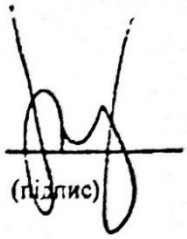
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

«До захисту допущено»
Завідувач кафедри
В.М.Терещенко



«___» _____20__ р.

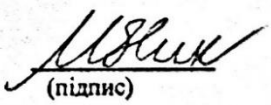
Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки
на тему:

АВТОМАТИЗАЦІЯ РОЗРОБКИ КАРТ РІВНІВ ТА ФРАКТАЛЬНОЇ ГРАФІКИ ДЛЯ КОМП'ЮТЕРНИХ ІГОР

Виконала студентка 4 курсу
Ніколаєва Марина Миколаївна



Науковий керівник:
асистент
Тарануха Володимир Юрійович

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Київ – 2022

РЕФЕРАТ

Обсяг роботи 83 сторінки, 15 ілюстрацій, 22 джерела посилань, 4 додатки.

АЛГОРИТМІЧНЕ СТВОРЕННЯ ОБ’ЄКТІВ, ВИПАДКОВЕ ПРОХОДЖЕННЯ, ВІДЕОГРА, ГЕНЕТИЧНИЙ АЛГОРИТМ, ГРАВЕЦЬ, КЛІТИННИЙ АВТОМАТ, КОМП’ЮТЕРНА ГРА, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, ФРАКТАЛ, ФРАКТАЛЬНА ГРАФІКА, ЦІКАВІСТЬ ДЛЯ ГРАВЦЯ.

Об’єктом роботи є процес розробки засобу для спрощення процесу роботи над комп’ютерною грою в жанрі платформер із застосуванням методів процедурної генерації, а також засобу для створення графіки, що може стати текстурою для гри. Предметами є два програмні продукти: для формування мапи рівня гри і для зображення простої фрактальної графіки для подальшої обробки.

Метою роботи є дослідження методів автоматичного створення об’єктів, що можуть слугувати наповненням для гри, виявлення переваг та недоліків, аналіз складності вивчення, реалізації та роботи цих методів, оцінка якості та цікавості рівнів та текстур, що могли б бути створені з їх застосуванням, з точки зору гравця, а також розробка програмних продуктів для демонстрації можливостей методів.

Методи розробки: основи комп’ютерної графіки, методи процедурної генерації мап, розробка програмних продуктів на основі фрактальної графіки та генетичних алгоритмів. Інструменти розробки: безкоштовне, вільно поширюване інтегроване середовище розробки Microsoft Visual Studio 2022 Community, мова програмування C#; безкоштовне, вільно поширюване інтегроване середовище розробки Apache NetBeans IDE 12.0, мова програмування Java.

Результати роботи: виконано дослідження жанрів відеоігор та їх привабливості для гравців, процесу розробки гри в жанрі платформер, методів процедурної генерації мапи для гри та створення простої фрактальної графіки, схеми роботи генетичних алгоритмів. Проаналізовано переваги та недоліки роботи алгоритму в

порівнянні з іншими можливостями створення наповнення для ігор. Розроблено три програмні продукти: генератор мапи рівня платформера, створений на основі методу процедурної генерації «Випадкове проходження» та генетичного алгоритму (останній потребує значного перероблення), генератор основи мапи рівня, виконаний на основі методу процедурної генерації «Клітинний автомат», та система для зображення фрактальної графіки, зроблений з використанням можливостей середовища програмування в області комп'ютерної графіки та із застосуванням рекурсивних алгоритмів малювання фракталів.

Генератор мапи рівня платформера може застосовуватися для створення основи для рівнів гри жанру платформер в іншому рушії або стати базою для покращення та розширення функціоналу.

Генератор основи мапи рівня може бути використаний для створення бази для рівнів гри жанру платформер в іншому рушії, однак створені ним мапи потребуватимуть більшої обробки порівняно з попереднім. Тому він також може стати базою для покращення, розширення та поглиблення функціоналу.

Система для зображення фрактальної графіки може застосовуватися для отримання уявлення про вигляд простих двовимірних фракталів окремо чи в поєднанні різних видів, для прийняття рішення про їх використання в роботі, для створення основ зображень для подальшої обробки, або стати базою для покращення, суттєвого розширення та доповнення функціоналу.

ЗМІСТ

ВСТУП

Оцінка сучасного стану об’єкта дослідження. Припускається, що достатня кількість людей, які вирішили вивчати тонкощі створення програмних продуктів, отримали натхнення в комп’ютерних іграх. З часів своєї появи ігри пройшли довгий шлях розвитку – змінювалися старі та з’являлися нові жанри, через постійне поліпшення технологій відповідно графіка ставала більш детальною, звук – якіснішим тощо. Але потужний поштовх людей до розробки зробила поява ігрових рушіїв – програм, що значно спрощували процес програмування для людей із досвідом, а також так званих конструкторів ігор, що дозволяли створювати ігри людям навіть без досвіду. Станом на 2022 рік існує багато таких програм, що мають безкоштовні версії для початківців і стають платними після досягнення користувачем певного прибутку на рік (відомі приклади – Unity, Unreal Engine). Тому розробити гру зараз може майже кожен, хто має ідею, натхнення, терпіння та бажання вчитися.

Звичайні конструктори дозволяють робити досить прості за своїми механіками ігри. Наприклад, у конструкторі рівнів можна з блоків-заготовок створити мапу майбутнього рівня та використати відразу чи завантажити для подальшого переносу на більш потужний рушій. Але складніші системи створення наповнення для гри все ж вимагають навичок програмування. Такими є процедурна генерація рівнів, малювання об’єктів засобами мов програмування тощо. Вже створено багато відомих ігор із застосуванням таких способів.

Актуальність роботи та підстави для її виконання. Комп’ютерні ігри з кожним роком набирають популярність та приваблюють все більшу кількість людей. Кожний гравець має свій улюблений жанр. Для створення ігор, які, найімовірніше, задовольняють потреби ринку, велика кількість студій, від глобальних до зовсім малих, наймають фахівців в усіх областях індустрії комп’ютерних ігор.

Інша справа – розробники-аматори, що створюють продукти не заради виконання замовлення, а за власним бажанням. Іноді розробкою займається одна людина, частіше – невелика група, зазвичай не кожний із членів цієї групи має досвід у програмуванні. Створення гри для такого розробника часто стає тягарем, що вимагає років часу та багато зусиль. Спрощення процесу роботи над одними частинами розробки може дати більше можливостей для роботи над іншими.

Мета й завдання роботи. Метою дипломної роботи є дослідження методів автоматичного створення об’єктів, що можуть слугувати наповненням для гри, виявлення їх переваг та недоліків, аналіз складності вивчення, реалізації та виконання цих методів, оцінка якості та цікавості рівнів та текстур уявної гри, що могла б бути створена із їх застосуванням, з точки зору гравця. Для досягнення мети було сформовано такі завдання.

1) Дослідити методи автоматичної генерації загальної мапи рівня гри жанру платформер, можливості їх використання;
2) Визначити методи деталізації мапи рівня гри;
3) Дослідити можливості не спеціалізованих IDE (Integrated Development Environment, інтегроване середовище розробки) у напрямку комп’ютерної графіки та її створення за допомогою фракталів;
4) Розробити програми для реалізації окремо завдань 1 (в об’єднанні з 2) та 3, провести аналіз їх часової та просторової складностей, оцінити якість отриманих результатів.

Об’єкт, методи і засоби розробки. Об’єктом дослідження є процес розробки засобу для спрощення процесу роботи над комп’ютерною грою в жанрі платформер із застосуванням різних методів процедурної генерації, а також засобу для автоматичного створення графіки, що може стати текстурою для гри.

Розробці передувало вивчення методів процедурної генерації зображень, схожих на мапи для рівнів, вивчення та отримання практичних навичок роботи з базовими засобами комп’ютерної графіки та генетичними алгоритмами під час

виконання лабораторних робіт із курсів «Обчислювальна геометрія та комп’ютерна графіка» та «Інтелектуальні системи» відповідно.

В якості інструментів для створення програм було обрано два інтегрованих середовища розробки. Для задачі процедурної генерації рівнів обрано Microsoft Visual Studio 2022, версія Community – IDE, яке є безкоштовним, вільно поширюваним, ліцензійним за умови виконання входу в профіль Microsoft. Мова програмування – C#. Для задачі роботи з графікою обрано Apache NetBeans IDE 12.0 – IDE, яке теж є безкоштовним і вільно поширюваним, а також не вимагає реєстрації. Мова програмування – Java. Для роботи з нею необхідно додатково встановити Java Development Kit – набір інструментів для розробки.

Бібліотеки C#, що завантажуються разом із середовищем розробки або зі сторонніх джерел, дозволяють включати до програми багато додаткового функціоналу, точний набір якого різний в залежності від версії .NET, для якої створюється продукт. Під час роботи із мовою треба звертати увагу на попередження, які показує середовище – деякі класи працюють тільки на операційній системі Windows.

Java та обране мною середовище розробки також має багато вбудованих класів та бібліотек для виконання різноманітних задач. Java працює на усіх операційних системах, і більшість її бібліотек теж мають цю властивість.

Можливі сфери застосування. Розробка комп’ютерної гри в жанрі платформер – дуже складний процес, і будь-які засоби його спрощення можуть бути корисними для малих студій, окремих команд або розробників-одинаків. Робота демонструє декілька засобів створення мапи рівня, а також засоби створення простої графіки з використанням фракталів. Дане дослідження може бути використане для вивчення можливостей процедурної генерації та генетичних алгоритмів у цій сфері розробки, отримання уявлення про вигляд простої фрактальної графіки. Програмні засоби, за їх вагомого покращення, можливо застосувати для створення повноцінних мап для рівнів для подальшого завантаження, наприклад, до рушію Unity.

РОЗДІЛ 1 ІГРИ, ПРОЦЕС ЇХ РОЗРОБКИ ТА ГРАВЦІ

1.1 Історія комп'ютерних ігор

Комп'ютерні ігри зараз займають вагоме місце в житті багатьох людей, створюються як спеціалізованими студіями, так і програмістами-одинаками, і часто приносять чималі гроші авторам. Але перші ігри не випускалися для широкої аудиторії, а зароджувалися на комп'ютерах у лабораторіях учених.

В 1947 році в США був запатентований перший пристрій, що виводив зображення на екран, реагуючи на дії гравця. Але на майбутнє відеоігор він не вплинув, і думки щодо можливості назвати його власне відеогрою розходяться.

1950-ті роки були для США так званою епохою фальстартів. Одиничні зразки пристроїв створювалися для тестувань або демонстрацій та розбиралися, бо їх автори не вважали за потрібне розвивати свої ідеї, а ігри називали пустими витратами часу. Потужність комп'ютерів стрімко зросла у 1960-х роках, і це значно розширило спектр задач, які вони могли виконувати, а також зробило техніку доступною для ширшого кола користувачів.

У 1962-му році Стів Рассел у складі групи студентів Массачусетського технологічного інституту, США, створив програму Spacewar! – першу відому комп'ютерну гру. Комп'ютер PDP-1, на якому її розроблено, коштував 120 тисяч доларів США, його купляли для лабораторій та інститутів, тому кількість проданих копій гри обмежилася декількома десятками. Але саме Spacewar! стала першою грою, яку можна було встановити на декілька комп'ютерів.

Загалом протягом 1960-тих та 1970-тих років створювалось багато ігрових проектів для навчання та практики в програмуванні або для розваги. Їх складність та якість зростали разом зі зростанням доступності та потужності технологій, а також із розвитком мережевих технологій, в першу чергу з появою прототипу сучасного Інтернету – мережею APRANET.

В 1967 році американський інженер Ральф Баер почав розробку проекту пристрою для ігор, який можна було б під'єднати до телевізора. Прототип отримав назву Brown Box. В 1971 році було підписано контракт із компанією Magnavox, і пристрій отримав назву Magnavox Odyssey. Це перша домашня ігрова приставка (консоль). Ральфа Баера через багато років назвуть батьком відеоігор.

Приставка завоювала величезну популярність, попри свою високу вартість. Одна із ігор для неї стала джерелом натхнення для виробника ігрових автоматів – компанії Atari, які в 1975-му випустили свою гру Pong для власних спеціальних приставок. Саме Atari в 1977-му році дали початок другому поколінню консолей, випустивши Video Computer System (Atari 2600), яка мала прилади для керування (джойстики) та змінні картриджі, що могли відображати на екрані кольорове зображення. Першим представником цього покоління в 1976-му році стала Channel F – ігрова приставка компанії Fairchild.

Розповсюдження домашніх комп'ютерів підштовхувало тогочасних розробників до створення ігор для них. Empire, створена Уолтером Брайтом у 1977-му році – перша гра, що не була адаптацією карткової чи настільної гри. Вона мала складний алгоритм дії і вважається прототипом сучасних комп'ютерних стратегій.

1978-й рік став роком народження відразу двох популярних ігор. Zork вважається прабатьком більшості сучасних текстових та графічних квестів, а Space Invaders – ігор жанру shoot 'em up. В 1979-му році з'явилася компанія Activision – перша компанія, що розробляла ігри, не виробляючи пристроїв для них.

До 1983-го року ринок відеоігор США став перенасиченим, що призвело до банкрутства декількох виробників консолей та комп'ютерів. Відродження індустрії розпочалося, коли на американський ринок прийшла японська компанія Nintendo Entertainment System (NES). На японському ринку вона стала відомою ще в 1980-му році, після появи автоматів із грою Donkey Kong. У тому ж 1980-му в Японії компанія Namco випускає гру Pac-Man, що отримала величезну популярність після виходу в США і особливо після перевидання для майже усіх тогочасних

персональних комп’ютерів. В 1983-му році Nintendo створює Super Mario Bros, яка з плином часу перетворилася на серію ігор, яку полюбляють гравці усього світу.

1982-й рік – рік появи компанії Amazin’ Software, пізніше перейменованої на Electronic Arts. Вона стала першою компанією, яка займалася популяризацією людей, що працювали над іграми. У 1984-му році популярності набув жанр платформер. 1986-й – рік появи компанії Ubisoft. 1987-й – створено стандарт VGA, що відображав на комп’ютерних екранах графіку з 256-ма кольорами.

У 1989-му році Nintendo випускає нову, 8-бітну консоль – Game Boy, разом із грою Tetris, а вже в 1991-му також японська компанія SEGA вперше запускає гідного конкурента – Sega Genesis, на якій вийшла гра Sonic the Hedgehog. Ця подія стала початком так званої консольної війни між Nintendo та SEGA. В подальшому SEGA отримає велику популярність у США, але не зможе зробити цього в рідній Японії.

1992-й рік став роком появи таких відомих ігор, як Mortal Combat, Wolfenstein 3D (вважається першою грою в жанрі шутер від першої особи. Батьком цього жанру є Hovertank 3D, вихід якого відбувся в 1991-му році), Alone in the Dark (дала початок жанрам survival horror та action-adventure), Dune II (заклала принципи стратегій в реальному часі).

1996-й рік – рік зародження кіберспорту разом із виходом Quake. Resident Evil закріпив поняття survival horror, вихід Diablo популяризував жанр hack & slash.

1990-ті роки в загальному вважаються роками інновацій. Відбувався перехід від растрової графіки до полігонального 3Д, з’являлися нові жанри, а також ігри, які на 2022-й рік стали серіями, наприклад, DOOM (1993-й) та Need for Speed (1994-й). Також ігри робили все більший акцент на сюжеті, а згодом і на атмосфері, і приваблювали гравців вже не тільки ігровим процесом. Озвучення персонажів мало все більше значення, сюжети почали виходити на передній план. З’явилася концепція відкритого світу, яким гравець міг би пересуватися за власним бажанням, а не тільки за вимогами гри. Вийшли відомі ігри жанру постапокаліпсис – Half-Life (1998-й) та Fallout (1997-й), останній дав гравцю можливість пройти гру великою

кількістю способів і отримати різні результати. В тому ж 1997-му з'явилася Grand Theft Auto (GTA), в 1999-му – Silent Hill.

Персональні комп'ютери мали більшу потужність порівняно з консолями, могли читати з дисків зображення і звук, а також, завдяки миші та клавіатурі, могли забезпечити виконання більшої кількості ігрових команд та швидкого наведення курсору на ціль. Але водночас велика їх кількість використовувала DOS, на яку важко було щось встановити, а розробники могли використовувати різні звукові карти, через що ледь не кожна гра виходила разом із комплектом драйверів. Із виходом Windows 95 ця ситуація значно покращилася.

Невдовзі до початку нового тисячоліття SEGA представила нову консоль – Dreamcast, першу 128-бітну консоль. Саме для неї з'явилися багато успішних ігор цієї компанії, починаючи із виходу Sonic Adventure (1998-й). Вже в 2000-х роках Sony представила PlayStation 2(2000-й), Nintendo випустила GameCube (2000-й), Microsoft увійшли на ринок консолей із Xbox (2001-й) та грою Halo.

У лютому 2000-го року вийшов симулятор життя The Sims і завоював величезну популярність. Із захистом коду комп'ютерних ігор були проблеми, ігри швидко копіювалися і неконтрольовано поширювалися (з'явилося так зване піратство). Тому багато розробників комп'ютерних ігор або припинили роботу, або випускали ігри пізніше, ніж їх консольні версії.

2001-й рік став знаковим також для української ігрової компанії GSC Game World – вийшла гра Козаки: Європейські війни, яка згодом стала популярною в усьому світі. Російська гра Дальнобойщики 2 стала відомою на території колишнього СРСР, отримала англомовне перевидання.

Жанр ігор MMORPG (Massively multiplayer online role-playing game, масова багатокористувацька онлайнова рольова гра) почав набувати популярності із виходом Lineage II та World of Warcraft (2004-й). Із поширенням дешевого та швидкісного Інтернету з'явилася технологія Flash, із застосуванням якої було

зроблено величезну кількість ігор, в які стало можливим грати в інтернет-браузері без необхідності встановлювати додаткові програми.

2007-й рік – рік виходу українського S.T.A.L.K.E.R.:Тінь Чорнобиля. Гра отримала статус культової на території колишнього СРСР. У 2010-му вийшла Metro 2033 також від українських розробників 4A Games. Обидві гри стали широко відомими за межами України, отримали англомовні видання. На ринку США Microsoft представили Xbox 360 (2005-й) та випустили гру Halo 3 (2007-й). Nintendo Wii (2006-й) отримав революційний контролер, що відстежував рух руки гравця і використовував цю інформацію для керування персонажами гри. Вже в 2010-му Microsoft показали технологію Kinect, що за допомогою камер стежила за рухами всього тіла гравця.

У 2010-тих роках зросла популярність смартфонів, впали ціни на них. Це дало запит на розвиток мобільних відеоігор. Першим значним успіхом стали Angry Birds (2011-й). В успішних досі iPhone (Apple) з’явився конкурент – смартфони з операційною системою Android. Була винайдена можливість керувати напрямком руху персонажа, нахиляючи смартфон або планшет. Мобільний ринок досі займає значне місце в індустрії, щорічно з’являються десятки нових ігор.

Ще у 2011-му році світ побачив прототип технології AR (Augmented Reality, доповнена реальність) разом із грою Skylanders: Spyro’s Adventure. Сучасна доповнена реальність створюється за допомогою камери пристрою, що сканує навколишній простір та «домальовує» речі, необхідні програмі.

Технології віртуальної реальності (VR, Virtual Reality) беруть початок ще у 1980-х роках, коли з’явилися перші програми, що дозволяли користувачу маніпулювати предметами на екрані рухами руки. Сучасні VR-пристрої можуть бути як лише спеціальними окулярами та парою контролерів, так і великими кімнатами з рухомою підлогою. Такі засоби застосовують не тільки для ігор, а й для військових та наукових цілей.

1.2 Жанри відеоігор та їх особливості

Жанром називають категорію ігор, що мають схожі риси ігрового процесу. Жанр визначається не обставинами, що оточують гравця, або історією, яку гра розповідає, а яким чином гравець взаємодіє з оточуючим середовищем.

Жанри охоплюють велику кількість ігор, через що деякі жанри можуть поділятися на менші, так звані піджанри. Сучасні ігри часто поєднують риси декількох жанрів, які їм в подальшому присвоюють, різниця між жанрами також іноді розмивається. Різні дослідники виділяють різні списки жанрів.

