МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Комп'ютерна система генерації тривимірного ігрового світу**

Виконав: студент 2 курсу, групи 8.1219-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
С. О. Волік
(ініціали та прізвище)
Керівник доцент, к. т. н. О.М.Міхайлуца
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «АйтіДіменшн»
В. С. Тряпічко
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кафедра _____ програмного забезпечення автоматизованих систем
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри_В.Г. Вербицький
"_01_" __вересня_____2020_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

_____
(прізвище, ім'я, по батькові)

1.Тема роботи ___Комп'ютерна система генерації тривимірного ігрового світу_____

керівник роботи __О.М. Міхайлуца, доцент, канд. технічних наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від _"25"_травня_2020_року № 600-с_____

2. Строк подання студентом кваліфікаційної роботи___30.11.2020____
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми процедурної генерації будинків та методів її реалізації;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
___слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.09.2020

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	02.09-10.09.20	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.09-12.09.2020	виконано
3	Аналіз існуючих методів рішення	13.09-17.09.19	виконано
4	Дослідження області процедурної побудови приміщень	18.09-24.09.19	виконано
5	Узгодження подальших дій з науковим керівником	25.09-26.09.19	виконано
6	Аналіз теоретичних відомостей	27.09-15.10.19	виконано
7	Проектування інтерфейсу та архітектури побудови будівель	15.10-23.10.19	виконано
8	Узгодження інтерфейсу та архітектури з науковим керівником	23.10-24.10.19	виконано
9	Реалізація функціоналу процедурної побудови будівель	25.10-10.11.19	виконано
10	Представлення отриманих результатів науковому керівнику і узгодження плану подальшого дослідження	11.11-13.11.19	виконано
11	Реалізація тестування та доповнення програмного застосунку	14.11-20.11.19	виконано
12	Проведення аналізу можливостей розробленого щастосунку	21.11-22.11.20	виконано
13	Оформлення звіту	23.11-27.11.20	виконано

Студент _____ С.О. Волік
(підпис) (прізвище та ініціали)

Керівник роботи _____ О. М. Міхайлуца
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 100

Рисунків: 88

Таблиць: 2

Джерел: 29

Волік С.О. Комп’ютерна система генерації тривимірного ігрового світу.

Кваліфікаційна робота для здобуття ступеня вищої освіти магістра за спеціальністю 121 — Інженерія програмного забезпечення, науковий керівник Міхайлуца О.М. Інженерний навчально-науковий інститут ЗНУ, 2020.

Мета кваліфікаційної роботи полягає у вивченні методів процедурного генерування приміщень з побудовою їх у вигляді 3D, а також у створенні власної програмної системи, яка зможе створювати різноманітні види будівель.

Досліджено методи побудови процедурних приміщень, їх проблематику і можливості розробки. Для розробки початкового продукту використовувалися мова програмування C# та game engine Unity3D для реалізації 3D відображення.

Ключові слова: *ПРОЦЕДУРНЕ ГЕНЕРУВАННЯ, ПЛАН ПРИМІЩЕННЯ, СТРУКТУРНА СІТКА, C#, UNITY3D, ПОЛІГОН, АЛГОРИТМ ПРЯМОГО СКЕЛЕТУ.*

SUMMARY

Pages: 100

Figures: 88

Tables: 2

Sources: 29

Volik S.A. Computer system for generating a three-dimensional game world.

Qualification work for obtaining a higher education degree of a master in specialty 121 — Software Engineering, scientific adviser O. M. Mikhailutsa. Engineering educational and scientific institute of ZNU, 2020.

The purpose of the qualification work is to study the methods of procedural creation of buildings with their construction in 3D, as well as to create their own software system that can create various types of buildings.

Methods of procedural creation of buildings, their problems and the possibilities of developing and using the system are investigated. To develop the original product, was used the C # programming language and the game engine Unity3d to implement 3d rendering.

Key words: *PROCEDURAL GENERATION, FLOOR PLAN, STRUCTURAL GRID, C #, UNITY3D, POLIGON, STRAIGHT SKELETON ALGORYTHM.*

ЗМІСТ

ВСТУП

Актуальність теми

У сучасному світі, а саме в ігровій індустрії є дві основні існуючі проблеми. Перша – швидке створення ігрового контенту. Тобто необхідно за мінімальну кількість часу створити добре працюючий продукт за допомогою автоматизація цього процесу. Друга проблема – це якість опрацювання ігрових об’єктів, тобто відображення ігрового об’єкту в найбільш реалістичному або потрібному вигляді. Нажаль з цим поки що в найкращій якості може справитися лише людина.

Щорічно ігрова індустрія в світі збільшує свої обороти, впроваджуються нові технології, збільшуються бюджети ігор, але нажаль щоб створити чудову гру потрібно мати великі ресурси, як грошові так и людські, і це є основною перепоною для розробників ігор з невеликими бюджетами. В наш час існує три варіанти створення ігрового контенту. Перший – повністю людською працею, тобто 3D художники створюють контент завдяки стандартним програмам для 3D моделювання. Другий – напівавтоматичний, коли 3D художник використовує допоміжні інструменти, які пришвидшують рутинну роботу та зменшують витрачений час на створення контенту. Третій варіант –майже повністю автоматичний, а інколи і повністю автоматичний. Кожен з цих варіантів має свої плюси та мінуси. Перший варіант дуже дорогий, як в грошовому еквіваленті так і в людино-годинах, але результати дуже якісні. Тому ААА ігрові студії [27] зазвичай використовую такий варіант. Другий варіант – це баланс усього і затрат по часу, і затрат по людино-годинам, але результати гірші ніж в першому варіанті. Третій варіант насправді не такий і простий як здається, його великий мінус в тому, що він інколи може буду не дуже реалістичний і поганої якості, але він відкриває нові можливості для самих ігор. Основна перевага процедурного генерування над ручною працею в тому, що за допомогою автоматизованого процесу (без участі людини) мо-

жуть генеруватися різноманітні варіанти ігрового контенту (генерація світів, генерація ігрових персонажів, генерування лабіринтів і т.д.).

Написана дипломна робота базується на другому варіанті створення контенту, бо співпраця людина-машина дає оптимальний результат по якості та по швидкості створення.

Проблема процедурного генерування яка була розібрана в кваліфікаційній роботі – це процедурне генерування приміщень. Ідея в тому, що програма зможе генерувати приміщення з можливістю змінювати налаштування від кількості поверхів, зміни типу даху до налаштування генерування плану будівлі, зміни моделей стін, вікон та інше. Це допоможе збільшити варіативність будівель, як зовнішньо (форми будівель), так і внутрішньо (плану приміщення), а також може зменшити витрачені людино-години при створені більш деталізованого ігрового об’єкту будівлі.

Мета і завдання дослідження

Метою дослідження є вивчення методів створення планів будівель автоматизованим чином, а також створення власної системи за допомогою якої можна створювати плани приміщень в реальному часі.

Об’єкт дослідження

Об’єктом дослідження є сама будівля, як явище, та її структура (архітектурні особливості, планування будівлі та інше).

Предмет дослідження

Автоматизоване генерування планів будівель.

Методи дослідження

Для розв’язання представлених завдань використовуються такі методи дослідження:

- Аналіз джерел про генерування планів будівель і способи вирішення проблеми.
- Проведення аналогії серед існуючих на ринку систем генерування будівель.
- Синтез отриманих результатів досліджень.
- Порівняльний аналіз програмних продуктів.

Наукова новизна одержаних результатів

Одержані результати є наочним відображенням переваг та недоліків автоматизованих систем створення контенту. Було проведено аналіз предметної області та на основі цього було розроблено програмну систему здатну генерувати будівлі з внутрішнім плануванням поверхів. На основі цих даних у майбутньому можлива розробка ігор, наприклад для генерування процедурно створеного міста або створення моделі міст та інше.

Головною відмінністю розробленої системи від більшості вже існуючих є те, що вона має повноцінний контроль над генеруванням приміщень, система має можливості по зміні всіх моделей та матеріалів, які використовуються під час генерування будівлі.

Практичне значення одержаних результатів

Практичне значення одержаних результатів таке:

- Проведений аналіз існуючих рішень.
- Готовий програмний продукт, який генерує плани будівель у 3D.
- Проведений аналіз методів оптимізації процедурного генерування

Апробація результатів

Результати роботи було представлено на науково-технічних конференціях студентів, магістрантів, аспірантів, молодих вчених [28,29].

Глосарій

План будинку — це розріз горизонтальною площиною на рівні, трохи вищому від підвіконня. Для багатоповерхового будинку плани виконують для кожного поверху.

Процедурна генерація (англ. Procedural content generation, скор. PCG) — автоматичне створення ігрового контенту за допомогою алгоритмів. PCG є програмне забезпечення, яке може створювати ігровий контент самостійно, або спільно при взаємодії з гравцями або геймдизайнерами.

Структурна сітка, модульна сітка — регулярні рамки орієнтирів, до яких прив'язані розміри основних структурних компонентів плану будівлі, а в міському плануванні — мережа шахових вулиць і проспектів, що утворюють основну планування міста або селища.

Unity — крос-платформова середа розробки комп'ютерних ігор, розроблена американською компанією Unity Technologies. Unity дозволяє створювати додатки, що працюють на більш ніж 25 різних платформах, що включають персональні комп'ютери, ігрові консолі, мобільні пристрої, інтернет-додатки та інші.

Прямолінійний скелет — це спосіб подачі багатокутника його топологічним скелетом. Прямолінійний скелет подібний до певної міри серединним осях, але відрізняється тим, що складається з відрізків, в той час як серединні осі багатокутника можуть включати параболічні криві.

Полігóн (багатокýтник, многокýтник) — геометрична фігура, замкнена ламана (сама, або разом із точками, що лежать усередині). Вершини цієї ламаної називають вершинами многокутника, а відрізки ламаної — сторонами многокутника.

Тріангуляція (обчислювальна геометрії) — це розкладання полігональної області (простого многокутника) P на множину трикутників, тобто знаходження множини трикутників, які попарно не перетинаються і об'єднання яких дорівнює P.

Тріангуляцію можна розглядати як спеціальний випадок плоского прямолінійного графу. Коли немає дірок або доданих точок, тріангуляція утворює максимальний зовнішньопланарний граф.

Blender — пакет для створення тривимірної комп'ютерної графіки, що включає засоби моделювання, анімації, рендерінгу, після-обробки відео, а також створення відеоігор (цю можливість видалили у версії 2.80) .

UV розгортка — процес в 3D моделюванні, який полягає в накладанні двовимірного зображення на тривимірну модель. Літерами U і V позначають осі координат площини розгортки, оскільки літери X, Y і Z використовуються для позначення просторових координат.

AAA («три А», англійською вимовляється як «triple-A») — умовна підмножина відеоігор, що створюються й розповсюджуються середніми й великими видавництвами, що зазвичай мають багато коштів на розробку й рекламу. Чітких мірил належності певної гри до підмножини AAA немає, тож зазвичай ідеться про клас передових ігор з великим бюджетом, розробка яких пов'язана зі значним економічним ризиком і потребою високих показників збуту задля забезпечення прибутковості.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРОБЛЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ПРИМІЩЕНЬ

1.1 Загальні відомості про процедурне генерування

В цьому розділі буде йти мова про процедурне генерування контенту та його різновиди. Під контентом розуміється створення рівнів гри, карти ігрового світу, правил гри, текстур, сюжетів, предметів, квестів, музики, зброї, транспортних засобів, персонажів nf інше [2]. В даному контексті під іграми розуміються комп'ютерні ігри, відеоігри, настільні ігри, карткові ігри, головоломки та інші [2]. Ключовою особливістю створюваного контенту є те, що він повинен бути іграбельним — гравець повинен бути в змозі пройти створений рівень, мати можливість використовувати згенеровану зброю, піднятися по згенерованої сходах тощо [2].

Терміни «процедурний» і «генерація» відносяться до комп'ютерів, так як PCG має бути запущено у вигляді процедури на комп'ютері, яка видає на вихід щось для подальшого застосування. При цьому процедура є частиною чогось, наприклад всієї гри або інструменту геймдизайнера [2].

1.2 Приклади задач процедурної генерації

В якості характерних прикладів використання PCG називають: генерацію без участі людини, наприклад генерація підземель в пригодницьких іграх (таких як The Legend of Zelda), яка створює новий світ при кожному запуску. Також процедурну генерацію можна представити як систему, що створює нові типи озброєнь в грі космічного сеттінгу в залежності від дій гравця. Генерація збалансованої настільної гри теж підпадає під значення терміну процедурна генерація. Внутрішня процедура ігрового двигуну (швидко заповнює ігровий світ рослинами; інструмент, що дає можливість створювати карти для стратегічної гри, і при задані змін параметрів перераховує карту для її поліпшення, а також пропонує варіанти, що дозволяють зробити карту

більш збалансованою і цікавою. У той же час, простий редактор карт, штучний інтелект для настільної гри або інструмент інтеграції створеного контенту не відносяться до PCG [2].

Метою використання PCG може бути створення ігрового контенту без участі людини (що може бути як менш затратно, так і допомагати геймдизайнер у вирішенні їхніх завдань), розробка інших типів ігор (поліпшення показників різноманітності і реіграбельності), адаптація ігор під гравця «на льоту», поліпшення контенту за допомогою алгоритмічних рішень, а також формалізація геймдизайну як широку наукову задачу [2].

Перші широко відомі застосування PCG відносяться до початку 1980-х років, коли при обмежених ресурсах комп'ютерів можна було створювати великі і різноманітні світи. Характерними прикладами таких ігор є Rogue і Elite [2], скріншоти яких зображено на рисунках 1, 2.

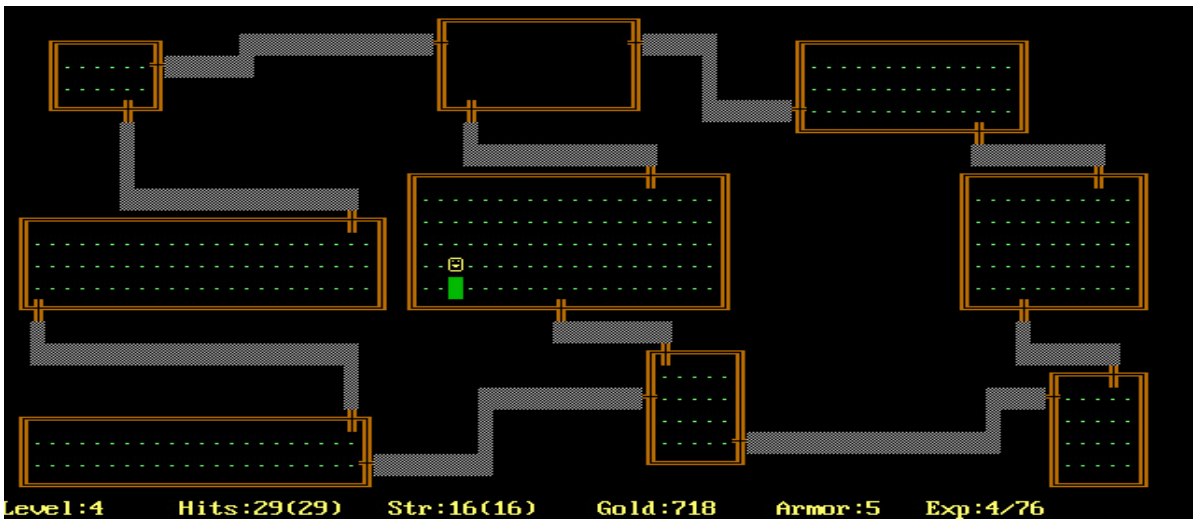

Рис. *1 Згенероване підземелля гри Rogue*

Рис. *2 Генерування випадкової системи зірок в грі Elite*

В подальшому PCG використовується в комерційних іграх, серед яких рольовий бойовик Diablo. В цій грі процедурна генерація застосовується для створення карт, числа монстрів та предметів [2], на рисунку 3 зображено процедурно згенероване підземелля.

Рис. *3 Гра Діабло*

В грі Spore розробники застосовували процедурну генерацію майже до всього, наприклад до генерування випадкових персонажів, сферичних планет, а головною особливістю цієї гри було створення процедурної анімації для істот [2]. Приклад процедурного генерування зображено на рисунку 4.

Рис. *4 Гра Spore з випадково згенерованими ворогами та анімаціями*

В серії ігор Civilization PCG використовується для створення карт [2], приклад згенерованої карти гри зображено на рисунку 5

Рис. *5 редактор карт в грі Civilization 1*

В грі Minecraft процедурна генерація використовується для створення ігрового світу [2]. Приклад згенерованого відкритого світу зображено на рисунку 6.

Рис. *6 Карта світу в Minecraft*

1.3 Використання процедурної генерації в розробці гри The Witcher 3

Один із сучасних прикладів процедурної генерації можна побачити у доповіді розробників компанії [17] CD Project Red, які займались розробкою The Witcher 3. Значною проблемою при розробці гри було створення великих відкритих просторів. У компанії CD Project Red на той час було недостатньо художників, які займались створенням відкритих просторів, що примусило розробників знаходити шляхи для рішення цієї проблеми. Одним із запропонованих рішень було створення інструментів для генерування ландшафтів та рослинності.

Програмісти розробили алгоритм, який може розпізнавати два типи поверхонь: природні та рукотворні. До природних система відносила ґрунт, пісок, каменисті елементи, а до рукотворних – цеглу, бруківку, дерев'яні поверхні. Це дозволило по-різному накладати ефекти на будь-які типи матеріалів, наприклад, на стіні дому сніг виглядав не так як на землі. За допомогою цього розробники програмного продукту змогли отримати плавні пере-

ходи між текстурою бруківки та природною поверхнею землі, так щоб можна було розгледіти контури деякої бруківки (Рис. 7).

Рис. *7 Перехід між текстурою землі та дороги у грі The Witcher 3*

Використання процедурної генерації дозволило розставляти по поверхні текстури невеликі об’єкти, такі як камені та траву. Художникам довелось намалювати лише десять видів трави, а алгоритм змінював їх колір під колір землі. Для цього левел-артисти використовували пігментну карту – текстуру локації з видом зверху в низькій роздільній здатності. Колір конкретного поля чи лугу накладався на траву градієнтом, щоб ближче до кореня рослинність була кольору землі, а вище зберігався оригінальний колір і не порушував відтінки плодів та суцвіть.

У грі «Відьмак» процедурно згенерована трава не перекривала вхідні текстури, а доповнювала їх. Для цього програму навчили виділяти ті місця, на яких може щось рости, наприклад якщо на старій дорозі є невелика кількість землі то в цьому місті може вирости трава. В процесі генерування художники мали змогу налаштовувати густину трави. Цей параметр працював на двох рівнях. З однієї сторони він визначав наскільки текстура трави перекривала землю. З другої — налаштовував саму кількість та висоту рослин-

ності. В залежності від цих параметрів художники могли створити як голу скалу з маленькими вкрапленнями трави, так і заповнений травою холом. Приклад генерування трави наведено на рисунку 8.

Рис. *8 Генерування трави в грі The Witcher 3*

Для створення дерев та кущів художники компанії CD Project Red використовували інший інструмент – пензлики з набором рослинності (Рис 9). Їх можна було налаштовувати під конкретну локацію, вказуючи насиченість ґрунту водою, кількість сонячних променів і навіть напрям розповсюдження насіння.

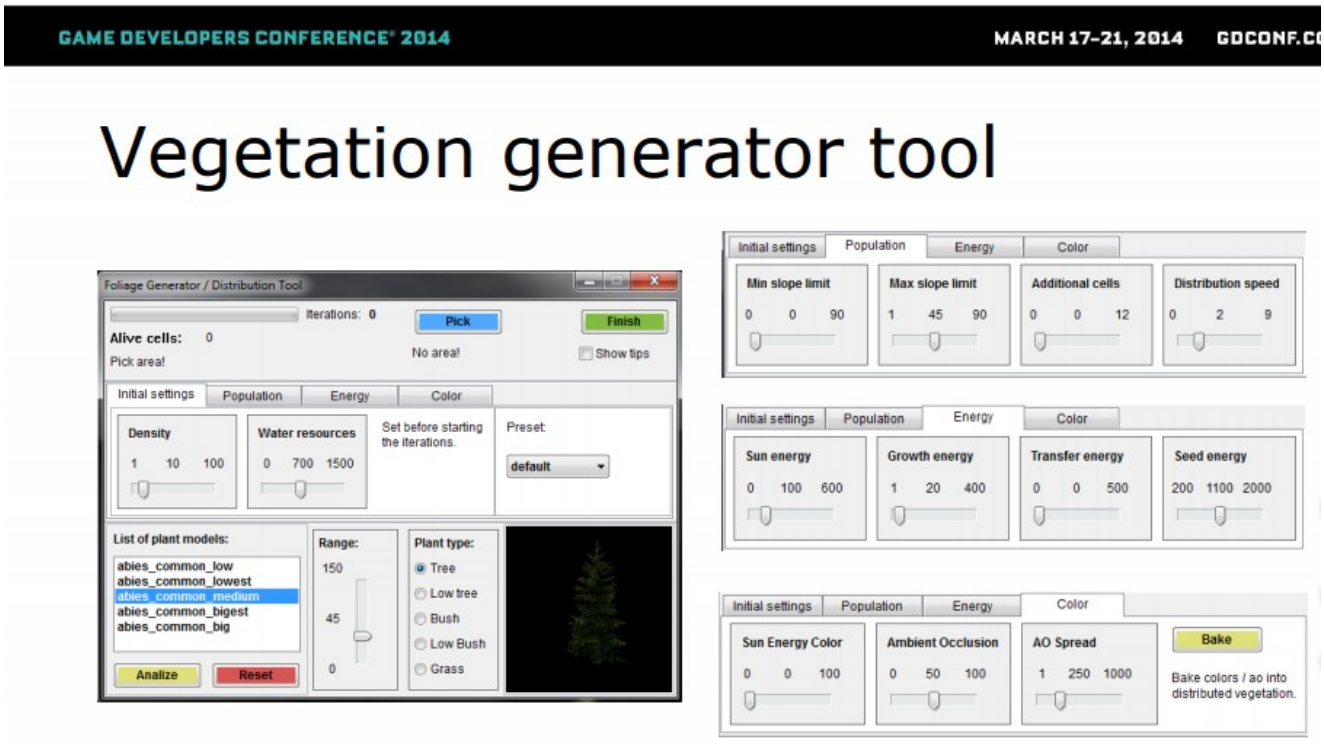

Рис. *9 Інструмент для створення рослинності у грі Witcher 3*

Алгоритми пензлів розпізнавали особливості рельєфу. Наприклад вони визначали, в яких місцях вода під час дощу стікає по схилах долин – там ліс виростав більш густим. В процесі система порівнювала рельєф з рельєфом сусідніх локацій, щоб визначити траєкторію сонця і наскільки територія освітлена.

Усі зібрані дані порівнювались з типами рослинності, які художники закладали до пензлів Набір дерев в пензлях південного лісу сильно відрізняється від пензлів для світлих гаїв на півдні, а ті в свою чергу, від рослинності на гірській місцевості.

Якщо для конкретного дерева не вистачало води або сонця, то пензель виключав цей тип із локації. Також якщо ресурсів було достатньо, об’єкти все одно порівнювалися з заданими «ідеальними умовами».

Також ці пензлі давали різні результати при повторному використанні. Якщо художник використовує цей пензлик на одній ділянці один раз, у нього виходить легкий підлісок: чагарник, одне стояче дерево. Але якщо повторити використання пензля, ефект стає більш вираженим – чагарники роз-

ростаються, а дерев стає більше та їх висота зростає. На рисунку 10 відображено скріншо з гри The Witcher 3.

Рис. *10 Скріншот гри The Witcher 3*

1.4 Шари для процедурних біомів у грі Horizon: Zero Dawn

Працюючи над грою Horizon Zero Dawn, програмісти компанії Guerrilla Games також використовували процедурну генерацію [18]. Для відтворення великого відкритого світу вони розробили алгоритм, який створював поверхні на основі шарів.

Для генерування місцевості використовувалися 4 чорно-білі текстури: карта топології, розміщення дерев, водних потоків та карта доріг. Топологія карти місцевості має вигляд дерева. На першому кроці між собою поєднувалися рельєф місцевості та розміщення дерев. На другому кроці отримана карта поєднувалась з водними руслами і на останньому етапі додавалися дороги. Відображення дерева побудови місцевості в грі Horizon Zero Down можна побачити на рисунку 11.

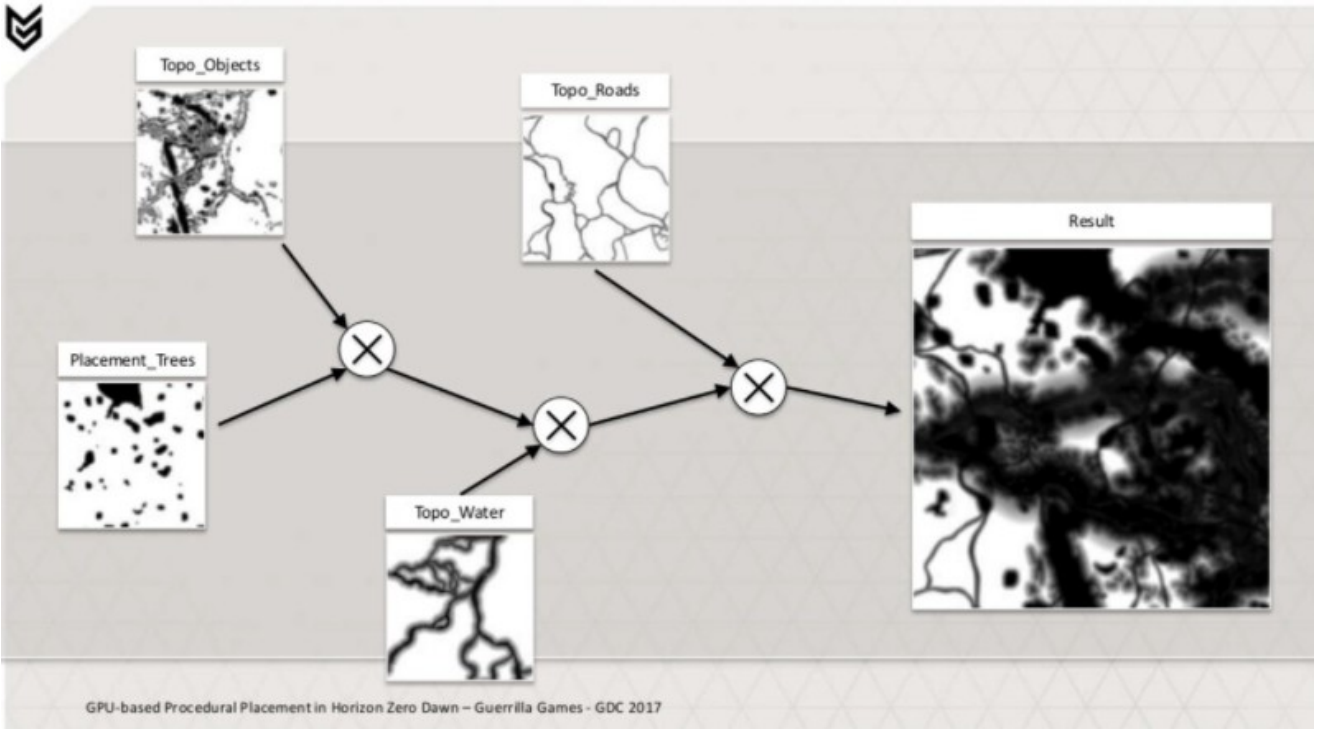

Рис. *11 Карта побудови ландшафту у грі Horizon: Zero Dawn*

Щоб згенеровані локації не просто працювали, але і виглядали різноманітними, художникам студії довелось створити багато асетів та шаблонів рослинності, але це все одно виявилось швидшим і вигіднішим, ніж повністю ручна робота художників. Три людини в Guerrila Games намалювали 500 типів рослин, а шаблонами займався один технічний художник.

Налаштувавши параметри генерації один раз, художники студії могли легко створювати визначені поєднання рослинності в різних частинах локацій. Алгоритм однаково успішно працював і з густими лісами, і з полями, чагарниками та травою.

Щоб моделі рослинності не заважали один одному, для кожної вказувався параметр footpoint, який задавав мінімальну дистанцію між об’єктами одного типу. Для того щоб результат не виглядав штучним, художники налаштовували параметри, які хаотично змінювали деякі характеристики та логіку розміщення.

В застосунку Horizon: Zero Dawn гравець досліджує багато різних біомів, від середньоєвропейських лісів до засніжених тундр. Самі розробники

назвали їх «екосистемами»: з точки зору інструменту це були просто набори налаштувань для створення локації.

