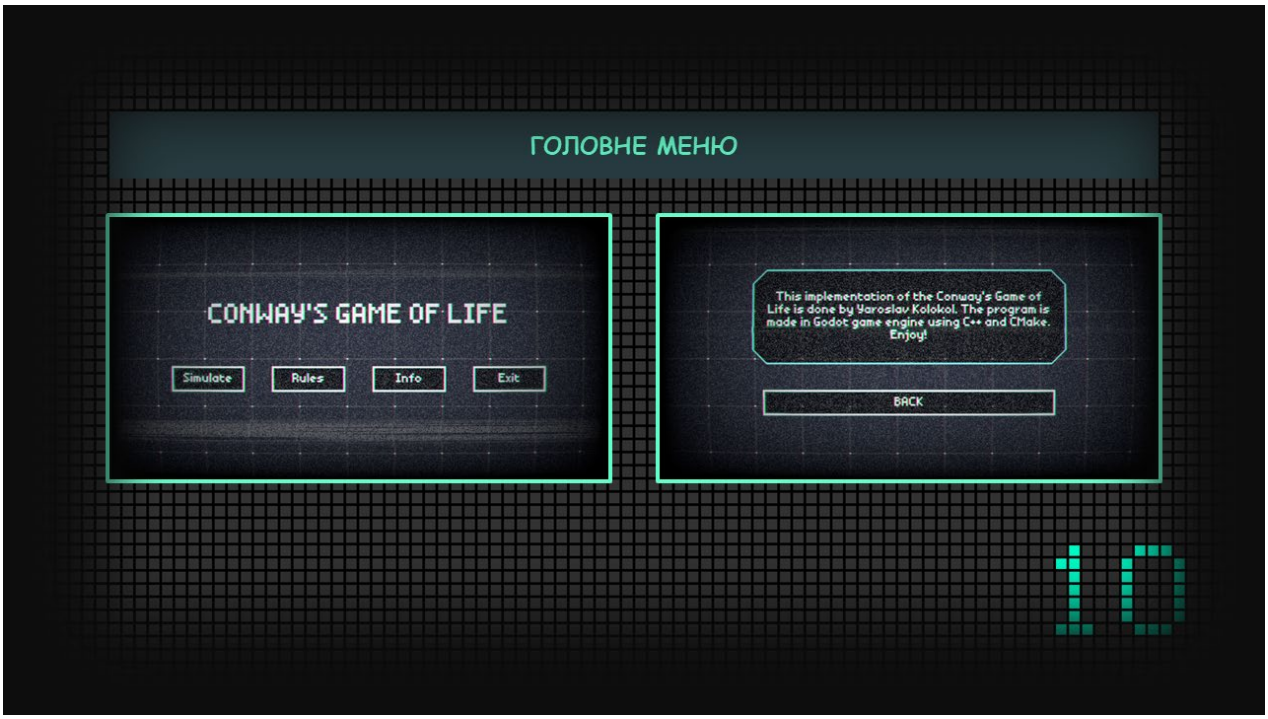

Рисунок Б.10 – Слайд №10

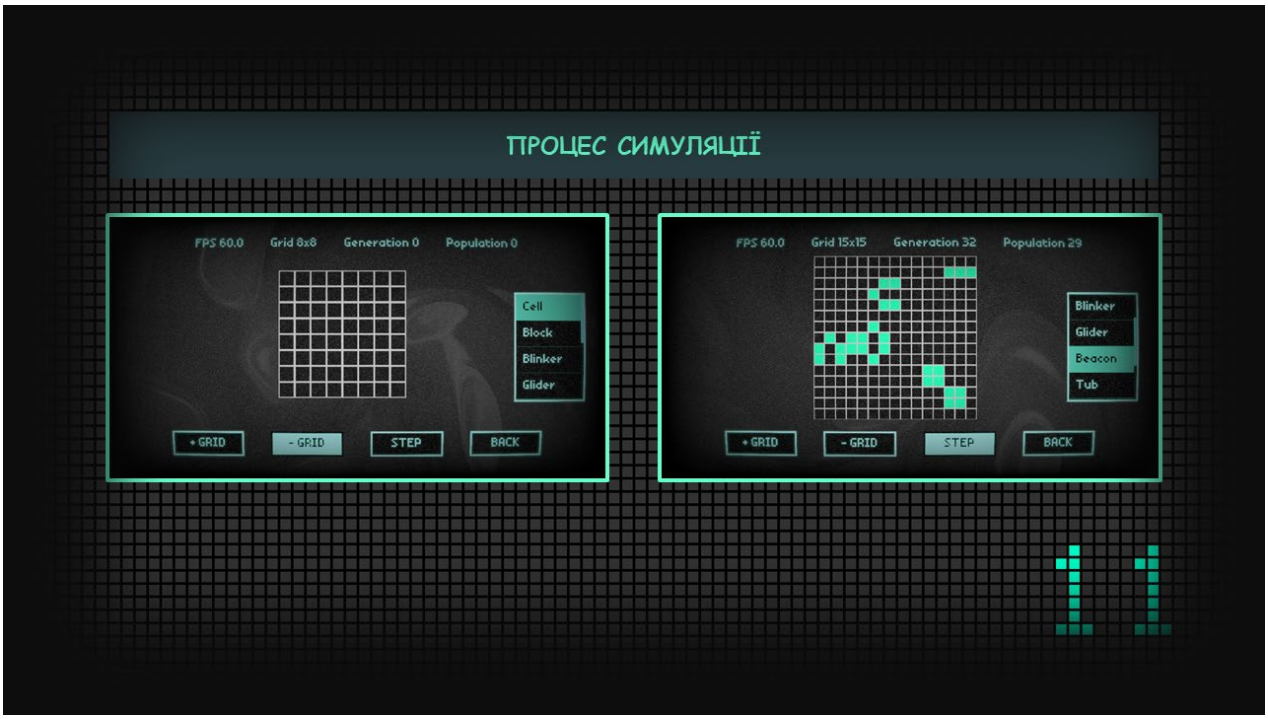