Основні жанри комп'ютерних ігор такі:

- Екшн (Action, бойовик) – гравця приваблюють своєрідним випробуванням його фізичних можливостей, наприклад, швидкості його реакції, координації очей і дій рук. Один із найстаріших жанрів, що породив багато піджанрів. Основні приклади таких:
  1) Платформер (Platformer) – зосереджує гравця на необхідності стрибати та лазити ігровим простором. Персонаж зазвичай може стрибати набагато вище власного зросту. Рівні можуть бути просто мапами, які треба пройти за певним маршрутом, а можуть містити ворогів;
  2) Шутер (Shooter – стрілець) – гравець має вибір із великої кількості вогнепальної зброї, за допомогою якої може знищувати ворогів на відстані. Конкретні задачі гравця залежать від особливостей ще дрібніших піджанрів. Основний поділ шутерів відбувається за перспективою: від першої особи (First person shooter) чи від третьої (Third person shooter);
  3) Файтинг (Fighting, битва) – на відміну від шутерів, зосереджуються на боях на близькій відстані, як рукопашних, так і з застосуванням холодної зброї. Зазвичай мають багато персонажів, яких можна вибрати для гри, а битви проходять у форматі один на один;

4) Стелс (Stealth, скритність) – гравець має слідкувати за тим, щоб його персонажа не помітили вороги, і розправлятися з ними, не привертаючи уваги. Ігри цього піджанру часто містять шкали, наскільки видимий персонаж та наскільки добре його чути;
5) Ритм (Rhythm game) – кидає виклик почуттю ритму гравця. Більшість таких ігор пов'язані з музикою, і гравцю необхідно вчасно натискати вказані на екрані кнопки;

- Пригода, або Квест (Adventure) – найважливішими у такій грі є дослідження світу та сюжет, а ігровий процес найчастіше складається із головоломок. Побічні активності, пов'язані, наприклад, з боями, мінімальні або відсутні. Теж дуже старий жанр. З часом породив свої піджанри, основні з яких:
  1) Текстовий (інтерактивна розповідь) – гравець читає опис оточення на екрані і друкує, що зробить далі. Започаткував жанр квестів;
  2) Графічний – гравець роздивляється зображення якогось місця і друкує, що робити далі. Із все ширшим застосуванням комп'ютерних мишок перетворився на жанр point & click («вкажи і натисни»), де гравець може кліком миші на об'єкт на екрані виконати якусь дію;
  3) Інтерактивний фільм (Interactive movie) – велика частина гри проходить у форматі фільму, але в певних ситуаціях гравець може вирішити, наприклад, у який бік піти чи яку репліку скаже персонаж. Всі дії, що можна зробити, вписані до гри, повної свободи гравець не має;
- Пригодницький бойовик (Action-Adventure) – цей жанр поєднує елементи двох зазначених у його назві. Зазвичай проходить від третьої особи. Від квестів береться акцент на сюжеті, велика кількість персонажів, головоломки тощо. Від бойовиків – активне переміщення локаціями, збільшена кількість бойових сцен, а також пряме керування персонажем;
- Від зазначеного вище Action-Adventure в окремий жанр відокремився Survival Horror (виживання у жахах). Такі ігри зосереджуються на

насиченій атмосфері страху та тривоги, серед якої персонаж гравця має вижити. Якщо бої із ворогами дозволені, гравцю не дають такої впевненості в собі, як у бойовику, через певні обмеження – малу кількість чи ламкість зброї та броні, малий запас здоров'я, низьку видимість тощо. Гравець має розв'язувати головоломки та шукати шлях до нових областей мапи;

- Головоломка (Puzzle) – вимагає від гравця розв'язування різних задач, для чого часто треба застосувати логіку, зрозуміти процес роботи тощо. Ігри такого жанру поділяються на різні піджанри залежно від конкретних задач, які вони пропонують. Це фізичні (де застосовуються реалістичні закони фізики), експериментальні (trial-and-error; гравець має зрозуміти, як працюють механізми на локаціях), три-в-ряд (tile-matching; мапа рівня складається із певних об'єктів, які гравець має видаляти, суміщаючи по три чи більше однакових в рядок чи стовпець) тощо;
- Рольова гра (Role-playing game, RPG) – гравець приміряє на себе роль персонажа, що протягом гри стає сильнішим та більш досвідченим. Гра робить великий акцент на необхідності розвитку персонажа через певні класи, кожен з них має свої особливості, і гравець має вибирати ті, які хоче використовувати далі. У поєднанні з бойовиком утворює Action-RPG, яскравим прикладом яких є Diablo;
- Rougelike (rouge-подібні ігри) іноді визначають як піджанр RPG, іноді – як окремий жанр. Його особливістю є майже невідворотна загибель персонажа, випадкова генерація рівнів за певним алгоритмом та покроковий ігровий процес;
- Симуляція (Simulation) – якомога точніше відтворює реальне явище чи процес разом із усіма його властивостями і дає гравцю можливість керувати ним. Основні піджанри:
    1) Симулятор менеджменту та будування – гравець займається розширенням та розбудовою певних установ, від магазинів чи спільнот

до держав, має реагувати на події, що можуть залежати від його попередніх дій або бути випадковими, і приймати відповідні рішення;

2) Симулятор життя – гравець має розвивати одну чи декілька віртуальних живих істот. Ігровий процес може зосереджуватися на окремій істоті, відносинах між декількома істотами або на цілій екосистемі;
3) Симулятор керування транспортом – такі ігри часто вимагають додаткових спеціальних пристроїв, наприклад, контролерів у вигляді керма та педалей. Гра демонструє детальне відтворення реальних механізмів, часто має складне керування;

- Стратегія (Strategy) – гравець має керувати не окремим персонажем, а умовними групами, задачі, які йому треба виконувати, достатньо глобальні. Ними можуть бути, наприклад, створення та розвиток імперії і водночас боротьба з іншими гравцями за ресурси, будівництво міста за умови обмеженості ресурсів, командування армією під час бою тощо. Основний поділ стратегій відбувається за так званою часовою організацією: у покрокових стратегіях (Turn-based strategy) гравці діють почергово, у стратегіях в реальному часі (Real-time strategy) – одночасно;
- Спортивна (Sports) – такі ігри відтворюють особливості видів спорту, зазвичай у вигляді змагань із тренуваннями. Супротивники гравця можуть бути як людьми, так і підконтрольними штучному інтелекту;
- Гра жахів (Horror game) – згаданий вище Survival horror є його піджанром. Такі ігри зосереджуються на викликанні напруженості, тривоги, страху, і майже завжди мають пророблену історію, намагаються налякати гравця;
- Пісочниця (Sandbox) – не зовсім жанр, а більше набір елементів ігрового процесу. Може не мати кінцевої мети. Гравець має можливість робити із зазвичай відкритим ігровим світом все, що забажає. Такі ігри дають великий простір для творчості.

1.3 Етапи розробки гри

Розробка гри – це комплексний процес, який потребує часу, технічних можливостей та, зазвичай, розумових зусиль. Необхідність цих ресурсів не залежить від того, є гра результатом дипломної роботи студента, проектом розробника-одинака, зробленим заради реалізації ідеї, мобільним додатком групи розробників, створеним лише заради прибутку, чи продуктом відомої ігрової студії, виконаним спеціально для світового ринку.

Залежно від дослідника, процес розробки гри поділяється на різну кількість етапів, але часто риси дрібних, вказаних в одній статті, можна об’єднати в більший, вказаний в іншій. Будь-яка гра, незалежно від її подальшої успішності, має пройти ці етапи. Деякі команди не приділяють достатньої уваги плануванню, через що виникає безлад в організації, і процес роботи сповільнюється. Кожний великий крок розробки ставить перед розробником запитання, на які він має дати відповідь.

Перший етап – підготовчий, або передуючий (Pre-Production). Іноді в окремий етап, що має відбутися до зазначеного, виносять планування.

Планування ставить перед виконавцями питання про:

- Бюджет (який, хто його надає, чи треба віддавати відсотки з продажів);
- Тип гри (2Д чи 3Д, якщо 3Д, то який саме);
- Ігрову платформу (персональний комп’ютер – яка система чи для всіх; консоль – яка саме; мобільні пристрої – iOS (операційна система пристроїв Apple), Android чи якась інша; браузерна гра);
- Жанр гри (може залежати від бюджету, але не обов’язково);
- Цільову аудиторію (для кого розробляється гра).

Під час планування відбувається також аналіз ринку для виявлення тенденцій і передбачення, гра якого типу може мати успіх у майбутньому, обраховуються та зважуються ризики. Вже тут можуть з’являтися ідеї майбутньої рекламної кампанії,

роздуми над отриманням та розподілом усіх типів виробничих ресурсів, а також концепт-арти – ескізи графіки, за якими потім можуть бути створені об’єкти у грі.

Коли всі зазначені вище питання вирішені, починається складання дизайн-документу, в якому міститься більш детальна інформація про майбутню гру. Саме цю частину називають підготовчою, якщо планування винесене до окремого етапу. В дизайн-документі містяться відповіді на запитання про:

- Сюжет гри (чим більше деталей, тим краще) та яким чином він буде показаний гравцю;
- Ігровий процес (як гравець буде бачити ігровий світ та взаємодіяти з ним);
- Ігрові механіки (дії, які може виконувати гравець та інші персонажі. Тут можуть вказуватися системи нагород, досягнення, обмеження тощо);
- Персонажів;
- Зовнішній вигляд гри (до документу можуть бути додані малюнки, які створюють уявлення про вигляд гри. Результат може відрізнятися від них);
- Монетизацію (чи буде гра платною, і якщо ні, як отримати з неї дохід – за допомогою вбудованої реклами чи можливості купляти щось всередині гри за гроші, чи зробити гру безкоштовною).

Під час підготовчого етапу проводиться оцінка технічних можливостей. Вже можуть бути окреслені перші обмеження до системних вимог майбутнього продукту (мінімальні – на якій конфігурації в гру можна буде грати хоча б з мінімальними налаштуваннями, та рекомендовані – на якій конфігурації можна відчути та побачити усі деталі, закладені до гри). Також тут складається розклад та план подальшої роботи над проектом. Дизайн-документ є інструкцією для програмістів та художників, тому до нього включаються всі можливі деталі, від вигляду початкового екрану гри до дрібних елементів.

Вже на цьому етапі програмісти мають створити прототип гри, щоб показати можливий вигляд ігрового процесу (детальна графіка на цій стадії не потрібна,

замість об’єктів використовують прості замінники) та перевірити загальну схему роботи зазначених механік.

Якщо цей етап узгоджено, відбувається перехід до наступного.

Етап розробки (Production) – другий великий етап, мабуть, найвідоміший з усіх, він же найдовший, а часто й найскладніший. Іноді його об’єднують з тестуванням, підготовкою до випуску та власне випуском.

Під час розробки гра проходить декілька кроків розвитку, час досягнення кожного з них необхідно означити в плані:

- Прототип (Prototype) зазначено вище;
- Перша спроба (First Playable) – до прототипу додається проста графіка, замінники перетворюються на повноцінні ігрові елементи, і результат дає уявлення про процес гри майбутнього продукту;
- Вертикальний зріз (Vertical Slice) – повністю готовий невеликий шматок гри, який проходиться за час від декількох хвилин до півгодини та дозволяє оцінити, як виглядатиме вся гра. Такі шматки застосовуються для привернення уваги інвесторів та для підготовки рекламної кампанії;
- Пре-Альфа (Pre-Alpha) – вводиться більша частина деталей гри. Саме тут приймаються рішення про додавання чи вилучення певних речей, іноді дуже значних;
- Альфа (Alpha) – весь функціонал додано, всі ігрові механіки працюють, гру можна пройти від початку до кінця. Графіка ще може бути не завершеною. До роботи беруться тестувальники, які шукають помилки та повідомляють про них розробникам;
- Бета (Beta) – всю графіку додано, основні помилки виправлено. Розробники зосереджуються на оптимізації та «поліруванні»;
- Готовий продукт (Release / Gold master) – гра повністю готова до випуску.

Власне програмування – значна, але тільки частина розробки гри. У великих студіях програмістів поділяють на окремі групи, кожна з яких займається своєю задачею, для прискорення роботи. Різні програмісти можуть відповідати за створення ігрових механік, рівнів, поведінки персонажів та навіть ігрового рушія, якщо є така необхідність.

Інша складна частина – створення графіки. Для двовимірної графіки необхідно створити ескіз, кольоровий малюнок, а потім – анімацію. Тривалість такого процесу для одного об'єкту залежить від кількості деталей. Тривимірна проходить процес створення простої полігональної моделі, більш складної обробки з деталізацією, накладання текстури (зовнішній вигляд), підготовки та власне анімації. Це вимагає знань більшої кількості професійного програмного забезпечення. Іноді заготовку для тривимірної графіки створюють у студіях, фіксуючи рухи людини у спеціальному костюмі – ця технологія носить назву захоплення руху (Motion Capture).

Також увага приділяється оформленню рівнів – це окрема задача розробки. Її можуть виконувати як дизайнер, так і спеціально призначений програміст. Саме він займається наповненням ігрового світу об'єктами.

Звуковому дизайну почали приділяти увагу ще з часів зародження відеоігор. Якщо спочатку звуком була тільки фонова музика, то далі почали додаватися звуки руху персонажів, виконання ними певних дій, додаткові фонові ефекти, що створювали атмосферу, а коли розмови перестали бути тільки текстовими – і озвучування кожної репліки.

Тестування – окремий етап розробки. Тестувальники займаються перевіркою всього, що зробили інші члени команди, виконують всі можливі дії та їх комбінації з метою пошуку помилок та вразливих місць, а також перевірки роботи програми в цілому та кожного елементу окремо. Знайдені помилки документуються та передаються програмістам для виправлення. Також тестування може виявити, якщо гру пройти неможливо, надто складно або, навпаки, надто легко, якщо гра нудна, якщо якісь елементи гри «ламають» її та дозволяють, наприклад, пройти рівень

простішим шляхом (так званий експлойт, з англ. exploit – використання у власних інтересах). Також тут слідкують за балансуванням гри, щоб для кожного елементу існував інший, який можна йому протиставити (якщо його немає, такий елемент називають сленговим словом «імба», з англ. imba, скорочення від imbalanced – незбалансований).

Наступний етап – підготовка до випуску (Pre-Launch). Графік доходить до Альфа- та Бета-кроків. Розгортається широка рекламна кампанія, щоб привернути увагу потенційних покупців. Якщо існує можливість, гру показують на спеціальних конференціях. Тут може розпочатися так зване відкрите бета-тестування – окремим гравцям дозволяють пограти у гру до запуску та допомогти знайти помилки, які ще не помічено. За відгуками таких людей розробники дізнаються іншу точку зору на гру і, за необхідності, приймають рішення про певні корегування.

Остання частина – випуск (Launch). Завершуються останні підготовчі дії, виправляються помилки. Якщо додається щось нове, то найчастіше воно стосується покращення графічної складової. Випуск гри – це її поява в магазинах, де гравці можуть її купити або завантажити, якщо гра безкоштовна.

Третій великий етап – супровід (Post-Production), який, на відміну від інших, не розділяють на менші частини. Гравці починають давати відгуки про свій досвід під час гри та надсилати розробникам повідомлення про знайдені помилки. Весь цей матеріал збирається і використовується для виправлень. Розробники можуть випускати патчі (Patch – латка) з цими виправленнями, а можуть включати їх до оновлень. Оновлення зазвичай включає також нові елементи гри, від зміни графіки до введення нових механік, рівнів, персонажів тощо. Можливі випуски окремих доповнень із більшою кількістю додаткових матеріалів – так зване DLC (Downloadable Content). Розробники слідкують за реакцією гравців на оновлення та вводять зміни відповідно до цього.

1.4 Як люди грають в комп'ютерні ігри

Ігри – настільки ж давня річ, як і саме людство. Але саме комп'ютерним іграм приділяють значну увагу світові засоби масової інформації та дослідники. Ігри дитини з об'єктами навколишнього світу досить легко пояснити бажанням пізнати цей світ та зрозуміти, як він влаштований. Але любов підлітків та дорослих до несправжніх світів та героїв, що в них живуть, готовність проводити у цих світах велику кількість часу навіть сьогодні може викликати подив та запитання – чому люди грають в комп'ютерні ігри?

Відповідей на це питання можна дати досить багато.

По-перше, для багатьох людей їх повсякденне життя швидко стає нецікавим, із нього зникають різноманітність та нові враження. Ігри зі своїми проробленими світами, персонажами та історіями дають можливість компенсувати нестачу вражень, зробити щось нове, або й створити та прожити інше життя. Велика кількість доступних ігор дозволяє отримати купу різноманітних вражень.

По-друге, ігри майже завжди містять у собі систему заохочень та нагород. Бажання їх отримати підштовхує гравця грати краще, швидше та уважніше, щоб опанувати гру. Відчуття перемоги зазвичай є важливим для гравця.

По-третє, деяким людям у реальному житті може не вистачати свободи дій. Вони занурюються в ігри, де щось контролюють, роблять те, що захочуть. Також ігри з відкритим світом дозволяють мандрувати, вивчати його, знаходити нові області та предмети. Цікавість та бажання гравця дізнатися, що де заховано, утримують його у світі гри.

По-четверте, ігри – це можливість так званої безпечної помилки. Ризики, з якими доводиться стикатися в житті, часто несуть із собою відповідальність, яку не кожна людина хоче брати на себе. Ризики у грі (навіть у багатокористувацькій) такого не мають. Якщо ворог вбив персонажа, можна почати рівень спочатку. Якщо програно змагання, можна перейти до іншого. Визнавати свою поразку в реальному житті також значно складніше, ніж у грі.

По-п’яте, деякі ігри є готовим полотном для творчості. Дитина будує башту з конструктора, дорослий – із віртуальних блоків. Такі блоки стають для людини засобом самовираження. Створення та розвиток персонажа також може бути метою гравця, котрий бажає творити.

По-шосте, ігри дають гравцям можливість ризикувати та випробовувати себе безпечним чином, даруючи сплески емоцій, яких іноді не вистачає людям. Прагнення перемогти, докладання зусиль, велика кількість спроб і, нарешті, досягнення результату – ейфорія від цього моменту знайома багатьом гравцям. Інший своєрідний ризик виникає, якщо гра дає можливість проходити її різними способами, а зміни у віртуальному світі залежать від попередніх дій гравця. Деякі ігри пропонують завдання, які вимагають прийняття рішень з точки зору моралі, де необхідно зважувати наслідки того чи іншого вчинку. У багатокористувацьких іграх також треба зрозуміти, з якими гравцями як спілкуватися, слідкувати, щоб випадково не вдарити союзника, у битві гравець ризикує померти та втратити всі свої речі та ресурси. Відчуття ризику та небезпеки всередині гри і одночасно відсутність їх в реальному житті часто подобаються гравцям.

По-сьоме, якщо гравець не відчуває мети в житті, гра може дати йому хоча б тимчасову мету, якої він буде прагнути. Мета в реальному житті часто буває достатньо абстрактним поняттям, а шлях до неї може здаватися біговою доріжкою – сили витрачаються, а результату не видно. У грі легше як означити шлях до віртуальної мети, так і відстежувати свій прогрес. Це також вабить гравців.

Ігри з реалістичними сценами насильства часто провокують критику. Деякі дослідники вважають, що такі ігри роблять гравців більш жорстокими (найчастіше в цьому звинувачують шутери). Але більш нові дослідження не виявляють сильного зв’язку між іграми та посиленням жорстокості. Навпаки, все частіше вчені доходять думки, що подібні ігри дозволяють гравцю розслабитися та виплеснути до гри накопичену за день злість.

Звичайно, всі гравці є різними, і їхні цілі, з якими вони сідають за улюблені ігри, різні, але загальні тенденції все ж можна прослідкувати. Грають люди також по-різному. Особливо сильно це помітно, якщо одну й ту саму гру можна пройти декількома способами. Тоді у, наприклад, двох блогерів, що знімають проходження цієї гри для сервісу YouTube, можуть вийти протилежні або просто різні результати. Хтось прагне розслабитися після важкого дня, погуляти ігровим світом та помилуватися віртуальними краєвидами. Хтось – зігнати злість на несправжніх ворогах. Хтось – відчути себе переможцем у змаганнях з іншими людьми. Хтось – отримати контроль над віртуальним світом та зробити його кращим або гіршим. Відповідно змінюються й стилі гри. Але є те, що приваблює якщо не всіх, то принаймні велику кількість гравців – ігрові системи нагород. Ці речі підштовхують гравця, а їх досягнення приносить йому задоволення. Навіть в іграх, де немає явного кінця, зазвичай є певні досягнення або така свобода вибору, що гравець може сам визначити свою мету (наприклад, у грі Minecraft умовна мета – дістатися останнього виміру та вбити дракона, але гра по цьому не завершується і продовжується нескінченно довго).

Окрім нагород, гравцю також можуть приносити задоволення історія, яку розповідає гра, графіка та звук, що складають атмосферу гри (ледь не основна причина любові гравців до серії ігор S.T.A.L.K.E.R.), можливість контролювати те, що відбувається в ігровому світі, і, частіше в багатокористувацьких іграх, спілкування з іншими людьми.

1.5 Висновок із розділу 1

Після вивчення теорії можна зробити такі висновки.

Розробка гри починається з аналізу ринку (за можливості), визначення цільової аудиторії та жанру гри. Якщо велика студія з відповідним фінансуванням може дозволити собі промахнутися (хоча це майже неодмінно вдарить по репутації, тому цього намагаються не допускати – варто згадати досвід Cyberpunk 2077), то розробникові-одинаку, який хоче створити гру і планує компенсувати витрачені зусилля отриманим прибутком, невірне розставлення пріоритетів може коштувати принаймні задоволення від процесу розробки.

Але якщо план вже складено (пройдено підготовчий етап), найбільш складними для одинака або невеликої команди ймовірно стануть процеси створення графіки та рівнів для гри. Саме їх можна спрощувати, щоб розширити можливості розробників, надаючи їм додатковий час на роботу над іншими частинами гри.

Для подальшого розгляду обрано жанр платформер у двовимірному просторі. Нові приклади таких ігор з’являються постійно, а статистика відгуків та завантажень чи покупок свідчить про популярність жанру як серед старшої аудиторії, так і серед молодої. Ігри цього типу випускаються як для персональних комп’ютерів, так і для консолей і мобільних пристроїв, тому гра може охопити велику кількість гравців.

Відповіді на запитання розділу 1.4 натякають на необхідність зацікавити гравця, заохотити його до продовження гри. Тому під час подальшої роботи буде враховано фактор цікавості рівнів з точки зору гравця, не обмежуючись тільки можливістю пройти гру.

РОЗДІЛ 2 СПОСОБИ ПРОГРАМНОГО СТВОРЕННЯ ЕЛЕМЕНТІВ ГРИ

2.1 Способи та особливості роботи з графікою

Комп'ютерна графіка як науковий термін – частина комп'ютерних наук, що вивчає методи створення та зміну зображень засобами комп'ютера. Найчастіше цей термін застосовують до створення тривимірних зображень, але він охоплює і двовимірні, і обробку зображень. У побуті термін «комп'ютерна графіка» використовується у значенні віртуального об'єкта чи зображення, створеного засобами комп'ютера, тобто тільки у значенні результату, а не процесу.

Способи створити зображення чи його анімацію з часом змінювалися. Найкраще розглянути цей процес на прикладі фільмів та мультфільмів.

Перша анімація для них (кінець XIX – початок XX століття) малювалася покадрово вручну на папері. Кожний кадр потребував окремої обробки, бо створювався разом із заднім планом, який мав відповідати створеним раніше. Така анімація вимагала великої кількості часу навіть від великого колективу художників.

Пізніше було зроблено відокремлення заднього плану від переднього та переміщення їх на прозору плівку. Декілька шарів плівки зображували, наприклад, задній план, нерухомі та рухомі частини тіл персонажів окремо. Використання такого способу значно спростило роботу аніматорів, дозволило розділити їх задачу на частини і поставити працю «на конвеєр». Але створення мультфільму у стандартному форматі 24-х кадрів на секунду таким способом – складний процес.

Разом із розвитком комп'ютерів поступово з'являлися програми для малювання різної складності. Спочатку лише для двовимірної графіки, пізніше виникли додатки, що дали змогу перейти до тривимірного простору. Зі зростанням потужності комп'ютерів відповідно розширювалися можливості програм.

Сучасну двовимірну графіку для гри можна створити декількома способами. Покадрова ручна анімація на папері досі існує, але є досить рідкісним явищем

(яскравий приклад гри, створеної із такими зображеннями – Cuphead). Значно більш розповсюдженою є анімація у спеціалізованих програмах.

Для малювання кадру зараз використовують програмне забезпечення для растрової чи векторної графіки. Векторна графіка, на відміну від растрової, підтримує масштабування без розкладання зображення на пікселі, але її об'єкти більш важкі для збереження та обробки. Програми пропонують широкий вибір функціоналу для створення ефектів на зображенні (розмиття різними способами, викривлення, скручування тощо), багатошаровість, що підтримує напівпрозорі шари, повний спектр кольорів тощо. Є можливість малювати як за допомогою миші, так і використовуючи графічний планшет із пером. Приклади програм: Adobe Photoshop (для растрової графіки) та Adobe Illustrator (для векторної графіки).