Параметри біомів зачіпали не лише типи асетів, а й їх розповсюдження, погоду, візуальні ефекти і звукові ефекти. Тому результат генерування двох різних типів екосистем навіть на одній локації виглядав по-різному — джунглі на деякій горі сильно відрізнялися від звичайного лісу в тій же локації. На рисунку 12 відображено скріншот з гри Horizon: Zero Dawn.

Рис. *12 Скріншот гри Horizon: Zero Dawn*

1.5 Алгоритм для процедурних вулиць

У 2019 році компанія Insomniac Games розробила алгоритм процедурної генерації міста [19] для гри «Marvel’s Spider-Man». Загалом алгоритм в кінцевому варіанті створив 18 квадратних кілометрів віртуального Манхетену (Рис. 13). В Insomniac Games намагалися зробити так, щоб 80% роботи алгоритму попало в фінальну гру, і тому для створення алгоритму було витрачено багато ресурсів.

Розробники почали свою роботу над процедурним містом з розмітки. На першому етапі створення міста на карту наносилися основні вулиці, границі локацій і місця для сюжетних місій. На другому етапі використовувалась процедурна генерація: система сама розставляла будівлі і прикрашала їх деталями. На третьому етапі художники вручну допрацьовували деякі місця.

Процедурна генерація в грі Spider-Man використовувалась не тільки для створення вулиць і архітектури, їй також довірили створення реалістичного трафіку машин, пішоходів, звуків і різним дрібних деталей на локації.

З будівлями алгоритм процедурного генерування працював поетапно. Спочатку із простих форм (кубів, циліндрів) генерація розставляла прості заготовки уздовж вулиць, опираючись на задану художниками етажність і розміри. Потім, використовуючи асети для потрібного району чи типу будівлі, система розташовувала на оболонках зовнішній декор – вивіски, вікна і сходи.

Результат перевіряли художники і якщо комбінація елементів виглядала не природно, то вони переставляли їх вручну. Наприклад, дуже часто художники правили дахи, бо вони постійно попадали в поле зору гравця.

Рис. *13 Місто згенероване алгоритмом у грі Spider-Man*

З автомобільним рухом генерація працювала по-іншому – тут у алгоритму було більше свободи. Так як художники наносили на карту тільки самі впізнані проспекти та вулиці реального Нью-Йорка, створенням невеликих вулиць займався алгоритм генерування міста.

Алгоритм використовував задані художниками параметри: ширину доріг, кількість смуг, направлення руху. На їх основі він створював вулиці там, де це було потрібно з точки зору транспортної прохідності. При цьому тихі вулиці зіставалися тихими, а великі проспекти – шумними і завантаженими (Рис. 14).

Рис. *14 Згенерований алгоритмом вулиці та трафік в грі Spider-Man*

Генерація управляла не лише прокладанням вулиць, але й кількістю машин на них. Алгоритм враховував можливі об’їзди, правила паркування, фази світлофору та інше, щоб створити реалістичний автомобільний рух і реалістичні «пробки». Пішоходи теж були частиною цієї системи. Генерація розставляла їх так, що більшість прохожих здавалися практично статичними об’єктами – вони могли рухатися лише в невеликому радіусі від точки де з’явилися, тому на загальний трафік вони не впливали. Динамічні же перехо-

жі активно рухалися по місту і взаємодіяли з транспортом – пропускали машини, користувалися пішохідними переходами.

Процедурні алгоритми в Spider-Man вміли створювати не тільки статичні елементи, такі як каналізаційні люки, клумби і паркани. Використовуючи данні про пішоходів і транспорту, система створювала світлофори там, де рух був найбільш щільним, а ліхтарі там, де було темніше за все. Всі ці елементи включатись, виключались і активно взаємодіяли з оточенням.

Тісний зв'язок між різними системами генерації виявився і одним із проблем розробки Spider-Man, він створював ефект доміно. На різних етапах створення міста у геймдизайнерів були готові лише деякі механіки гри, а акробатика була додана частково. Сюжет гри також постійно доповнювався і переписувався, через це змінювалися важливі для історії локації.

Щоб звести ці випадки до мінімуму, tools-інженерам студії довелось переписувати алгоритм генерації. Новий варіант працював більш модульно і слабкіше зв’язував різні елементи системи. Помінявши той самий провулок в новій системі, художники могли зіштовхнутися з зміненим плануванням кварталу, а не цілого району.

Також розробникам прийшлось продумати і змінити сам пайплайн створення міста. Художника Insomniac Games, які відповідали за створення частини локації міста для основного сюжету гри потрібно було мати впевненість в тому, що їх зроблена робота не буде знищена при наступному використанні алгоритмів генерування міста, тому програмісти які займалися розробкою алгоритмів генерування додали можливість заморожувати деякі квартали та будівлі міста. Вони регулярно додавали нові елементи, текстури, і декоративні об’єкти, котрі генерація підхоплювала і розповсюджувала по локації.

Щоб створена вручну частина локації не зникала через побочні зміни, її «заморожували» — після того, як художники і дизайнери вносили всі необхідні правки, алгоритму заборонявся доступ до конкретної частини міста. Причому разом з будинком, «заморожувалися» і найближчі до нього тротуари та

вулиці, щоб майбутні зміни не створювали нових конфліктів. На рисунку 15 зображений реальний скріншот з гри.

Рис. *15 Скріншот з гри Spider-Man*

1.6 Цільові властивості генерації

Створюваний контент повинен задовольняти певним умовам та вирішувати відповідні проблеми. На практиці найчастіше розглядаються наступні властивості:

- *Швидкість*: в залежності від завдання вимоги різняться від мілісекунд до місяців, але в загальному випадку контент повинен бути створений вчасно для задоволення потреб ігрового процесу.
- *Надійність*: деякі генератори створюють «купу чогось», в той час як інші можуть гарантувати виконання заданих критеріїв, наприклад завжди забезпечувати можливість прохідності гравцем до виходу лабіринту.
- *Контролепригідність*: здатність контролювати згенерований контент виходячи з ситуації і надання геймдизайнер відповідної

волі (ця властивість далеко не завжди потрібна); наприклад, генерація гладкого довгастого каменю або створення рівня з певною атмосферою.

- *Різноманітність*: створення такого контенту, який був би якомога різним на різних запусках, і при погляді на нього гравець не відчував одноманітність.
- *Креативність і правдоподібність*: генерація такого контенту, який виглядає так, як ніби його створила людина, а не генератор.

1.7 Методи процедурної генерації будівель

Магістерська робота Олександра Дахла «Procedural Generation of Indoor Environments» [5] посвячена процедурній генерації плану приміщення. Перше на чому акцентує увагу Олександр – це вхідні дані. В його роботі до вхідних даних входять: точки зовнішніх контурів приміщення, розташування вікон, розміри вікон та висота поверху приміщення. Друге, на чому він акцентував увагу – це генерація головного коридору приміщення за допомогою алгоритму Straight skeleton.

Після створення ділянки коридору, яка розділила будинок на регіони (багатокутники, які відділені коридором), після чого всі ці регіони поділяються на під регіони (кімнати), і на останок відбувається розстановка вікон та дверей. Приклад згенерованого приміщення можна побачити на рисунку 16.

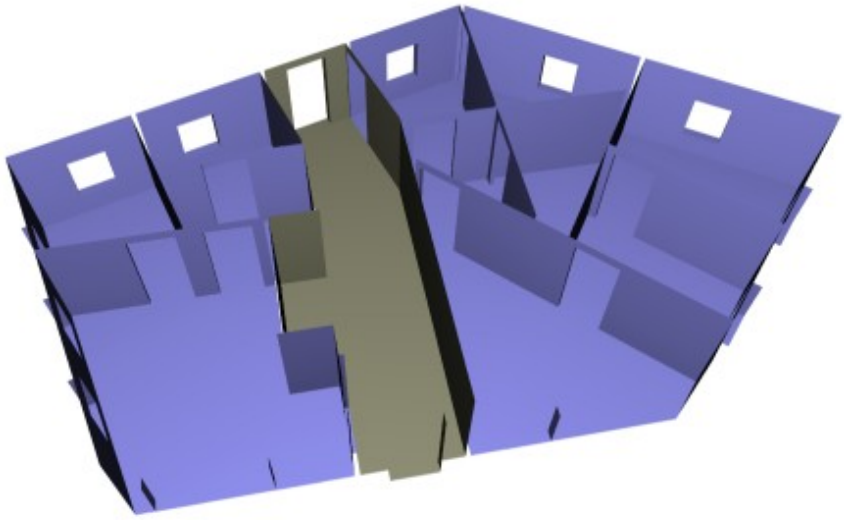

Рис. *16 3D план згенерованої будівлі*

Andersson S. автор магістерської роботи «Detailed Procedurally Generated Buildings» [20] спробував реалізувати процедурну генерацію приміщення (екстер'єр) разом з створенням плану приміщення з внутрішнім інтер'єром та розстановкою предметів в будівлі. Основні відмінності від схожих робіт в тому, що він виділяє життєво важливі види кімнат (такі як кухня, спальня та інше) і життєвої важливі види предметів для кімнати (такі як ліжко в спальні, обідні стіл у кухні та інше). Коридор для приміщення генерується коли в ньому є замкнуті кімнати, і він служить для того, щоб з кожної кімнати був вихід. Дверні проходи додаються шляхом знаходження сусідніх стін між двома кімнати.

Автор статті «Procedural Modeling of Buildings Composed of Arbitrarily —Shaped Floor-Plans: Background, Progress, Contributions and Challenges of a Methodology Oriented to Cultural Heritage» [21] Pádua L. провів аналіз існуючих підходів віртуальної візуалізації основаних на ручному відображенні і процедурному — за характеристиками адекватності, вимогам і рівню автоматизації. Основний висновок такий, що процедурний підхід найбільш перспективний в плані автоматизації, але має недоліки в плані деталізації. Автор проінспектував всі найвідоміші алгоритми і підходи використання процедурної генерації споруд.

Kužel V. автор магістерської роботи «Procedural building reconstruction from building outlines» [22] представив метод швидкої процедурної генерації правдоподібних будівель з їх контурів. Існує кілька методів, що розрізняються за швидкістю і кількістю деталей в результатах. Всі ці методи можуть бути використані в різних випадках. Користувач може віддати перевагу швидкості або більш привабливим результатам. Метод, представлений у цій роботі, вважає за краще швидкість і напівавтоматичний підхід. Він приймає контури будівлі в якості вхідних даних разом з деякими параметрами які задає користувач, такими як висота. Програмна реалізація генерування будівлі автора створює будівлю на основі контуру з точок. В цій роботі головний акцент був зроблений на створення різних видів фасадів, а особливо дахів будівлі. Автор спробував охопити всі відомі види дахів будівель. Для створення каскадного даху був застосований алгоритм Straight skeleton [10]. Приклад згенерованої будівлі з дахом зображена на рисунку 17.

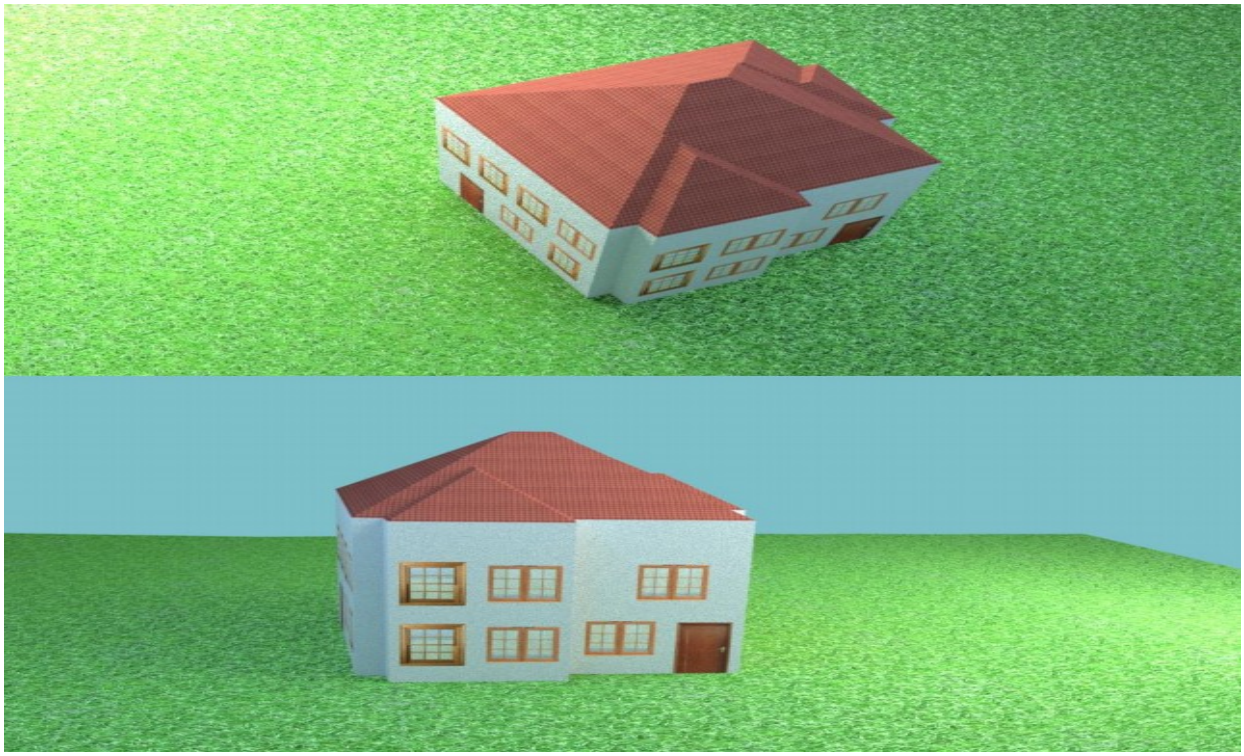

Рис. *17 Приклад будівлі з каскадним дахом*

Автор «A method for growth-based procedural floor plan generation» [6] Camozzato D. спробував реалізувати генерацію плану приміщення. Вхідні

дані його програми це контури будівлі і виходи з неї (вікна, двері). Після введення вхідних даних автор на основі контурів будівлі створює подобу сітки, для розміщення кімнат. Далі він вибирає осередки в яких присутні виходи назовні і поміщає в них кімнати, після чого ітеративно збільшує кімнати на одну клітинку. Також він використовує спеціальну оцінку осередків для вибору оптимального осередку для даної кімнаті. В кінці для отримання доступу до всіх кімнатах створюється коридор. Приклад згенерованого приміщення будівлі зображено на рисунку 18.

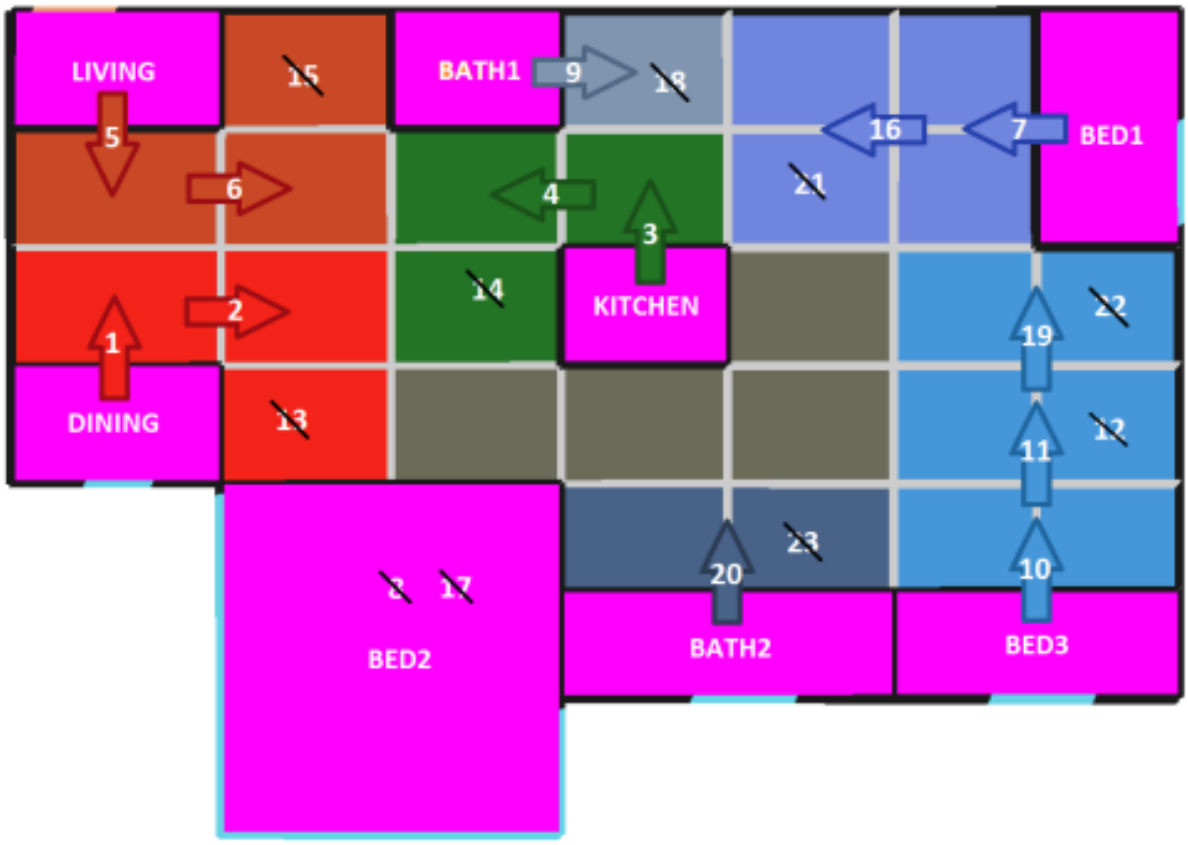

Рис. *18 Приклад створенного плану приміщення*

Автори доповіді «Real-time Procedural Generation of Personalized Façade and Interior Appearances Based on Semantics» [23] Da Silveira, D. Camozzato, F. Marson представили обчислювальну модель для процедурної генерації фасадних і внутрішніх стилів будівель для використання в тривимірних віртуальних середовищах ігор та симуляцій.

Модель використовує два типи вхідної інформації: геометричну і семантичну. Геометрична інформація відноситься до двовимірного плану бу-

дівлі, його частин і розмірами, а також позиціях для дверей і вікон. Семантична інформація дозволяє створювати архітектурні стилі, пропонуючи варіанти матеріалів і текстур для фасаду та внутрішніх деталей, а також для форм і розмірів дверей та вікон. Зміна одного або декількох вхідних параметрів змінює остаточний вид результату. Запропонована обчислювальна модель може використовуватися для створення великих віртуальних середовищ, оскільки вона дозволяє змішувати різні плани поверхів і архітектурні стилі для досягнення візуального розмаїття. Основними характеристиками роботи є процедурна генерація тривимірних будівель в реальному часі, індивідуалізація фасадів та інтер'єрів будівель, а також використання семантики для додання значення різних елементів будинку. Приклади згенерованих будівель зображені на рисунку 19.

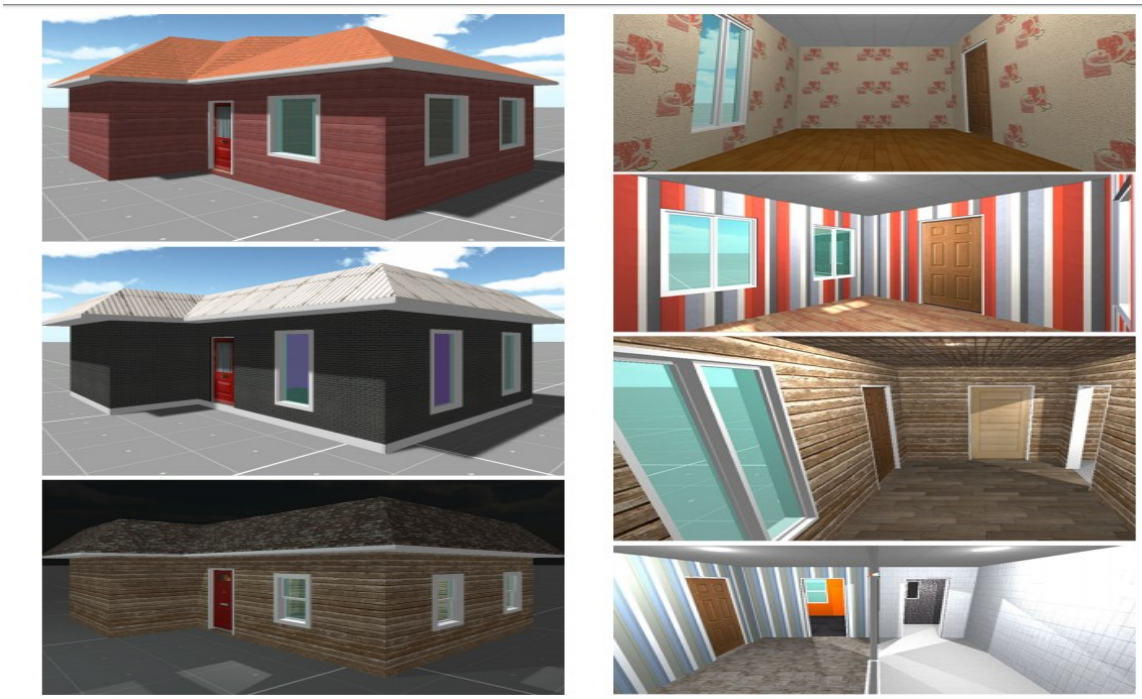

Рис. *19 Приклад будівель*

Автор роботи «Towards the Procedural Generation of Urban Building Interiors» [24] Bradley B. описав процес створення системи для побудови плану приміщень з прямими кутами. Власне вхідні дані подаються у вигляді полігону (контуру приміщення), далі створюється прямокутна сітка в даному

контурі. Після створюються зв’язки між осередками сітки. Далі створюються кімнати, а наступний етап це створення вже трьох вимірного представлення приміщення. Останній етап — створення вікон і дверей між кімнатами. Основна перевага розробленого рішення в тому, що немає необхідності в введенні вікон і дверей. Процес створення будівлі зображений на рисунку 20.

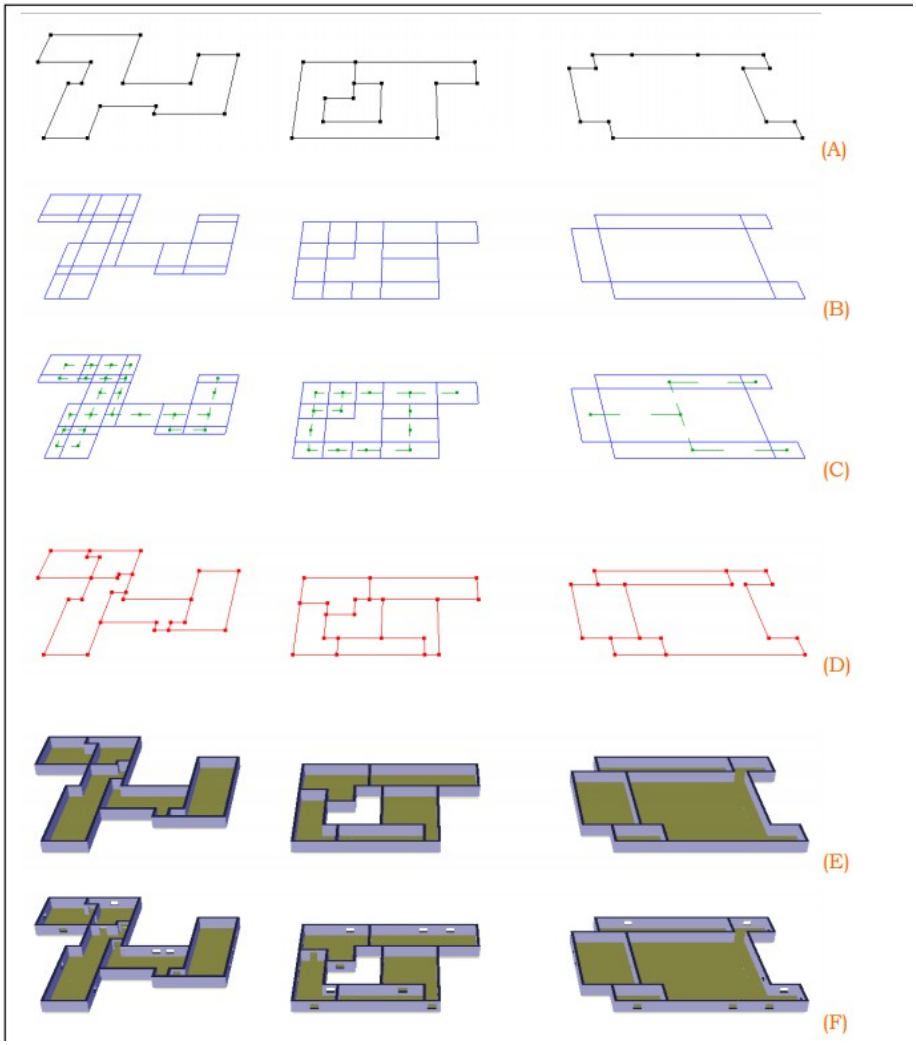

Рис. *20 Приклад плану приміщення*

Автори статті «Rule-based layout solving and its application to procedural interior generation» [25] T.Tutenel, R. Bidarra, R. M. Smelik, Klaas Jan de Kraker розповідають, яким чином можна достовірно розставити предмети в приміщенні. Вся суть зводитися до того, що існують деякі закономірності в розстановці предметів в приміщені. Тобто журнальний столик зазвичай розташований між телевізором і диваном, а диван спрямований на телевізор.

Основне завдання генерування інтер’єру в приміщенні, це рівномірний розподіл місця в приміщенні і канонічний вид цих розташованих предметів в порівнянні з реальним світом. Приклад згенерованого інтер’єру приміщення зображений на рисунку 21.

Рис. *21 Приклад интерьеру приміщення*

В роботі розробників компанії Procedia Technology «Procedural Generation of Traversable Buildings Outlined by Arbitrary Convex Shapes» T.Adão, F. Pereiraa, E. Peresb, L. Magalhãesb представлена система процедурного моделювання [26], яка створює тривимірні будівлі обмежені довільний опуклий багатокутник. Система отримує в якості вхідних даних XML на основі онтології, також відомий як XML4BD, який містить визначення певного будівлі, включаючи пов'язані з ним частини будівлі, поверхи, кімнати і переходи. Файл проходить через модератора XML, який читає і пере-

віряє структуру XML4BD. Наведений модератор також завантажує структуру класів на основі онтології і виконує кілька тестів, щоб забезпечити узгодженість, пов'язану з планом поверху (наприклад, усунення переходів між кімнатами, у яких немає загальної стіни). Цей модуль перебирає структуру будівлі для створення тривимірної моделі. Випробування системи продемонстрували універсальність цього інструменту при створенні будівель з декількома конфігураціями і топологями. Результати випробувань довели, що цей інструмент ефективний у виробництві будівель, обмежених чисто опуклим форми. Крім того, можна також включити в будівлю спеціальні приміщення з внутрішніми структурами, такими як басейни або сади з колонами.

РОЗДІЛ 2 ДОСЛІДЖЕННЯ ЗАСОБІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ПРИМІЩЕНЬ ТА НЕОБХІДНИХ ЕЛЕМЕНТІВ

2.1 Основні методи процедурної генерації приміщень

В цьому розділі буде глибокий розбір розроблених методів процедурної генерації приміщень та їх особливостей і недоліків.

Метод процедурного формування внутрішнього середовища будівлі

Перший метод процедурної генерації приміщення був розроблений в магістерській роботі Олександра Дахла «Procedural Generation of Indoor Environments» [5].

Перше з чого почав доповідати автор – це з вхідних даних, які приймає програмна система:

- форма будівлі (у вигляді точок полігона);
- позиції та розміри вікон та дверей;
- висота поверху майбутньої будівлі;
- товщина зовнішньої стіни.

Наступний крок – це побудова Building skeleton (видозмінений алгоритм прямолінійного скелету [13,16] . Його суть в тому, що маючи зовнішні точки будівлі їх можна стягувати всередину будівлі і отримати структуру схожу на форму даху – це канонічний прямолінійний скелет, а Building skeleton – це видозмінений варіант Олександра Дахла в якому будується коридор будівлі замість форми даху. Процес створення Building skeleton зображений на рисунках 22-23. На наступному етапі додаються входи в будівлю (Рис. 24), а потім коридор огортає стінами (Рис. 25). Також розроблена програна система може виділяти місце під сходи при необхідності (Рис. 26).