Рисунок Б.11 – Слайд №11

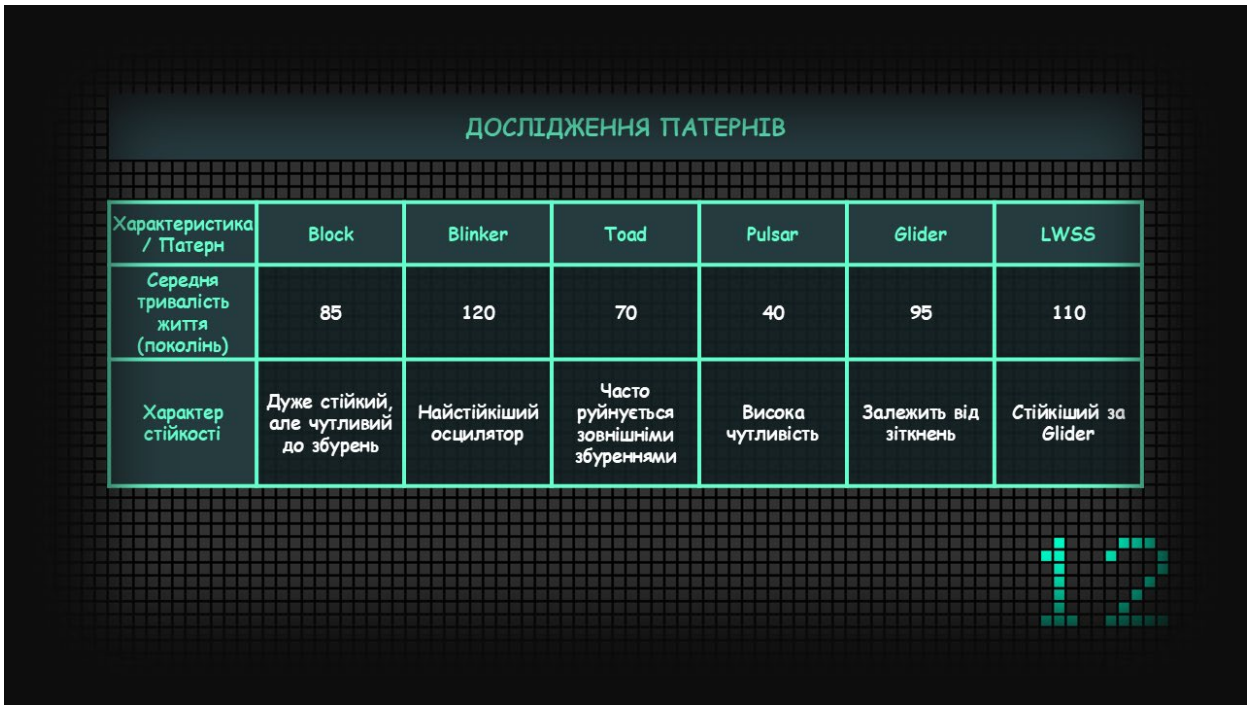

Рисунок Б.12 – Слайд №12

Рисунок Б.13 – Слайд №13