3Д-моделювання вимагає, в першу чергу, спеціального програмного забезпечення. Приклад такого – Blender, яке дозволяє також створювати тривимірну анімацію. Саме 3Д-анімацію найчастіше називають комп'ютерною графікою. Вона має широке застосування як у проектах аматорів, так і в масштабних кінофільмах – за її допомогою створюються різноманітні об'єкти, від вибухів до цілих локацій.

3Д-анімацію можна створити й, наприклад, за допомогою Motion Capture, або захоплення руху. На відміну від ручної анімації, тут відстежуються закріплені на об'єкті датчики, зміна їх положення передається на комп'ютер, і готова основа переноситься на тривимірну модель. Так анімують, в першу чергу, людиноподібних персонажів, а особливо часто його використовують для передачі виразу обличчя або руху губ під час розмови. Приклад гри – Detroit: Become Human, а також саме його застосування демонстрували GSC Game World при розробці S.T.A.L.K.E.R. 2. Головний недолік цього виду анімації – ціна, використання студії motion capture коштує більших грошей, ніж суто комп'ютерна анімація. Також такі зйомки обмежені законами фізики, а реалістичні рухи іноді важко пристосувати до моделі (наприклад, у змодельованого товстого персонажа руки можуть «провалюватися» в тіло, навіть якщо модель зробили на основі людини відповідної комплекції).

2.2 Фрактали та їх застосування в комп'ютерній графіці

Фрактал (з лат. Fractus – зламаний, розбитий, роздрібнений) як математичний термін – множина, що має властивість самоподібності. Самоподібним є об'єкт, що має таку ж форму, що й одна або декілька його частин. В загальному сенсі фракталом називають об'єкт, який має принаймні одну із трьох властивостей:

- Нетривіальна структура за будь-якого масштабу. Будь-яка регулярна фігура (еліпс, коло тощо) за значного збільшення масштабу буде схожа на елемент прямої. Із фракталами такого не відбувається – на будь-якому рівні масштабування можна побачити складну картину;
- Самоподібність, принаймні наближена;
- Його метрична розмірність є дробовою (розмірність точки – нуль, кривої – рівно один, гладкої поверхні – рівно два, об'єкта ненульового об'єму – рівно три, може мати дробові значення) або перевищує топологічну (приклади аналогічні, але строго цілі).

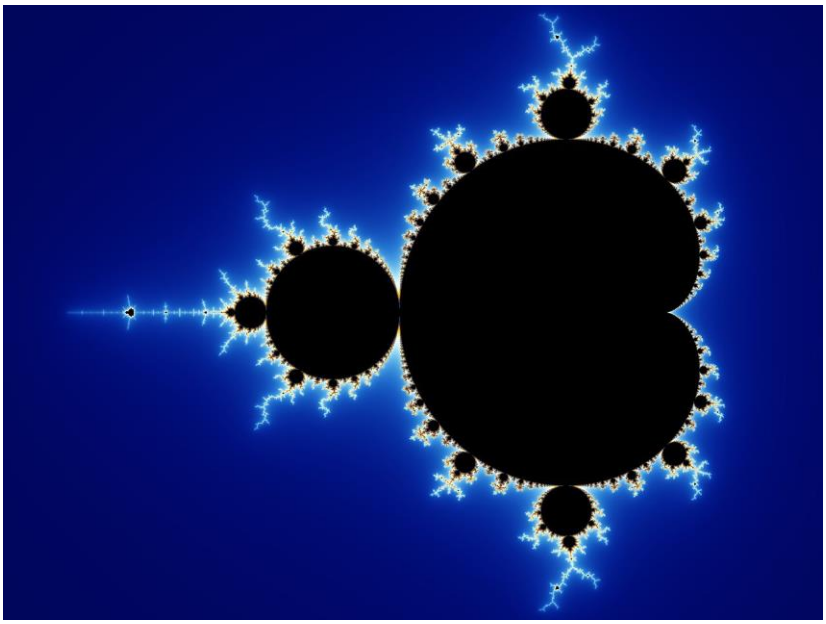


Рисунок 1. Множина Мандельброта

Класичний приклад фракталу – множина Мандельброта. Її елементи – точки c (в комплексному просторі) рекурентного співвідношення $z_{n+1} = z_n^2 + c$, де $z_0 = 0$ і значення всіх подальших ітерацій залишаються в обмеженій послідовності. Базовий

варіант цього фракталу чорно-білий (точка або належить множині (чорний), або не належить (білий)), а кольорові варіанти відповідають модифікаціям (наприклад, за кількістю кроків, після яких можна сказати напевне, що точка не належить множині).

Велика кількість природніх об’єктів мають фракталоподібну структуру. Такі об’єкти мають назву квазіфрактали – вони не мають точного та повного повторення при збільшенні масштабу, а при досягненні певного рівня масштабування фрактальна структура повністю зникає.

Прикладами природніх фракталів можуть слугувати корали, морські зірки та їжаки, рослини (приклади – папороть, капуста романеско, дерева), листя дерев, квіти, бронхи у людей та тварин тощо. У неживій природі це візерунки на склі після морозу, сніжинки, кристали, блискавки, берегові лінії та гірські хребти тощо.

Рисунок 2. Капуста романеско

Фрактали отримали особливу популярність після появи спеціального програмного забезпечення, що дозволяло візуалізувати їх, збільшити масштаб та роздивитися. Велика кількість формул, за допомогою яких можна сформувати фрактал, та широкі можливості програм для їх відображення та зміни спричинили появу фрактальної графіки, як двовимірної, так і тривимірної.

Перевага фрактальної графіки очевидна – до рендерингу (завантаження готового зображення) в пам’яті комп’ютера відповідна програма може не зберігати нічого, крім декількох рівнянь, що описують фрактал. Саме ці ідеї стали причинами появи розробок технології стиснення інформації за алгоритмами.

Однією формулою можна описати навіть достатньо складний двовимірний фрактал. Наприклад, папороть є фракталоподібною структурою, її формула має назву папороть Барнслі. Сніжинка Коха – достатньо точна формула опису природніх сніжинок. Якщо двовимірний фрактал можна певним чином намалювати вручну, то тривимірні, навіть за найпростішими формулами, стають надто складними. Зображення, виконані тривимірними фракталами, можуть носити назву фрактального мистецтва – самостійного напрямку образотворчого мистецтва.

У розробці комп’ютерних ігор фрактали мають своє місце. За допомогою цих структур створюються об’єкти, схожі на природні – дерева, гірські ландшафти, морські поверхні та хвилі, контури континентів та островів, кордони держав тощо. Існують програми, що дозволяють писати музику за допомогою фрактальних формул. Тривимірні фрактали також реалізуються за допомогою програмного забезпечення, як безкоштовного, так і платного.

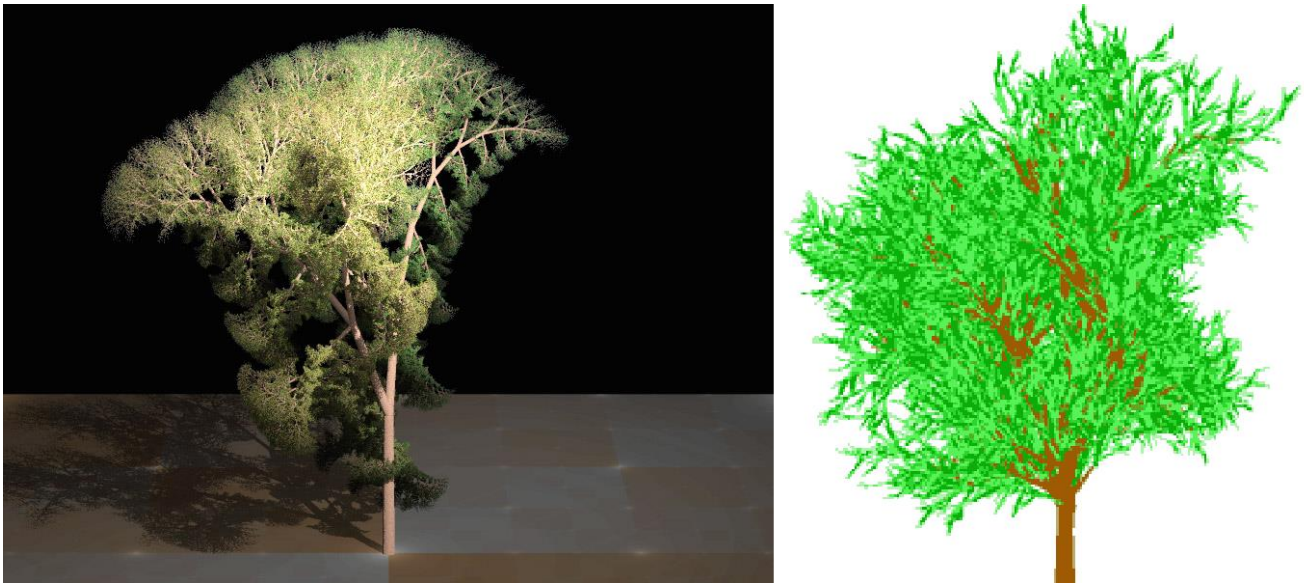


Рисунок 3. Три- та двовимірне фрактальні дерева

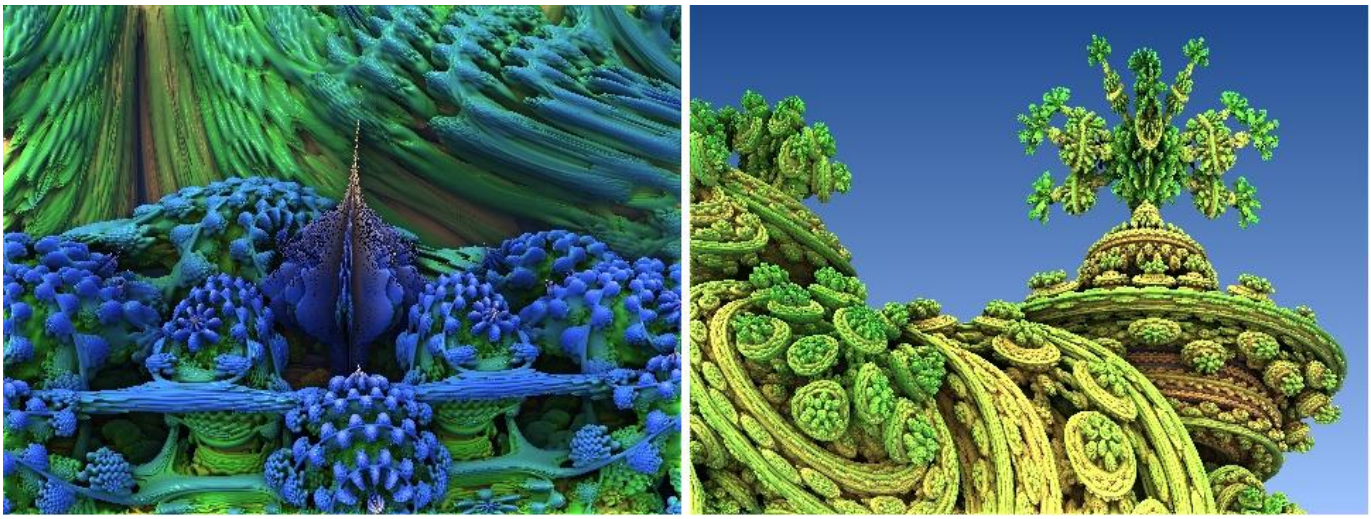


Рисунок 4. Частини тривимірних фракталів

Тривимірні фрактали на рисунку 4 створені програмою Mandelbulb 3D (безкоштовна, тільки для Windows).

Навіть не дуже якісні рендеринги частин даних фракталів схожі на інопланетні пейзажі. Фрактал на лівому рисунку генерується за формулою Integer Power, фрактал на правому – BT1Pine, обидва зі стандартними параметрами.

За допомогою фракталів також можна створювати плани будівель, і не тільки вигляд згори. Наприклад, відносно проста губка Менгера (тривимірний аналог килиму Серпинського) схожа на футуристичну будівлю з вікнами різних розмірів. Двовимірні фрактали можуть генерувати малюнки, що виглядають майже як ісламські геометричні візерунки.

Фрактали вважаються прикладом ідеального балансу між впорядкованістю та хаосом, чим приваблюють велику кількість людей. Звичайно, всім подобаються різні вигляди фракталів. Наприклад, на веб-сайті Deviantart тег fractal містить велику кількість зображень як дво-, так і тривимірних фракталів, кожний із яких, як майже порожні, так і дуже складні, має певну кількість вподобань.

2.3 Особливості рівнів платформера та їх створення

Як зазначено у розділі 1.2, платформер ставить гравцеві завдання дістатися кінцевої точки, стрибаючи платформами та, можливо, знищуючи або уникаючи ворогів та перешкод. Персонаж зазвичай вміє ходити, бігати та стрибати, а гравець може керувати ним навіть у повітрі. У деяких іграх є можливість стрибку в повітрі (подвійного стрибку), відштовхування від стін та/або біг по них, політ та плавання на обмежених відрізках рівня тощо.

Двовимірна гра в жанрі платформер має вигляд збоку (рух вліво, вправо, вгору та вниз), у тривимірній камеру найчастіше закріплено за спиною персонажа або, рідше, в ізометричній перспективі («діагонально», під певним кутом).

Платформерами також можна назвати так звані раннери (Runner – бігун). Їх головна відмінність полягає в тому, що персонаж раннеру автоматично біжить вперед (або вбік, якщо гра двовимірна), а гравець має своїми діями, наприклад, вказувати йому стрибати чи нагинатися, щоб уникнути його смерті.

Як вказано у розділі 1.5, для розгляду обрано двовимірний платформер.

Основні види платформ, на які персонаж гравця може стати, такі:

- Статичні – персонаж може стояти на них необмежений час і не рухатись;
- Падаючі – персонаж може стояти на них протягом певного проміжку часу. Після цього платформа може зникнути (персонаж впаде «крізь неї») або впасти (залежно від гри персонаж або теж впаде «крізь неї», або падатиме разом із нею та може стрибнути з неї під час падіння);
- Рухомі – найчастіше рухаються горизонтально або вертикально, існують і більш складні моделі руху (коло, дуга тощо). Персонаж рухається разом із платформою. В деяких іграх можуть бути слизькі платформи, які «виїжджають» з-під ніг персонажа, платформи, що обертаються, або можливість померти, тому що платформа притиснула персонажа до землі або стіни;

- Трампліни – як платформи, так і просто ігрові об’єкти. Стрибнувши на таку, персонаж підскочить вище. Нерухомо стояти на такій платформі неможливо;
- Конвеєри – платформа нерухома, але рухає гравця в певному напрямку;
- Небезпечні – можуть як наносити ураження або вбивати відразу, так і мати певний час до спрацювання (як після того, як персонаж на неї стане, так і незалежно від цього, за заданим алгоритмом).

До платформ іноді відносять також ліани та сходи, якими можна піднятися чи спуститися на іншу платформу.

Якщо на рівнях присутні «живі» вороги, вони відрізняються своєю поведінкою. Наприклад, типові вороги у платформері можуть вміти:

- Ходити (за фіксованим маршрутом в одному чи змінному напрямку; переслідувати гравця. Іноді падають з платформ);
- Літати (за фіксованим маршрутом в одному чи змінному напрямку; вільно; переслідувати гравця);
- Стрибати (на визначену або змінну висоту; через визначені проміжки часу або разом із гравцем);
- Стріляти (через визначені проміжки часу; в напрямку гравця або в чітко визначену точку; варіант – скидати предмети на гравця, якщо ворог літає);
- Бити (якщо персонаж гравця помирає не з одного удару);
- Блокувати удари (варіант – ворог із голками, якого неможливо вбити стрибком згори);
- Вороги з більш складним штучним інтелектом можуть імітувати поведінку персонажа гравця або змінювати її, підлаштовуючись під стиль гри, який обрав гравець.

Для платформера характерна наявність більш сильного ворога після певної кількості рівнів. Такий ворог носить назву бос (boss – головний), найчастіше він має

складнішу поведінку, інші механіки дії, більший розмір, значний запас здоров’я та сильнішу атаку. Щоб його перемогти, найчастіше необхідно зрозуміти його схему дії та виробити власну тактику.

Платформер містить багато рівнів, що можуть відрізнятися за стилем, окремими механіками та поведінкою ворогів. Рівні протягом гри зростають за складністю. Гра може мати окремий рівень для ознайомлення з керуванням та основними механіками. Якщо він відсутній, на першому рівні для цього присутні спеціально зроблені частини. Нові механіки часто вводяться поступово.

У грі зазвичай прописано жорсткий порядок рівнів, але існують платформери з відкритим світом, які дозволяють вільно рухатися та діставатися як старих, так і нових рівнів (області, куди іти ще зарано, зазвичай перекриті). Приклад такої гри – Sonic Adventure. Дістатися одного рівня іноді можна декількома шляхами.

Ігрові рівні, окрім платформ та ворогів, містять різні об’єкти, заохочуючи гравця до їх збору. Взяття предмета можуть нагородити гравця ігровими очками, за збір певної кількості персонажу часто видають додаткове життя, предмети можуть виступати в якості ігрової валюти тощо. Деякі предмети знаходяться в прихованих локаціях, за їх знаходження видається більша кількість очків. Інші дають гравцю бонуси, наприклад, захист від ударів (до першого удару або тимчасова повна невразливість) або збільшення певних характеристик.

Іноді платформер містить систему досвіду. Накопичення очків досвіду дає можливість покращення характеристик персонажа, іноді – відкриває доступ до раніше недосяжних областей мапи, рівнів або предметів.

Створення рівнів для платформера можливе різними способами.

По-перше, можна зробити мапу рівня вручну. Цей спосіб, особливо якщо розробник любить ігри цього жанру, дозволяє створювати рівні, що точно відповідають бажанням автора – він здатен точно прописати вигляд платформ та їх функціонал. Головних недоліків цього методу два. Він вимагає величезної кількості

часу, особливо для масштабних ігор, та певних якостей розробника – стійкості, терпіння й багатої фантазії.

По-друге, можна використати програму-конструктор. Це програми, зроблені для спрощення першого способу. З їх допомогою можна просто розставити об’єкти-заготовки на порожній мапі і завантажити для подальшого перенесення до іншого рушія. Деякі ігрові рушії містять вбудовані конструктори, у них є вже прописані моделі поведінки ворогів, платформ, персонажів, переходів між рівнями, регулювання ігрової гравітації та її зміни тощо. Конструктори можуть бути як платними, так і безкоштовними, але останні часто мають урізану функціональність.

По-третє, можна застосувати методи автоматичної генерації мап рівнів. Залежно від способу отримуються різні рівні з різними стилями, а залежно від складності – як накидані випадковим чином набори платформ, так і рівні, що гарантують виконання заданих критеріїв (принаймні можливість його пройти).

Існують програми для алгоритмічної генерації графіки (наприклад, ландшафту для тривимірного рівня), але над мапами рівнів розробник має працювати самостійно. Мережа Інтернет містить багато інструкцій зі створення рівнів, деякі з яких буде розглянуто в наступному розділі.

2.4 Процедурна генерація рівнів, її методи

Процедурна генерація (Procedural Generation) як термін – спосіб створення даних із використанням алгоритмів, де людина надає власне алгоритм, обмеження та вхідні дані для нього, а комп'ютер – потужність та швидкість, а також додає до результату випадковість.

Як було згадано в попередньому розділі, процедурну генерацію найчастіше застосовують для роботи з графікою. Візуалізація фракталів, особливо тривимірних – також результат дії подібних алгоритмів. Якщо подивитися на методи накладання текстур (надання об'єкту певного зовнішнього вигляду), тут процедурна генерація часто спрацьовує – з її допомогою створюються зображення металів, каменю, дерева, води, магми, піску, клітин або схожих на них предметів (наприклад, купи камінців) тощо.

Використання алгоритму як основи створення рівнів в процесі гри відбулося в 1980-му році, з виходом гри Rouge, першого відомого представника жанру rougelike. За умов тогочасних обмежень кількості пам'яті та розмірності екранів це був засіб економії ресурсів комп'ютера – в пам'яті зберігався лише алгоритм, час роботи витрачався тільки на генерацію, а ігровий процес рівнів-результатів був достатньо складним, щоб зацікавити гравця.

Зі зростанням можливостей комп'ютерів проблеми нестачі пам'яті та розмірності екранів відпали. Процедурна генерація перетворилася на засіб додавання різноманітності до ігрових елементів – текстур, рівнів, персонажів тощо. В окремих іграх алгоритми лягли в основу генерації базової мапи, забезпечуючи унікальний світ за кожного нового проходження гри. Приклади таких ігор – Minecraft (2009-й, фактично – 2011-й; 3Д, від першої або третьої особи), Spore (2008-й; 3Д, від третьої особи), The Binding of Isaac (2011-й; 2Д, вигляд згори).

Застосування методів процедурної генерації до двовимірних ігор також можливе. Вони можуть сильно відрізнятися від схем для тривимірних, а їх

результати часто потребують додаткової оцінки людиною, хоча, за умови достатньої складності алгоритму, існує можливість вбудувати процес оцінювання до програми.