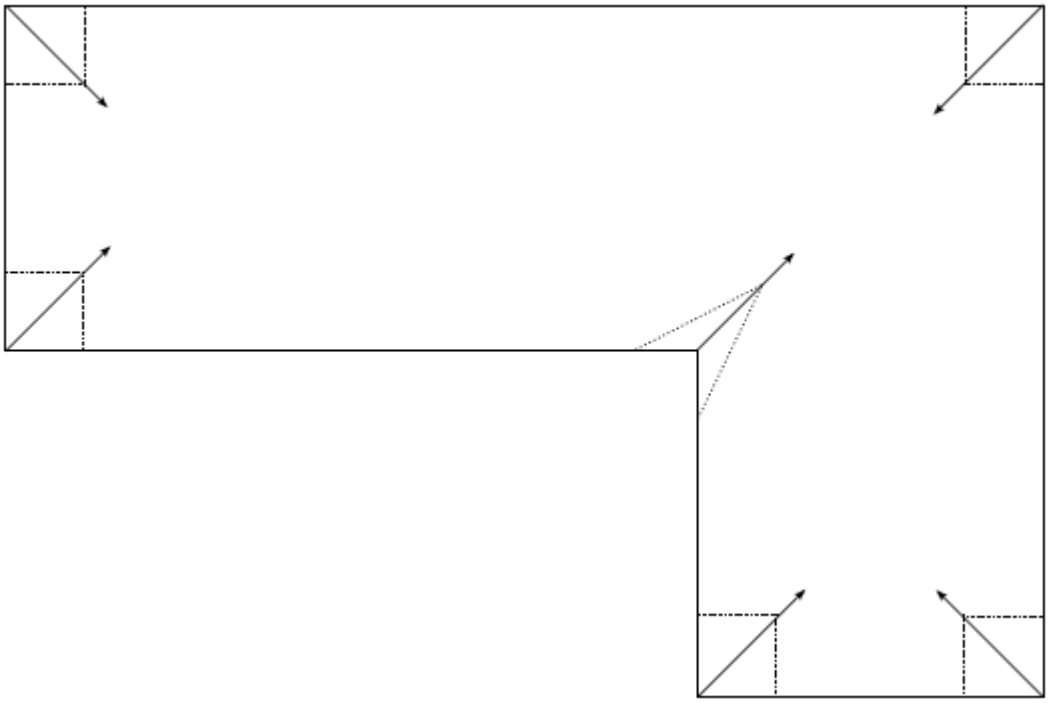

Рис. *22 Початкові краї скелета з позначеними стінками.*

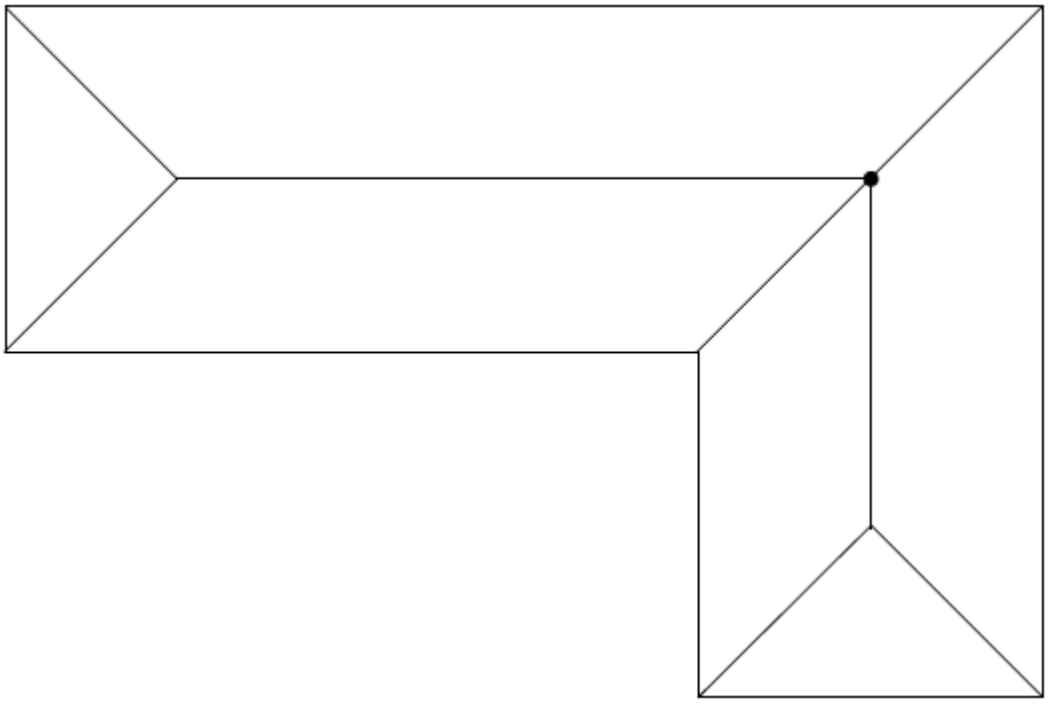

Рис. *23 Скелет з позначеним коренем (до того, як двері були прикріплені).*

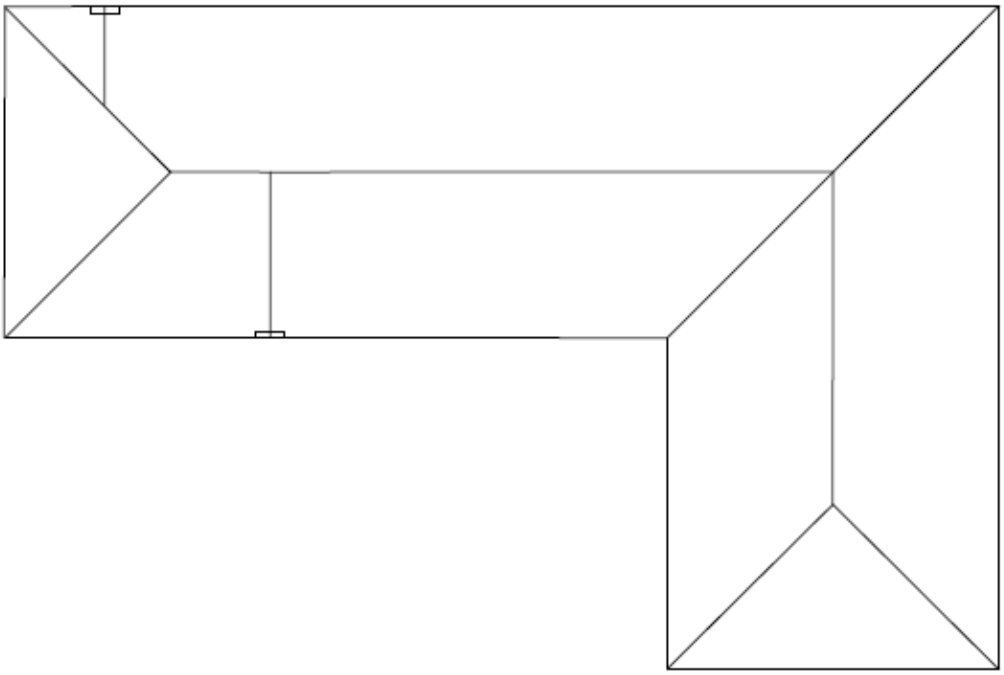

Рис. *24 Скелет з прикріпленими дверима*

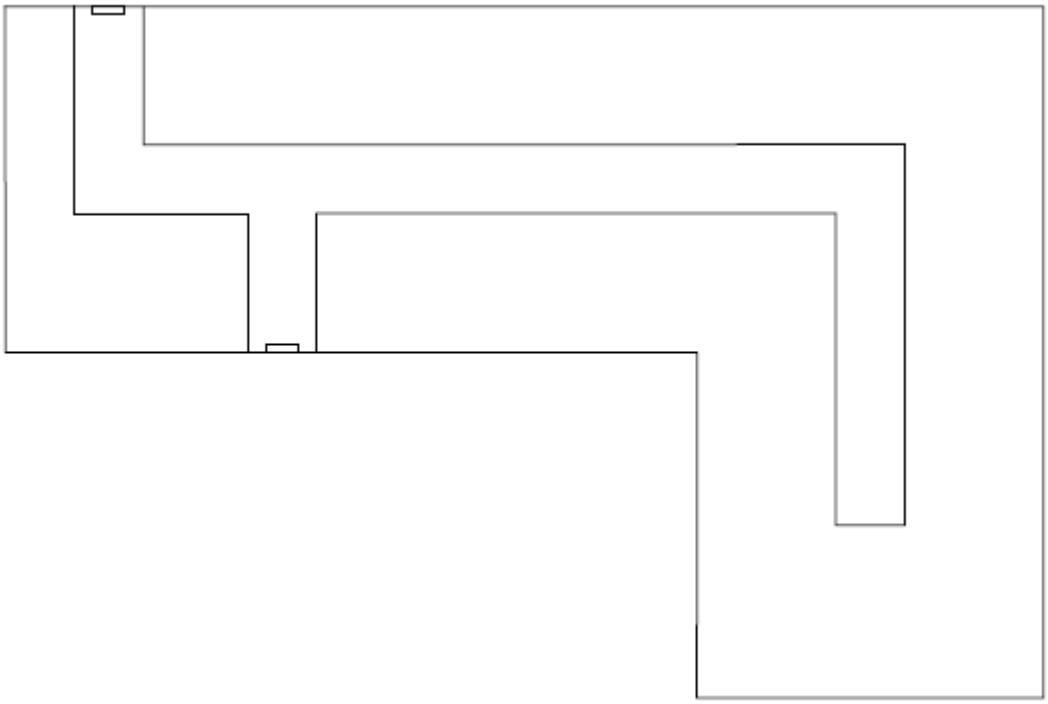

Рис. *25 Скелет після огортання ребер скелету стінами*

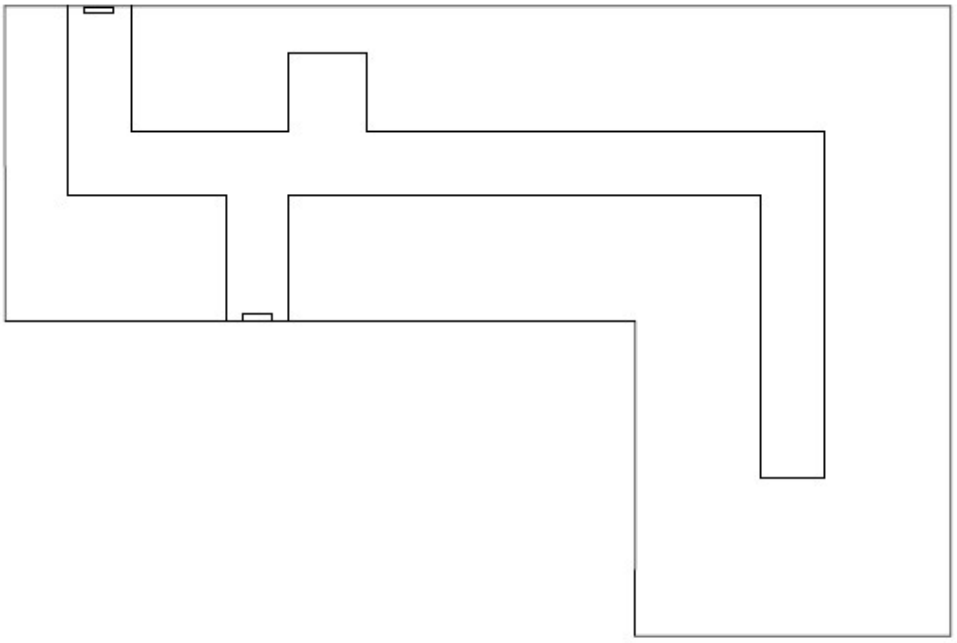

Рис. *26 Скелет після виділення місця під східці*

На наступному етапі автор виділяє так звані регіони будівлі, які можна визначити після попереднього кроку (Рис. 27).

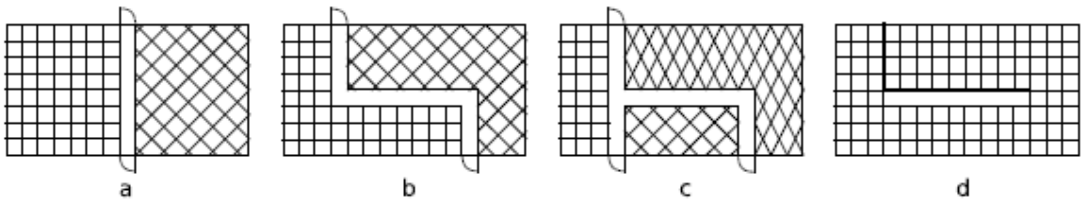

Рис. *27 Варіанти регіонів будівлі. a,b — будівля з двома регіонами; c — будівля з трьома регіонами; d — будівля з одним регіоном;*

В подальшому йде робота з цими знайденими регіонами. На цьому кроці створюються кімнати за допомогою діаграми Вороного та S-Space простору. S-Space простір створено нанесенням перпендикулярних ліній між вікнами та внутрішніми кутами будівлі. Сукупність цих ліній буде утворювати сітку, яка розділить будівлю на прямокутні частини. Зображення цих ліній можна побачити на рисунку 32 у вигляді пунктирних ліній. Початкові точки алгоритму вороного наносяться на вікна, двері та на внутрішні кути будівлі, візуалізація

процесу нанесення точок зображено на рисунку 28. Після нанесення точок входу алгоритму Вороного виконується перший прохід алгоритму, який відображений на рисунку 29-30, де символ ⊗ — це точка для якої знайдена область Вороного. На рисунку 31 зображено всі знайдені області алгоритмом Вороного.

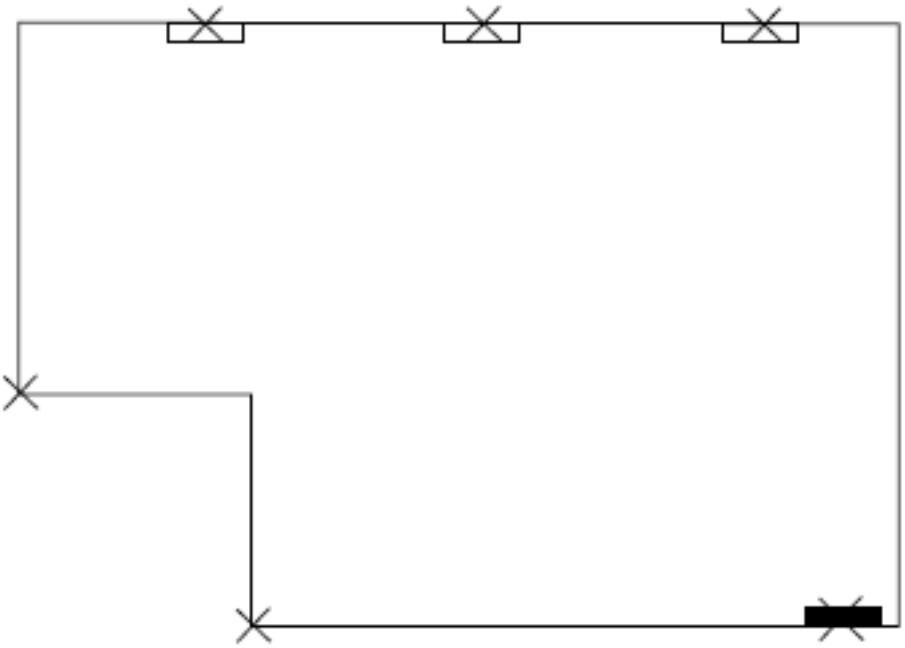

Рис. *28 Зразок апартаментів з початковими точками алгоритму Вороного*

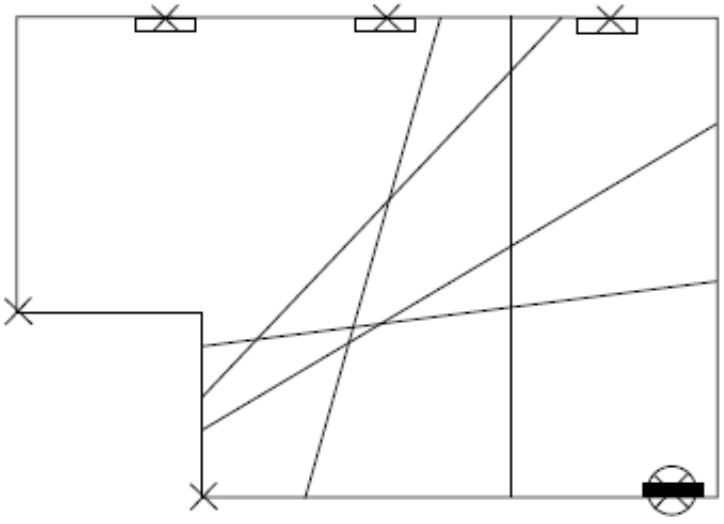

Рис. *29 Лінії зрізу діаграми Вороного для першого кроку*

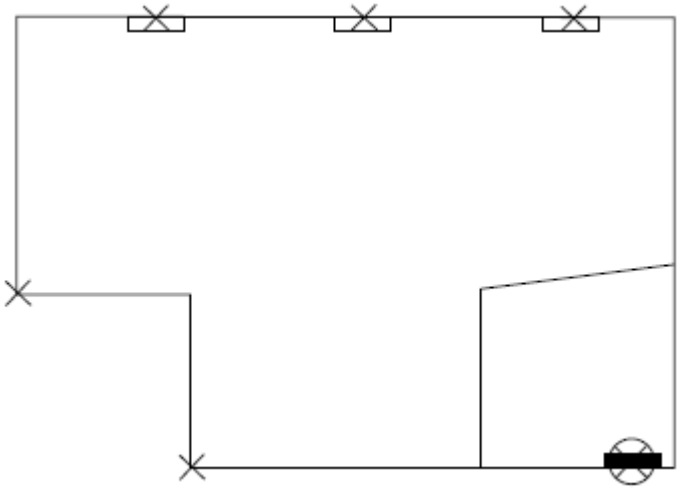

Рис. *30 Перша область діаграми Вороного*

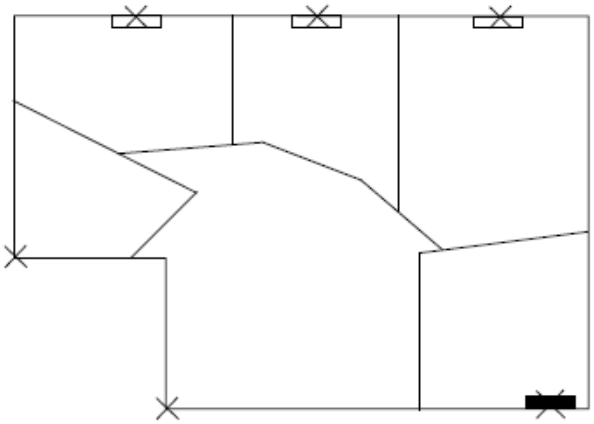

Рис. *31 Усі області діаграми Вороного*

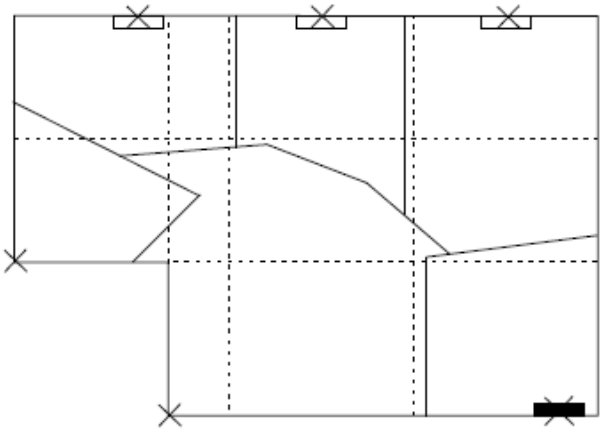

Рис. *32 Області діаграми Вороного та скелет S-Space*

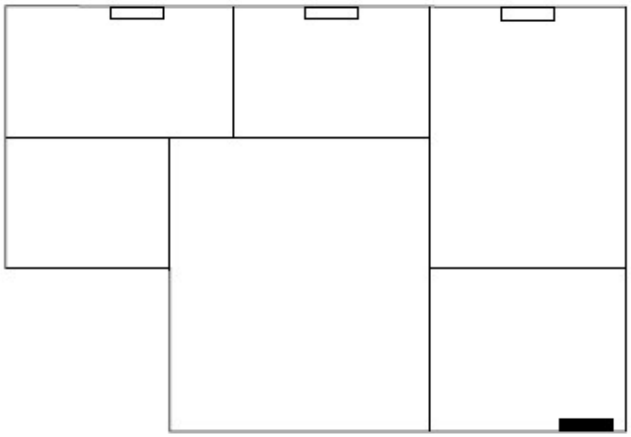

Рис. *33 Створенні кімнати регіону*

Далі після створення кімнат йде етап додавання дверей та вікон. Основна ідея полягає в тому, що в кімнатах сусідніх з коридором знаходяться можливі позиції дверей і за допомогою генерації випадкових чисел обираються відповідні позиції дверей. Вікна додаються на кожній стіні будівлі з деяким кроком. Додавання вікон і дверей можна побачити на рисунках 34, 35.

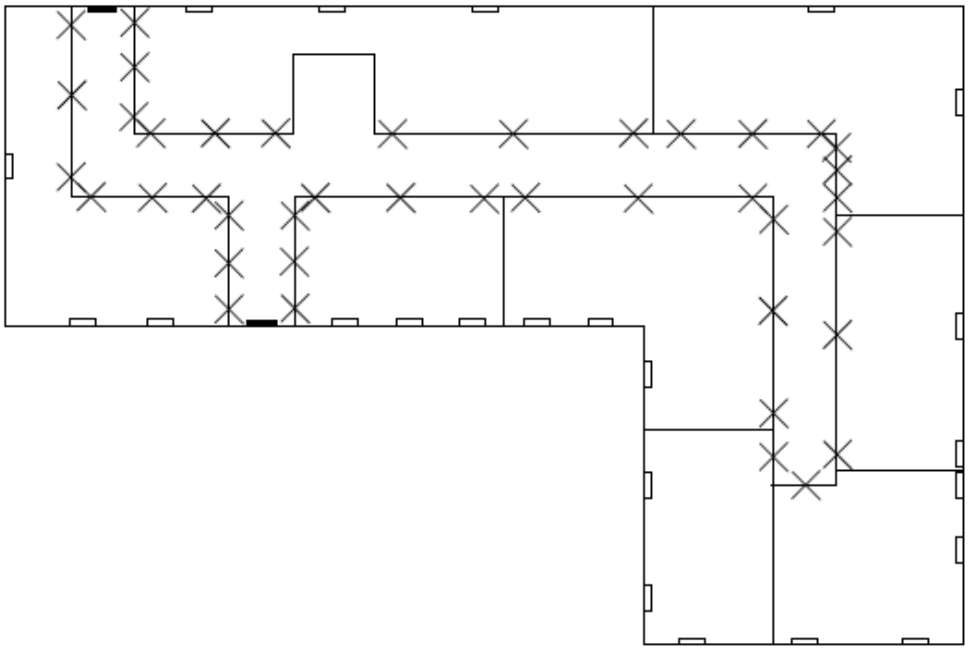

Рис. *34 Позначені субрегіони з можливими положеннями дверей кімнат*

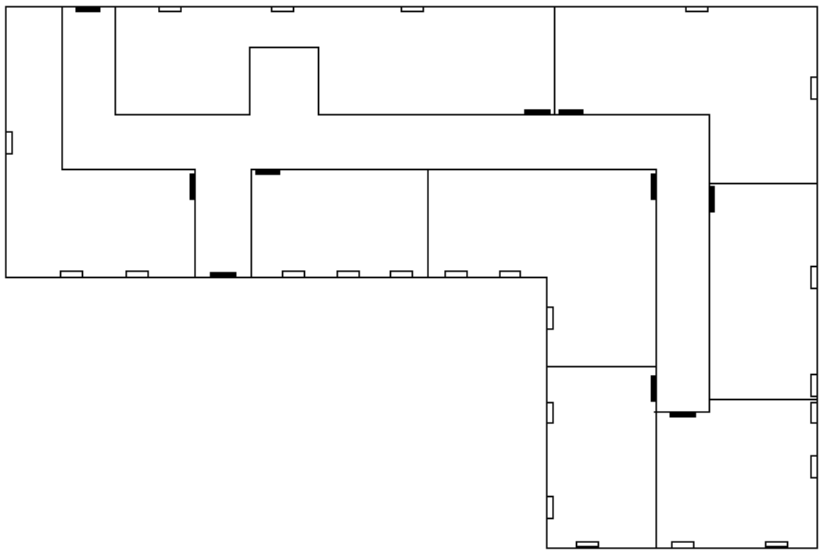

Рис. *35 Будинок із створеними апартаментами*

Процедурне вирощування плану приміщення

Метод процедурного вирощування плану приміщення будівлі [6] орієнтований на розширенні кімнат в будівлі. Тобто система приймає деяку форму будівлі у вигляді точок полігона. Далі з нього алгоритм створює щось подібне до архітектурної сітки, який поділяєте будівлю на клітинки деякого розміру (це будуть початкові клітинки для заповнення їх кімнатами). На наступному кроці алгоритм за допомогою деяких оцінок sw_y ss_y sp_y se_y sh_y sd_y sk_y створених клітинок обирає для кожної кімнати найбільш оптимальну. Усі параметри які використовуються в системі наведено у таблиці 1. Після обрання клітинок для всіх кімнат починається етап вирощування, де за допомогою вже нових оцінок йде розширення кімнат в одному з чотирьох напрямів: угору, вниз, вліво, вправо.

Вхідні данні:

- форма будівлі у вигляді полігону;
- розташування вікон та дверей;
- реквізити кімнати, що надаються користувачем;
- реквізити приєднання кімнат (якщо потрібно).

Таблиця 1 Параметри генерування приміщень

Умовні позначення	Опис умовних позначень
K_o	Оцінка розширення, яка орієнтована на R_R
kb_o	Правило заблокованого напряму, частина оцінки розширення кімнати K_o
kn_o	Правило необхідності, частина оцінки розширення K_o
kw_o	Правило витраченого вікна, частина оцінки розширення K_o
o	Параметр розширення (вгору, вниз, ліворуч праворуч)
P_o	Ймовірність вибору варіанту розширення *o*
P_y	Ймовірність вибору відповідної комірки *y*
R_C	Реквізити з'єднання кімнат
R_R	Реквізити кімнат
S_y	Оцінка підходящої клітинки *y*
sd_y	Правило вхідних дверей, частина оцінки S_y
se_y	Правило еластичного з'єднання, частина оцінки S_y
sh_y	Правило статичного важкого з'єднання, частина оцінки S_y
sk_y	Правило малого вікна, частина оцінки S_y
sp_y	Правило приватної кімнати, частина оцінки S_y
ss_y	Правило соціальної кімнати, частина оцінки S_y
sw_y	Правило вікна, частина оцінки S_y
t_D	Поріг відстані для прийнятних кліток
t_W	Поріг довжини стіни
y	Придатна клітинка, де можна розмістити кімнату

Усі реквізити кімнат задаються у файлі XML. Налаштування кімнати наведено у лістингу 1.

Лістинг 1 Приклад налштування кімнату у XML

```
<Room name="BEDR1" type="Private" priority="10">
     <Width min="2.0" max="5.0" />
     <Depth min="2.0" max="5.0" />
     <Area min ="6.0" max="20.0" />
```

```
    <Window required="yes" />
</Room>
```

У лістингу 2 зображено налаштування з’єднання між кімнатами. Реквізити кімнати мають наступні параметри:

- тип кімнати (соціальний, приватний, сервісний);
- мінімальна/максимальна довжина та ширина, а також площа;
- чи необхідне вікно в кімнаті;
- пріоритет, щоб визначити порядок розміщення та розширення кімнат.

Лістингу 2 Приклад налаштування для з’єднання між кімнатами.

```
<ConnectionRequisites>
    <Connection name="BATH2,BEDR1" type="Door">
        <Link a="BEDR1" b="BATH2" />
    </Connection>
    <Connection name="DIN1,KITC1" type="OpenWall">
        <Link a="DIN1" b="KITC1" />
    </Connection>
</ConnectionRequisites>
```

На наступному етапі алгоритм вже має реквізити кімнат та вхідні данні. Цей етап будує сітку будівлі для подальшого вирощування кімнат. На рисунку 36 зображено вигляд сітки для побудови будівлі.