Можливі методи генерації мапи для двовимірної гри:

- Випадкове розміщення кімнат (прямокутних) – більше підходить для ігор з виглядом згори або міських рівнів з виглядом збоку. Початковий простір для мапи заповнюється «стінами» (через такий блок персонаж пройти не здатен). Випадковим чином обирається місце для кімнати та її розмір. Якщо кімнату можна розмістити, блоки всередині неї замінюються порожніми. Кімнати випадковим чином з'єднуються коридорами. Основний недолік (без додаткових перевірок) – можливість отримати недосяжну кімнату;
- Розміщення кімнат випадковим поділом навпіл – теж для ігор з виглядом згори або міських рівнів із виглядом збоку. Початковий простір необхідну кількість разів поділяється навпіл, всередині таких секторів випадково розміщуються кімнати та з'єднуються коридорами. Перевага – значно менше порожнього місця порівняно з попереднім методом. Недолік – можливість отримати недосяжну кімнату;
- Клітинний автомат (Cellular Automaton) – створює мапу, більше схожу на печеру для вигляду збоку. Початковий простір заповнюється випадковим чином – із заданим шансом може згенеруватися не «стіна», а порожній блок. Зазначену кількість разів до кожної клітини застосовуються правила – за певної кількості порожніх сусідів «стіна» стане порожньою, за заданої кількості порожній блок перетвориться на «стіну». Переваги – швидкість відпрацювання алгоритму та його відносна простота навіть за великого розміру мапи. Недолік – можливість отримати недосяжну область;
- «Проходка п'яниці», або випадкове проходження (Drunkard's walk / Random walk) – теж для генерації мап, схожих на печери для вигляду збоку. Початковий простір заповнюється «стінами», випадковим чином обирається початкова точка, яка стає порожньою. На кожному із заданої

кількості кроків випадковим чином обирається наступна точка із сусідів теперішньої, вона перетворюється на порожню та відбувається перехід до неї. Перевага – згенерована таким методом мапа не має недосяжних областей. Недолік – час, необхідний для генерації, швидко зростає разом із кількістю кроків та розміром початкового простору (можна покращити збільшенням площі кроку);

- Агрегація, обмежена дифузією (Diffusion Limited Aggregation) – підходить як для вигляду згори, так і для вигляду збоку. Початковий простір заповнюється «стінами». Ближче до центру обирається початкова точка. Вона та її сусіди стають порожніми. На порожньому місці випадковим чином створюється «бігун» або декілька «бігунів», кожен з яких може рухатися або тільки за сторонами, або тільки за діагоналями, або в усіх напрямках. Кожен «бігун» випадковим чином рухається мапою, поки не натрапить на «стіну». Місце, на яке він натрапив, стає порожнім, і процес повторюється. Переваги – згенерована мапа не має недосяжних областей; використання «бігунів» з різними можливостями дає велику кількість результатів. Недоліки – час роботи зростає разом із розміром мапи; для одночасного використання декількох «бігунів» необхідно писати програму для роботи з декількома потоками (паралельна програма), що ускладнює розробку недосвідченому програмісту.

Алгоритми можна комбінувати для створення більш різноманітних мап.

Процедурна генерація спрощує роботу, але без додаткового функціоналу не здатна створити готовий рівень для платформера. Вона може залишати окремі області, що в подальшому будуть використані в якості платформ (якщо не додавати згладжування – прибирання надто малих областей), але шанс появи такого явища відносно малий і залежить від алгоритму, тому покладатися на нього не варто. Платформи та ворогів на рівні можна або розставляти вручну, або доручити алгоритмам, але вже іншим. Це буде детальніше розглянуто в розділах 3.2 та 3.3.

2.5 Сприйняття рівня платформера гравцем

Платформер кидає виклик гравцю, перевіряючи його вміння одночасно швидко та уважно оцінювати місце навколо, приймати рішення та реагувати. Швидкість та точність реакції у грі цього жанру виходять на передній план.

Оскільки ігор в жанрі платформер із часів їх появи вийшла дуже велика кількість, гравці, відкриваючи нову подібну гру, вже здогадуються, що їх очікує, і налаштовують себе відповідним чином. Здивувати досвідченого гравця елементами ігрового процесу можливо, але достатньо складно. Головне для рівня платформера – забезпечити достатню складність, щоб гра не здавалася нудною. Цікавість рівня також частково визначається його складністю. Розглянемо деякі елементи ігрового процесу, що впливають на зацікавленість гравця.

Складність і цікавість рівня з точки зору гравця забезпечуються перш за все темпом проходження. Необхідність довго чекати певного моменту для дії майже напевно роздратує більшість гравців. Але невеликі проміжки часу між стрибками, навпаки, додають тиску та підштовхують до проходження, особливо якщо ці проміжки не очевидні та їх треба зрозуміти. Приклад – рухомі платформи, що рухаються між іноді спрацьовуючими небезпечними частинами рівня. Гравцю необхідно прослідкувати схему, за якою відбувається рух, визначити, як діяти, та пройти наміченим маршрутом. Малий проміжок між платформами, у який треба застрибнути, може викликати азарт у гравця, особливо якщо після такого проміжку він отримає нагороду.

Інший спосіб зробити рівень складнішим з точки зору гравця – обмежити час на проходження. Це можна зробити як безпосередньо (встановити таймер, по завершенню якого персонаж помирає), так і опосередковано, наприклад, за допомогою системи штрафів (наприклад, у серіях ігор Sonic the Hedgehog та Super Mario Bros кількість отриманих за проходження рівня очків залежить від швидкості – чим менше часу витрачено, тим більша нагорода). Таймер, що веде зворотній відлік, чинить постійний тиск на гравця, змушує його діяти швидше.

Також складним сприймається рівень, що містить значну кількість ворогів. Якщо камера не показує відразу весь рівень, раптова поява ворога, особливо нового, швидше за все викличе у гравця напруженість, але не відчуття складності. Але потрапляння на відрізок із великою кількістю ворогів, яких необхідно обійти або знищити, підвищить складність рівня в очах гравця.

Рівні платформера часто містять елементи головоломок. Якісь частини можуть бути закриті дверима, до яких необхідно знайти ключ. Ключем може стати простий предмет, який необхідно підняти – тоді, щоб ускладнити знаходження, його можна розмістити в області, якої складно дістатися, або поставити «охорону» із ворогів. Рідше двері необхідно відчиняти шляхом знищення всіх ворогів у певній області, натисканням однієї чи декількох кнопок/важелів відразу або у визначеній комбінації тощо. Головоломки з плином часу та появою різних ігор стали майже постійною частиною жанру платформер, і їх наявність зробить рівень цікавішим.

Гравця можуть зацікавити вороги зі складним штучним інтелектом, оскільки треба здогадатися, як з ними боротися, а відчуття перемоги – один із найважливіших факторів задоволення від гри. Використання платформ зі складною схемою руху (хвиля, маятник, дуга, більш заплутана траєкторія) також підвищить складність.

Всі згадані елементи, а також багато інших, в поєднанні один з одним і за правильного балансування складають основну частину складності рівня. На цікавість у більш загальному сенсі впливають також графіка (розглядання фонів чи окремих текстур іноді захоплює гравця більше, ніж власне ігровий процес) та звук, що створюють атмосферу рівня. Найголовніше – балансування, оскільки через надто складний рівень гравець може закинути гру, а надто легкий не зацікавить його. Але якщо утримати баланс між простотою і так званим хардкором (hardcore – для ігор найближчий переклад – жорсткий), результат може сподобатись більшості гравців.

Деяким гравцям подобаються значно складніші рівні, на проходження яких витрачається багато часу та зусиль. До більшості ігор додаються налаштування складності, а для таких гравців можуть вводитися окремі спеціально зроблені рівні.

2.6 Висновок із розділу 2

Після вивчення більш детальної інформації можна зробити такі висновки.

Засоби створення наповнення для рівнів існують, їх використовують для генерації графіки, схожої на елементи реального світу. Застосування фракталів для таких задач дозволяє досягти достатньої схожості зі справжніми об’єктами.

Складні тривимірні фрактали створюють видовищні фони для рівнів, але можуть налякати розробника об’ємом роботи та досліджень, на відміну від двовимірних. Для отримання уявлення про такий фрактал достатньо поцікавитись його назвою в мережі Інтернет. Саме генерацією таких структур розробник може зацікавитися. Забезпечення його програмним продуктом для створення простих фракталів за різними формулами дасть можливість створювати просту графіку для подальшої обробки, що значно спростить роботу в порівнянні з малюванням з нуля.

Створено багато алгоритмів процедурної генерації мапи рівня, але всі вони залишаються в межах інструкцій в мережі Інтернет. Навіть якщо вони містять зразки коду, прикладів їх реалізації у вигляді програмного засобу знайдено не було.

Для створення основи для рівня платформера найкраще підходять алгоритми клітинного автомату та випадкового проходження. Їх реалізацію буде виконано та описано в наступному розділі.

Повноцінні рівні неможливо створити без платформ, що висять у повітрі. Про це згадувалося в розділі 2.4. Але обрані алгоритми або взагалі не залишають таких місць, або залишають з невеликим шансом та випадковим чином, що не забезпечує можливості пройти рівень. Для заповнення рівня платформами буде застосовано додатковий функціонал, опис якого також буде надано в наступному розділі.

З урахуванням розділу 2.5 необхідно оцінити цікавість рівнів-результатів з точки зору гравця.

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА

3.1 Робота з фрактальною графікою

Якщо розповісти людині простими словами про фрактали і спитати, що вона собі уявляє, імовірно, вона скаже про рослину. Як було згадано в розділі 2.2, найбільш очевидний приклад природного фракталу – дерево.

Класичний деревовидний фрактал починається з однієї лінії (стовбура), від кінця якої відходять дві коротші (гілки) під певним кутом, і процес повторюється для кожної з них. Таке дерево також називають бінарним. Якщо кут між гілками дорівнює 90 градусів, дерево має назву дерево Піфагора.

Побудувати деревовидний фрактал можна рекурсивно. Кожний виклик малюватиме свою гілку на екрані, перевірятиме, чи необхідно продовжувати, і якщо так, виконуватиме два виклики для менших гілок. Недолік рекурсивних функцій – можливість переповнення доступної пам’яті – для такого алгоритму не виключено, але у пам’яті зберігається одночасно не більше викликів, ніж кількість ітерацій, тому для відносно простих задач (до 15-ти – 20-ти ітерацій) переповнення не відбудеться.

Жорстка фіксація всередині коду куту нахилу наступних гілок відносно попередніх призведе до створення однакових дерев за кожного запуску. Якщо для поодинокого дерева можна допустити таку роботу, то для лісу вона непридатна. Малювання лісових фонів за допомогою фрактальних дерев значно спростило б роботу перед подальшою обробкою в графічному редакторі. У справжньому лісі є схожі дерева, але немає абсолютно однакових. Отже, необхідно дати користувачу можливість редагувати кути нахилу вручну.

Також можна змінювати довжину початкової лінії, множник, який визначає скорочення лінії за кожної ітерації, початкову товщину в пікселях тощо.

Очевидною є необхідність ручного вибору кольору для фракталу. Кольором за замовчуванням встановлено чорний, кольором фону – білий.

Справжнє дерево поступово звужується від стовбура до листків. Відобразити це звуження можна за допомогою зміни товщини ліній, які будуть намальовані. Якщо припустити, що останні два кроки мають бути в один піксель товщиною, то, маючи початкову товщину, можна визначити коефіцієнт, з яким буде відбуватися звуження під час наступних ітерацій.

```
private void treeFractal(int iters, double length, double thickness, double x, double y, double angle){

    double radian = angle * Math.PI / 180;
    double newX = x + length * Math.sin(radian);
    double newY = y + length * Math.cos(radian);
    if (length < 1) length = 1;
    if (thickness < 1) thickness = 1;
    Line2D.Double line = new Line2D.Double(x, y, newX, newY);
    graph.setStroke(new BasicStroke((float)thickness));
    graph.draw(line);

    if (iters - 1 >= 0 && length >= 1 && thickness >= 1){
        treeFractal(iters - 1, length * treeFractLineShrink, thickness * treeFractLineThin, newX, newY, angle + treeFractAngleLeft);
        treeFractal(iters - 1, length * treeFractLineShrink, thickness * treeFractLineThin, newX, newY, angle + treeFractAngleRight);
    }
    DrawingPanel.paintComponents(graph);
    DrawingPanel.setVisible(true);
}
```

Рисунок 5. Код функції малювання фракталу-дерева

Для генерації фрактального лісу програма запускається декілька разів. Обираються різні кольори, різні кути нахилу обох гілок та початкової лінії.

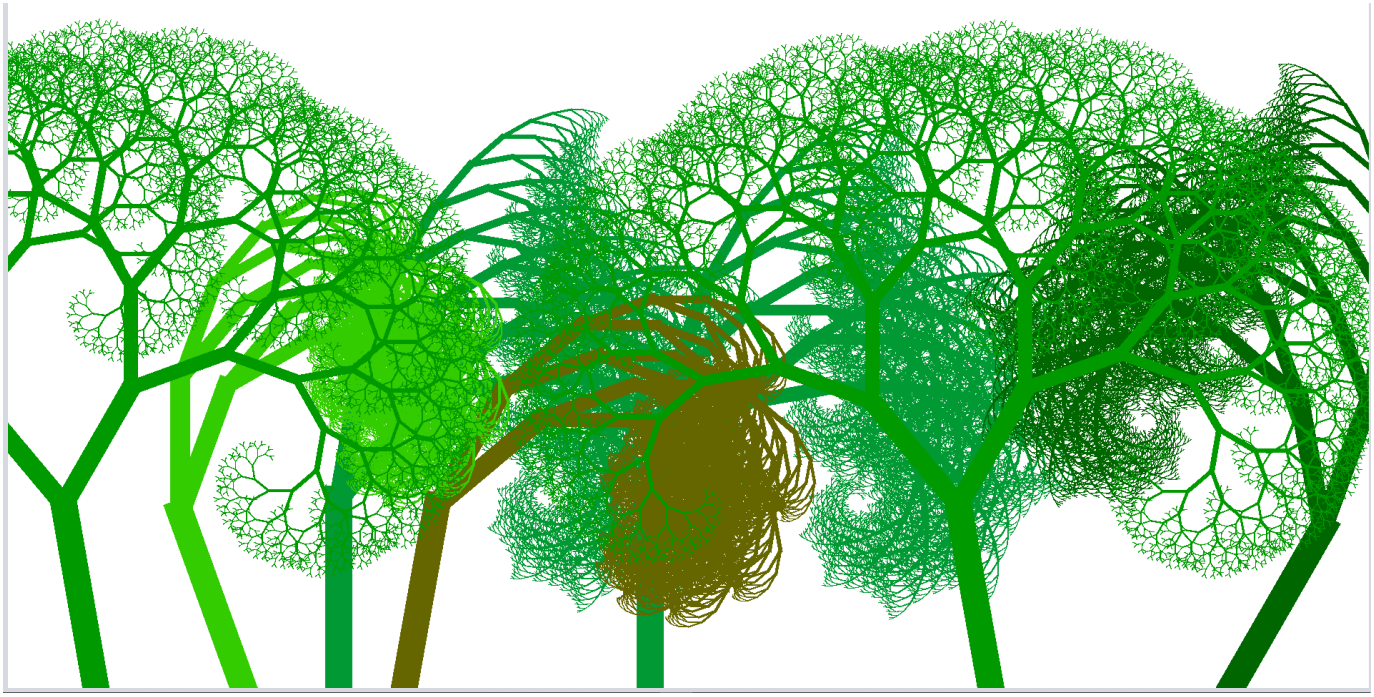


Рисунок 6. Ліс, складений із деревовидних фракталів, зменшено

Відразу помітний недолік – за певного кута нахилу гілок стають видимими «заломи» між ними та стовбуром.

Іншими відомими простими фракталами є зірки.

Зірковий фрактал починається з однієї точки – центру. Від неї під однаковими кутами один від одного відходять промені, інший кінець кожного з них вважається центром для наступного кроку, і процес повторюється.

Зірку зручно малювати рекурсивно. Для малювання потрібні координати центру, кількість та початкова довжина променів, а також множник для скорочення подальших. Всі ці параметри необхідно змінювати вручну та надати таку можливість користувачу.

Схема збереження викликів у пам'яті така ж, як у дерев. Але для зіркового фракталу швидше стають помітними затримки при малюванні – так, п'ять ітерацій відпрацьовують достатньо швидко, шість необхідно почекати, починаючи з семи час очікування стає відчутним (перевірка проводилася для зірки із сімома променями).

```
private void starFractal(int iters, double length, int lineNum, double lineShrink, double x, double y){
    double degree = 360 / lineNum;
    double curDegree = 360 / lineNum;
    ArrayList<XYPair> pairs = new ArrayList<>();
    for (int i = 0; i < lineNum; i++){
        double radian = curDegree * Math.PI / 180;
        Line2D.Double line = new Line2D.Double(x, y, x + length * Math.sin(radian), y + length * Math.cos(radian));
        pairs.add(new XYPair(x + length * Math.sin(radian), y + length * Math.cos(radian)));
        graph.draw(line);
        curDegree += degree;
    }

    if (iters - 1 > 0){
        for (int i = 0; i < pairs.size(); i++){
            starFractal(iters - 1, length / lineShrink, lineNum, lineShrink, pairs.get(i).getX(), pairs.get(i).getY());
        }
    }
    DrawingPanel.paintComponents(graph);
    DrawingPanel.setVisible(true);
}
```

Рисунок 7. Код функції малювання зіркового фракталу

Тут клас XYPair зроблено для збереження пар координат, що вказують на кінці щойно намальованих нових променів.

Змінюючи товщину лінії, якою малюється зірка, та накладаючи декілька таких фракталів один на одного, можна створити заготовку для орнаменту.

Приклади такого наведено далі. Під час малювання товщина лінії чорної зірки – десять пікселів, червоної – п’ять, синьої – чотири, білої та блакитної – один.

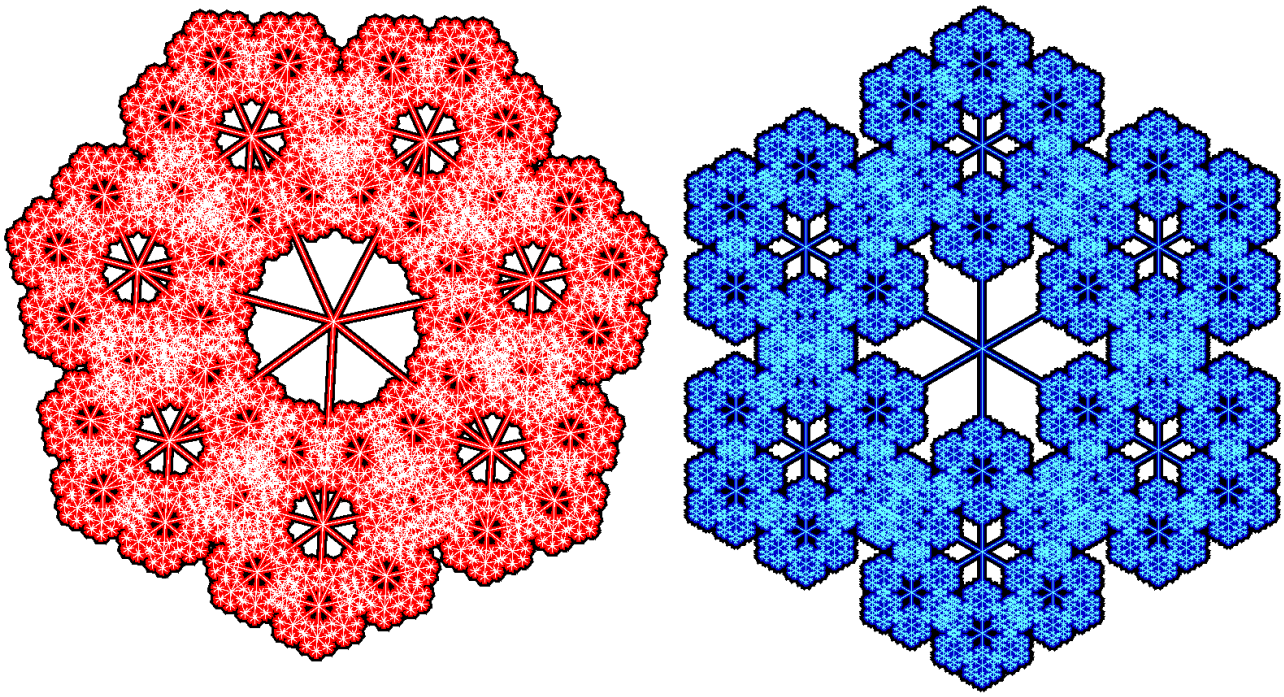


Рисунок 8. Приклади багатошарових зіркових фракталів, зменшено

Відразу помітно асиметрію зірки-результату з непарною кількістю променів. Справжні об’єкти нечасто бувають абсолютно симетричними, тому малювання саме таких зірок може стати корисним для розробника, що робить відносно реалістичну графіку. Зірки з парною кількістю променів можуть бути застосовані в якості сніжинок, які найчастіше мають правильну симетричну форму.

Інший спосіб, який можна застосувати для створення більш реалістичних дерев – це L-система, або система Ліденмайєра. Це вид формальної граматики, що складається із алфавіту символів, які описують майбутні дії. Класичний алфавіт системи такий: F (рух із малюванням), G або X («порожні» команди, іноді рух без малювання), + та - (повертання на певний кут проти чи за годинниковою стрілкою), [(збереження поточного положення),] (повернення до збереженого положення). На початку роботи задається початковий символ та набір правил (аксіом), за якими буде побудовано інструкцію для малювання.

Якщо скласти достатньо складні правила, результуюча система дозволить малювати реалістичні дерева. Приклад дерева, намальованого такою системою, наведено в розділі 2.2, поруч із тривимірним фракталом. L-система іноді вважається фракталом, ймовірно, через властивість про розмірності (розділ 2.2, визначення фракталу, пункт третій). Але самоповторення для неї не характерне.

Недолік L-системи – її інструкції стрімко розростаються зі збільшенням кількості ітерацій. Для деяких мов програмування (наприклад, для Java) ця проблема є критичною через обмеженість довжини стрічки, яка може зберігатися в пам’яті. Рішенням цієї проблеми буде збереження інструкції до файлу, адреса якого зберігається програмою, та подальше посимвольне читання з нього.

Нехай створення інструкції починається із символу F і програма містить єдину аксіому: F -> FF+[+F-F-F]-[-F+F+F]. Вибір кута, на який програма буде повертати, зроблено випадковим у межах від 1 до 45 градусів. Лінії скорочуються з певним коефіцієнтом. Виконання є послідовним. Програмі вказано п’ять ітерацій.

Рисунок 9. Порівняння L-системи (зменшено) та хвоща болотяного

Це дуже далеко від дерева, але зміна формули та підбір потрібної може дати інші малюнки та зробити результат більш схожим на деревовидну структуру.

Також достатньо популярними є фрактали, що складаються із кіл або квадратів. Ці фігури зручно використовувати для створення орнаментів, із квадратів також можуть складатися дерева (перше дерево Піфагора було побудоване саме так). Перевага таких фракталів – варіативність. Змінами коду можна створювати величезну кількість розташувань наступних фігур відносно попередніх.

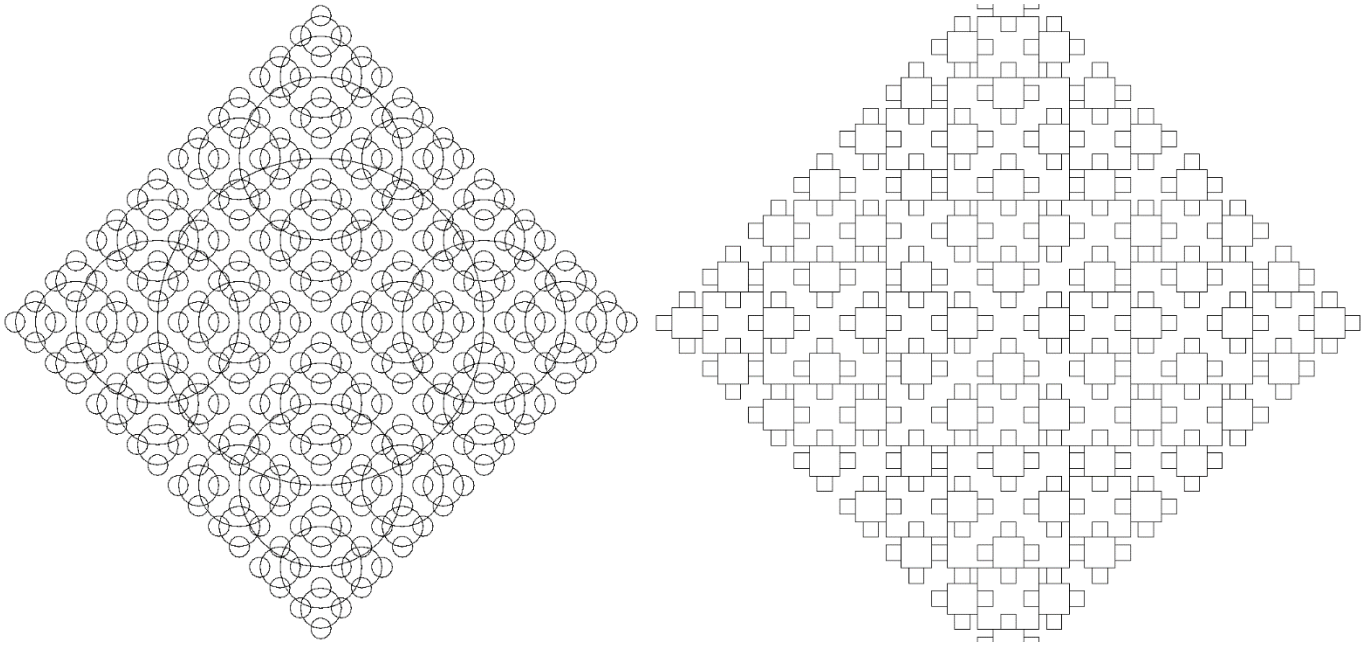


Рисунок 10. Фрактали з кіл та квадратів після п'яти ітерацій, зменшено

Зміни розташування елементів надто складні, щоб дозволяти виконувати їх користувачу. Але можна зробити заготовки найбільш розповсюджених варіантів та передати їх для використання.

Існує ще дуже велика кількість фракталів. Папороть Барнслі, перо, множина Кантора, трикутник Серпинського, криві Коха та Пеано, ламана дракона (фрактал Хартера-Хейтуея) тощо. Охват їх усіх однією програмою вимагатиме часу, але цілком можливий.

Код функції для L-системи, а також для функцій фракталів із кіл, квадратів та трикутнику Серпинського наведено в Додатку А.

3.2 Обрані методи процедурної генерації мапи та процес їх розробки

Як було сказано у розділі 2.6, для реалізації генератора мап рівня обрано два методи, які буде розглянуто окремо один від одного – клітинний автомат (Cellular Automation) та випадкове проходження (Random Walk).

Клітинний автомат – метод, що з'явився у 1970-х роках. Математик Джон Конвей випустив статтю, в якій описав гру The Game of Life. Це навіть не гра, а симуляція – прямокутний простір випадковим чином заповнювався клітинами («живими» та «мертвими»), до яких на кожному з подальших кроків застосовувався такий набір правил:

1) Якщо у «мертвої» клітини рівно три «живих» сусіди, вона оживає;
2) Якщо у «живої» клітини менше двох «живих» сусідів, вона помирає;
3) Якщо у «живої» клітини два або три «живих» сусіди, вона продовжує жити;
4) Якщо у «живої» клітини більше трьох живих сусідів, вона помирає.

Сусідом клітини вважається клітина, що має з нею або спільну сторону, або спільну вершину кута. Змінюючи вказані цифри, можна отримати зовсім різні результати та, відповідно, зображення або мапи.

Схема, яку буде застосовано для перших тестів – «народження» клітини відбувається за шести чи більше «живих» сусідів, «смерть» – за менше, ніж трьох (така система матиме умовну назву «схема 6 на 3»).

Проблема методу проявилася під час реалізації та тестування. Результати його роботи створюють зображення, подібні до печер, тільки на дуже малих розмірностях. Для більших зображень результат стає більше схожим на зразок камуфляжної тканини або стіну, прорізану тріщинками.

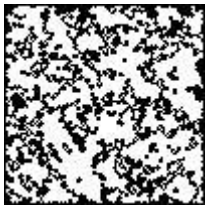


Рисунок 11. 100x100 пікселів для клітинного автомату, справжній розмір

Рівні для платформера можуть бути різних розмірів. Якщо прийняти розмір персонажа гравця за один піксель, рівень 100x100 (висота x ширина) об’єктів розміром з нього на перший погляд може здатися великим. Але насправді, якщо платформер не побудовано на розв’язку однієї-двох головоломок на рівень, розміри мають бути збільшені. Рівні платформера частіше прямокутні, а не квадратні, але на схему роботи це не впливає. Проблема виникає саме зі збільшенням розміру.

Рішення – виконувати початкову генерацію на розмірностях, для яких результат виглядає принадно, а по завершенню виконувати збільшення та згладжування. Схему генерації також було неодноразово змінено.

Після тестування принадним став розмір сторони у п’ятдесят пікселів. Для збереження пропорційності сторін зменшеного прямокутника та початкового до початку генерації проводиться ряд додаткових обчислень.

Зменшений прямокутник випадковим чином із заданим шансом заповнюється значеннями істини (порожнє місце) та фальші (стіна). Здійснюється задана кількість ітерацій за схемою: якщо у клітини менше трьох сусідів «істина», вона стає «фальшю», якщо п’ять чи більше – «істиною» («схема 5 на 3»). Виконується заповнення початкового прямокутника, для якого кожен піксель зменшеного перетворюється на квадрат певного розміру. Починається процедура згладжування.

Згладжування – процедура, що дозволяє закруглити кути великих квадратів та зробити рівень більш придатним для гри – перепади висоти, що набагато більші за персонажа, навряд чи сподобаються гравцю. Під час неї застосовується інша схема правил до великого прямокутника. Клітина стає «фальшю», маючи менше п’яти сусідів «істина», і стає «істиною», маючи п’ять чи більше («схема 5 на 5»). Після деякої кількості кроків згладжування готовий масив перетворюється на зображення («істина» – білий, «фальш» – чорний) та завантажується на комп’ютер до вказаної директорії, щоб користувач міг оцінити результат.

Під час вироблення схем було помічено різницю зображень залежно від того, яка кількість клітин була задана для «оживання» і яка – для «смерті». Якщо надати

користувачу можливість налаштовувати цю схему вручну, він отримає більшу кількість різних результатів.

Тестування виявило ще одну велику помилку, яку, однак, можна застосувати й на користь користувача. Виконуючи згладжування, програма має перевіряти всіх сусідів кожної клітини на «істинність» і змінювати значення клітини відповідно до отриманого результату. Під час цього процесу спочатку перевірявся та змінювався один і той самий масив, що призводило до змішування старих та нових даних. Такого не можна допускати, однак один із отриманих таким чином результатів наведено нижче.

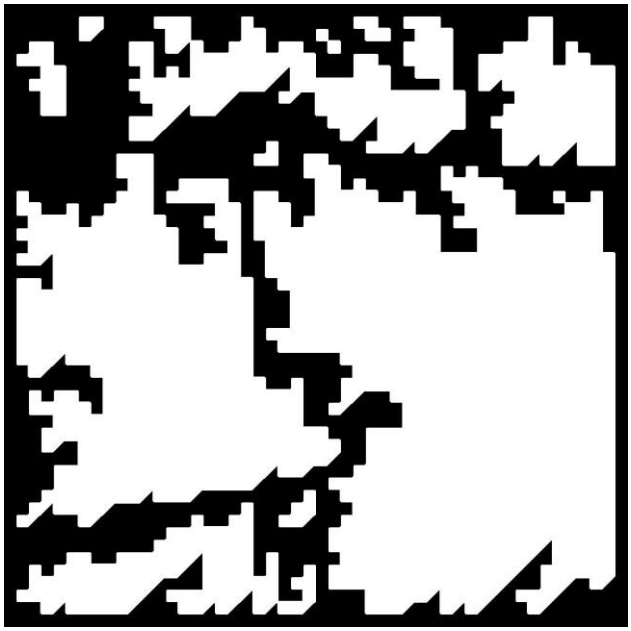


Рисунок 12. 1000x1000 пікселів, згладжування за "схемою 5 на 4", основна схема не зберіглася, зменшено

Хоча така мапа потребуватиме ще багатьох кроків обробки, але вона вже виглядає як основа для рівня, тільки для іншого стилю. Можливо, ця мапа могла б стати основою не для рівня-печери, а для рівня-зруйнованого міста або гірського з гострим камінням.

Найкращі результати показала «схема 5 на 3» для основи разом із «схемою 5 на 5» для згладжування. Зроблено також тимчасовий масив для уникнення змішування старих та нових даних під час згладжування (під час генерації основи таких проблем помічено не було).

Рівень розміром 1000x1000 об’єктів буде очевидно надто великим. Методом спроб та помилок було з’ясовано відносно принадний розмір – 300x600 (також можливі 200x400 та 100x200). Приклад генерації такого рівня наведено.

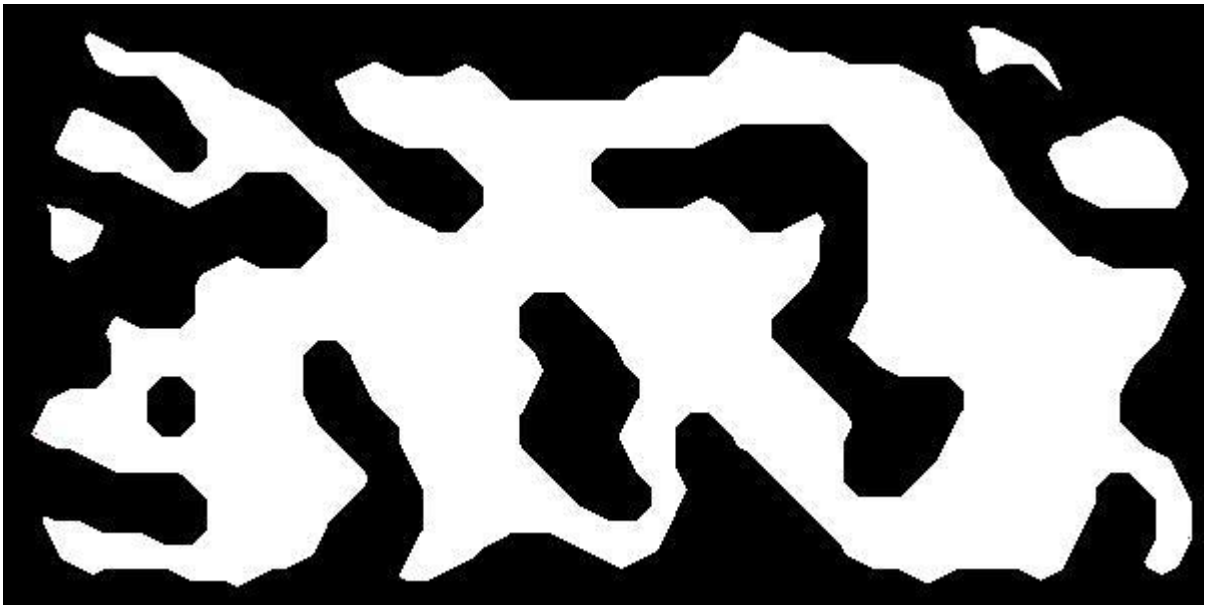


Рисунок 13. 300x600 пікселів, "схема 5 на 3" для основи, "схема 5 на 5" для згладжування, шанс появи білої клітини 45%

Якщо придивитися до цього зображення, можна помітити два окремих пікселі, що висять у повітрі. Це точки початку (алгоритм шукає якомога нижчу та ближчу до лівого краю точку, що з усіх боків має порожні точки, а знизу – підлогу. Пошук відбувається зліва направо (зовнішній цикл, відпрацьовує довше) та знизу догори (внутрішній, відпрацьовує швидше)) та кінця (відповідно якомога вищу та ближчу до правого краю. Зовнішній цикл – справа наліво, внутрішній – згори вниз).

Недолік цього алгоритму, зазначені у розділі 2.4, очевидний – необхідна розробка додаткового функціоналу для з’єднання недосяжних порожніх областей з іншими, що потребуватиме часу як для розробки, так і для роботи. Код розміщено в Додатку Б.

Перейдемо до розгляду наступного алгоритму.

Випадкове проходження – один із найпростіших для розуміння та реалізації алгоритмів. Початкова точка обирається або в центрі майбутньої мапи, або у випадковому її місці. Для подальшого розгляду обрано другий варіант.

На відміну від клітинного автомату, якість результату роботи методу випадкового проходження не залежить від розмірів мапи настільки безпосередньо. Результат даного методу залежить відразу від декількох факторів:

1) Співвідношення розміру мапи та кількості ітерацій. Наприклад, для масиву розміром 50x100 пікселів доречно виконати менше кроків, ніж для масиву 300x600;
2) Розміру кроку. Якщо одна ітерація охоплює квадрат розміром 5x5 пікселів, доведеться виконати значно меншу їх кількість, ніж для 1x1;
3) Удачі. Випадковість тут має значно сильніший вплив, ніж у попередньому методі. Можливо, доведеться запустити алгоритм більшу кількість разів із різними параметрами для отримання необхідного виходу.

Схему роботи методу повністю описано в розділі 2.4, тому далі буде розглянуто процес розробки генератора на його основі.

Перше, що необхідно зробити – обмежити територію, куди може потрапити наступна точка. Для клітини розміром 1x1 піксель достатньо меж, заданих довжиною та висотою двовимірного масиву, який зберігає майбутню мапу. Але для кроків більших розмірів, початок (верхній лівий кут) яких може потрапити будь-куди, необхідно означити додаткове обмеження – якщо подвоєний розмір клітини виходить за межі мапи, клітина не змінюється, і обирається інший напрямок руху.

Головний недолік методу випадкового проходження – час його роботи. Наприклад, для мапи розміром 1000x1000 пікселів та розміру клітини в один піксель для отримання прийнятного результату необхідно задати декілька десятків тисяч ітерацій, що значно сповільнить роботу програми. Користувач може чекати, але в довгостроковій перспективі такі затримки роздратують його і змусять шукати іншу програму, або й створювати власну. Тому необхідно дозволити користувачу обирати розмірність мапи, розмір кроку та їх кількість вручну.

На відміну від клітинного автомату, випадкове проходження не вимагає зменшення розмірності мапи для покращення результату роботи. Але мапа, зроблена

відмінними від 1x1 піксель кроками, потребує згладжування. Для процедури можна застосувати схему з попереднього методу – клітинний автомат зі «схемою 5 на 5».

Далі наведено приклади двох різних згенерованих мап.

Рисунок 14. 300x600 пікселів, крок 5x5, 10000 ітерацій, без згладжування, зменшено

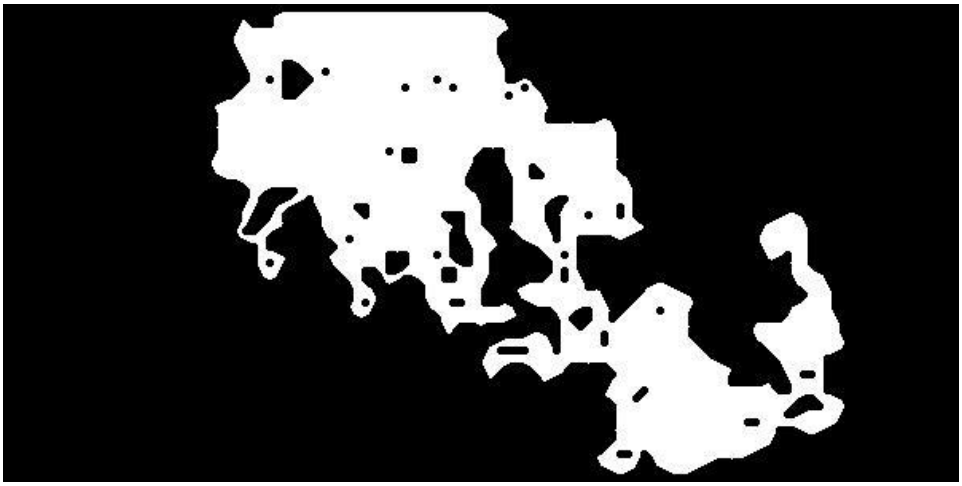


Рисунок 15. 300x600 пікселів, крок 5x5, 10000 ітерацій, 10 ітерацій згладжування, зменшено

Наведене в розділі 2.4 твердження про платформи для цього методу не настільки актуальне, як для попереднього – частина менших чорних областей залишаються навіть по завершенню процедури згладжування. Але розробник, що має досвід в іграх жанру платформер, помітить проблему – навіть якщо персонаж стрибає далеко, платформ надто мало, щоб показати гравцеві всю мапу. Спосіб додавання нових буде розглянуто в наступному розділі. За основу взято генератор із методом випадкового проходження. Його код розміщено в Додатку В.

3.3 Метод додавання платформ та його розробка

Найпростіший спосіб розміщення платформ на рівні – віддати цю можливість до рук користувача. Робота вручну дозволить, наприклад, одразу перевіряти, чи вийде пройти рівень та наскільки швидко. Останнє необхідне, якщо розробник бажає показати гравцеві всю мапу рівня – тоді необхідно робити найкоротший маршрут від початку рівня до його кінця якомога довшим. Але, як вже було згадано, ручна робота займає набагато більше часу, ніж програмна.

Можна розставляти платформи випадковим чином за допомогою генератора. Але ця схема матиме очевидні недоліки – подібний механізм розставлятиме платформи, не перевіряючи, чи можна їх поставити в обраному ним місці, але навіть якщо зробити додаткову перевірку, рівень, створений таким чином, з великою ймовірністю виявиться непрохідним.

Засобом вирішення відразу усіх цих проблем, крім часової, може стати генетичний алгоритм.

Генетичний алгоритм (Genetic Algorithm) застосовується для задач оптимізації та моделювання, виконує випадковий вибір, комбінування та варіацію заданих параметрів за допомогою механізмів, що існують у дикій природі – спадкування, мутації, відбір та розмноження (кросинговер). Алгоритм є евристичним, а саме такі використовують для вибору кращого рішення із отриманих, якщо точне не може бути знайдене. Задача оптимального розташування платформ на мапі відноситься до таких, тому застосування генетичного алгоритму є доречним.

Для створення та запуску подібного механізму необхідно спланувати та реалізувати декілька елементів та їх роботу:

1) Ген (Gene). Найменша складова, комбінації яких будуть змінюватися в подальшому. Для поставленої задачі роль гену виконує платформа;
2) Хромосома (Chromosome), або Індивід (Individual). Набір генів, що бере участь в усіх операціях та є частиною популяції. Для поставленої задачі цю роль виконує набір платформ, розташованих у різних місцях мапи;

3) Для індивіда необхідно означити функцію придатності (Fitness Function). За її допомогою виконується обчислення, наскільки відносно гарним є даний розв'язок задачі. За цими значеннями має виконуватися сортування індивідів всередині популяції. Для поставленої задачі функція має визначати довжину найкоротшого шляху, яким можна пройти від початку до кінця рівня, і повертати її або -1, якщо такий шлях знайти неможливо;
4) Популяція (Population) – набір індивідів, відсортований за придатністю. Тут також необхідні додаткові параметри, наприклад, кількість найкращих результатів, які будуть зберігатися. Цю роль найкраще виконає список;
5) Відбір (Selection) – функція, що вибирає індивідів, які складатимуть популяцію та будуть розмножуватися. Тут застосовна кількість найкращих, які гарантовано потраплять до списку;
6) Схрещування, або Кросинговер (Crossing over) – функція, що виконує створення нових індивідів, використовуючи дані двох обраних «батьків». Для поставленої задачі один із «батьків» віддає «дитині» свої платформи, розташовані до середини мапи (розділ є вертикальним), а другий – після середини. «Дитина» також перевіряється на придатність;
7) Мутація (Mutation) – функція, яка для поставленої задачі, з невеликим шансом, вилучає випадкову платформу із індивіда, якого опрацьовує, та у випадковому порожньому місці ставить нову платформу. «Мутант» перевіряється на придатність і, якщо немає протирічь, додається до популяції.

Після кожного циклу відбір-схрещування-мутація відбувається сортування індивідів за допомогою функції придатності від найкращого до найгіршого. Якщо даний крок був останнім, обирається перший із отриманого списку індивід (тобто індивід з найкращим результатом), і його платформи наносяться на основну мапу.

Перевага такого методу роботи – відносна схожість результату на роботу людини. Навіть якщо виконано одну ітерацію, можна отримати рівень, який

можливо пройти і на який не витрачено зусиль розробника. Для більшої їх кількості результат буде тільки покращуватися.

Основний недолік програми – складність. Детальніше її буде розглянуто в наступному розділі. Для рівня великого розміру час навіть першої генерації може стати значним, або й неприйнятним через велику кількість необхідних перевірок.

Особливо довгою за часом виконання функцією стане пошук найкоротшого шляху. Обраний для неї алгоритм – A*, алгоритм пошуку шляху на графі, що дає одні з найкращих результатів порівняно з іншими. Якщо перетворити кожну з точок, розташованих відразу над платформами, на вершину графа і з'єднати їх (якщо гравець може їх досягти) ребрами, алгоритм зможе знайти найкоротший шлях між ними та підрахувати відстань, яку доведеться пройти – або повернути від'ємне значення, якщо шляху немає. Проблема цієї функції – розмірність отриманого графа. A* – відносно стислий за часом алгоритм, але що більший граф, то довше він працюватиме, особливо якщо шлях знайти не вдасться. Тому для рівнів розміром 200x400 об'єктів він працюватиме швидше, ніж для 300x600.

Під час тестування роботи генетичного алгоритму прийнятного часу вдалося досягти тільки для рівня розміром 10x30 об'єктів та з елементарною структурою (єдиною порожньою областю). Для рівнів більшого розміру час роботи швидко та значно зростає. Причину цього розглянуто в наступному розділі.

Код розміщено за посиланням [1].

3.4 Аналіз результатів та оцінка складності програм

Складність трьох розроблених програмних продуктів буде оцінено окремо.

Часова складність системи для створення фрактальної графіки суттєво залежить від обраного для відображення фракталу та від кількості ітерацій, що йому задані. Нехай кількість ітерацій дорівнює К, власне малювання – атомарна операція (O(1)), всі підрахунки також вважатимуться атомарними.