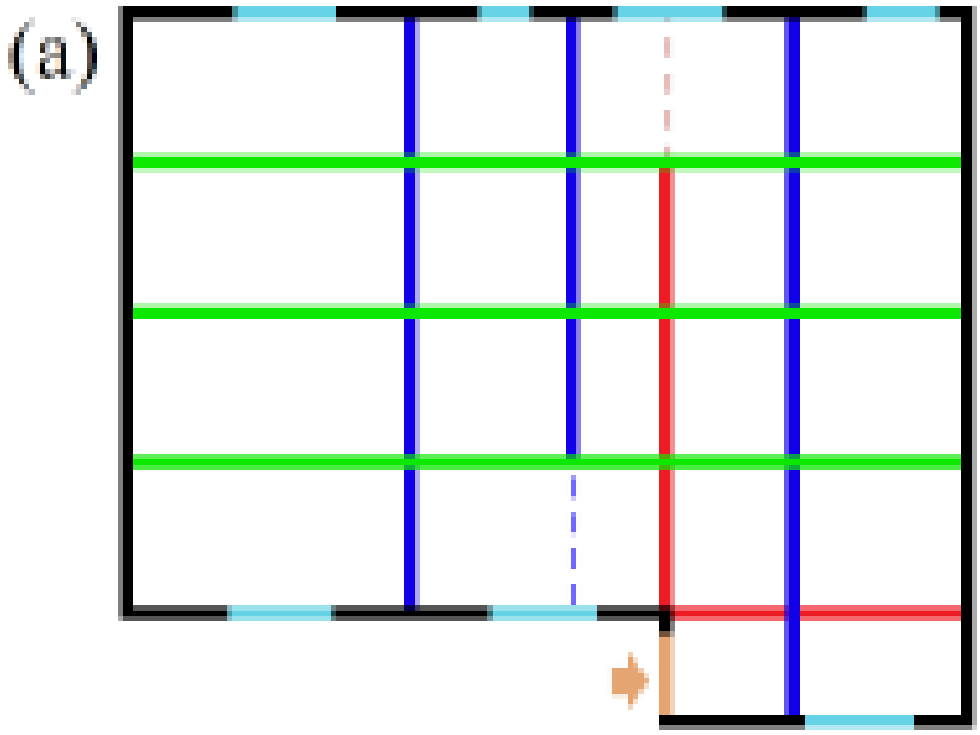

Рис. *36 Створення архітектурної сітки*

Правила створення сітки:

- якщо є два вікна, то додається перпендикулярна пряма між ними (синій колір);
- якщо вікон поруч немає, то додається перпендикулярна пряма з кроком (зелений колір);
- якщо при подовженні стіни маємо перетин сегментів стін, то додається подовжений сегмент до сітки.

Наступний етап створення будинку — це знаходження відповідних клітинок для заповнення їх кімнатами (Рис 38). По-перше, обираються клітинки, які мають вікна та двері. По-друге – клітинки, які рівновіддалені від цих клітинок на однакову задану відстань. На рисунку 38 рожеві клітинки – це ті, що містять вікна та двері, додаткові зелені клітинки, що відповідають вимогам,

вони вираховуються завдяки параметру t_D, що означає прийнятну відстань між іншими клітинами.

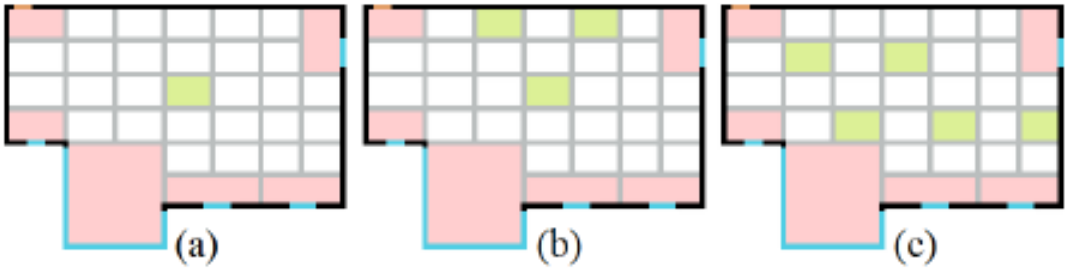

Рис. *38 Комірки для створення кімнат; a — відстань t_D дорівнює 3 метри між клітинками; b — відстань t_D дорівнює 2.5 метри між клітинками; c — відстань t_D дорівнює 2 метри між клітинками*

Після обрання підходящих кандидатів-клітинок вираховуються оптимальні клітинки для усіх кімнат за формулою:

$$S_{y=¿\, sw_y * ss_y * sp_y * se_y * sh_y * sd_y * sk_y, ¿}$$

де умовні позначення sw_y, ssy, sp_y, se_y, sh_y, sd_y, sk_y варіюються між 0 та 1. Для кожної змінної є своє правило, яке вираховує оцінку. Деякі з них застосовуються лише за певних умов, наприклад клітинку оцінюють за приватним правилом, яке впливає лише на реквізити кімнати R_R приватного типу. Коли правило не застосовується його асоційований бал вважається рівним 1.

Правило вікна вираховується за умовами, зображеними в формулі:

$$sw_y = \begin{cases} 1, \textit{якщо вікно присутнє в } y, \\ ¿\, 0.5, \textit{якщо вікно не потрібно за } R_R, \textit{але воно наявне в } y, \\ ¿\, 0, \textit{інакше} \left(\textit{якщо вікно потрібне за } R_R, \textit{але воно відсутнє}\right) \end{cases}.$$

Візуальне відображення роботи цього правила зображено на рисунку 39.

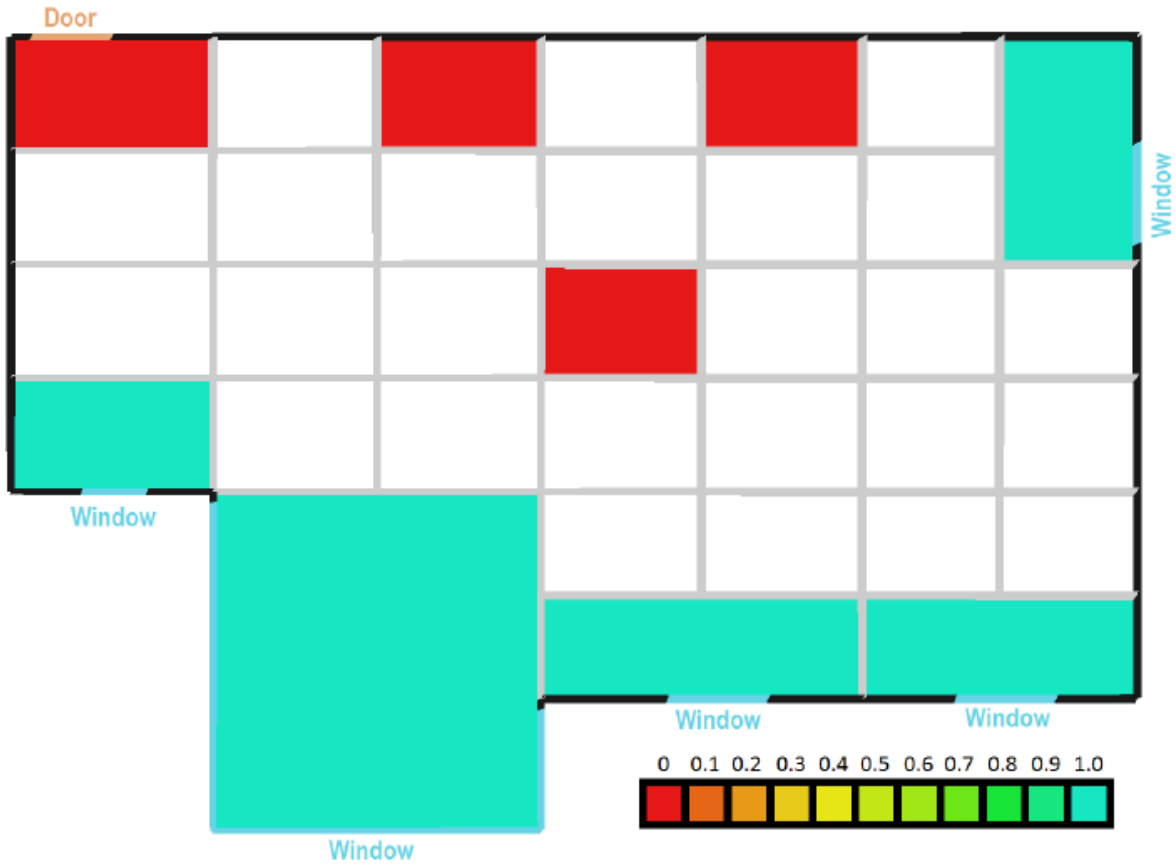

Рис. *39 Візуалізація правила вікна*

Соціальне правило вираховується за формулою:

$$SS_y = 1 - \frac{dist(cell_y, cell_D)}{max_i(dist(cell_i cell_D))},$$

де *dist (celly, cellD)* – дистанція між обраною клітинкою і клітинкою з вхідними дверима, *maxi (dist (celli, cellD))* – максимальна дистанція між деякою клітинкою та вхідними дверима. У випадку цього правила соціальні кімнати будуть розташовуватись ближче до виходу. Візуалізація соціального правила зображене на рисунку 40.

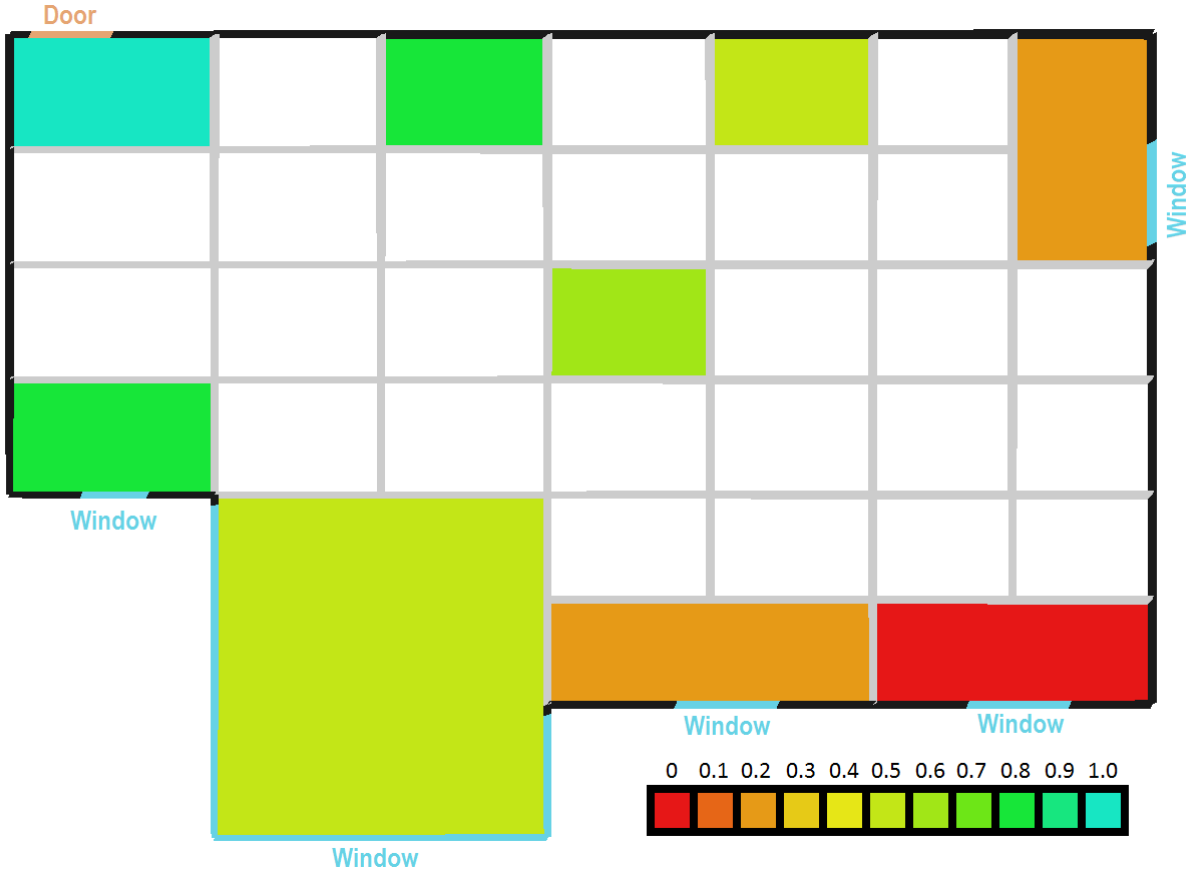

Рис. *40 Візуалізація соціальної оцінки*

Правило приватності вираховується за формулою:

$$sp_y = \frac{dist(cell_y, cell_D)}{max_i(dist(cell_i, cell_D))},$$

де *dist (celly, cellD)* – дістанція між обраною клітинкою і клітинкою з вхідними дверима, *maxi (dist (celli, cellD))* – максимальна дистанція між деякою клітинкою та клітинкою входу. У випадку цього правила приватні кімнати будуть розташовуватись як най далі від виходу. Візуалізація правила приватності зображене на рисунку 41.

Рис. *41 Візуалізація правила приватності*

Правило еластичного з'єднання обчислюється за формулою:

$$se_y = \sum_{i=1}^{n} 0.5^{dist\,(cell_y,\, cell_j)},$$

де *dist (celly, cellj)* – відстань між обраною коміркою *celly* та коміркою *cellj*, в якій розташована кімната з тим самим типом, що і поточна кімната. Ідея цього правила у знайдені комірки, яка буде рівновіддалена від усіх кімнат того ж типу. Візуалізація правила еластичного з’єднання зображене на рисунку 42.

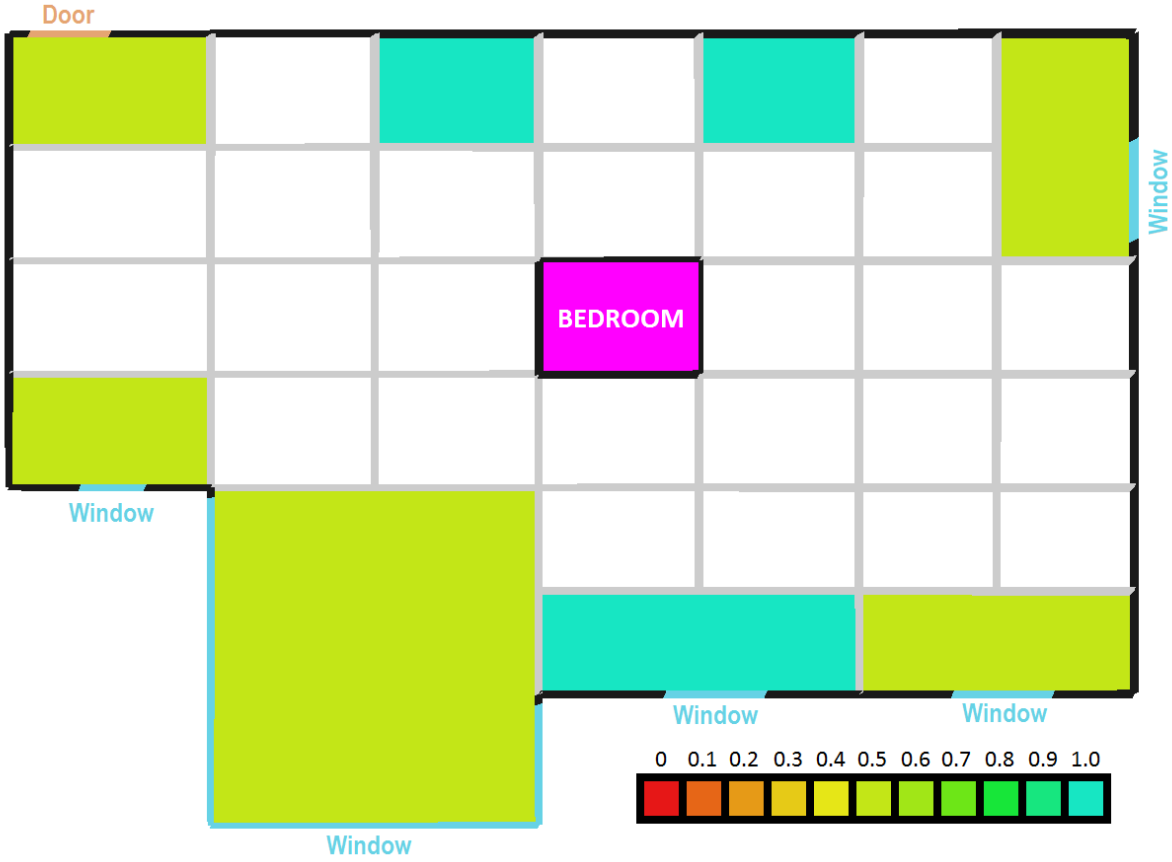

Рис. *42 Візуалізація еластичного з'єднання*

Правило важкого з'єднання обчислюється за формулою:

$$sh_y = \begin{cases} 0.5^{dist(cell_y, cell_h)}, if\ dist(cell_y, cell_h) < t_H, \\ ¿0, в\ іншому\ випадку \end{cases}$$

де *dist* ($cell_y$, $cell_h$) - відстань між придатною коміркою *y* та клітиною *h*, що містить раніше розміщене приміщення, з яким необхідно зберігати сусідство. Порогова відстань t_H визначається як подвійна довжина t_W (поріг довжини стінки сітки). Тобто фактично для будь-якої комірки, що не відповідає вимогам (перевищує порогову відстань t_W), бал встановлюється на нуль. Суть цього правила у гарантуванні того, що кімнати, які задані користувачем як сусідні, були поруч. Візуалізація правила важкого з'єднання зображене на рисунку 43.

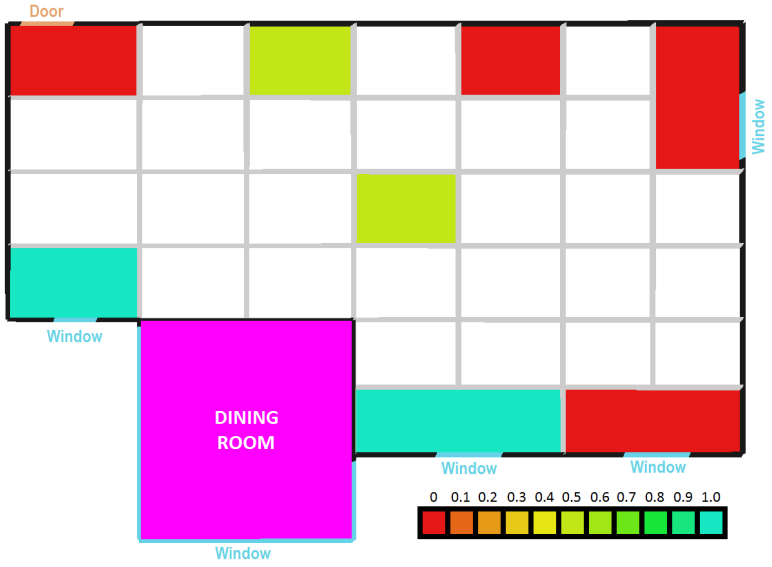

Рис. *43 Візуалізація важкого з'єднання*

Правило вхідних дверей обчислюється за формулою:

$$sd_y = \begin{cases} 0, \text{якщо двері присутні в комірці } y, \text{але за } R_R \text{ кімната не соц. типу} \\ ¿\,1, \text{в ішному випадку} \end{cases},$$

де 0, якщо є вхідні двері, але кімната соціального типу, 1 – в іншому випадку.

Правило маленького вікна обчислюється за формулою:

$$sk_y = \begin{cases} 0, \text{якщо вікно присутнє в } y, \text{але за розміром в } R_R \text{ воно не підходить} \\ ¿\,1, \text{в іному випадку} \end{cases},$$

де 0, де воно не потрібно або мале, 1 – в іншому випадку.

Після того, як оцінка кожної придатної комірки *y* була визначена для приміщення, що розміщується, масив балів *S* нормалізується таким чином:

$$P_y = \frac{S_y}{\sum_{x=1}^{n} S_x},$$

де *x* — кожна клітина, яка відповідає вимогам, для якої була розрахована оцінка *S*. Таким чином, кожна нормалізована оцінка P_y — це ймовірність того, що дана відповідна комірка *y* буде вибрана випадковим чином.

Далі йде етап ітеративного розширення кімнат. Кімнати розширяються в чотирьох напрямах: вгору, вниз, вліво, вправо. Для кожного напряму вираховується оцінка за формулою:

$$K_o = kb_o\, kw_o\, kn_o,$$

де kb_o, kw_o, kn_o – змінні, які варіюються між 0 та 1. Вони також вираховуються за правилами: заблокованого напряму, витраченого вікна та правилом необхідності.

Правило заблокованого напряму обчислюється за формулою:

$$kb_o = \begin{cases} 0, \text{якщо цей крок призводить до виходу кімнати з будинку} \\ ¿\,1, \text{в іншому випадку} \end{cases},$$

де 0, якщо цей крок приводить до виходу кімнати з області будинку, 1 – в іншому випадку.

Правило витраченого вікна обчислюється за формулою:

$$kw_o = \begin{cases} 1, if\ n_w = 0 \vee n_w = 1 \\ ¿\,\dfrac{0.2}{(n_w - 1)}, if\ n_w > 1 \end{cases},$$

де n_W – це кількість вікон і 1, якщо n_W 0 або 1, інакше якщо n_W більше 1 застосовується формула $\dfrac{0.2}{(n_W - 1)}$.

Правило необхідності обчислюється за формулою:

$$kn_o = \begin{cases} 1, if\ dim ¿\, R_{minW} \wedge dim ¿\, R_{minL}, \\ ¿\,0.5, if\ dim ¿\, R_{minW} \oplus dim ¿\, R_{minL}, \\ ¿\,0.1, otherwise, \end{cases}$$

де 1, якщо розмірність по ширині R_{minW} та довжині R_{minL} менше мінімально необхідної, 0.5, якщо розмірність по ширині R_{minW} більше мінімальної, а по довжині R_{minL} менше або більше, 0.1 – в іншому випадку.

Нормалізація результату обчислюється за формулою:

$$P_o = \frac{K_o}{(\sum_{x=1}^{n} K_x)}.$$

де K_o – одна з оцінок розширення кімнати, а K_x елемент суми цих оцінок.

Кімнати розширюються протягом трьох різних етапів: першого кроку розширення, кроку фіксації та другого кроку розширення.

Метою першого кроку розширення є вирощування кожної одноклітинної кімнати до її максимального розміру, як визначено в налаштуваннях приміщення. По-перше, кімнати сортуються відповідно до їх пріоритету. По-друге, кожна кімната обирається по черзі, від найвищого до найнижчого пріоритету, і розширюється до тих пір, поки не буде досягнуто максимального розміру. Коли усі кімнати розширились до максимального заданого розміру, наступає етап фіксації.

Крок фіксації розширює кімнати, які залишаються заблокованими після першого кроку розширення. Для цього алгоритм звільняє простір уздовж клітинок, які заблокували кімнату, щоб виконати свій R_R. Щоб звільнити простір у заданому напрямку (вгору / вниз або вліво / вправо), кожна кімната, що прилягає до заблокованого приміщення (у відповідному напрямку), скорочує одну клітинку. Після звільнення сусідніх клітинок, кімната яка була затиснена зростає до мінімального розмірів.

РОЗДІЛ 3 ПРОЕКТ ПРОГРАМНОЇ СИСТЕМИ ГЕНЕРУВАННЯ ПРИМІЩЕНЬ

3.1 Архітектура системи

Програмну систему процедурної генерації можна поділити на 2 основних модулі:

- Програмна реалізація генерування 2D плану будинку, яка включає в собі усі компоненти, такі як:
  - підготовка форми будівлі;

- знаходження потрібної площі;
- планування різних поверхів будинку (підвал, звичайних поверх, дах);
- генерування квартир, та кімнат.

- Програмна реалізація 3D візуалізації згенерованих 2D планів будинків. 3D візуалізація реалізована завдяки ігровому двигуну Unity3D, який має досить гнучкі інструменти для відображення 3D об’єктів.

3.1.1 Архітектура 2D плану будівлі

Архітектура 2D плану будинку базується на ієрархії класів, представленій на рисунку 44.

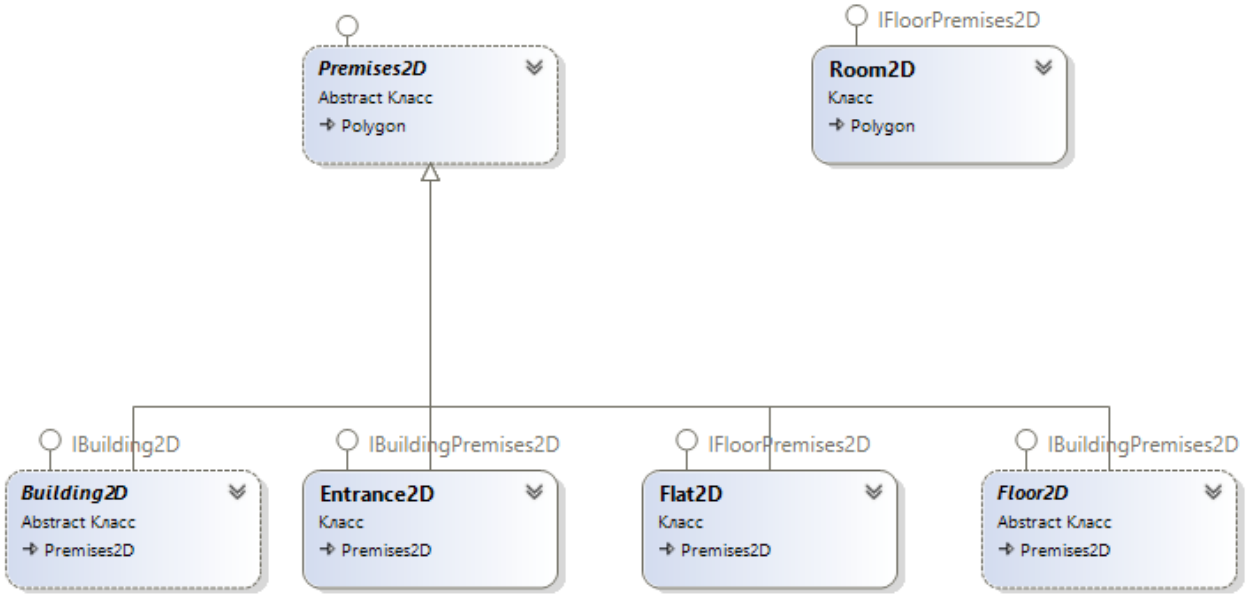

Рис. *44 Ієрархія 2D плану*

Суть цієї ієрархії полягає в тому, що будівля може мати в собі список елементів інтерфейсу *IBuildingPremises2D* (внутрішні елементи будівлі) – це можуть бути елементи *Entrance2D* (під’їздів) або *Floor2D* (поверхів). Це зроблено тому, що будівля може мати або вигляд багатоповерхівки з під’їздами, або вигляд звичайного дому без під’їздів. Абстрактний клас

Floor2D (поверху) тримає в собі колекцію елементів інтерфейс *IFloorPremises2D* (внутрішні елементи будівлі), такі як: квартири або кімнати.

На етапі створення поверху та квартири було реалізовано абстрактний клас *PlanProcessor2D* (Рис. 46), який реалізує планування будинку, тобто додавання вікон, коридору, сходів та інше. Також цей *PlanProcessor2D* містить допоміжний клас *GrowthProcessor2D* (Рис. 45), який реалізує вирощування кімнат або квартир. У випадку багатоповерхівки з квартирами в кінцевому результаті спочатку на рівні поверху, вирощуються квартири, а потім вже в створених квартирах йде вирощування кімнат.