В обраного фракталу із **кіл** під час кожного виклику малюється одне коло та, якщо необхідно продовжувати, виконуються послідовно чотири рекурсивні виклики. Для нього часову складність можна оцінити як $\mathbf{O\ (1 + 4^{K-1})}$.

Аналогічною є часова складність для обраного фракталу із **квадратів**.

Для **трикутника Серпинського** складність можна оцінити як $\mathbf{O\ (1 + 3^{K-1})}$, якщо виконувати відображення тільки його контурів. Якщо обрано зафарбування, необхідно враховувати процес створення полігонів для даного процесу, але на складність це впливає мінімально.

Деревовидний фрактал має складність $\mathbf{O\ (1 + 2^{K-1})}$.

Для **зіркового** фракталу необхідно ввести параметр, що відповідає кількості променів зірки. Нехай він має назву М. Тоді за кодом часову складність зірки можна оцінити як $\mathbf{O\ (M + M^{K-1})}$. За цією складністю в порівнянні з попередніми очевидна причина згадки про помітні затримки у розділі 3.1.

Складність роботи функції, що створює інструкцію для L-системи, оцінити важче, оскільки вона залежить не тільки від кількості ітерацій, а також від кількості та складності правил побудови. Її можна оцінити, вона буде наближеною до $O\ (B_1 + B_2 + \ldots + B_K)$, де кожне B_i – кількість символів у вхідному файлі на момент обробки (і в межах від 1 до К). Якщо прийняти кількість символів у файлі-результаті за Р, то часова складність роботи функції відображення буде оцінена як O (P). Сумарна оцінка часової складності роботи функцій **L-системи** – $\mathbf{O\ (B_1 + B_2 + \ldots + B_K + P)}$.

Роботу генератора на основі клітинного автомату необхідно оцінювати по частинах (код знаходиться в Додатку Б).

Нехай Н – висота мапи, М – ширина мапи, Нс та Мс – відповідно висота та ширина зменшеної мапи, З – кількість кроків згладжування. Заповнення випадковими числами малої мапи відбувається за О (Нс * Мс). Виконання алгоритму клітинного автомату – за О (І * (Нс – 2) * (Мс – 2)), де І – кількість кроків (в програмі І незмінно дорівнює двадцяти). Перенесення схеми малої мапи на повну – за О ((Нс – 2) * (Мс – 2) * $С^2$), де С – коефіцієнт збільшення, що відображає співвідношення сторін зменшеної та повної мап. Найважча операція – згладжування: О (З * ((Н – 2) * (М – 2) + Н * М)). Копіювання масиву до зображення – О (Н * М). Пошук початкової та кінцевої точок – додаткові О ((М – 2) * (Н – 5)).

Складність роботи генератора оцінюється за найважчою операцією зі спрощенням всіх схожих операцій – **О (З * Н * М).**

Роботу генератора на основі випадкового проходження буде розглянуто окремо від генетичного алгоритму (код знаходиться в Додатку В та за посиланням [1] відповідно).

Нехай Н – висота мапи, М – ширина, К – розмір кроку (сторона квадрату, менше за М та Н), Л – кількість кроків, З – кількість ітерацій згладжування. Початкове заповнення мапи – О (Н * М). Вибір точки старту – О ($К^2$). Виконання проходження – О (Л * $К^2$) (цикл вибору напрямку не враховується через неможливість передбачити кількість його спрацювань до переривання). Операції згладжування та пошуку початку та кінця співпадають з методами попереднього алгоритму, тому і їх оцінки будуть співпадати.

Складність роботи генератора, за найважчою операцією – **О (З * Н * М)** (для не дуже великого розміру кроку, інакше час проходження буде більшим за виконання згладжування).

Генетичний алгоритм також вимагає поетапного оцінювання.

Нехай Н – висота мапи, М – ширина, К – кількість вертикальних смуг, на які буде розбито мапу ($К_1$ – ширина однієї), Л – кількість секторів, на які буде розбито

кожну смугу ($Л_1$ – висота одного), В – кількість ітерацій, Ж – кількість індивідів у поколінні, $П_I$ – кількість платформ в індивіда I.

Етап підготовки – приблизно O (К * Л * $К_1$ * $Л_1$) (складність роботи циклу while оцінити складно, тому його випущено). Підсумовуючи – O (Н * М).

Етап створення першої популяції – O (Ж * (К * Л + Н * М + $Н^2$ * М ^{2})) (роботу менших циклів всередині можна випустити). Підсумовуючи – O (Ж * $Н^2$ * М ^{2}).

Етап відбору – O ($Ж^2$). Етап кросинговеру – O (Ж * $П_1$ * $П_2$). Етап мутації – O ($П_I$) (роботу двох циклів while оцінити складно через їх залежність від генераторів псевдовипадкових чисел, тому їх випущено).

За цими оцінками можна визначити загальну оцінку одного повного циклу відбір-кросинговер-мутація (частини етапу симуляції). Вона буде рівною приблизно O (Ж * $Н^2$ * $М^2$). Оцінка всього етапу симуляції – O (В * Ж * $Н^2$ * $М^2$).

Оцінка розташування платформ в початковому масиві порівняно невелика – O ($П_I$ * перша розмірність платформи * друга розмірність платформи).

За найбільшою оцінкою, складність роботи генетичного алгоритму – **O (В * Ж * $Н^2$ * $М^2$).**

Через таку високу складність стає зрозуміло, що час виконання алгоритму стрімко зростатиме разом зі зміною будь-якого з його параметрів, особливо розмірностей мапи. Припускається, що ця оцінка може бути вищою, якщо структура згенерованої мапи є складною, та, можливо, досягати O (В * Ж * $Н^3$ * $М^3$) або ще більших степенів.

Алгоритм такої складності вимагає значної оптимізації та ретельної переробки коду, оскільки створення рівня великого розміру за допомогою цього алгоритму може стати довшим, ніж виконання аналогічної роботи вручну.

3.5 Оцінка якості мап рівнів з точки зору гравця

Якість рівня, як було згадано в розділі 2.5, для гравця визначається багатьма факторами, але основні з них – помірна складність та надання можливості випробувати свої вміння.

Оскільки автор дипломної роботи також є досвідченим гравцем у жанрі платформер, вона може оцінити отримані результати та висловити свою думку.

Генератор на основі клітинного автомату вимагає значної доробки навіть як основа. Можливо, його покращить зменшення ймовірності появи білої точки за першої генерації або зміна схеми перетворення основи. Але перспектива у методу є, і за певних покращень та достатньої удачі даний програмний продукт зможе створювати пристойні мапи.

Генератор на основі випадкового проходження як основа показує себе значно краще. Малі статичні місця, які він залишає і які в подальшому стають платформами, додають створеним за його допомогою рівням деталізації та зменшують об’єм роботи для генетичного алгоритму. Його мапи навіть без доробки генетичним алгоритмом значно більше схожі на рівні.

Через проблеми з часом роботи генетичного алгоритму мапи, створені з його допомогою, не можуть бути оцінені. Серед досвідчених гравців було проведене опитування, спрямоване на отримання думок про якість мап, зроблених генератором із випадковим проходженням та генератором із клітинним автоматом. Ці мапи розміщено в Додатку Г. За їх думками можна підбити такий підсумок.

Результати генератора на основі випадкового проходження приваблюють деталізацією, відсутністю недосяжних областей, більш складним виглядом і застосовністю для великих мап. Не подобаються гравцям отримання надто вузьких та покручених областей, а також деяка хаотичність.

Результати генератора на основі клітинного автомату приваблюють більшою структурністю, простотою, зрозумілістю дизайну, відсутністю хаотичності та застосовністю для менших мап. Не подобаються гравцям недосяжні області.

3.6 Перспективи та можливі покращення

В області графіки основна перспектива – розширення списку фракталів для використання користувачем. Їх частину було згадано в розділі 3.1. Для структур із кіл чи квадратів, що мають значну кількість варіантів, можливий відбір найбільш розповсюджених та їх введення до програми.

Створення інтерфейсу налаштувань для кожного з фракталів, що є у списку, також є важливим. Наприклад, для L-системи інтерфейс має містити налаштування стартової формули, списку правил, початкової довжини й товщини ліній, коефіцієнту скорочення та кількість ітерацій, що матимуть товщину в один піксель.

Додаткове необхідне покращення – додавання можливості завантажити отримане зображення на комп’ютер.

Для деревовидних фракталів можна запропонувати відразу декілька покращень. По-перше, під час їх малювання помітні заломи між наступними та попередніми гілками. Рішенням буде використання згладжування або більша кількість перевірок для уникнення цього ефекту.

По-друге, для великої кількості ітерацій і, відповідно, менших гілок доречним є зміна кольору деяких із них на колір фону для створення асиметрії, характерної для справжніх дерев. Робити перефарбування краще з певною ймовірністю, яка буде зростати під час роботи (наприклад, для гілки відразу біля стовбура вона буде меншою, ніж для більш віддалених). Зміна має відбуватися один раз для кожної гілки – якщо вона спрацювала, гілки, що відходять від неї, також не будуть відображені. Можливо, для створення багатьох дерев цю ймовірність варто зробити незмінною, а для єдиного лишити зростаючою.

По-третє, випадковим чином (варіант – також зі зростаючою ймовірністю залежно від гілок) розміщувати на вже відображеному дереві листя. Це створить шум і зробить менш помітними заломи між гілками, через що зображення наблизиться до справжнього дерева.

Для усіх фракталів покращенням буде можливість обрати декілька кольорів для розфарбування та створити градієнт.

В області генераторів мапи можна запропонувати покращення, деякі з яких були згадані в попередніх розділах.

Для генератора на основі клітинного автомату очевидною перспективою є додавання функціоналу для з'єднання недосяжних областей одна з одною та з більшим простором. Цей функціонал можна зробити опціональним, тоді також додати можливість розміщувати у недосяжних областях демонстрацію ворогів чи зразків механік, які чекають гравця на наступних рівнях. Таку роботу можна виконувати як вручну, передавши керування користувачу, так і з застосуванням алгоритмів. Додавання цієї процедури до генератору на основі випадкового проходження є не настільки необхідним, але також не зайвим.

Для генетичного алгоритму необхідністю є оптимізація й прискорення роботи.

Для генератора разом із генетичним алгоритмом значною перспективою є додавання функціоналу для розташування ворогів. Але не всіх – ворогів з більш складною поведінкою та траєкторією руху краще розміщувати вручну, контроль над таким функціоналом необхідно передати користувачу. Вороги, які вміють тільки ходити, лазити чи літати за простими маршрутами, можуть бути розставлені за допомогою алгоритму.

Також важлива перспектива – завантаження отриманого результату не до зображення, а до, наприклад, текстового файлу у вигляді стрічки елементів, який в подальшому можна буде зчитати за допомогою, наприклад, скриптів більш потужного рушія, на якому розробляється гра.

Значним покращенням буде відділення генератора від генетичного алгоритму та передача функцій розташування точок початку та кінця користувачу. Це дасть йому можливість, за бажанням, змінювати їх положення та отримувати різні мапи платформ за кожного запуску генетичного алгоритму без зміни основи мапи. У розробленій програмі цей функціонал нерозривно пов'язаний.

3.7 Висновок із розділу 3

Виконавши поставлені завдання та оцінивши результати їх роботи, можна зробити такі висновки.

Для створення основи майбутнього рівня добре застосовними є алгоритми процедурної генерації. Із двох обраних, реалізованих та протестованих алгоритмів кращі для гри жанру платформер результати показав алгоритм випадкового проходження, оскільки він залишає більше малих областей, які в подальшому можуть слугувати в якості платформ, а також майже не залишає недосяжних порожніх областей. Обидва генератори потребують додаткового функціоналу, але його відсутність не є для них критичною – показані ними результати вже можуть бути застосовані в якості бази для мапи.

Розробка генетичного алгоритму для розміщення додаткових платформ виявила як переваги, так і недоліки його використання. Перевагою є перспектива автоматичного створення майже повноцінного рівня, що вимагає мінімальних редагувань в подальшому (якщо не враховувати необхідність додавання ворогів). Недоліком – час його роботи. Якщо брати до уваги складність обраної задачі та аналіз, виконаний в розділі 3.4, цей недолік є очевидним, однак певна переробка, значна оптимізація та прискорення роботи коду може допомогти його пом’якшити.

Розроблена програма для малювання фрактальної графіки вимагає великої кількості доробок, розглянутих у розділі 3.6. Але вже виконана її частина схожа на початку кроку Пре-Альфа (визначення знаходиться в розділі 1.3) проекту – тобто містить частину функціоналу в спрощеному вигляді і дає загальне уявлення про вигляд та роботу майбутнього продукту.

ВИСНОВОК

Генерація ігрових об’єктів засобами програмного забезпечення не є новою розробкою, але залишається актуальною. Особливо корисною вона може стати для розробників-одинаків або невеликих команд, що працюють заради втілення ідеї в реальність і не мають ресурсів, якими володіють великі ігрові студії.

Було проведено дослідження процесу розробки ігор від ідеї до запуску та подальшого супроводу, вивчення кожного етапу, особливостей різних жанрів та деталей, що приваблюють гравців.

Найважчими частинами етапу розробки гри вважаються створення графіки, а саме зовнішнього вигляду об’єктів, а також процес програмування. Для ігор жанру платформер, обраного для детального розгляду, у деяких потужних ігрових рушіях вже є заготовлений код, що відповідає за ігрову фізику, особливості руху персонажів та поведінку платформ тощо. Під час роботи з такими програмами задача програмування відступає на другий план, а на перший висувається інша – створення мапи рівня, якою гравець буде рухатися.

Для спрощення процесу малювання графіки було обрано застосування фракталів. Деякі природні об’єкти реального світу мають структуру, схожу на фрактальну, тому за їх допомогою можна створювати зображення, які в подальшому можуть бути використані в якості фонів для рівнів або текстур окремих предметів. Було створено програму, що дозволяє малювати фрактали декількох видів, змінювати їх колір, поєднувати або заміщувати. Такий продукт дає користувачу можливість ознайомитися з виглядом простої фрактальної графіки, прийняти рішення про її використання. За подальших серйозних покращень програма може стати засобом створення зображень для рівнів, що не потребуватимуть додаткової обробки. Можливостей не спеціалізованих середовищ розробки вистачає для створення подібних програмних продуктів, що було доведено практикою.

Задача автоматичного створення мапи рівня (без ворогів) помітно складніша, оскільки результатом роботи має бути не приблизне уявлення про вигляд майбутнього рівня, а конкретне рішення, яке користувачу залишиться лише затвердити та перенести до рушія. Після вивчення засобів процедурної генерації мап було обрано два методи для створення основи – клітинний автомат та випадкове проходження. Для кожного з них розроблено окремий генератор.

По завершенню тестування та проведення опитування серед гравців зроблено висновок – для ігор жанру платформер обидва генератори показали добрі результати, але ці результати не є фінальними. Для їх покращення вивчено, розроблено та протестовано генетичний алгоритм, що вводить до мапи додаткові платформи для забезпечення можливості пройти рівень та демонстрації якомога більшої частини мапи гравцю. Основний недолік, що не дозволяє повноцінне використання генетичного алгоритму для роботи – час його виконання.

Отримані програми дозволяють користувачу ознайомитися з простою фрактальною графікою й отримати мапи рівнів для гри у вигляді зображень. Розробникові-одинаку навіть такі програмні продукти можуть дуже спростити роботу та звільнити час для інших частин розробки.

Вивчення створення фракталів може стати складним в області відображення на екрані, але в області коду проблеми виникнуть з меншою ймовірністю через наявність в мережі Інтернет великої кількості ресурсів для допомоги. Генератори без додаткового функціоналу достатньо легкі для розуміння та сприйняття з тих же причин. Складним для вивчення може стати генетичний алгоритм, оскільки прикладів коду для них значно менше, ніж для згаданих вище задач, і найбільш імовірно, що кожний його етап необхідно досліджувати та створювати окремо.

Переваги та недоліки методів процедурної генерації було розглянуто в розділах 2.4 та 3.2. Аналіз якості з точки зору потенційних гравців – у розділі 3.5.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. GeneticAlgo-RandomWalkGenerator [Електронний ресурс] / MarinaNikolaieva / GitHub – 31 травня 2022 р. – Режим доступу до джерела: https://github.com/MarinaNikolaieva/GeneticAlgo-RandomWalkGenerator
2. Procedural Generation [Електронний ресурс] / Wikipedia, останнє редагування – 18 травня 2022 р. – Режим доступу до джерела: https://en.wikipedia.org/wiki/Procedural_generation
3. Generate Random Cave Levels Using Cellular Automata [Електронний ресурс] / Michael Cook / EnvatoTuts+ – 23 липня 2013 р. – Режим доступу до джерела: https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664
4. Cave-like Level Generation Using Cellular Automata [Електронний ресурс] / Martin Celusniak / LinkedIn – 19 березня 2019 р. – Режим доступу до джерела: https://www.linkedin.com/pulse/cave-like-level-generation-using-cellular-automata-martin-celusniak
5. The Cellular Automaton Method for Cave Generation [Електронний ресурс] / j2kun / Jeremykun – 29 липня 2012 р. – Режим доступу до джерела: https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/
6. Introduction to Genetic Algorithms – Including Example Code [Електронний ресурс] / Vijini Mallawaarachchi / TowardsDataScience – 8 липня 2017 р. – Режим доступу до джерела: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
7. The Nature of Code – Chapter 8. Fractals [Електронний ресурс] / Daniel Shiffman / NatureOfCode – 2012 р. – Режим доступу до джерела: https://natureofcode.com/book/chapter-8-fractals/

8. Fractal Tree with Lines/Polygons written in Java [Електронний ресурс] / Marijan Nikic / CodeProject – 3 червня 2020 р. – Режим доступу до джерела: https://www.codeproject.com/Articles/5269971/Fractal-Tree-with-Lines-Polygons-Written-in-Java
9. List of fractals by Hausdorff dimension [Електронний ресурс] / Wikipedia, останнє редагування – 5 квітня 2022 р. – Режим доступу до джерела: https://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension
10. How can I adapt A* pathfinding to work with platformers? [Електронний ресурс] / toinfinityandbeyond, Liam Lime / GameDev StackExchange – 26 березня 2016 р. – Режим доступу до джерела: https://gamedev.stackexchange.com/questions/118912/how-can-i-adapt-a-pathfinding-to-work-with-platformers
11. Amit's A* Pages [Електронний ресурс] / Amit Patel / Red Blob Games, Stanford University, останнє редагування – 21 квітня 2022 р. – Режим доступу до джерела: http://theory.stanford.edu/~amitp/GameProgramming/
12. History of video games [Електронний ресурс] / Wikipedia, останнє редагування – 24 травня 2022 р. – Режим доступу до джерела: https://en.wikipedia.org/wiki/History_of_video_games
13. A Brief History of Computer Games [Електронний ресурс] / Mark Overmars / Stichtingspel – 30 січня 2012 р. – Режим доступу до джерела: https://www.stichtingspel.org/sites/default/files/history_of_games.pdf
14. Відеоігри в Україні [Електронний ресурс] / Wikipedia, останнє редагування – 16 травня 2022 р. – Режим доступу до джерела: https://uk.wikipedia.org/wiki/%D0%92%D1%96%D0%B4%D0%B5%D0%BE%D1%96%D0%B3%D1%80%D0%B8_%D0%B2_%D0%A3%D0%BA%D1%80%D0%B0%D1%97%D0%BD%D1%96

15. List of video game genres [Електронний ресурс] / Wikipedia, останнє редагування – 17 травня 2022 р. – Режим доступу до джерела: https://en.wikipedia.org/wiki/List_of_video_game_genres
16. The 7 Stages of Game Development [Електронний ресурс] / Devin Pickell / G2 – 15 жовтня 2019 р. – Режим доступу до джерела: https://www.g2.com/articles/stages-of-game-development
17. 6 Key Stages of Game Development: From Concept to Standing Ovation [Електронний ресурс] / Victoria Mozolevskaya / Kevuru Games – 8 лютого 2021 р. – Режим доступу до джерела: https://kevurugames.com/blog/6-key-stages-of-game-development-from-concept-to-standing-ovation/
18. 15 Reasons People Play Video Games [Електронний ресурс] / Jeremy Edge / GameQuitters – 7 жовтня 2020 р. – Режим доступу до джерела: https://gamequitters.com/15-reasons-people-play-video-games/
19. Архаїчна геометрія природи [Електронний ресурс] / yaneva / Львівський національний університет імені Івана Франка, факультет механіко-математичний – 16 серпня 2013 р. – Режим доступу до джерела: http://www.mmf.lnu.edu.ua/ar/426
20. Notes on Procedural Map Generation Techniques [Електронний ресурс] / Christian Mills, Herbert Wolverson / Christianjmills – 9 грудня 2021 р. – Режим доступу до джерела: https://christianjmills.com/Notes-on-Procedural-Map-Generation-Techniques/
21. Плоскость восприятия: плюсы и минусы создания 2D-игр [Електронний ресурс] / Андрей Верещагин / DTF – 2 грудня 2018 р. – Режим доступу до джерела: https://dtf.ru/gamedev/32593-ploskost-vospriyatiya-plyusy-i-minusy-sozdaniya-2d-igr
22. Drawing Simple Generative Organics using L-systems [Електронний ресурс] / Vexlio, LLC – 2018 р. – Режим доступу до джерела: https://www.vexlio.com/blog/drawing-simple-organics-with-l-systems/

ДОДАТОК А

```
335     private void triangleFractal(int iters, double x, double y, double length, double thick){
336         if (iters == 1 && shapeFractFill)
337             drawTriangle(x, y, length, true);
338         else if (iters == 1)
339             drawTriangle(x, y, length, false);
340         else{
341             drawTriangle(x, y, length, false);
342             if (thick < 1)
343                 thick = 1.0;
344             triangleFractal(iters - 1, x, y, length / 2, thick * shapeFractLineThin);
345             triangleFractal(iters - 1, x + length / 4, y - Math.sin(Math.PI / 3) * length / 2, length / 2, thick * shapeFractLineThin);
346             triangleFractal(iters - 1, x + length / 2, y, length / 2, thick * shapeFractLineThin);
347         }
348     }
```