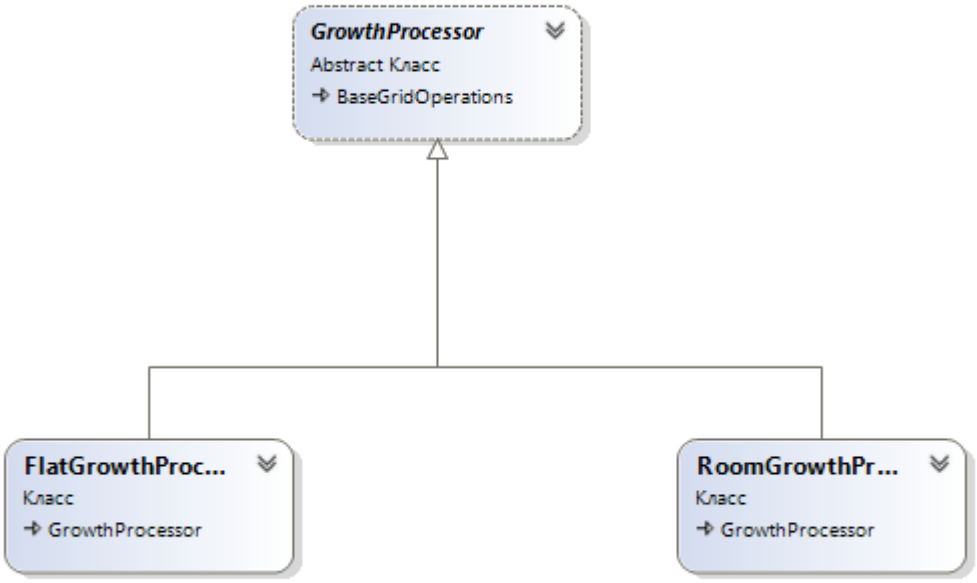

Рис. *45 Ієрархія GrowthProcessor*

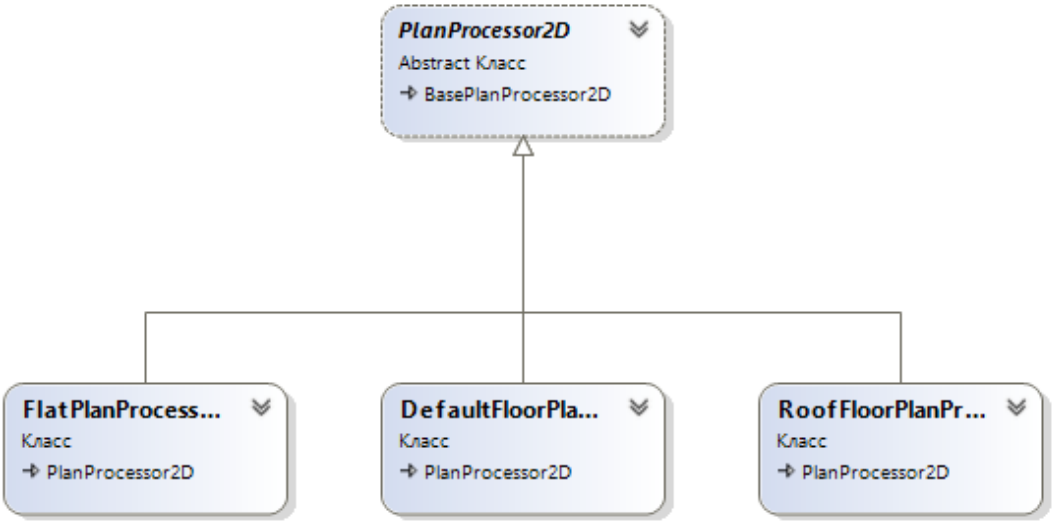

Рис. *46 Ієрархія PlanProcessor2D*

Побудована ієрархія класів дозволяє досить легко додати поверхи, які мають свої особливості, наприклад дах, який може мати різну форму (Рис. 47).

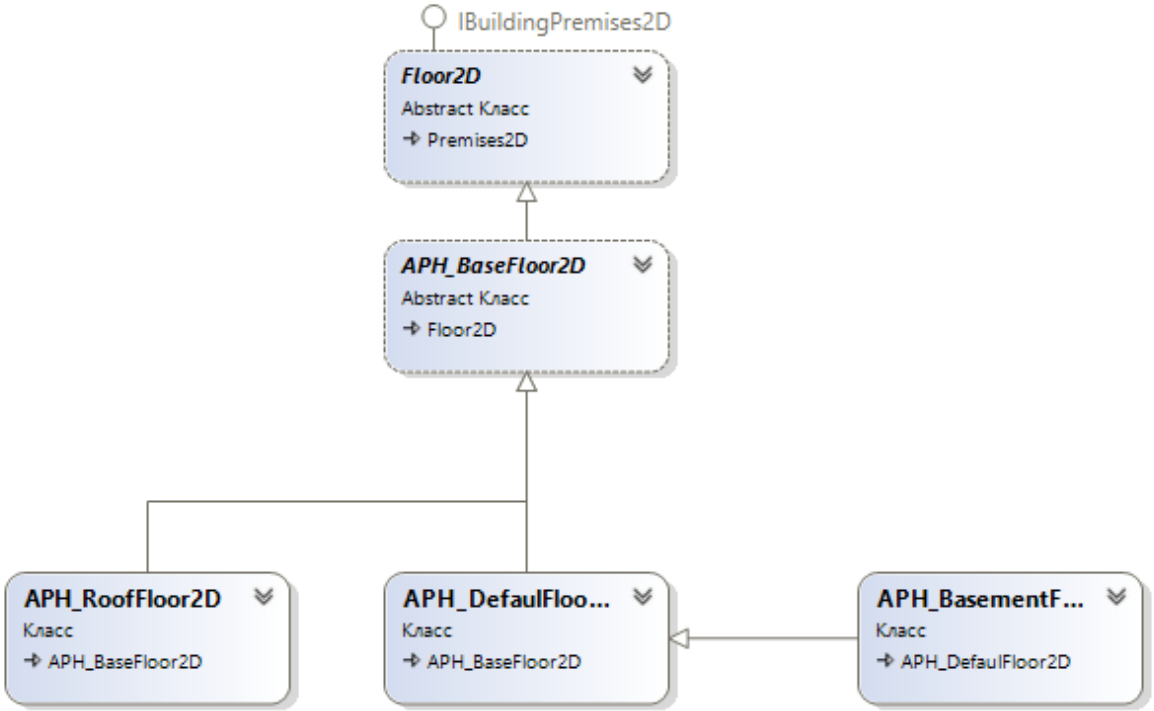

Рис. *47 Ієрархія класів Поверху*

3.2 Засоби реалізації

Для реалізації програмного продукту було використано наступні засоби:

- середовище розробки Visual Studio 2019 Community;
- мова програмування C#;
- ігровий двигун Unity3D;
- 3D редактор Blender.

Середовище розробки Visual Studio 2019 Community

Під час розробки програмної системи використано середовище розробки Visual Studio 2019 Community, яке має підтримку написання коду для ігрового двигуну Unity3D, а саме підтримка IntelliSense для Unity. Середовище розробки підтримує Debug режим спеціально під Unity.

Мова програмування C#

C# — об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи DOTNET.

Ігровий двигун Unity3D підтримує дві мови програмування для розробки: UnityScript (аналог JavaScript) та C#. В розробленому застосунку було використано мову C# через очевидні переваги:

- мова C# строго типізована;
- більшість коду для Unity написана на C#.

Ігровий двигун Unity3D

Unity — міжплатформенне середовище розробки комп'ютерних ігор [3]. Unity дозволяє створювати додатки, що працюють на більш ніж 25 різних

платформах, що включають персональні комп'ютери, ігрові консолі, мобільні пристрої, інтернет-додатки та інші [3]. Випуск Unity відбувся в 2005 році і з того часу йде постійний розвиток.

Для візуалізації побудови приміщень був використаний ігровий двигун Unity, оскільки він є безкоштовним для персонального використання і одним з найпопулярніших ігрових двигунів у світі та більше підходить для невеликих проектів.

3D редактор Blender

Blender — пакет для створення тривимірної комп'ютерної графіки, що включає засоби моделювання, анімації, рендерінгу, після-обробки відео, а також створення відеоігор [7].

Особливостями пакету є малий розмір, висока швидкість рендерінга, наявність версій для багатьох операційних систем — FreeBSD, GNU/Linux, Mac OS X, SGI Irix 6.5, Sun Solaris 2.8 (sparc), Microsoft Windows, SkyOS, MorphOS та Pocket PC. Пакет має такі функції, як динаміка твердих тіл (Rigid Body), рідин (Liquid simulation) та м'яких тіл (Soft body), систему гарячих клавіш (hot key), велику кількість легко доступних розширень, написаних мовою Python. Починаючи з версії 2.61 з'явились функції "відстеження камери" (англ. camera tracking), та "захоплення руху" (англ. motion capture або mocap). Програма є вільним програмним забезпеченням та розповсюджується під ліцензією GNU GPL.

У розробленій роботі магістра використано 3D редактор Blender для створення таких 3D моделей як: стіни будівлі, стіни з вікнами, двері, балкони, вхід до будинку та інші.

3.3 Модулі та алгоритми

Під час розробки програмного забезпечення для побудови процедурних планів будівель, було розроблено 2 модулі програми, а також проміжні функції для роботи з багатокутниками:

- Модуль 2D плану будівлі (будинок, під’їзд, квартира, кімната).
- Модуль 3D для візуалізації будинку за допомогою функцій Unity.
- Модуль роботи з полігонами (пошук площі багатокутника, чи точка лежить на краю полігону, чи точка в середини полігону та інше).

3.3.1 Обмеження програмної реалізації

Програмна система має два значних обмеження. Перше обмеження на вхід програмній системі можна подавати лише багатокутники з кутами у 90 градусів, це обумовлюється тим, що 3D частина будується на основі прямих стін з шириною у 2 метри. Друге обмеження базується на першому, усі моделі стін повинні мати параметри 2 метри ширини, 2.5 метри висоти. В цій програмній системі реалізована лише будівля у вигляді прямокутника, а саме панельний будинок.

3.3.2 Побудова 2D плану будівлі

Вхідні данні

Для вводу вхідних даних використовуються вбудовані в Unity контейнери даних ScriptableObject, які використовуються для зберігання об’ємних даних. Їх основна перевага – це серіалізація об’ємних даних і не залежність від скриптів гри.

Вхідні дані поділені на декілька контейнерів налаштувань ScriptableObject: будинку, 3D кімнат, 2D кімнат. Всі вхідні дані зображені на рисунках 47-52.

На рисунку 48 зображено частину вхідних даних будівлі, а саме 3D об’єкти та матеріали будівлі, які необхідні для візуалізації (основний вхід до під’їзду, зливні труби, сходи та інше).

Рис. *48 Вхідні 3D об'єкти будівлі*

На рисунку 49 зображено налаштування для кімнат.

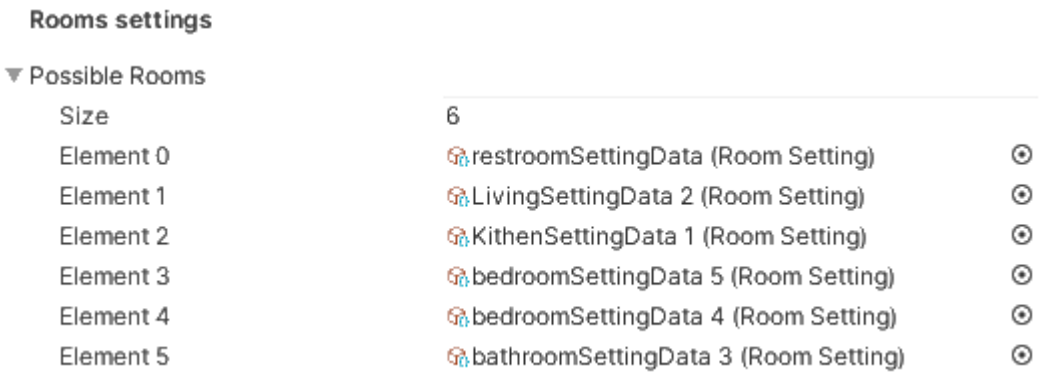

Рис *49 Налаштування можливих кімнат квартир*

На рисунку 50 представлені налаштування, які відповідають за площу та кількість поверхів та під'їздів.

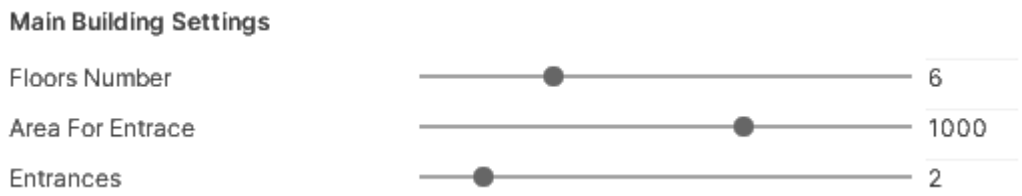

Рис. *50 Налаштування площі будинку*

На рисунку 51 зображено додаткові налаштування 3D моделей саме для кімнати, бо кожна кімната може мати свої особливості, такі як різні матеріали стін, різні вікна та інше.

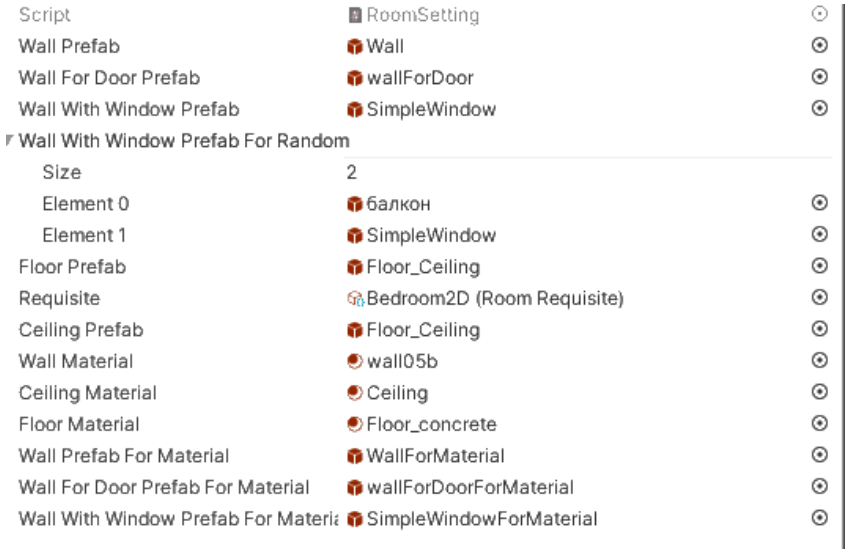

Рис. *51 Налаштування 3D кімнати*

На рисунку 52 задаються налаштування для 2D кімнати: площа, необхідність вікна, максимальна кількість з’єднань з кімнатами та налаштування зростання.

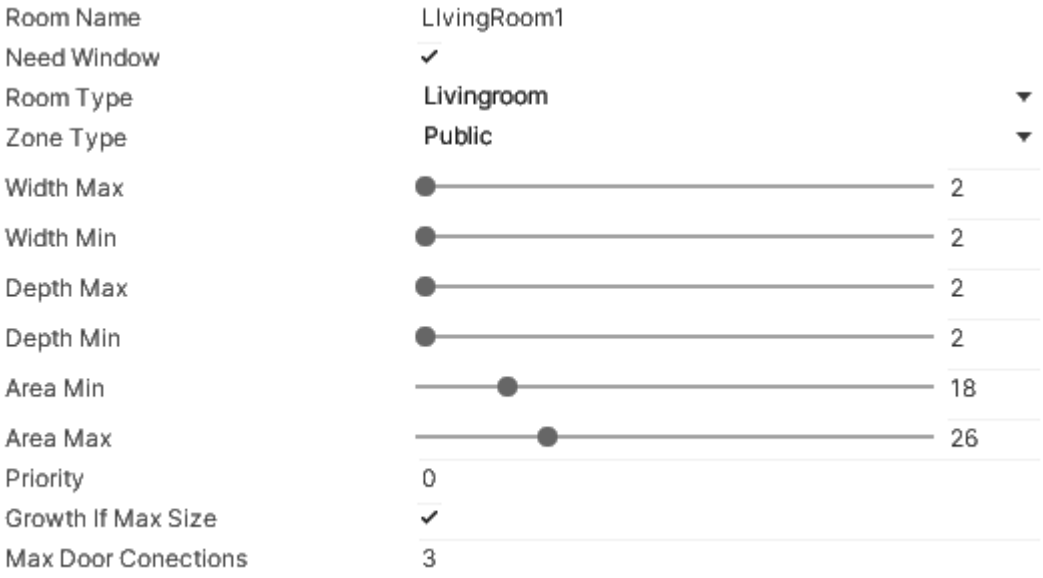

Рис. *52 Налаштування 2D кімнати*

Процес побудови приміщення 2D

Процес побудови приміщення розбивається на частини, будівля відповідає за розділення на під'їзди, під'їзд відповідає за розділення на поверхи, які відповідають за розділення на квартири, а квартири за розділення на кімнати. Схему побудови приміщення представлено на рисунках 53, 54.

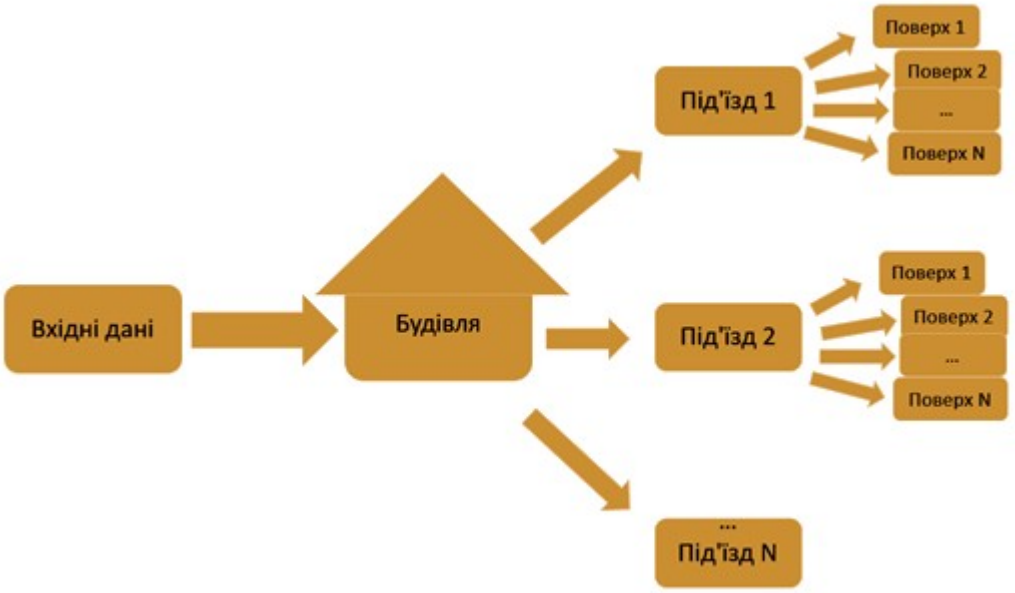

Рис. *53 Візуалізація ієрархії будинку частина 1*

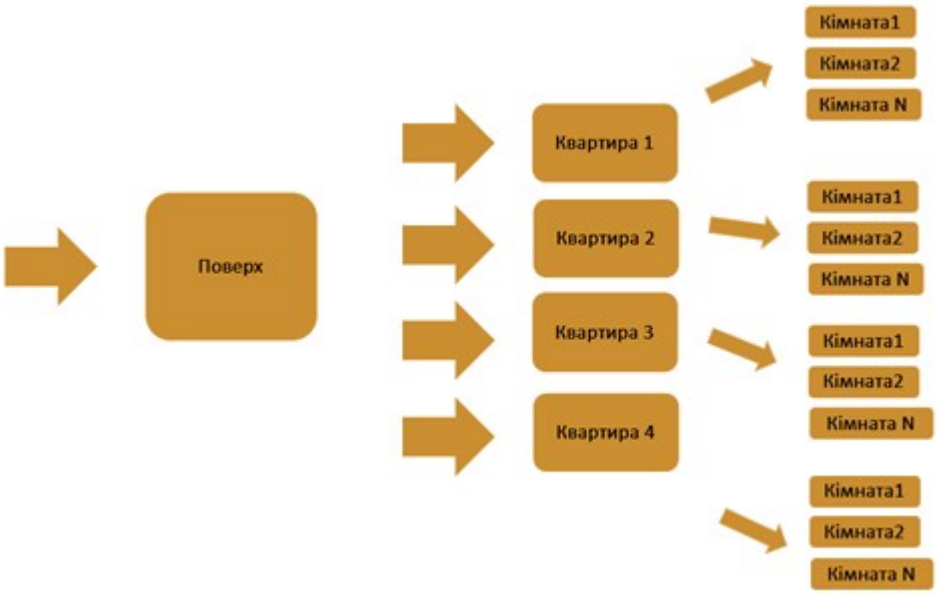

Рис. *54 Візуалізація ієрархії будинку частина 2*

На рівні будівлі вирішуються такі проблеми як:

- отримання форми будівлі;
- підгін площі будівлі від вхідні данні;
- розділення отриманої форми будівлі на під’їзди.

Рівень під’їзду відповідає за створення поверхів.

На рівні поверху вирішуються такі проблеми як:

- коректне розміщення квартир;
- коректне розміщення коридору;
- коректне розміщення східців.

На рівні квартири вирішуються такі проблеми як:

- коректне розміщення кімнат;
- коректне додавання дверей між кімнатам.

Реалізація алгоритму побудови плану приміщення

Концепція алгоритму побудови приміщення базується на застосуванні росту кімнат за допомогою оцінок підходящих кандидатів, яку запропонував Camozzato D. A. [6], суть цього алгоритму у розбитті контуру будівлі на зони у вигляді сітки, в які можна покласти кімнати для подальшого росту. В випадку алгоритму вирощування кімнат [6] побудована сітка базувалась на вхідних параметрах вікон та кутах будівлі, або ж при відсутності їх з фіксованим кроком. В дані розробленій програмній системі будується сітка на основі форми будівлі з деяких кроком у 2 метри (Рис. 55), тобто в результаті буде побудована сітка з квадратів 2 на 2 метри, це обумовлюється тим, що усі моделі стін мають ширину у 2 метри.

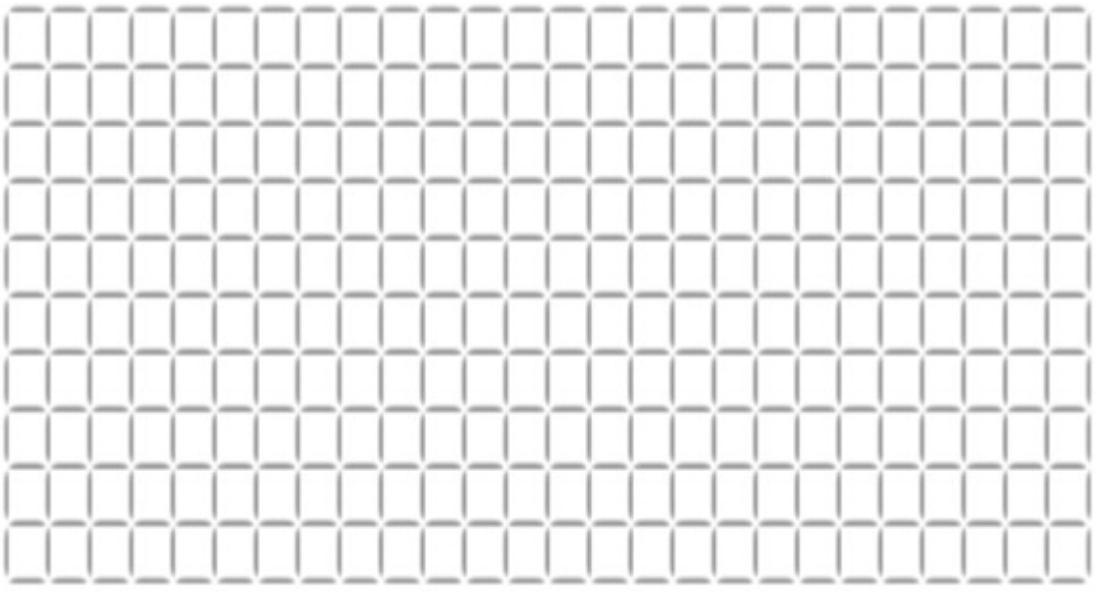

Рис. *55 Створена сітка для будівлі*

Ця сітка будується на рівні поверху та рівні квартири, бо вирощування застосовується спочатку для 4 квартир, а потім на основі форми квартири вже розширюються кімнати.

Процес створення кімнат має декілька етапів:

- оцінка клітинок для кімнат;
- вирощування кімнат та квартир на основі оцінок (зображене на рисунках 56-58).

В процесі вирощування квартир кожна з них розташовується в кутах поверху, а далі йде рівномірне вирощування квартир, до того моменту коли все вільне місце не буде зайняте.

Побудова кімнат та квартир делегуються класу PlanProcessor2D, який займається побудовою сітки, додаванням вікон та статичним додаванням кімнат — цей клас був реалізований для двох рівнів будинку. Для рівню поверху в класі FloorPlanProcessor2D та для рівню квартири FlatPlanProcessor2D (Рис. 56).

Вирощування кімнат делегується класу GrowthProcessor2D, а саме його дітям FlatGrowthProcessor та RoomGrowthProcessor. Перший відповідає за вирощування кварти на рівні поверху, а другий за вирощування кімнат на рівні квартири (Рис. 57, 58).

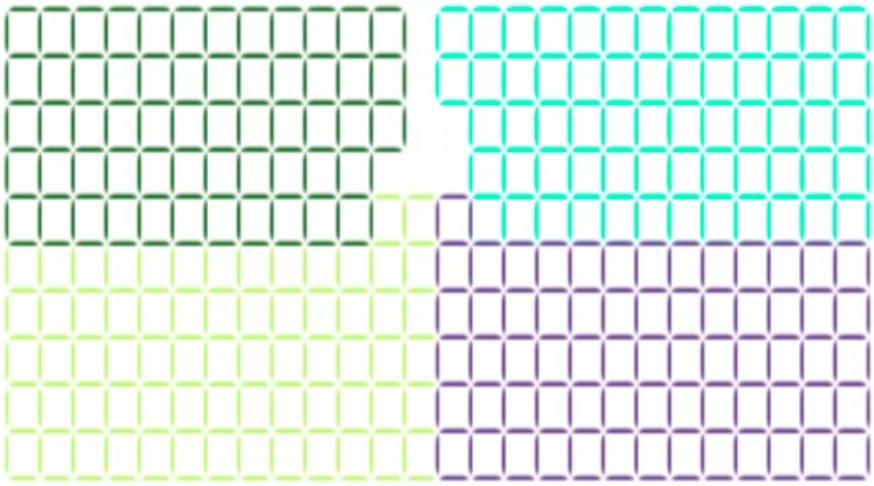

Рис. *56 Згенеровані квартири (без кімнат)*

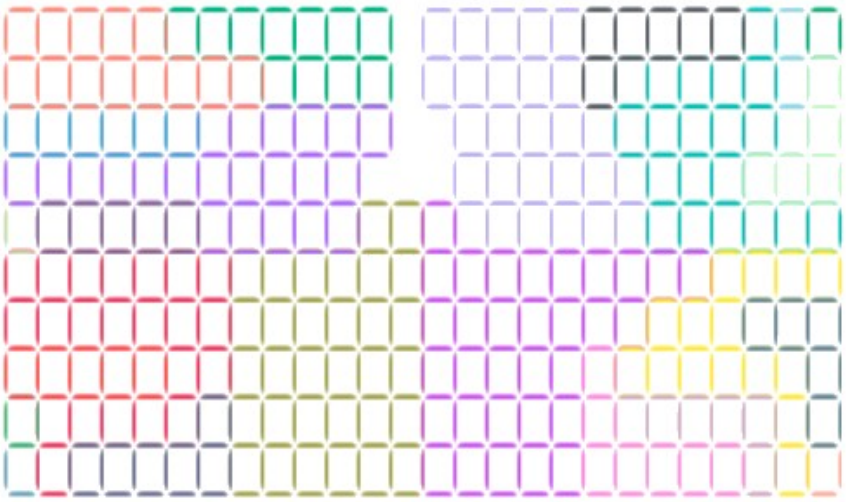

Рис. *57 Згенеровані квартири у вигляді комірок (з кімнатами)*

Рис. *58 Згенеровані квартири (з кімнатами)*

Алгоритм контроль площі будівлі

Хоча дана реалізація має форму прямокутника, наперед була додана можливість контролю площі для прямокутного багатокутника. Один з варіантів вираховування площі — це застосування алгоритму тріангуляції [11] для вхідного полігону, а потім вираховувати площу трикутників після цього дивлячись на площу якщо площа замала, то усі точки помножуються на деяке значення близьке до 1 для приближення наявної площі до шуканої. Далі необхідно округлити всі точки до цілих чисел кратних двом, бо з наших обмежень йде, що клітинки повинні мати розмір 2 на 2 метри.

Додавання під'їздів до будинку та входів до під'їздів

Реалізована будівля панельного типу має форму прямокутника до вхідних даних якої задається кількість під'їздів для будинку. На основі цієї цифри береться найдовше ребро прямокутника і ділиться на $n+2$ точок, де n – кількість під'їздів. В результаті на основі цих точок утворюються прямокутники під'їздів.

Додавання входу до під'їзду має схоже рішення, але в даному випадку у найдовшого ребра знаходиться точка середини і це є знайдений вхід в будівлю.

Реалізація додавання коридору, сходів, ліфту

Вирішення проблеми додавання сходів у випадку розробленої програмної системи таке: так як коридор та сходи повинні мати конкретне місце біля входу то алгоритм статично додає сходи до входу як кімнату 2x6 метрів. Коридор додається як кімната горизонтальна сходам розміром 6x2 метри. Ліфт було додано як кімнату на одну 2x2 метри. Відображення розташування кімнат ліфту, коридору та сходів зображено на рисунку 59.