Рисунок 16. Код функції для малювання трикутника Серпинського

```
350     private void drawTriangle(double x, double y, double len, boolean fill){
351         if (!fill){
352             graph.draw(new Line2D.Double(x, y, x + len, y));
353             graph.draw(new Line2D.Double(x, y, x + len / 2.0, y - Math.sin(Math.PI / 3.0) * len));
354             graph.draw(new Line2D.Double(x + len / 2.0, y - Math.sin(Math.PI / 3.0) * len, x + len, y));
355         }
356         if (fill && shapeFractFill){
357             double xs[] = {x, x + len, x + len / 2.0};
358             double ys[] = {y, y, y - Math.sin(Math.PI / 3.0) * len};
359             GeneralPath polygon = new GeneralPath(GeneralPath.WIND_EVEN_ODD, 3);
360             polygon.moveTo(xs[0], ys[0]);
361             for (int i = 1; i < xs.length; i++)
362                 polygon.lineTo(xs[i], ys[i]);
363             polygon.closePath();
364             graph.fill(polygon);
365         }
366     }
```

Рисунок 17. Код для відображення трикутника із заливкою чи без неї

```
387     private void circleFractal(int iters, double radius, double x, double y){
388         Ellipse2D elip = new Ellipse2D.Double(x - radius, y - radius, 2* radius, 2 * radius);
389         graph.draw(elip);
390         double halvedRadius = new Double(Double.toString(radius)) / 2;
391         if (iters - 1 > 0 & radius > 2){
392             circleFractal(iters - 1, halvedRadius, x - radius, y);
393             circleFractal(iters - 1, halvedRadius, x + radius, y);
394             circleFractal(iters - 1, halvedRadius, x, y - radius);
395             circleFractal(iters - 1, halvedRadius, x, y + radius);
396         }
397         DrawingPanel.paintComponents(graph);
398         DrawingPanel.setVisible(true);
399     }
400     //IDEA make different variations of this fractal
```

Рисунок 18. Код функції для малювання фракталу із кіл

```
402     private void squareFractal(int iters, double radius, double x, double y){
403         Rectangle2D rect = new Rectangle2D.Double(x - radius, y - radius, 2* radius, 2 * radius);
404         graph.draw(rect);
405         double halvedRadius = new Double(Double.toString(radius)) / 2;
406         if (iters - 1 > 0 & radius > 2){
407             squareFractal(iters - 1, halvedRadius, x - 2 * radius + halvedRadius, y);
408             squareFractal(iters - 1, halvedRadius, x + 2 * radius - halvedRadius, y);
409             squareFractal(iters - 1, halvedRadius, x, y - 2 * radius + halvedRadius);
410             squareFractal(iters - 1, halvedRadius, x, y + 2 * radius - halvedRadius);
411         }
412         DrawingPanel.paintComponents(graph);
413         DrawingPanel.setVisible(true);
414     }
415     //IDEA make a square fractal with new squares drawn on corners, not on sides
416     //IDEA combine them both??
```

Рисунок 19. Код функції для малювання фракталу із квадратів

```
439     private Path LSystemGen(int iters){
440         //ERROR! This function only allowns up to 2 runs in a row, then the program must be restarted
441         JFileChooser fc = new JFileChooser();
442         FileNameExtensionFilter filter = new FileNameExtensionFilter("Text Files", "txt");
443         fc.setFileFilter(filter);
444         int val = fc.showOpenDialog(this);
            String name = "";
446         Path p = null;
447         if (val == JFileChooser.APPROVE_OPTION){
448             try{
449                 for (int i = 0; i < iters; i++){
450                     File raf = fc.getSelectedFile();
451                     p = Paths.get(raf.getAbsolutePath());
452                     name = raf.getName();
453                     String path = raf.getParent();
454                     String newFileName = path + "\\temp.txt";
455
456                     Path newPath = Paths.get(newFileName);
457                     if (!Files.exists(newPath))
                            newPath = Files.createFile(newPath);
459                     File writeTo = new File(newFileName);
460
461                     //If there is something in the Init file before the start, clear it
462                     if (i == 0){
463                         if (raf.length() != 0){
464                             BufferedWriter write = new BufferedWriter(new FileWriter(raf));
465                             write.close();
466                         }
467                     }
468
                        BufferedReader read = new BufferedReader(new FileReader(raf));
470                     BufferedWriter write = new BufferedWriter(new FileWriter(writeTo));
471
472                     if (raf.length() == 0)
473                         write.write("F");
474                     else{
475                         while (read.ready()){
476                             char c = (char)read.read();
477                             if (c == 'F'){  //MAYBE TO DO: 2 letters in 1 byte?
478                                 write.write("FF+[+F-F-F]-[-F+F+F]");
479                             }
480                             else
481                                 write.write(c);
482                         }
483                     }
484                     read.close();
485                     write.close();
486                     boolean res = raf.delete();
487                     Path source = Paths.get(writeTo.getAbsolutePath());
488                     p = Files.move(source, source.resolveSibling(name));
489                 }
490             }
                catch(Exception ex){
492
493             }
494         }
495         return p;
496     }
```

Рисунок 20. Код функції для формування інструкції L-системи та повертання шляху до файлу

```
498  private void LSystemGenerator(double initLength, double initX, double initY, double initAngle, Path instruction){
499      double curLength = initLength;
500      double curX = initX;
501      double curY = initY;
502      double curAngleDegree = initAngle;
503      double keepX = initX;
504      double keepY = initY;
505      double keepLength = initLength;
506      double keepAngle = initAngle;
507      Random rand = new Random();
         double angleShift = 0.0;
         double radian = 0.0;
510      double shortage = 0.9;
511
512      try{
513          File file = instruction.toFile();
514          BufferedReader read = new BufferedReader(new FileReader(file));
515          while (read.ready()){
516              char now = (char)read.read();
                 switch (now){
518                  case 'F':  //if F, move with drawing a line
519                      radian = curAngleDegree * Math.PI / 180;
520                      Line2D.Double line = new Line2D.Double(curX, curY, curX + curLength * Math.sin(radian), curY + curLength * Math.cos(radian));
521                      graph.draw(line);
522                      curX += curLength * Math.sin(radian);
523                      curY += curLength * Math.cos(radian);
524                      curLength *= shortage;
```

Рисунок 21. Початок функції малювання L-системи

```
525                      DrawingPanel.paintComponents(graph);
526                      DrawingPanel.setVisible(true);
527                      break;
528                  case 'G':  //if G, move without drawing
529                      radian = curAngleDegree * Math.PI / 180;
530                      curX += curLength * Math.sin(radian);
531                      curY += curLength * Math.cos(radian);
532                      curLength *= shortage;
533                      DrawingPanel.paintComponents(graph);
534                      DrawingPanel.setVisible(true);
535                      break;
536                  case '+':  //if +, rotate for several degrees +
537                      angleShift = rand.nextInt(45) + 1 + rand.nextDouble();
538                      //angleShift = 25;
539                      curAngleDegree += angleShift;
540                      DrawingPanel.paintComponents(graph);
541                      DrawingPanel.setVisible(true);
542                      break;
543                  case '-':  //if -, rotate for several degrees -
544                      angleShift = rand.nextInt(45) + 1 + rand.nextDouble();
545                      //angleShift = 25;
546                      curAngleDegree -= angleShift;
547                      DrawingPanel.paintComponents(graph);
548                      DrawingPanel.setVisible(true);
549                      break;
```

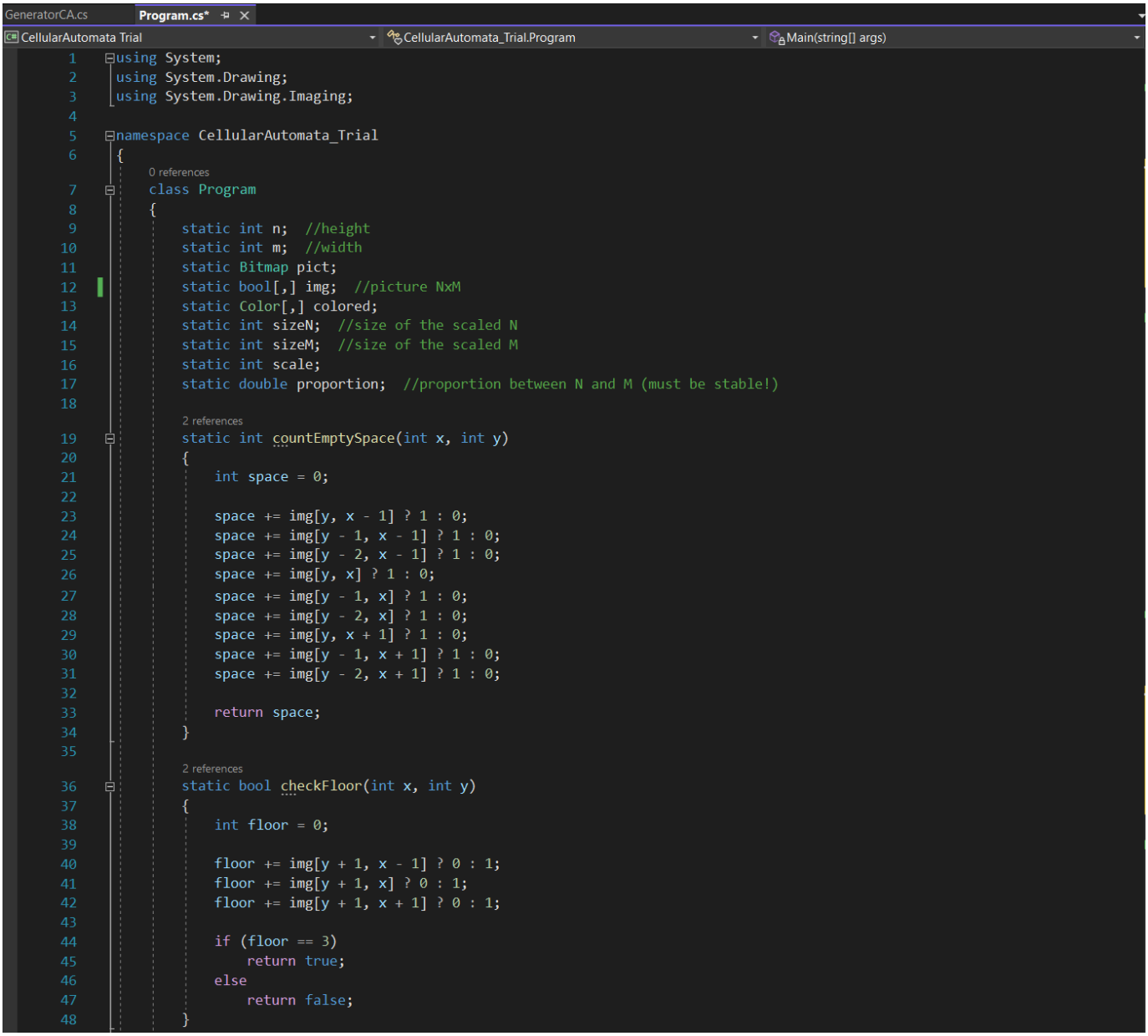
Рисунок 22. Середина функції малювання L-системи

```
550                      case '[':  //if [, save the current coordinates
551                          keepX = curX;
552                          keepY = curY;
553                          keepLength = curLength;
554                          keepAngle = curAngleDegree;
555                          DrawingPanel.paintComponents(graph);
556                          DrawingPanel.setVisible(true);
557                          break;
558                      case ']':  //if ], return to kept coordinates (zeros by default)
559                          curX = keepX;
560                          curY = keepY;
561                          curLength = keepLength;
562                          curAngleDegree = keepAngle;
563                          DrawingPanel.paintComponents(graph);
564                          DrawingPanel.setVisible(true);
565                          break;
566                      default:
567                          break;
568                  }
569              }
570          }
             catch(Exception ex){
572
573          }
574      }
```

Рисунок 23. Кінець функції малювання L-системи

ДОДАТОК Б

Головна функція (Program.cs):



GeneratorCA.cs | Program.cs*

CellularAutomata Trial | CellularAutomata_Trial.Program | Main(string[] args)

```
1   using System;
2   using System.Drawing;
3   using System.Drawing.Imaging;
4
5   namespace CellularAutomata_Trial
6   {
        0 references
7       class Program
8       {
9           static int n;  //height
10          static int m;  //width
11          static Bitmap pict;
12          static bool[,] img;  //picture NxM
13          static Color[,] colored;
14          static int sizeN;  //size of the scaled N
15          static int sizeM;  //size of the scaled M
16          static int scale;
17          static double proportion;  //proportion between N and M (must be stable!)
18
            2 references
19          static int countEmptySpace(int x, int y)
20          {
21              int space = 0;
22
23              space += img[y, x - 1] ? 1 : 0;
24              space += img[y - 1, x - 1] ? 1 : 0;
25              space += img[y - 2, x - 1] ? 1 : 0;
26              space += img[y, x] ? 1 : 0;
27              space += img[y - 1, x] ? 1 : 0;
28              space += img[y - 2, x] ? 1 : 0;
29              space += img[y, x + 1] ? 1 : 0;
30              space += img[y - 1, x + 1] ? 1 : 0;
31              space += img[y - 2, x + 1] ? 1 : 0;
32
33              return space;
34          }
35
            2 references
36          static bool checkFloor(int x, int y)
37          {
38              int floor = 0;
39
40              floor += img[y + 1, x - 1] ? 0 : 1;
41              floor += img[y + 1, x] ? 0 : 1;
42              floor += img[y + 1, x + 1] ? 0 : 1;
43
44              if (floor == 3)
45                  return true;
46              else
47                  return false;
48          }
```

```
0 references
static void Main(string[] args)
{
    //init picture dimentions
    Console.WriteLine("Please enter how many pixels do you want");
    Console.WriteLine("Height: ");
    n = int.Parse(Console.ReadLine());
    Console.WriteLine("Width: ");
    m = int.Parse(Console.ReadLine());

    proportion = n > m ? n / m : m / n;
    sizeN = n > m ? (int)(50 * proportion) : 50;
    sizeM = n < m ? (int)(50 * proportion) : 50;

    scale = n / sizeN;

    pict = new Bitmap(m, n);
    img = new bool[n, m];
    colored = new Color[n, m];

    //black - solid, white - open
    Console.WriteLine("How many smoothing steps do you want? ");
    int steps = int.Parse(Console.ReadLine());

    Random rand = new Random();
    //chance of the point to spawn white
```

```
    int spawnChance = 45;

    try
    {
        GeneratorCA gener = new GeneratorCA(n, m, sizeN, sizeM, scale, steps, spawnChance);
        gener.generate();
        img = gener.getArray().Clone() as bool[,];

        //Copying the image into the bitmap for export
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
            {
                if (img[i, j])
                {
                    pict.SetPixel(j, i, Color.White);
                    colored[i, j] = Color.White;
                }
                else
                {
                    pict.SetPixel(j, i, Color.Black);
                    colored[i, j] = Color.Black;
                }
            }

        //NEEDED Deal with empty unconnected spaces!
```

```
        //Now - set the Start & End points for the player
        bool stop = false;

        //start point
        for (int x = 1; x < m - 1; x++)
        {
            for (int y = n - 2; y > 3; y--)
            {
                int count = countEmptySpace(x, y);
                bool floor = checkFloor(x, y);
                if (count == 9 && floor)
                {
                    colored[y, x] = Color.Red;
                    pict.SetPixel(x, y, Color.Red);
                    stop = true;
                    break;
                }
            }
            if (stop)
            {
                stop = false;
                break;
            }
        }
```

```
198                 //end point
199                 for (int x = m - 2; x > 0; x--)
200                 {
201                     for (int y = n - 2; y > 3; y--)
202                     {
203                         int count = countEmptySpace(x, y);
204                         bool floor = checkFloor(x, y);
205                         if (count == 9 && floor)
206                         {
207                             colored[y, x] = Color.Red;
208                             pict.SetPixel(x, y, Color.Red);
209                             stop = true;
210                             break;
211                         }
212                     }
213                     if (stop)
214                     {
215                         stop = false;
216                         break;
217                     }
218                 }
219 
220                 //export the bitmap into the JPEG file
221                 pict.Save("C:\\Data\\Diploma\\image1st.jpeg", ImageFormat.Jpeg);
222             }
```

```
223             catch (Exception ex)
224             {
225                 Console.WriteLine("Exception caught");
226                 Console.WriteLine(ex);
227             }
228         }
229     }
230 }
231 
```

Генератор (GeneratorCA.cs):

GeneratorCA.cs | Program.cs

CellularAutomata Trial | CellularAutomata_Trial.GeneratorCA | n

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6 
7  namespace CellularAutomata_Trial
8  {
       3 references
9      class GeneratorCA
10     {
11         private int n;
12         private int m;
13         private int sizeN;
14         private int sizeM;
15         private bool[,] array;
16         private bool[,] arrayScaled;
17         private bool[,] tempArray;
18         private int scale;
19         private int steps;
20         private int spawnChance;
21 
           1 reference
22         public GeneratorCA(int N, int M, int nS, int mS, int s, int st, int sCh)
23         {
24             n = N;
25             m = M;
26             sizeN = nS;
```

```
27              sizeM = mS;
28              arrayScaled = new bool[sizeN, sizeM];
29              array = new bool[n, m];
30              tempArray = new bool[n, m];
31              scale = s;
32              steps = st;
33              spawnChance = sCh;
34          }
35
            2 references
36          private int countWhiteScale(int x, int y, bool scaled)  //here's where the coordinates will be transfered
37          {
38              int number = 0;
39              if (scaled)
40              {
41                  number += arrayScaled[y - 1, x - 1] ? 1 : 0;
42                  number += arrayScaled[y - 1, x] ? 1 : 0;
43                  number += arrayScaled[y - 1, x + 1] ? 1 : 0;
44                  number += arrayScaled[y, x - 1] ? 1 : 0;
45                  number += arrayScaled[y, x + 1] ? 1 : 0;
46                  number += arrayScaled[y + 1, x - 1] ? 1 : 0;
47                  number += arrayScaled[y + 1, x] ? 1 : 0;
48                  number += arrayScaled[y + 1, x + 1] ? 1 : 0;
49              }
50              else
51              {
52                  number += array[y - 1, x - 1] ? 1 : 0;
```

```
53                  number += array[y - 1, x] ? 1 : 0;
54                  number += array[y - 1, x + 1] ? 1 : 0;
55                  number += array[y, x - 1] ? 1 : 0;
56                  number += array[y, x + 1] ? 1 : 0;
57                  number += array[y + 1, x - 1] ? 1 : 0;
58                  number += array[y + 1, x] ? 1 : 0;
59                  number += array[y + 1, x + 1] ? 1 : 0;
60              }
61              return number;
62          }
63
            1 reference
64          private void fillOrigin(int x, int y)
65          {
66              for (int i = 0; i < scale; i++)
67              {
68                  for (int j = 0; j < scale; j++)
69                  {
70                      array[y * scale + i, x * scale + j] = arrayScaled[y, x];
71                      tempArray[y * scale + i, x * scale + j] = arrayScaled[y, x];
72                  }
73              }
74          }
```

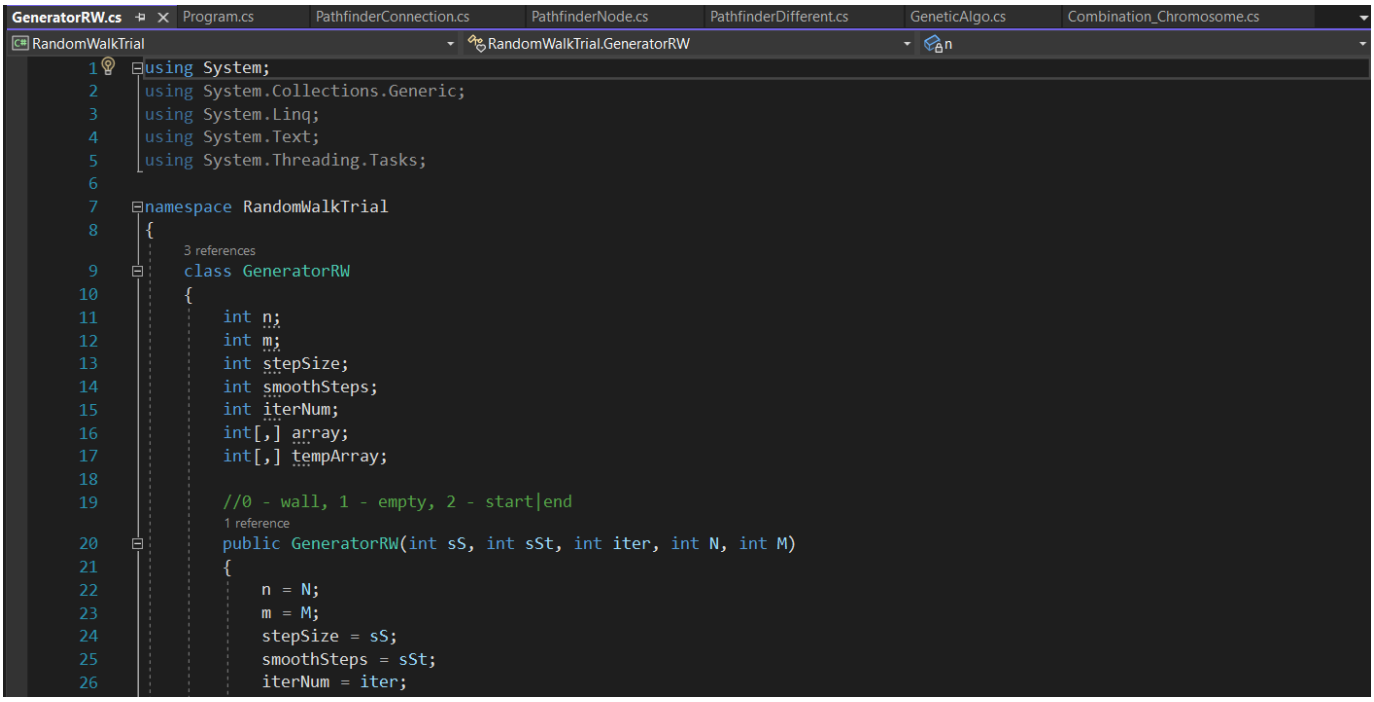
```
            1 reference
75          public void generate()
76          {
77              Random rand = new Random();
78              //Fill the scaled image
79              for (int i = 0; i < sizeN; i++)
80              {
81                  for (int j = 0; j < sizeM; j++)
82                  {
83                      if (i == 0 || j == 0 || i == sizeN - 1 || j == sizeM - 1)
84                          arrayScaled[i, j] = false;
85                      else if (rand.Next(1, 101) > spawnChance)
86                          arrayScaled[i, j] = false;
87                      else
88                          arrayScaled[i, j] = true;
89                  }
90              }
91
92              //implement the algorithm
93              //Apply the 5-born/3-stay scheme for the scaled picture
94              for (int i = 0; i < 20; i++)
95              {
96                  for (int x = 1; x < sizeM - 1; x++)
97                  {
98                      for (int y = 1; y < sizeN - 1; y++)
99                      {
```