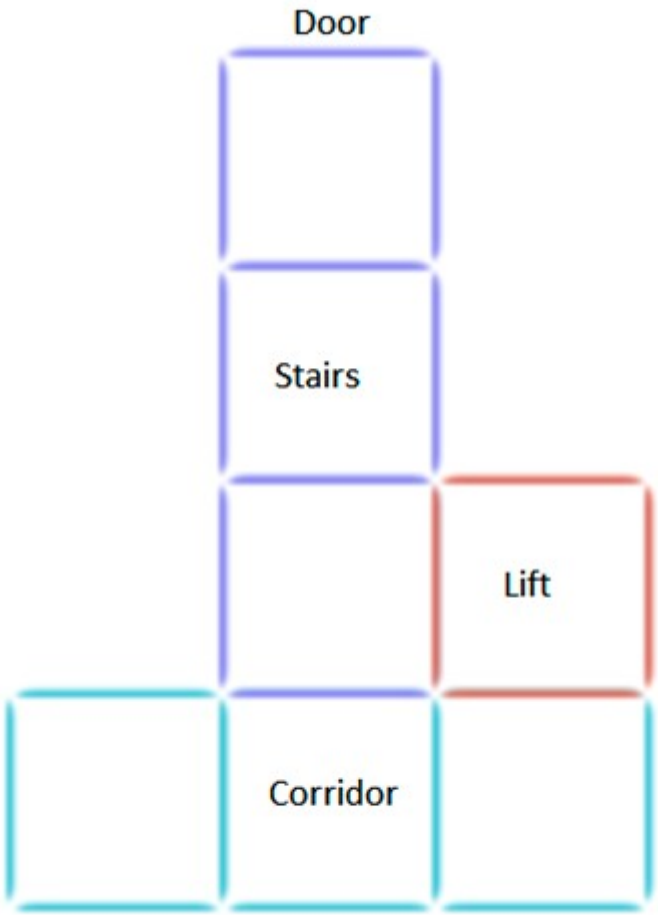

Рис. *59 Відображення статичної частини будівлі з помітками кімнат*

Додавання дверей між кімнатами

В Розробленій програмній системі під час заповнення налаштувань будинку додаються лінк-з’єднання між кімнатами та на їх основі будувати двері між кімнатами у випадковій стіні між кімнатами.

Реалізація різних дахів

В даній програмні системі існує два варіанти даху, перший – це реалізація на основі поверху з виходом на дах. Ця реалізація створюється як поверх будівлі, а саме дах з виходом на нього (Рис. 60).

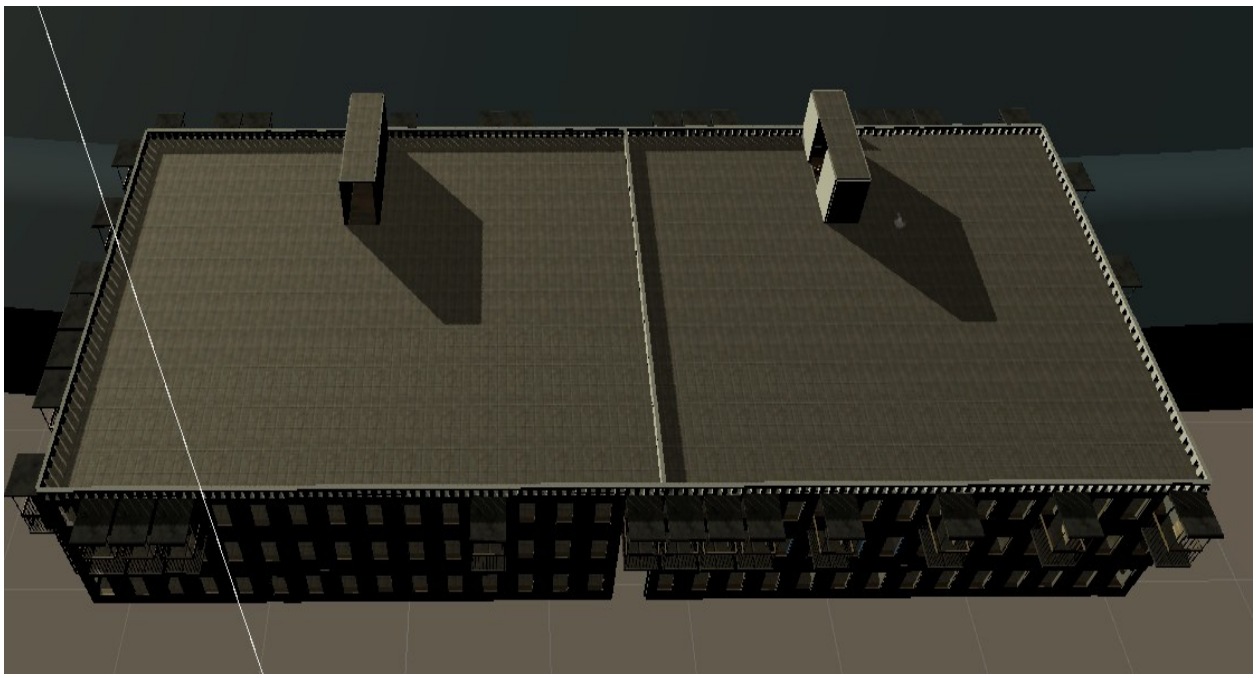

Рис. 60 *Плоский дах будівлі*

Для другого виду даху був використаний алгоритм Straight skeleton [10], який розроблений Aichholzer et al. [15] та розширений Aichholzer та Aurenhammer [16].

Цей алгоритм може працювати з будь-якими двовимірними полігонами, такими як: звичайні та з внутрішніми порожнинами. Це відкриває великі можливості створювати різноманітні варіанти дахів, які будуть виглядати доволі реалістично. Приклади візуалізації роботи алгоритму зображені на рисунках 61-62.

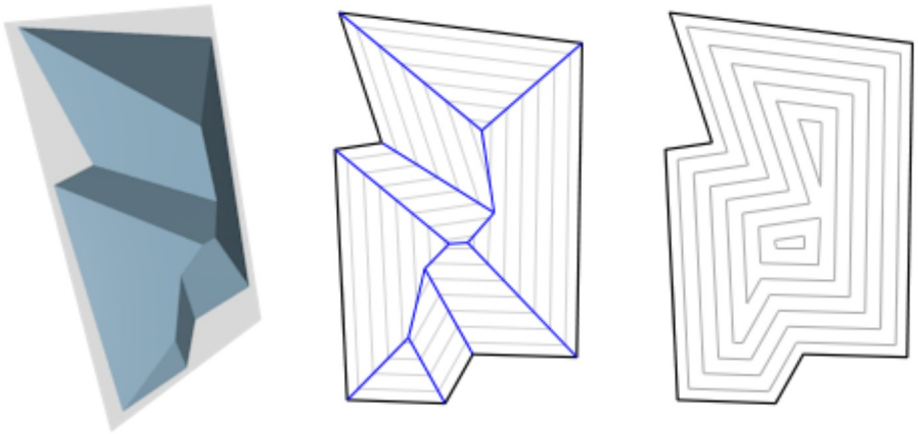

Рис. *61 Візуалізація роботи алгоритму Straight skeleton*

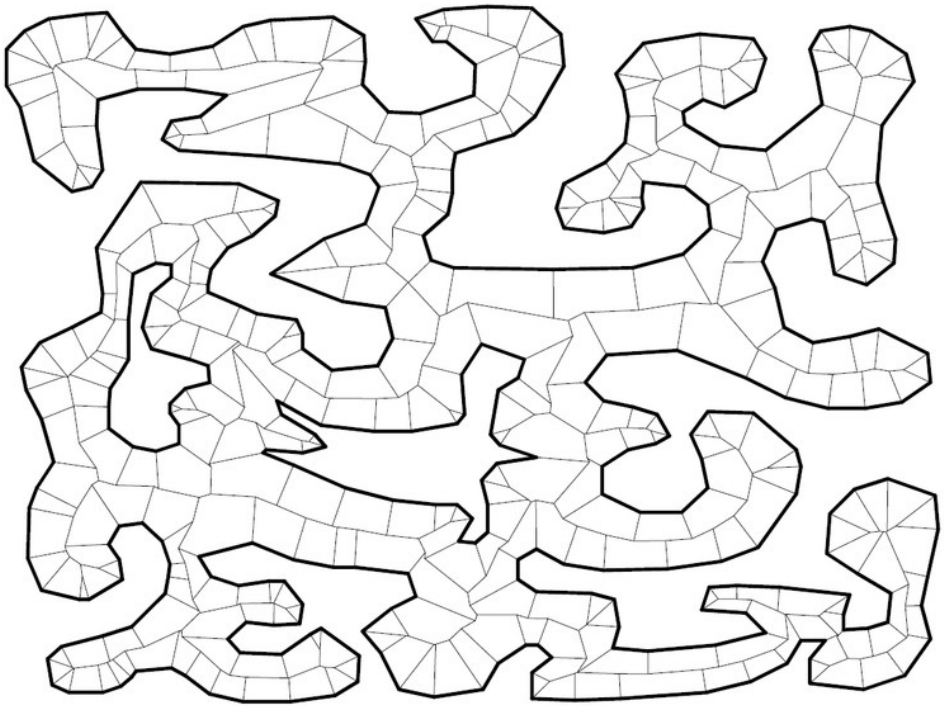

Рис. *62 Складний роботи алгоритму Straight Skeleton*

На рисунку 63 відображено створений дах створених розробленою програмною системою на основі алгоритму Straight skeleton.

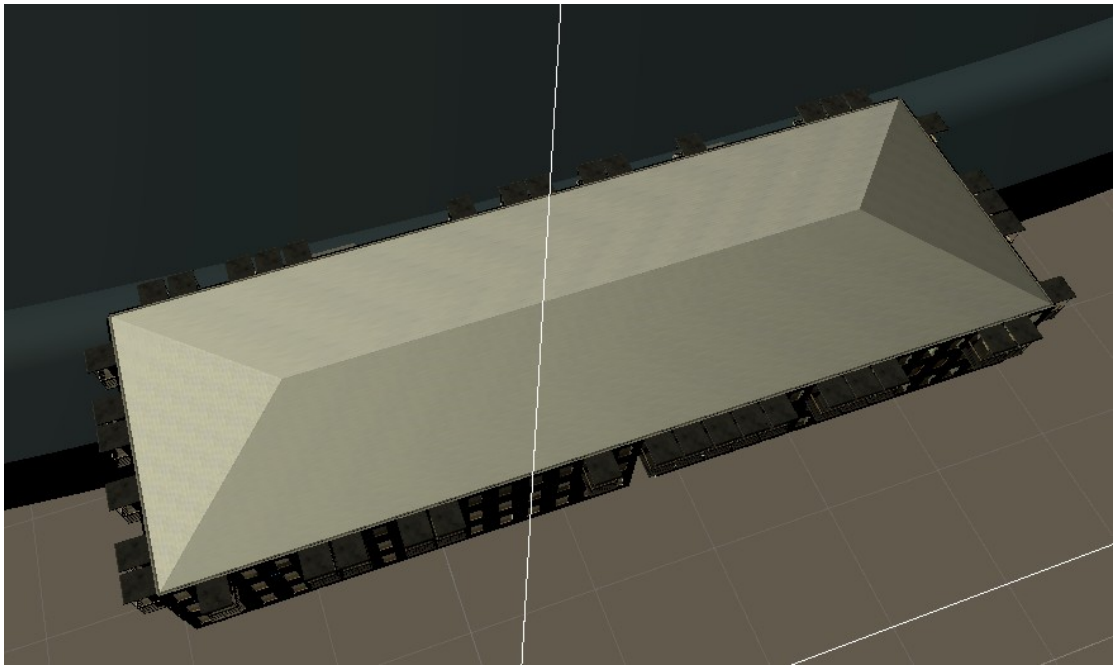

Рис. *63 Відображення каскадного даху за допомогою алгоритму Straight Skeleton*

У лістингу 3 відображена функція візуалізації каскадного даху за допомогою алгоритму Straight skeleton. Статична функція *Skeleton-Builder.BuildRoof(roofBoarder)* повертає дах у вигляді списку полігонів з 2D координатами. Після отримання полігонів даху необхідно перевести створені точки у 3D координати та задати висоту загальних точок полігонів каскадного даху. Після переведення даху у тривимірну систему координат залишається лише відобразити дах за допомогою меш-сітки.

Лістинг 3 Функція візуалізації каскадного даху

```
public void VisualizeRoof()
    {
        var roof = SkeletonBuilder.BuildRoof(_roofBoarder);
        var meshData = new RoofMeshData(roof, _roofBoarder,
_numberOfFloors - 2, _floorHight);
        var RoofRoot = new GameObject("RoofRoot");

        RoofRoot.transform.parent = _buildingRoot;

        for (var i = 0; i < meshData.verticesOfPolygons.-
Count; i++)
```

```
        {
            Mesh msh = new Mesh();
            for (var j = 0; j < meshData.verticesOfPoly-
gons[i].Count; j++)
            {
                meshData.verticesOfPolygons[i][j] +=
_buildingRoot.transform.position;
            }
            msh.vertices =
meshData.verticesOfPolygons[i].ToArray();
            msh.triangles =
meshData.indicesOfPolygons[i].ToArray();
            msh.RecalculateNormals();
            msh.RecalculateBounds();

            var emptyObj = new GameObject("roofPart" + i.-
ToString());
            emptyObj.transform.parent = RoofRoot.transform;

            emptyObj.AddComponent(typeof(MeshRenderer));

            var meshRender = emptyObj.GetComponent<MeshRen-
derer>();
            meshRender.material = _roofMaterial;

            Vector3[] vertices = msh.vertices;

            Vector2[] uvs = new Vector2[vertices.Length];

            if (vertices.Length == 4)
            {
                uvs[0] = new Vector2(0, 0);
                uvs[1] = new Vector2(0, 1);
                uvs[2] = new Vector2(1, 1);
                uvs[3] = new Vector2(1, 0);
            }
            else if (vertices.Length == 3)
            {
                uvs[0] = new Vector2(0, 0);
                uvs[1] = new Vector2(0, 1);
                uvs[2] = new Vector2(1, 1);
            }
            msh.uv = uvs;

            MeshFilter filter = emptyObj.AddCompo-
nent(typeof(MeshFilter)) as MeshFilter;
            filter.mesh = msh; }
```

Також була додана можливість комбінування дахів різних типів, а саме каскадного і плоского типів. Відображення комбінування дахів зображено на рисунку 64.

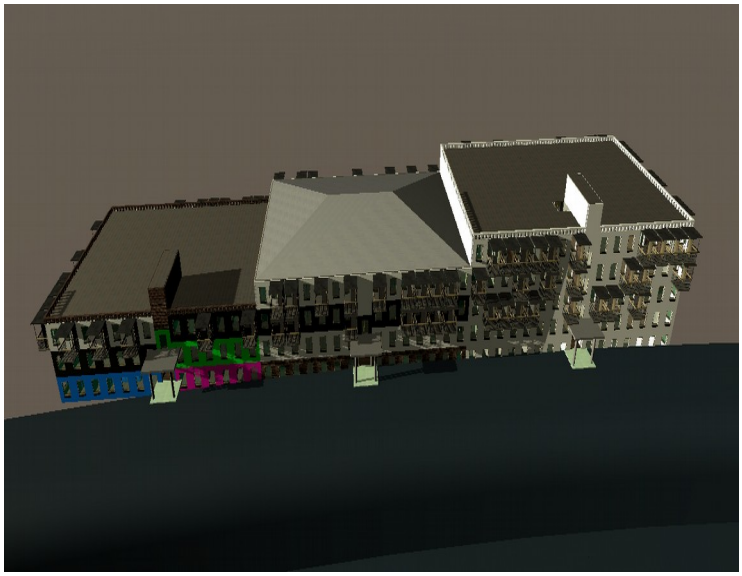

Рис. *64 Відображення комбінування даху*

Процес створення комбінованого даху наведено у лістингу 4.

Лістинг 4 Функція візуалізації комбінованого даху

```
private void VisualizeRoof()
{

          if (m_panelHousesettings.entraces.All(e =>
e.RoofType == RoofType.CASCADE && e.FloorsSettings.Count ==
m_panelHousesettings.entraces[0].FloorsSettings.Count))
              {
                  var roof = new CascadeRoof(BuildingForm,
FloorHight, NumberOfFloors, BuildingRoot.transform, m_pan-
elHousesettings.defalutRoofMaterial);
                  _roofs.Add(roof);
                  roof.VisualizeRoof();
              }
              else {

                  var CascadeRoofs = new
List<List<Vector2d>>();
                  var roofFloorForCreationRoof = new
List<int>();
                  var roofontinuationFloor = new List<int>();
```

```
                var curretRoofForm = new List<Vector2d>();

                Entaraces3D.ForEach(e =>
                {

                    if (e.HaveRoof && e.RoofType ==
RoofType.FLAT)
                    {
                        e.Roof.VisualizeRoof();

                        if (curretRoofForm.Count != 0)
                        {
                            CascadeRoofs.Add(curretRoof-
Form);

roofFloorForCreationRoof.Add(roofontinuationFloor[0]);
                            roofontinuationFloor.Clear();
                            curretRoofForm = new List<Vec-
tor2d>();
                        }


                    }
                    else if (e.RoofType == RoofType.CAS-
CADE)
                    {
                        if
(roofontinuationFloor.Exists(floor => floor == e.FloorNum-
ber) || roofontinuationFloor.Count == 0)
                        {
                            curretRoofForm.AddRange(e.Main-
Polygon);

roofontinuationFloor.Add(e.FloorNumber);
                        }
                        else
                        {
                            CascadeRoofs.Add(curretRoof-
Form);

roofFloorForCreationRoof.Add(roofontinuationFloor[0]);
                            roofFloorForCreationRoof.-
Clear();
                            curretRoofForm = new List<Vec-
tor2d>();
                        }
                    }
```

```
                });

                if (curretRoofForm.Count != 0)
                {
                    CascadeRoofs.Add(curretRoofForm);
                    roofFloorForCreationRoof.Add(roofontin-
uationFloor[0]);
                }

                for (var i = 0; i < CascadeRoofs.Count; i+
+)
                {
                    var roofForm = CascadeRoofs[i];
                    var maxX = roofForm.Max(v => v.X);
                    var maxY = roofForm.Max(v => v.Y);

                    var minX = roofForm.Min(v => v.X);
                    var minY = roofForm.Min(v => v.Y);

                    List<Vector2d> roofFOrm = new List<Vec-
tor2d>()
                    {
                        new Vector2d(minX, minY),
                        new Vector2d(minX, maxY),
                        new Vector2d(maxX, maxY),
                        new Vector2d(maxX, minY),
                    };

                    IRoof3D roof = new CascadeRoof(roof-
FOrm, FloorHight, roofFloorForCreationRoof[i], Building-
Root.transform, m_panelHousesettings.defalutRoofMaterial);
                    roof.VisualizeRoof();
                    _roofs.Add(roof);

                }
            }

        }
```

Реалізація алгоритму Straight skeleton була взята з відкритого коду GitHub користувача reinterpretcat [14].

3.3.3 Побудова 3D плану будівлі

Створення 3D моделей

Усі 3D моделі були створенні у 3D редакторі Blender [7]. Це програма з відкритим кодом, яка має сучасні інструменти для створення дуже складних 3D моделей. Наразі на ринку 3D моделювання існує декілька популярних рішень. Одним із головних конкурентів Blendey є 3Dmax [8], який має більший функціонал порівняно з Blender, але ця програма не є безкоштовною, тому усі моделі були створені у Blender. Побачити інтерфейс цих двох програм можна на рисунках 65, 66.

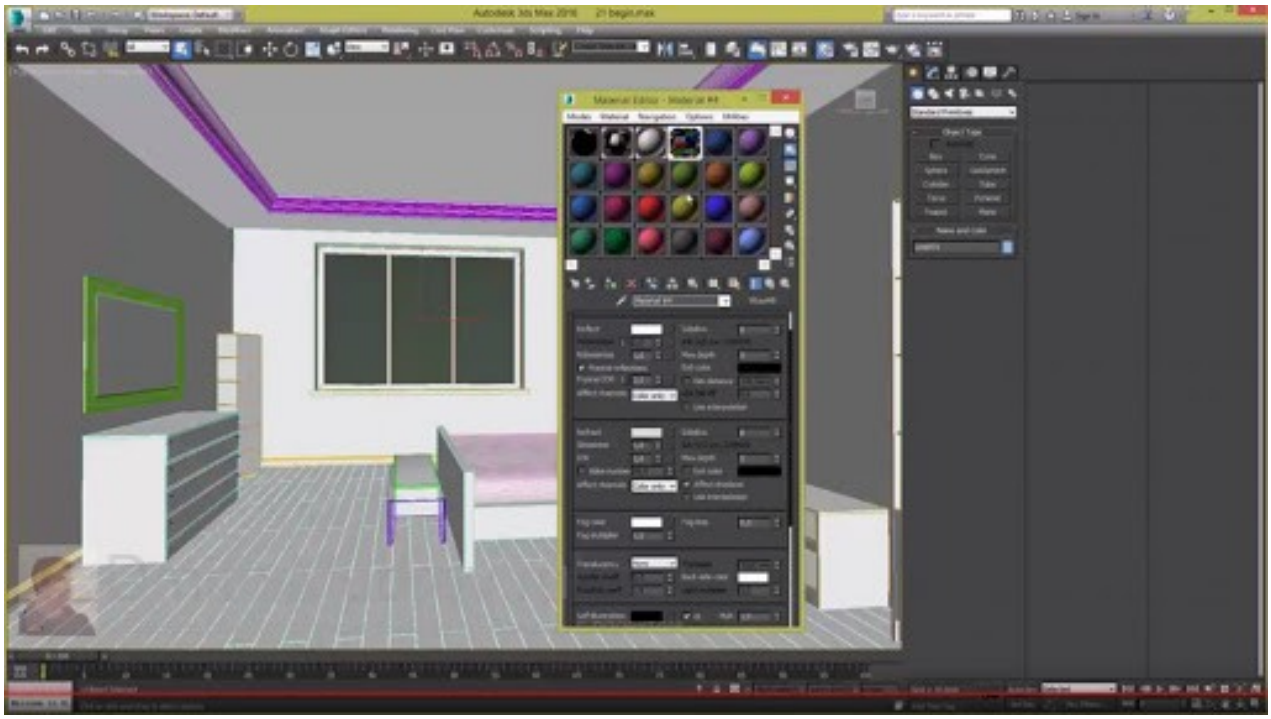

Рис. *65 Інтерфейс 3D max*

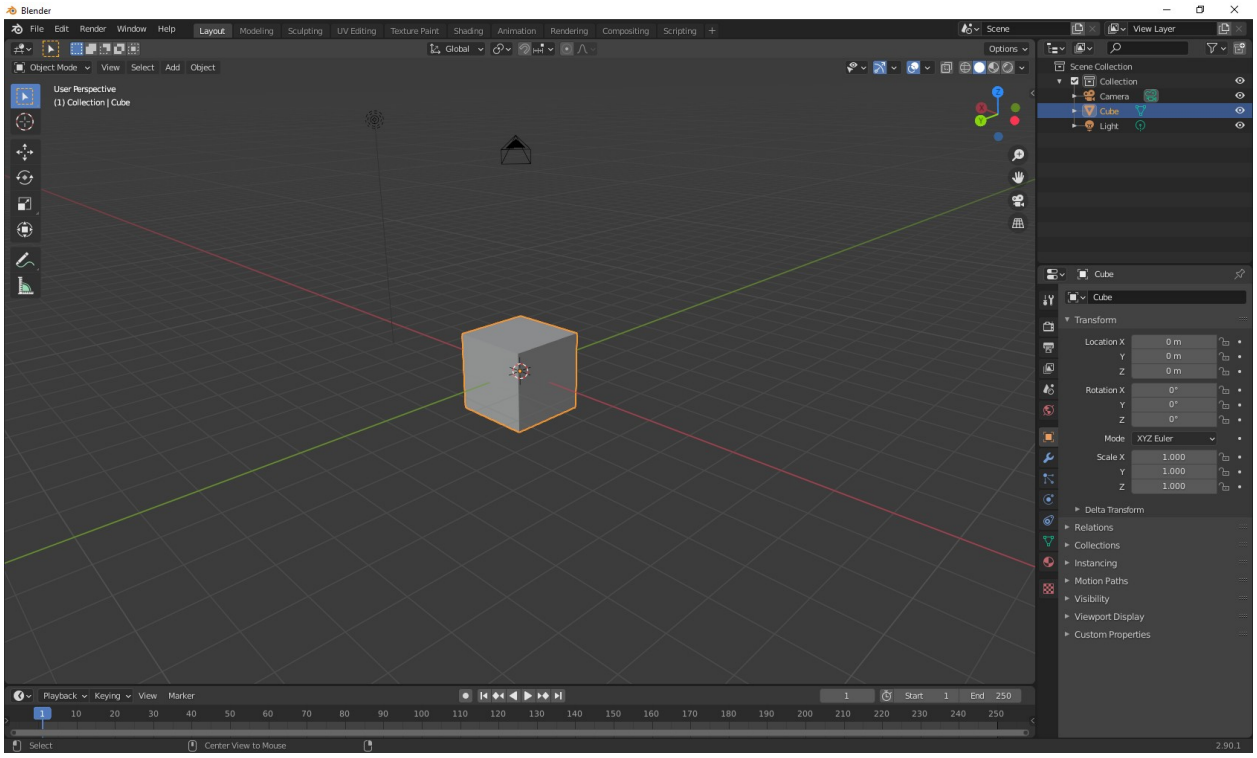

Рис. 66 *інтерфейс Blender*

Після вибору програми для створення моделей було створено набір стін 2.5 м в висоту та 2 метри в ширину, а також було створені додаткові елементи, такі як:

- звичайна стіна;
- стіна з вікном;
- балкон;
- вхід до підїзду;
- стіна з дверним отвором (для кімнат);
- стіна для підвалу з вікном;
- огорожа для даху.

Далі було створено моделі підлоги, стелі, сходів, а також елементи зливної системи для дощу, їх ви можете побачити на рисунках 67-72.

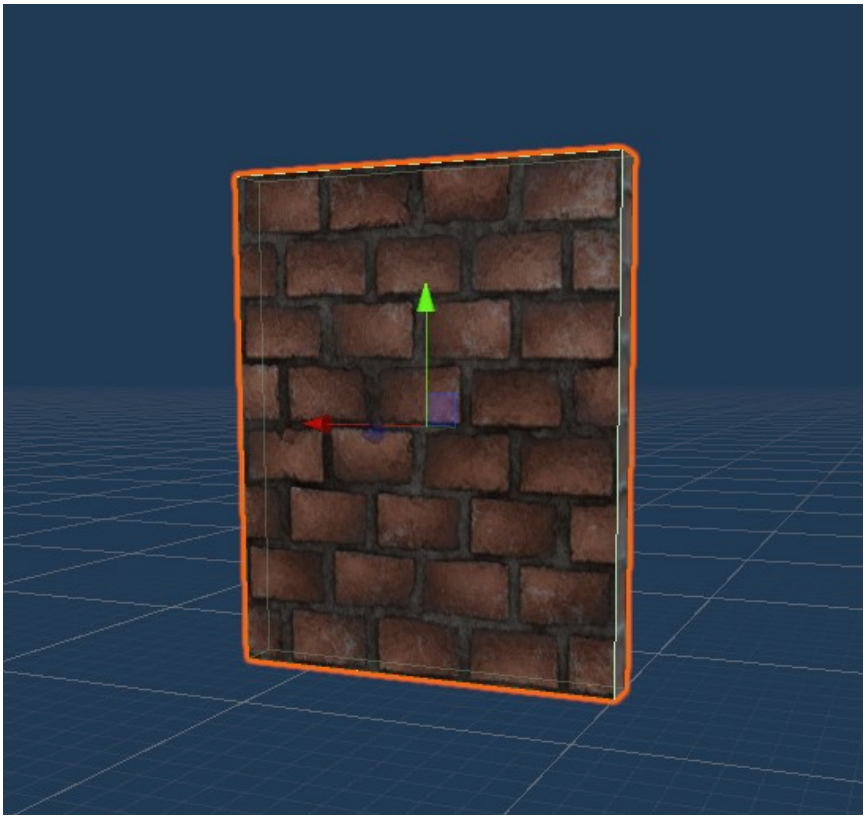

Рис. *67 Звичайна стіна*

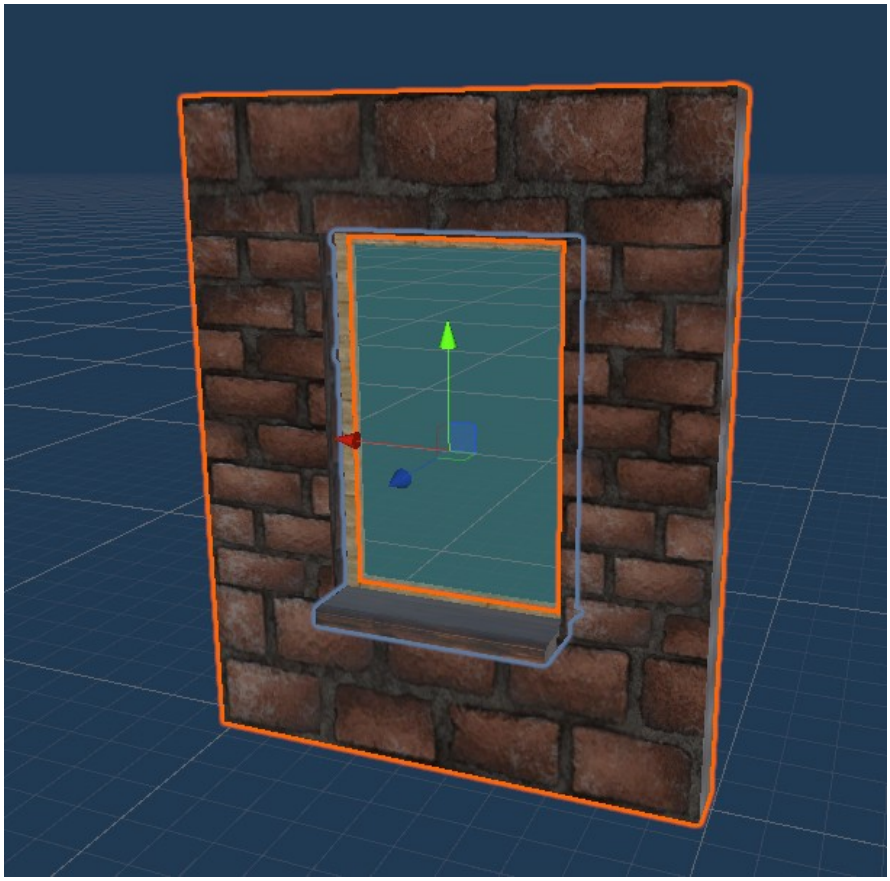

Рис. *68 Стіна з вікном*

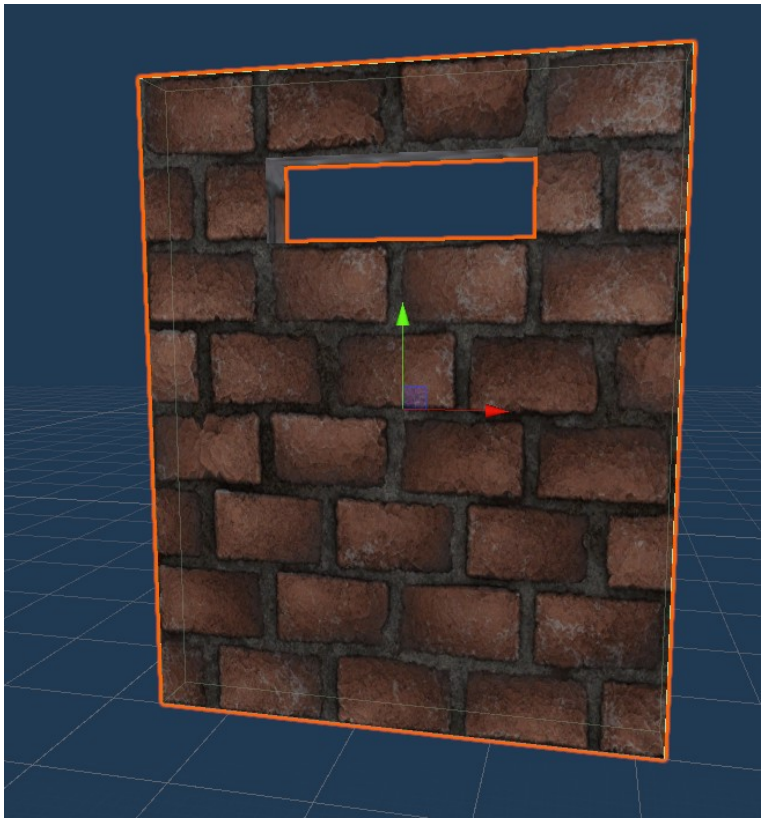

Рис. *69 Підвальна стіна*

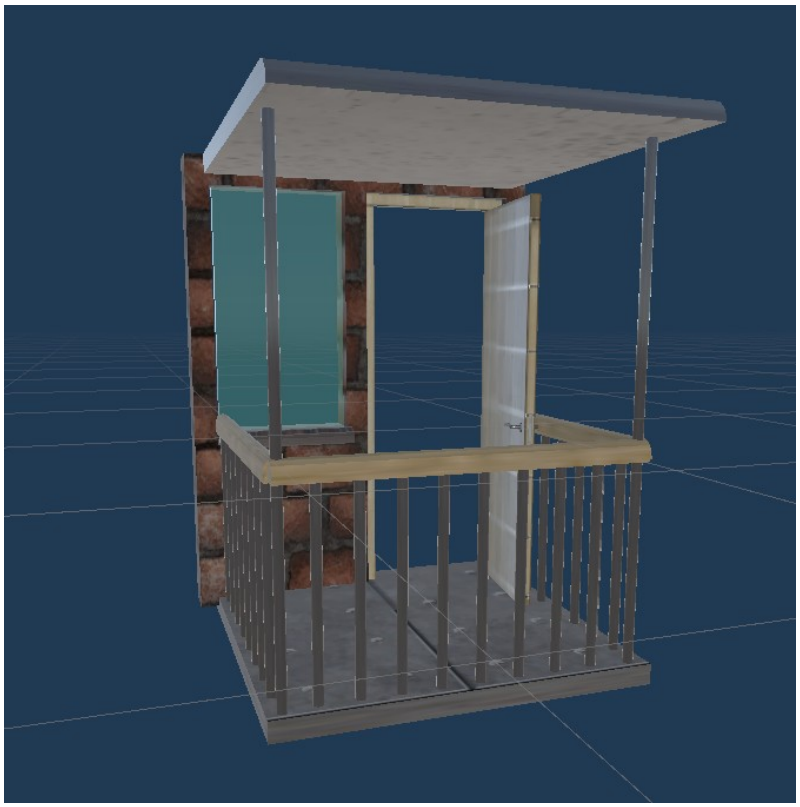

Рис. *70 Балкон*

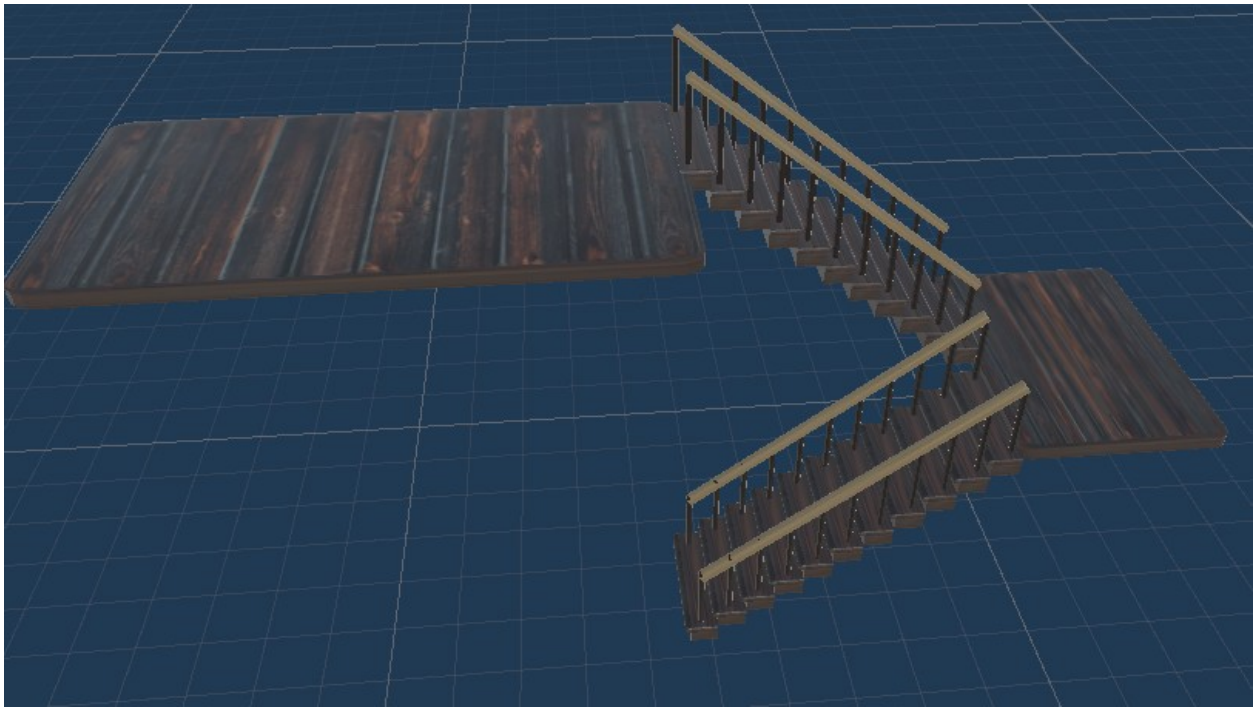

Рис. *71 Сходи*

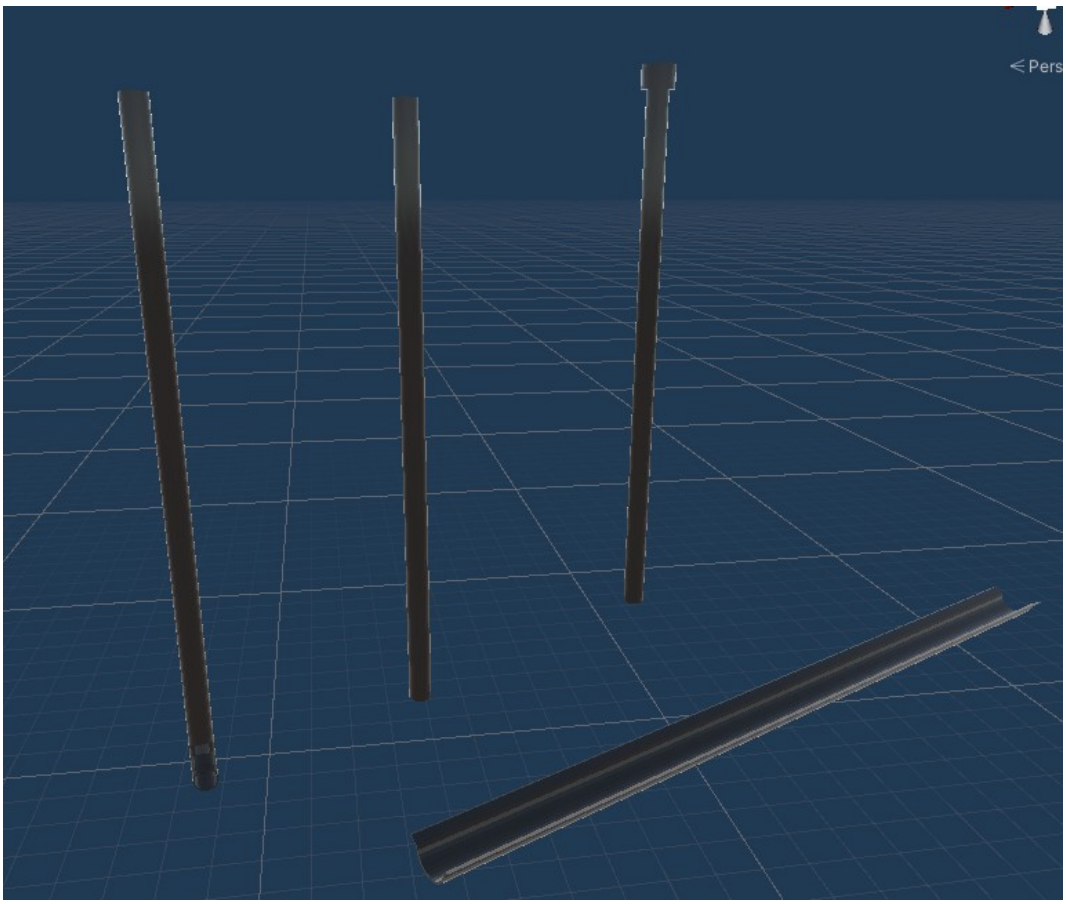

Рис. *72 Труби зливної системи*

Робота з UV-координатами

UV розгортка — процес в 3D моделюванні який полягає в накладанні двовимірного зображення на тривимірну модель. Літерами U і V позначають осі координат площини розгортки, оскільки літери X, Y і Z використовуються для позначення просторових координат.

Для UV розгортки зазвичай використовуються будь-яка текстура, наприклад текстура стіни або дерева. Зазвичай для стін та дерев'яної підлоги або дахів використовують безшовні текстури, які можуть продовжувати самі себе в будь-якому напрямі. Приклад безшовну текстури цегляної стіни можна побачити на рисунку 73.

Рис. *73 Приклад безшовної текстури*

Процес налаштування UV-координат в Blender має автоматичне налаштування UV, але цей процес інколи працює не дуже добре, тому налаштува-

ння граней стін інколи доводилося робити вручну. На рисунку 74 зображено вікно редактору для корегування UV-координат моделі.

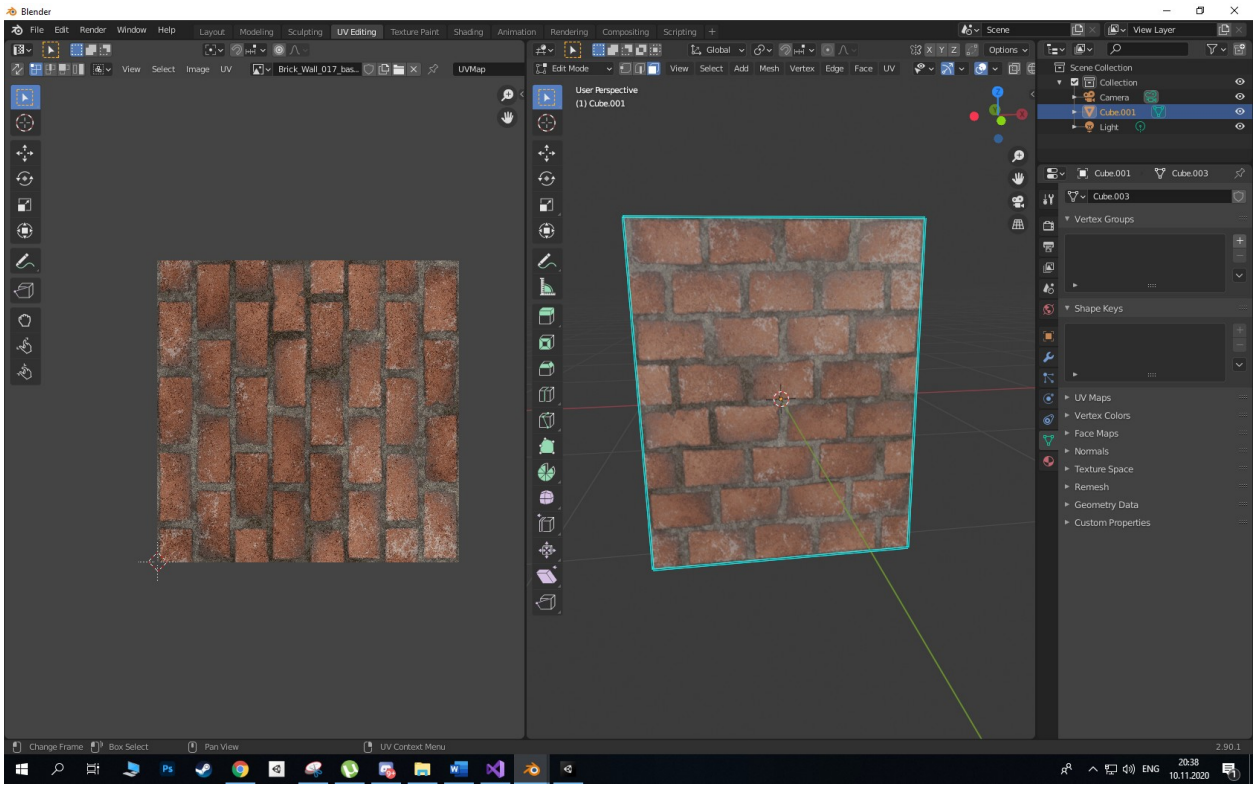

Рис. *74 редактор для роботи з UV текстурою*

Створення 3D будівлі на основі 2D плану

Після закінчення процесу побудови 2D плану приміщення відбувається відображення у 3D. Так як програмна система вже згенерувала плани поверхів, кімнат, квартир у вигляді списків стін у форматі: початок стіни, кінець стіни, центр стіни у 2D координатах. Процесс 3D відображення не є такою важкою задачею, але вона має досить багато нюансів.

Коректний імпорт та направлення моделей в Unity

Для коректного відображення у 3D усі моделі при імпорті в Unity повинні бути направлені в одному напрямку, бо в залежності від напряму стіни її необхідно повертати на 90, 180, 270 градусів відповідно і без цього коректного відображення не відбудеться. Коректний імпорт моделі повинен мати таке розташування світових векторів: синій (forward) вектор направлений у внутрішню сторону моделі, червоний (right) вектор направлений

праворуч від моделі, зелений (up) вектор направлений в гору моделі. На рисунку 75 зображено імпортовану модель.

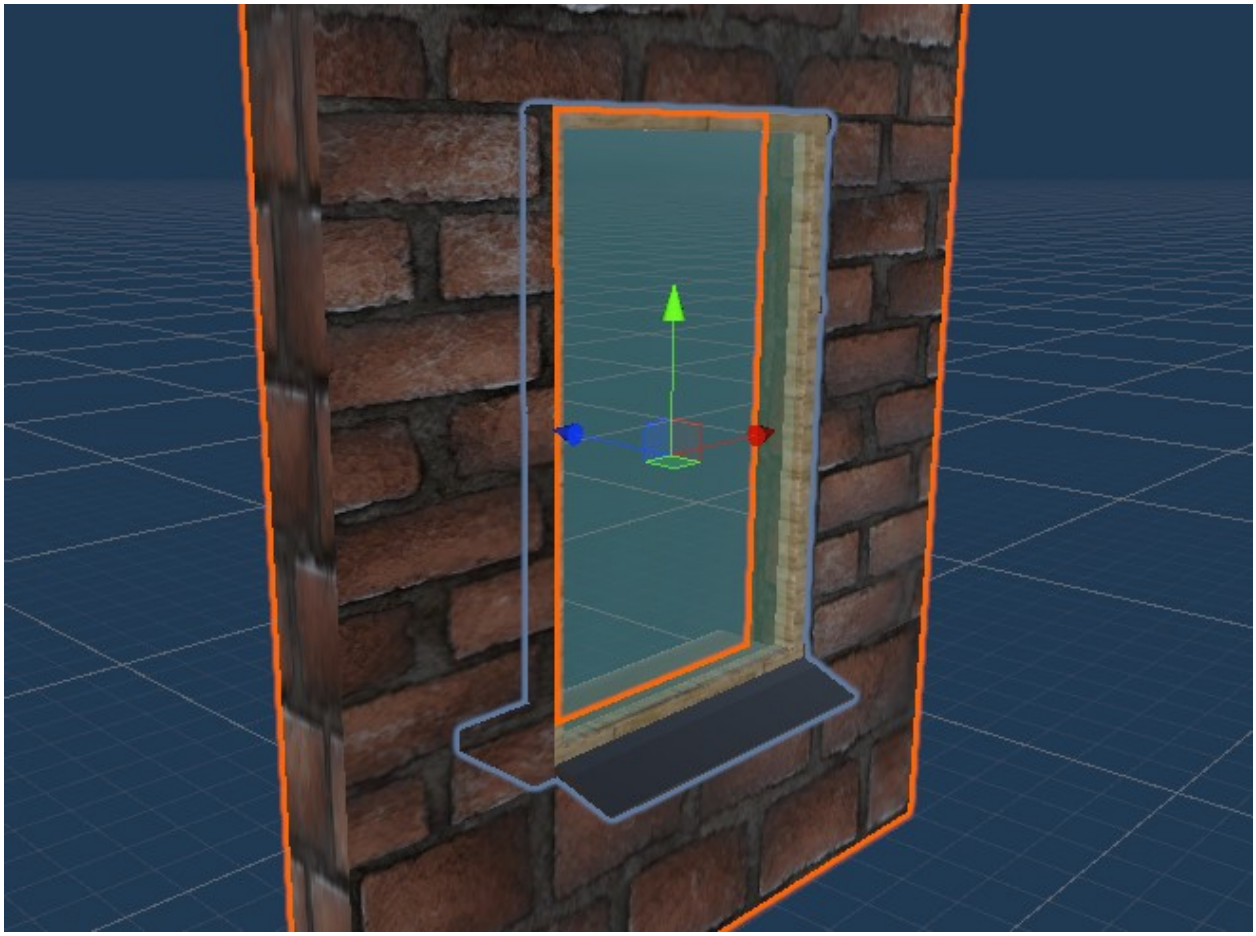

Рис. *75 Коректний імпорт моделі з правильним розташуванням векторів моделі.*

Взагалі яким чином відбувається поворот моделі? Ця проблема вирішується за допомогою векторної алгебри, бо за допомогою скалярного векторного добутку можна дізнатися, чи дивлять ці два вектори в одному напрямку чи ні. Якщо отриманий векторний добуток дорівнює одиниці то ці два вектори дивляться в одному напрямі, якщо мінус один то в різні. За допомогою цієї особливості скалярного векторного добутку дуже легко знайти поворот моделі, що наведено в лістингу 5.

Лістинг 5 — Функція пошуку кута повороту стіни

```
protected float FindWallRotation(PartOfWall wall, out float
xOffset, out float yOffset, float maxOffset = 0.1f, bool
moveDiractionInside = true)
{
            var point1 = new Vector3((float)wall.V1.X, 0,
(float)wall.V1.Y);
            var point2 = new Vector3((float)wall.V2.X, 0,
(float)wall.V2.Y);

            var vector = Vector3.ClampMagnitude(point2 -
point1, 1);

            float direction = 1;

            if (!moveDiractionInside)
                direction = -1;

            if (Vector3.Dot(vector, Vector3.forward) == 1)
            {
                yOffset = 0f;
                xOffset = maxOffset * direction;
                return 90;
            }

            else if (Vector3.Dot(vector, Vector3.forward)
== -1)
            {
                yOffset = 0f;
                xOffset = -maxOffset * direction;
                return 270;
            }

            else if (Vector3.Dot(vector, Vector3.right) ==
1)
            {
                yOffset = -maxOffset * direction;
                xOffset = 0;
                return 180;
            }

            else
            {
                yOffset = maxOffset * direction;
                xOffset = 0;
                return 0;
            }}
```

З цього лістингу можна побачити, що в залежності від векторного добутку можна визначити поворот стіни:

- якщо forward вектор і вектор стіни направлені в одному напрямку то поворот 90 градусів;
- якщо forward вектор і вектор стіни направлені в різні напрямки то поворот 270 градусів;
- якщо right вектор і вектор стіни направлені в одному напрямку то поворот 180 градусів;
- якщо right вектор і вектор стіни направлені в різні напрямки то поворот 0 градусів.

Проблема розташування сходів

Розташування сходів та продовження цих сходів на наступних поверхах має свої особливості та обмеження. По-перше, розмір моделі повинен підводитися під критерії 2 метри в ширину і 6 метрів в довжину та 5 метрів в висоту. Це необхідно, щоб сходи могли зістикуватися при додаванні їх на наступних поверхах. Самі сходи можна побачити на рисунку 62. Також існує проблема розташування сходів у кімнаті. Ця проблема вирішується завдяки знаходженню центру кімнати та розташування вже в ній сходів.

Цей елемент є також модульним, як стіни, і може бути змінений на розсуд користувач.

3.3.4 Результат

В результаті була розроблена система процедурного генерування будинку, яка здатна генерувати приміщення з досить великою варіативності. При використанні програмної системи її параметри можна змінювати від таких загальних налаштувань як кількість поверхів, площа, кількість під’їздів до найдрібніших параметрів, таких як налаштування росту кімнати, матеріали поверху, квартири, кімнати, а також можливість змінювати усі моделі такі як: стіни, ліфт, сходи, двері.

Процес генерування будівлі займає деякий час і це може дуже сильно вдарити по швидкодії гри, тому розроблена програмна система більше підходить до застосування в режимі редагування гри. Згенероване приміщення зображено на рисунках 76-80.