```
                    int neighbours = countWhiteScale(x, y, true);
                    if (neighbours < 3)
                        arrayScaled[y, x] = false;
                    else if (neighbours >= 5)
                        arrayScaled[y, x] = true;
                }
            }
        }

        //Rescaling the picture back to the original size
        for (int x = 1; x < sizeM - 1; x++)
        {
            for (int y = 1; y < sizeN - 1; y++)
            {
                fillOrigin(x, y);
            }
        }

        //Now the smoothing part starts
        //This is the 5-born/5-stay scheme on the original-sized picture
        for (int i = 0; i < steps; i++)
        {
            for (int x = 1; x < m - 1; x++)
            {
                for (int y = 1; y < n - 1; y++)
                {
                    int neighbours = countWhiteScale(x, y, false);
```

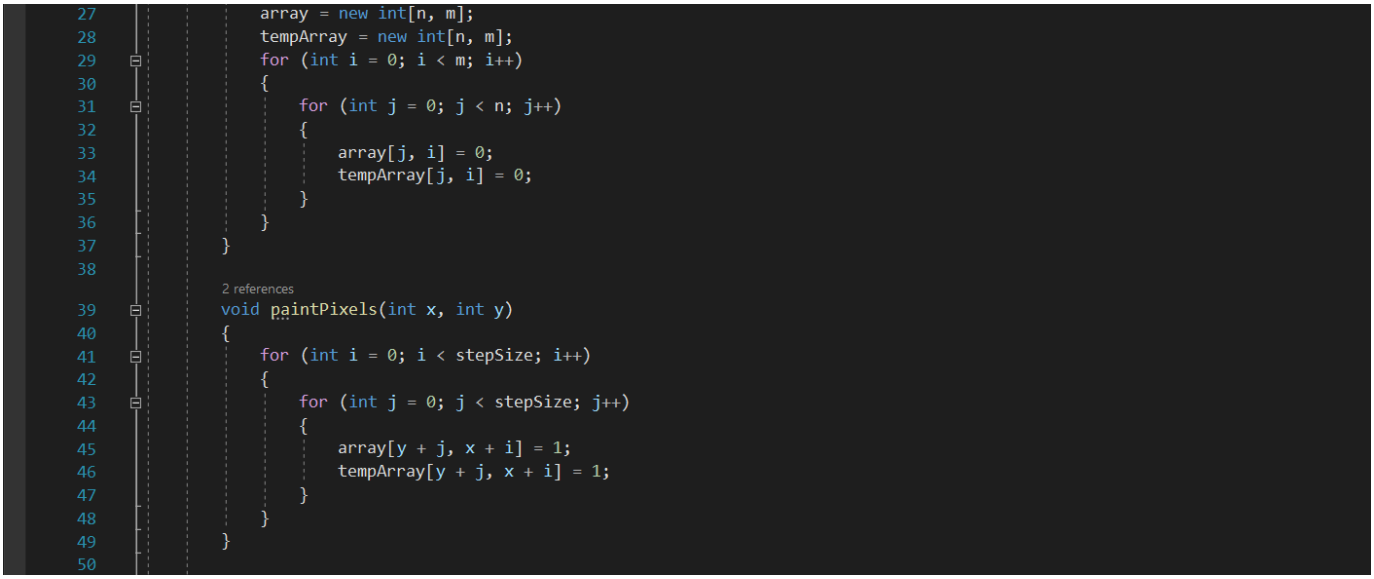
```
                    if (neighbours < 5)
                        tempArray[y, x] = false;
                    else if (neighbours >= 5)
                        tempArray[y, x] = true;
                }
            }
            for (int x = 0; x < m; x++)
            {
                for (int y = 0; y < n; y++)
                    array[y, x] = tempArray[y, x];
            }
        }
    }

    1 reference
    public bool[,] getArray()
    {
        return array;
    }

    0 references
    public bool[,] getArrayScaled()
    {
        return arrayScaled;
    }
}
}

```

ДОДАТОК В

Тільки генератор (GeneratorRW.cs), оскільки до головної функції включено також генетичний алгоритм:



```
GeneratorRW.cs | Program.cs | PathfinderConnection.cs | PathfinderNode.cs | PathfinderDifferent.cs | GeneticAlgo.cs | Combination_Chromosome.cs
RandomWalkTrial | RandomWalkTrial.GeneratorRW | n
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace RandomWalkTrial
8  {
       3 references
9      class GeneratorRW
10     {
11         int n;
12         int m;
13         int stepSize;
14         int smoothSteps;
15         int iterNum;
16         int[,] array;
17         int[,] tempArray;
18
19         //0 - wall, 1 - empty, 2 - start|end
           1 reference
20         public GeneratorRW(int sS, int sSt, int iter, int N, int M)
21         {
22             n = N;
23             m = M;
24             stepSize = sS;
25             smoothSteps = sSt;
26             iterNum = iter;
```



```
27             array = new int[n, m];
28             tempArray = new int[n, m];
29             for (int i = 0; i < m; i++)
30             {
31                 for (int j = 0; j < n; j++)
32                 {
33                     array[j, i] = 0;
34                     tempArray[j, i] = 0;
35                 }
36             }
37         }
38
           2 references
39         void paintPixels(int x, int y)
40         {
41             for (int i = 0; i < stepSize; i++)
42             {
43                 for (int j = 0; j < stepSize; j++)
44                 {
45                     array[y + j, x + i] = 1;
46                     tempArray[y + j, x + i] = 1;
47                 }
48             }
49         }
50
```

```
1 reference
int countWhiteScale(int x, int y)  //here's where the coordinates will be transfered
{
    int number = 0;
    number += array[y - 1, x - 1] != 0 ? 1 : 0;
    number += array[y - 1, x] != 0 ? 1 : 0;
    number += array[y - 1, x + 1] != 0 ? 1 : 0;
    number += array[y, x - 1] != 0 ? 1 : 0;
    number += array[y, x + 1] != 0 ? 1 : 0;
    number += array[y + 1, x - 1] != 0 ? 1 : 0;
    number += array[y + 1, x] != 0 ? 1 : 0;
    number += array[y + 1, x + 1] != 0 ? 1 : 0;
    return number;
}

2 references
int countEmptySpace(int x, int y)
{
    int space = 0;

    space += array[y, x - 1] != 0 ? 1 : 0;
    space += array[y - 1, x - 1] != 0 ? 1 : 0;
    space += array[y - 2, x - 1] != 0 ? 1 : 0;
    space += array[y, x] != 0 ? 1 : 0;
    space += array[y - 1, x] != 0 ? 1 : 0;
    space += array[y - 2, x] != 0 ? 1 : 0;
    space += array[y, x + 1] != 0 ? 1 : 0;
    space += array[y - 1, x + 1] != 0 ? 1 : 0;
    space += array[y - 2, x + 1] != 0 ? 1 : 0;

    return space;
}

2 references
bool checkFloor(int x, int y)
{
    int floor = 0;

    floor += array[y + 1, x - 1] != 0 ? 0 : 1;
    floor += array[y + 1, x] != 0 ? 0 : 1;
    floor += array[y + 1, x + 1] != 0 ? 0 : 1;

    if (floor == 3)
        return true;
    else
        return false;
}

1 reference
public void generate()
{
    Random rand = new Random();
    int x, y;  //init the starting point
    x = rand.Next(0, m - stepSize);
    y = rand.Next(0, n - stepSize);
```

```
            paintPixels(x, y);
            //pict.SetPixel(x, y, Color.White);

            int direction;
            //0 - up, 1 - down, 2 - left, 3 - right
            //loop the random walk
            for (int i = 0; i < iterNum; i++)
            {
                direction = rand.Next(0, 4);
                while (true)
                {
                    if (direction == 0 && y + 2 * stepSize >= n)
                        direction = rand.Next(0, 4);
                    else if (direction == 1 && y - 2 * stepSize < 0)
                        direction = rand.Next(0, 4);
                    else if (direction == 2 && x - 2 * stepSize < 0)
                        direction = rand.Next(0, 4);
                    else if (direction == 3 && x + 2 * stepSize >= m)
                        direction = rand.Next(0, 4);
                    else break;
                }
                if (direction == 0)
                    y += stepSize;
                else if (direction == 1)
                    y -= stepSize;
                else if (direction == 2)
                    x -= stepSize;
```

```
                else if (direction == 3)
                    x += stepSize;
                paintPixels(x, y);
                //pict.SetPixel(x, y, Color.White);
            }

            //Let's smooth the picture
            for (int i = 0; i < smoothSteps; i++)
            {
                for (int X = 1; X < m - 1; X++)
                {
                    for (int Y = 1; Y < n - 1; Y++)
                    {
                        int neighbours = countWhiteScale(X, Y);
                        if (neighbours < 5)
                            tempArray[Y, X] = 0;
                        else if (neighbours >= 5)
                            tempArray[Y, X] = 1;
                    }
                }
                for (int X = 0; X < m; X++)
                {
                    for (int Y = 0; Y < n; Y++)
                    {
                        array[Y, X] = tempArray[Y, X];
                    }
                }
            }
```

```
            }

            //Now let's place the Start and End points for the player
            bool stop = false;

            //start point
            for (int X = 1; X < m - 1; X++)
            {
                for (int Y = n - 2; Y > 3; Y--)
                {
                    int count = countEmptySpace(X, Y);
                    bool floor = checkFloor(X, Y);
                    if (count == 9 && floor)
                    {
                        array[Y, X] = 2;
                        stop = true;
                        break;
                    }
                }
                if (stop)
                {
                    stop = false;
                    break;
                }
            }
```

```

            //end point
            for (int X = m - 2; X > 0; X--)
            {
                for (int Y = n - 2; Y > 3; Y--)
                {
                    int count = countEmptySpace(X, Y);
                    bool floor = checkFloor(X, Y);
                    if (count == 9 && floor)
                    {
                        array[Y, X] = 2;
                        stop = true;
                        break;
                    }
                }
                if (stop)
                {
                    stop = false;
                    break;
                }
            }
        }
```

```

        1 reference
        public int[,] getArray()
        {
            return array;
        }
    }
}

```

ДОДАТОК Г

Робота генератора на основі випадкового проходження:

Рисунок 24. 100x200 пікселів, 2500 кроків, розмір кроку 2x2, 3 ітерації згладжування, справжній розмір

Рисунок 25. 100x200 пікселів, 2500 кроків, розмір кроку 2x2, 3 ітерації згладжування, справжній розмір

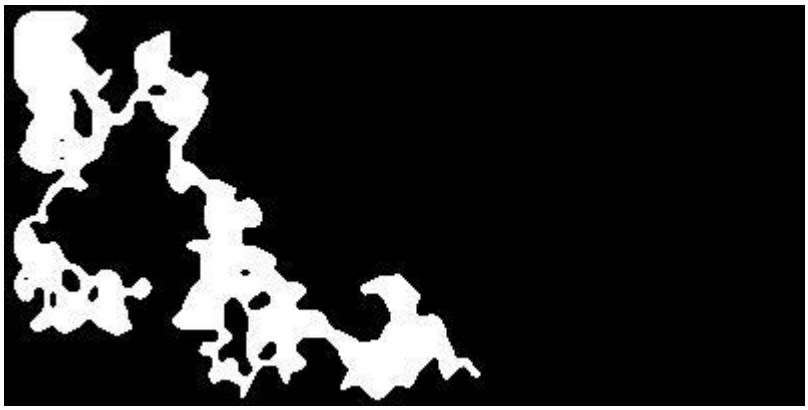


Рисунок 26. 200x400 пікселів, 5000 кроків, розмір кроку 3x3, 5 ітерацій згладжування, справжній розмір

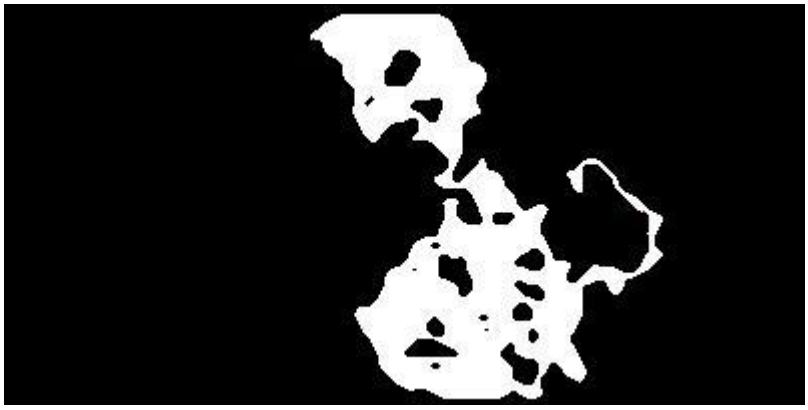


Рисунок 27. 200x400 пікселів, 5000 кроків, розмір кроку 3x3, 5 ітерацій згладжування, справжній розмір

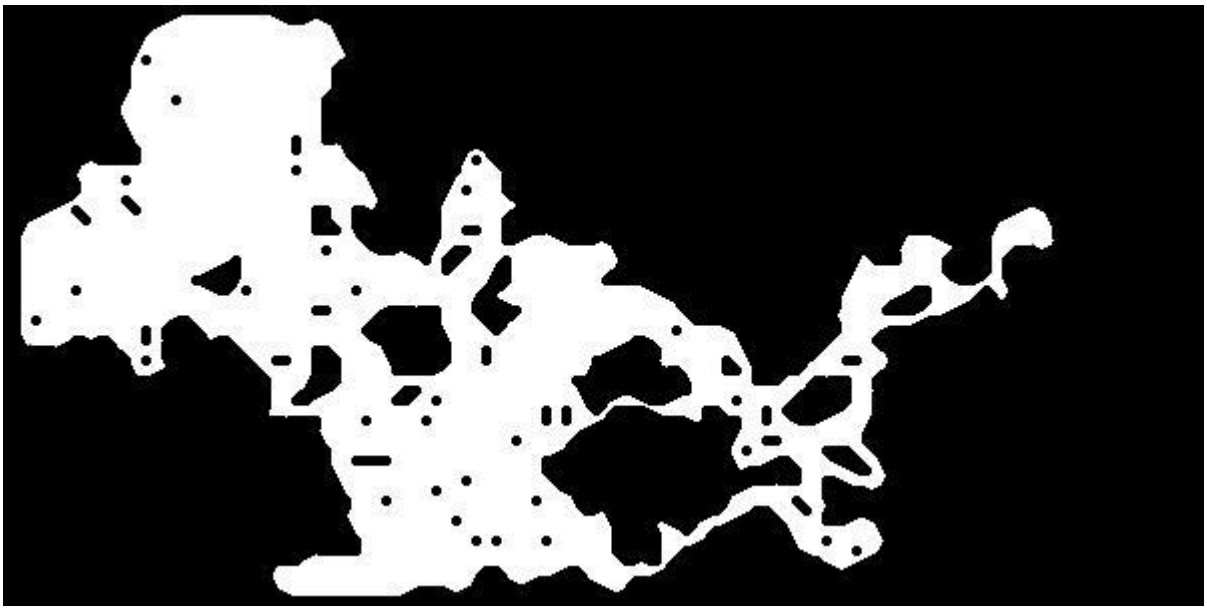


Рисунок 28. 300x600 пікселів, 10000 кроків, розмір кроку 5x5, 10 ітерацій згладжування, справжній розмір

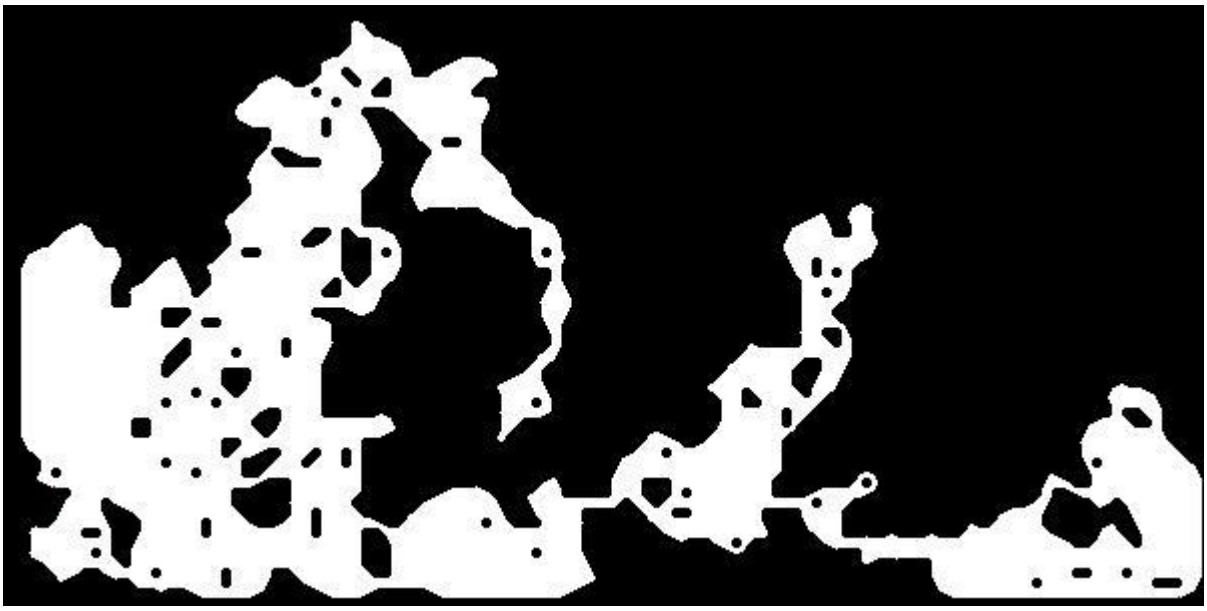


Рисунок 29. 300x600 пікселів, 10000 кроків, розмір кроку 5x5, 10 ітерацій згладжування, справжній розмір

Робота генератора на основі клітинного автомату:

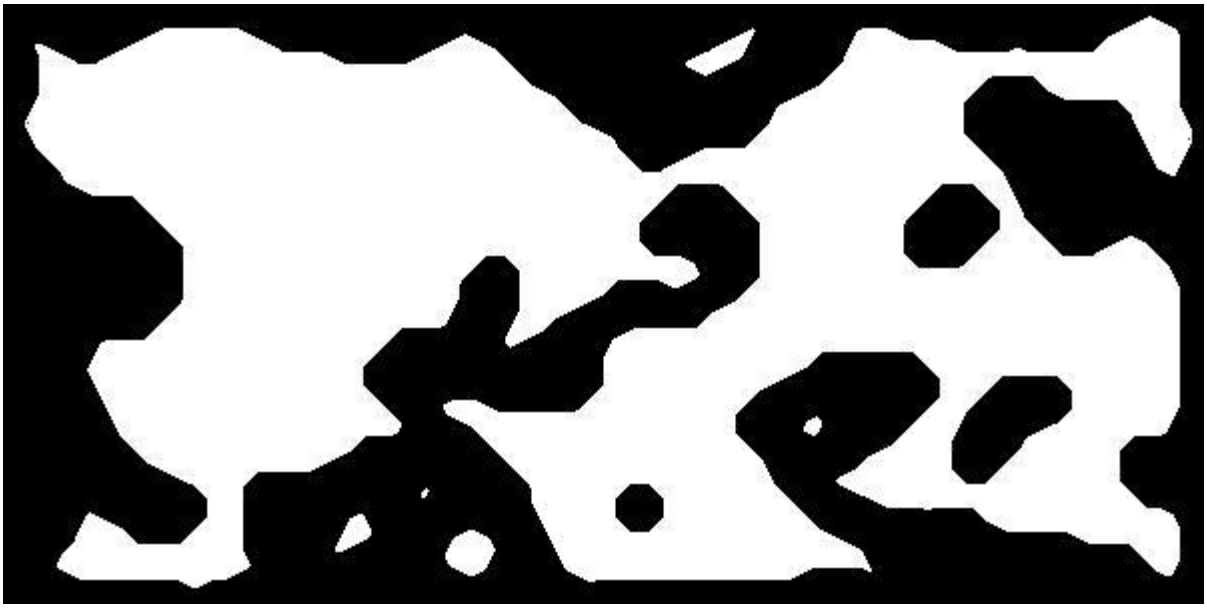


Рисунок 30. 300х600 пікселів, "схема 5 на 3" для основи, "схема 5 на 5" для згладжування, 10 ітерацій згладжування, справжній розмір

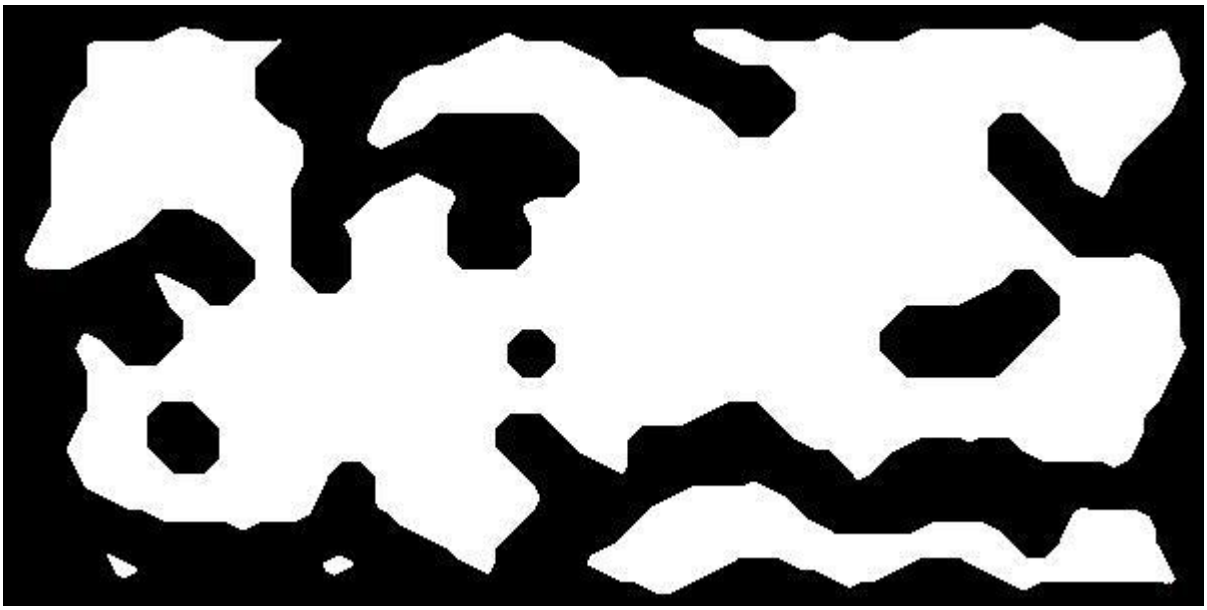


Рисунок 31. 300х600 пікселів, "схема 5 на 3" для основи, "схема 5 на 5" для згладжування, 10 ітерацій згладжування, справжній розмір