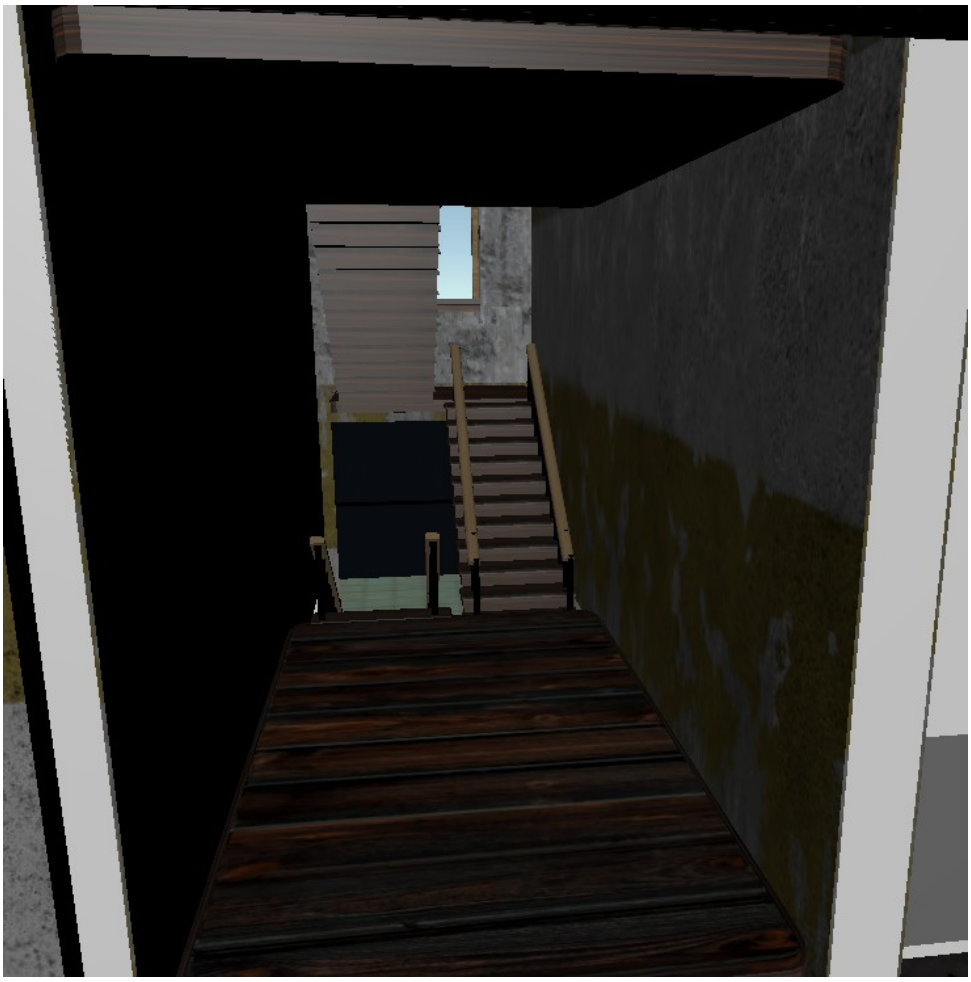

Рис. *76 Згенерована кімната зі сходами*

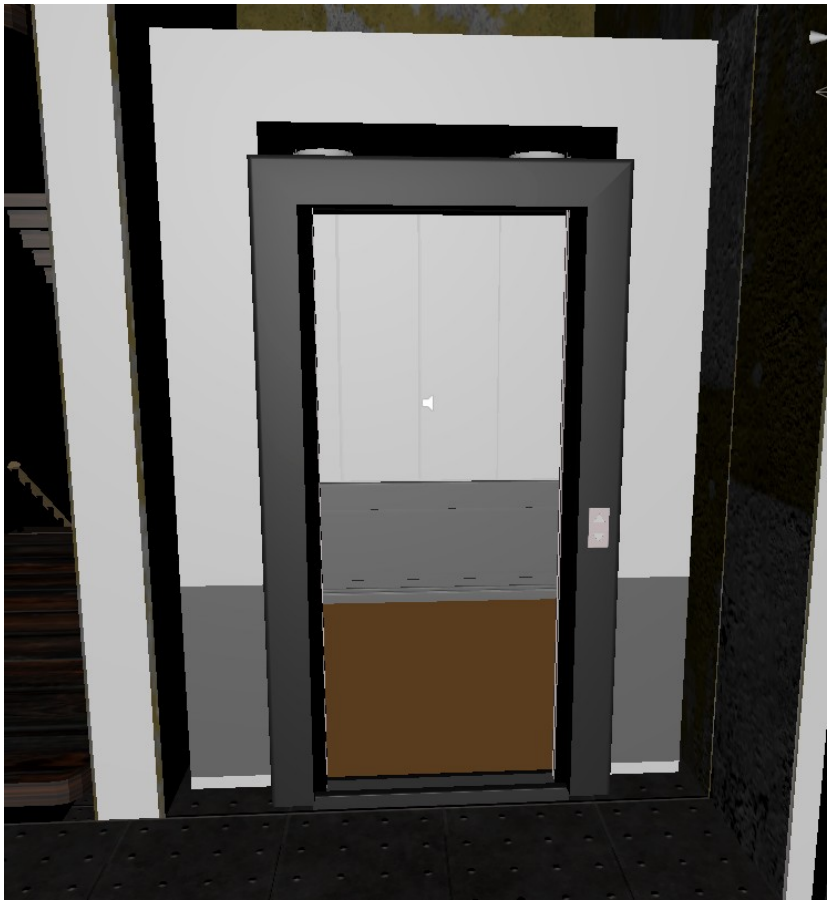

Рис. *77 Згенерована область для ліфту*

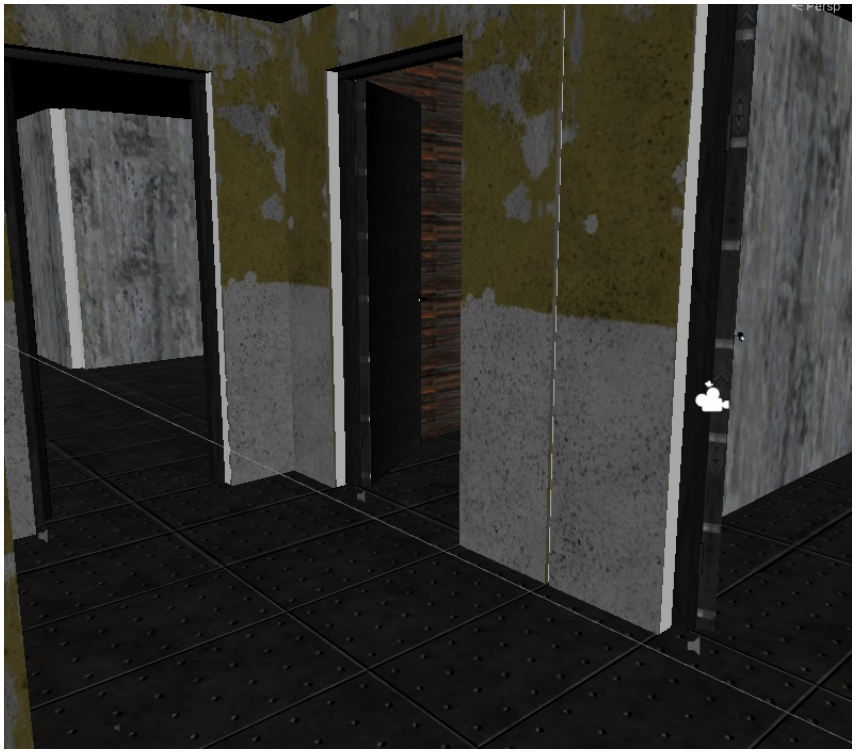

Рис. *78 Згенерована кімната коридор з дверима, які ведуть до кварти*

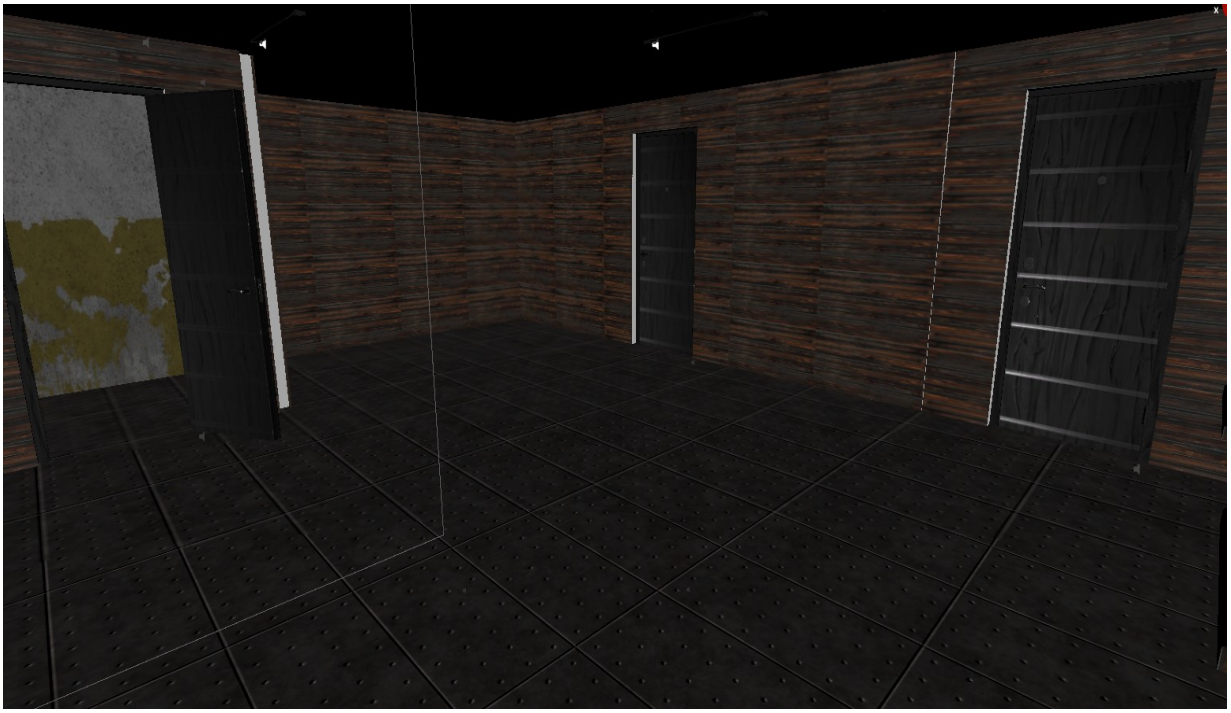

Рис. *79 Одна із згенерованих кімнат квартири*

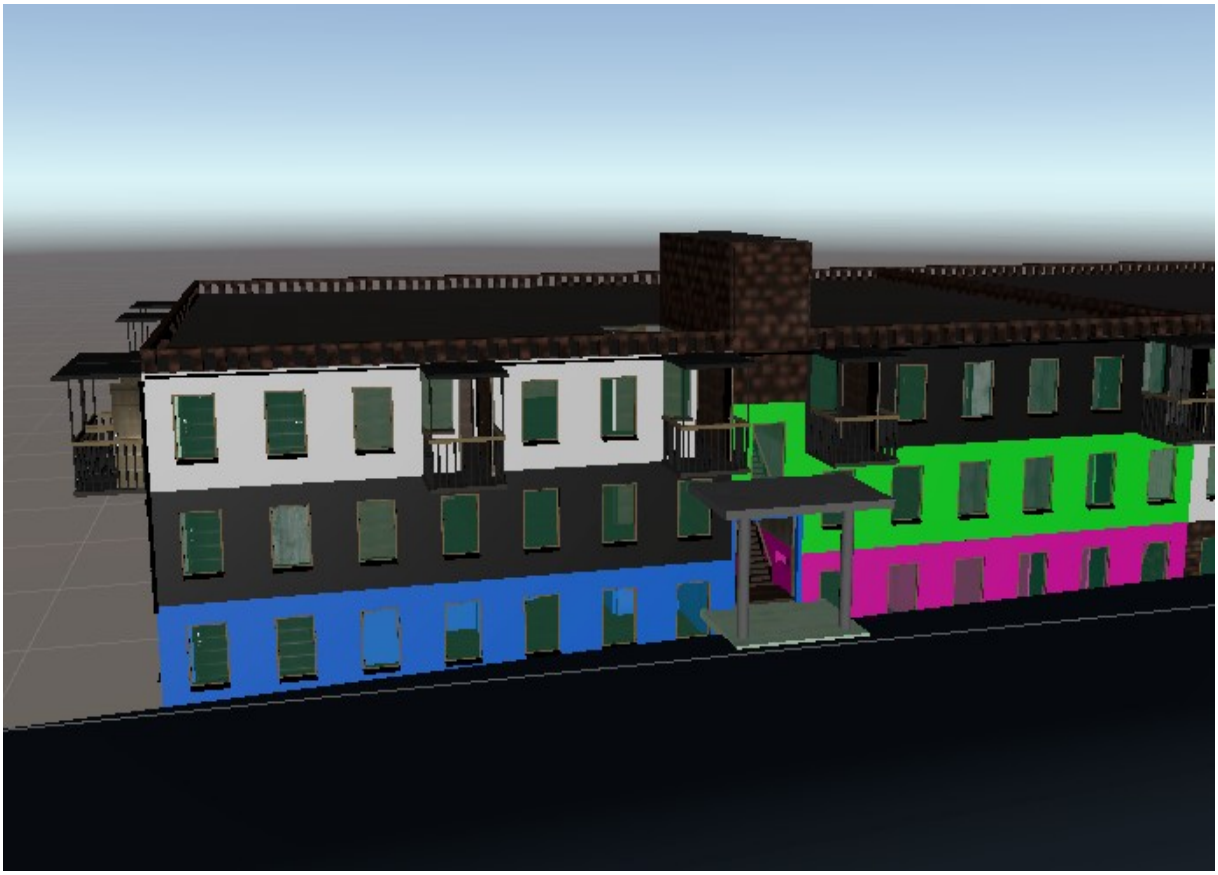

Рис. *80 Приклад можливості налаштування матеріалів стін для квартир*

РОЗДІЛ 4 АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ ПРОЦЕДУРНГО ГЕНЕРУВАННЯ ПРИМІЩЕНЬ

4.1 Проблема оптимізації рішення

Реалізація генерації процедурної будівлі за допомогою модульних стіна має недоліки в плані оптимізації, бо на ігровій сцені створюється багато графічних об’єктів у вигляді стін, підлоги, сходів, вікон і так далі. Так як на сцені дуже велика кількість об’єктів то слабкі комп’ютери починають сильно завантажуватись. Для вирішення цієї проблеми треба шукати шляхи оптимізації.

Перший дуже ефективний варіант оптимізації – це об’єднання меш-сіток об’єктів. Цей метод постійно використовується в процедурній генерації. Чому ж цей метод такий ефективний? Насправді відповідь дуже проста. Ігровий двигун Unity кожен кадр гра відображає всі візуальні візуальний об’єкти, які є в полі зору гравця. Та коли цих об’єктів дуже багато, то частота кадрів може різко знизитись. А об’єднання декількох подібних об’єктів в один зменшує навантаження ігрового двигуну і комп’ютеру в цілому.

За допомогою цього методу усі моделі стін, вікон, підлоги об’єднуються в одну меш-сітку зі збереженням матеріалів на них. Реалізація даного методу була взята з магазину Asset store від Unity, назва асету Mesh combiner [9]. Використання цього підходу при великій кількості об’єктів дозволило в даному випадку збільшити частоту кадрів майже у 8 разів, з 50 кадрів до майже 400 кадрів на секунду. Підтвердження можна побачити на рисунках 81-88 та у таблиці 2. Тестування було проведе на комп’ютері з такими характеристиками:

- операційна система Windows 10 x64;
- відео карта Geforce GTX 1660 super;
- процесор AMD Ryzen 7 2700x Eight-core processor 3.7 GHz;
- 16 Gb оперативної пам’яті.

Як додаток до оптимізації меш-сіток – це правильне моделювання 3D об’єктів, без зайвих точок на моделі, та з мінімізацією площін на ній. В Unity є обмеження на об’єднання меш-сіток, а саме починаючи з версії Unity 2017.3 або вище Unity може об’єднувати до 4 мільярдів мешів в один при умові, що кількість точок в одному меш-об’єктку менша за 65535. Раніше максимум був 65535 точок для об’єднання одного мешу.

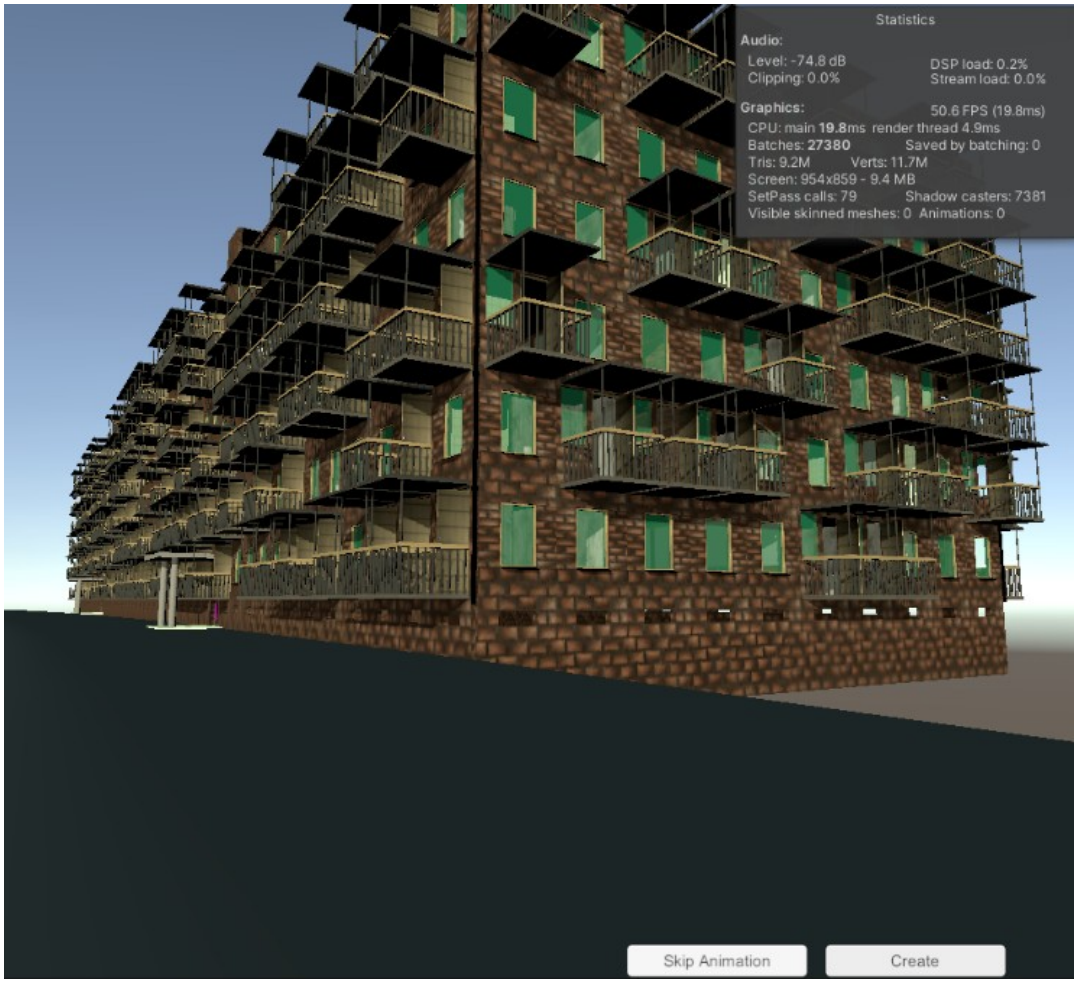

Рис. *81 Частота кадрів без об’єднання мешів 50 кадрів на секунду*

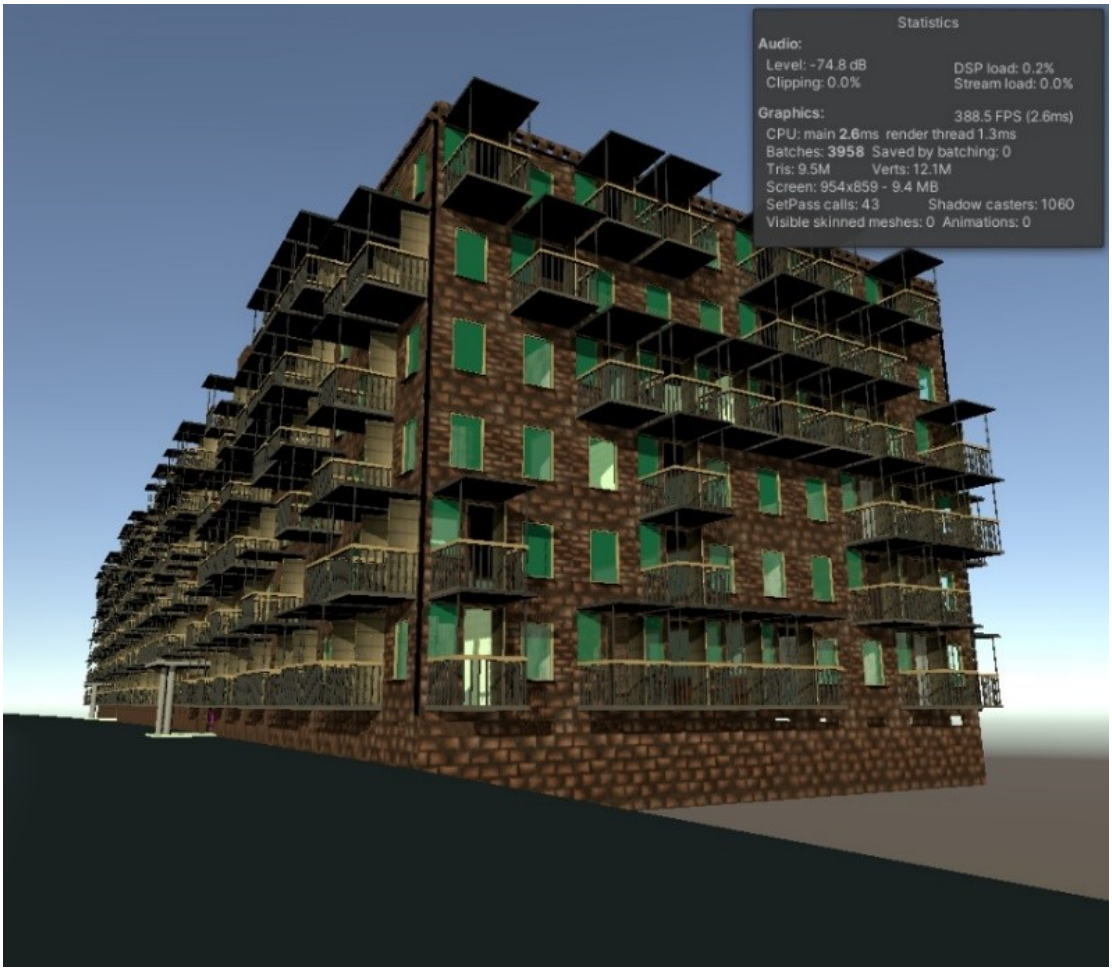

Рис. *82 Частота кадрів після об'єднання мешів ~400*

Другу проблема, яку має розроблена програмна система — це швидкодія створення будівлі. В її аналізі дуже допомагає такий вбудований інструмент як Unity profiler, рисунок 83. Цей інструмент насправді не показує повноцінну картину, бо він по-перше, веде запис в режимі редактору, а не в кінцевій грі. По-друге він сам уповільнює всю програму, а якщо він працює в deep режимі, то програма сповільнюється у два рази! Але цей недолік не заважає аналізувати швидкодії окремих функцій програми, бо картина хоча не повноцінна, але з неї можна побачити яка функція з усіх бере на себе дуже багато часу.

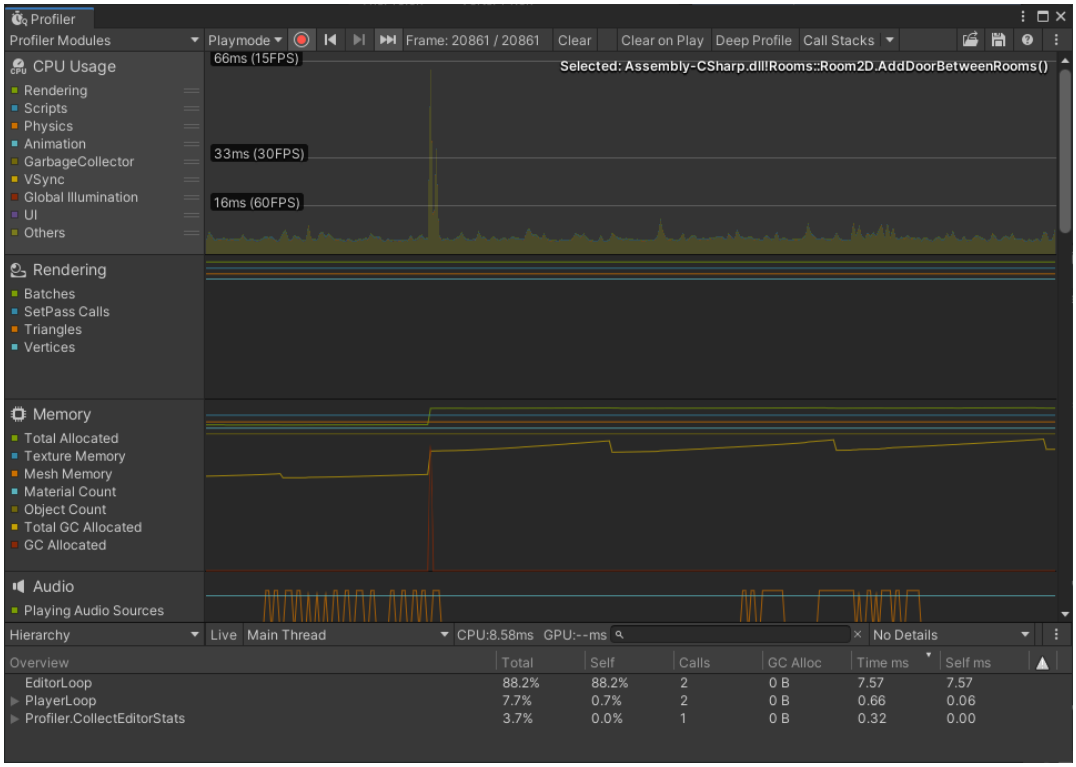

Рис. *83 Вікно Unity profiler для аналізу швидкодії*

На рисунку 84 можна побачити скільки часу витрачається на створення 3D об’єктів та поєднування маш-сшток при площі 12000 м2. Загалом у записі в deep режимі Unity profiler час склав 6119 мілісекунди або 6.1 секунди. Невлику частину часу для генерування приміщення витрачається на візуалізацію кімнати, бо створення більшості 3D об’єктів та робота з ними знаходиться на рівні кімнат. Більшість часу з усього витрачається на інстанціювання моделей на сцену та об’єднання їх в єдину меш-сітку, це можна побачити з функції *VisualizeWalls*, а також з функції Unity *AddComponent*, яка викликається автоматично при створенні ігрового об’єкту. Також об’єднання меш-об’єктів займає приблизно 1.5 секунди. На рисунку 85 можна побачити скільки часу займає побудова 2D плану будинку при записі у deep режимі профайлеру видається цифра у 4000 мілісекунд, хоча при записі в звичайному режимі цифра становить 500 мілісекунд (Рис. 86).

Building3D.Visualize()	56.7%	0.0%	1	125.5 MB	6119.10	0.00
ApartamentPanelHouse3D.InitializeSpaces3D()	56.7%	0.0%	1	125.5 MB	6118.81	0.10
APH_Entrance3D.Visualize()	53.7%	0.0%	2	115.6 MB	5790.81	0.04
List`1.ForEach()	53.7%	0.0%	2	115.6 MB	5790.67	0.01
APH_Entrance3D.<Visualize>b_2_0()	53.7%	0.0%	16	115.6 MB	5790.57	0.05
APH_Floor3D.Visualize()	53.7%	0.0%	16	115.6 MB	5789.80	1.36
Room3D.Visualize()	53.6%	0.0%	336	115.5 MB	5779.68	3.22
GameObject.AddComponent()	23.2%	0.0%	672	52.7 KB	2502.65	0.74
Room3D.VisualizeWalls()	14.2%	0.0%	336	5.6 MB	1536.29	3.06
MeshCombiner.CombineMeshes()	12.9%	0.0%	336	109.0 MB	1396.20	1.97
Room3D.VisualizeCeiling()	1.4%	0.0%	318	330.7 KB	154.85	0.64
Room3D.VisualizeFloor()	1.4%	0.0%	318	330.7 KB	152.53	0.57

Рис. *84 Створення 3D моделей будинку та поєднування мешів при записі у deep режимі*

ApartamentPanelHouse3D..ctor()	39.0%	0.0%	1	85.7 MB	4092.78	0.04
ApartamentPanelHouse2D..ctor()	39.0%	0.0%	1	85.7 MB	4088.05	0.02
ApartamentPanelHouse2D.Create2DSpaceInternal()	38.6%	0.0%	1	84.7 MB	4042.67	0.01
ApartamentPanelHouse2D.CreateFloorPlans()	38.5%	0.0%	1	84.7 MB	4039.44	0.06
Premises2D.Create2DSpace()	38.5%	0.0%	2	84.7 MB	4039.01	0.00
Entrance2D.Create2DSpaceInternal()	38.5%	0.0%	2	84.7 MB	4038.85	0.12
Premises2D.Create2DSpace()	38.5%	0.0%	16	84.7 MB	4038.32	0.01
APH_BaseFloor2D.Create2DSpaceInternal()	33.5%	0.0%	12	69.1 MB	3511.42	0.07
APH_BaseFloor2D.AddDoorForRooms()	27.1%	0.0%	12	58.9 MB	2841.49	0.08
List`1.ForEach()	27.0%	0.0%	24	58.7 MB	2836.51	0.04
APH_BaseFloor2D.<AddDoorForRooms>b_9_2()	21.0%	0.0%	48	39.1 MB	2200.00	0.05
Premises2D.Create2DSpace()	20.6%	0.0%	48	38.7 MB	2156.90	0.02
Flat2D.Create2DSpaceInternal()	20.6%	0.0%	48	38.7 MB	2156.77	0.14
FlatPlanProcessor2D.CreatePlan()	16.4%	0.0%	48	29.2 MB	1726.22	0.04
FlatPlanProcessor2D.AddDoorsToRooms()	12.1%	0.0%	48	19.8 MB	1275.24	4.90
GrowthProcessor.GrowthOfRooms()	4.3%	0.0%	48	9.3 MB	450.62	0.06
Mono.JIT	0.0%	0.0%	1	112 B	0.30	0.24
BasePlanProcessor2D.get_growthProcessor()	0.0%	0.0%	48	0 B	0.00	0.00
FlatPlanProcessor2D..ctor()	3.1%	0.0%	48	8.8 MB	325.86	0.15

Рис. *85 Створення 2D плану у deep режимі*

PlayerLoop	96.7%	0.0%	2	88.0 MB	533.47	0.07
Update.ScriptRunBehaviourUpdate	96.4%	0.0%	1	88.0 MB	532.21	0.00
RenderPipelineManager.DoRenderLoop_Internal()	0.0%	0.0%	1	0 B	0.43	0.09
FixedUpdate.PhysicsFixedUpdate	0.0%	0.0%	1	0 B	0.29	0.00
updateScene.Invoke	0.0%	0.0%	1	0 B	0.08	0.00

Рис. *86 Створення 2D плану в звичайному режимі*

Так як же оптимізувати таку довгу побудову процедурно створеного приміщення? По-перше, застосувати партерн проектування Object pool. Його суть у хешуванні деякої кількості об’єктів у оперативній пам’яті, а потім при потребі використовувати їх. Цей шлях дуже ефективний коли йде дуже багато запитів до жорсткого диску, бо операції з диском можуть викликати великі затримки по часу і завантажують центральний процесор. Цей варіант буде ефективно працювати, бо меш-сітки реальних об’єктів, які були створенні перед об’єднанням не знищуються і їх можна потім використовувати повторно.

У таблиці 2 зображено порівняння з створеною мешу-сітки та без неї. На рисунках 87 та 88 дані таблиці зображені у вигляді графіку.

Проаналізувавши ці данні можна прийти до думки, що час створення будівлі для об’єднаного мешу дуже великий і збільшується лінійно порівняно з варіантом без об’єднання в єдину меш-сітку. У випадку об’єднання в єдину меш-сітку спостерігається більша частота кадрів.

Таблиця *2 Порівняння з об’єднаною меш-сіткою та без неї*

Площа. M^2	Кадри. к/с	Час. с.	Кадри. к/с	Час. с.
	Тести з об’єднаною меш-сіткою		Тести без об’єднаною меш-сіток	
1800	~600	1	~340	0.15
3600	~372	1.8	~140	0.22
5400	~270	5	~81	0.39
7200	~203	7.34	~66	0.56
9000	~155	10.17	~54	0.78
10800	~150	13.15	~42	0.91
12400	~130	16.11	~36	1.24

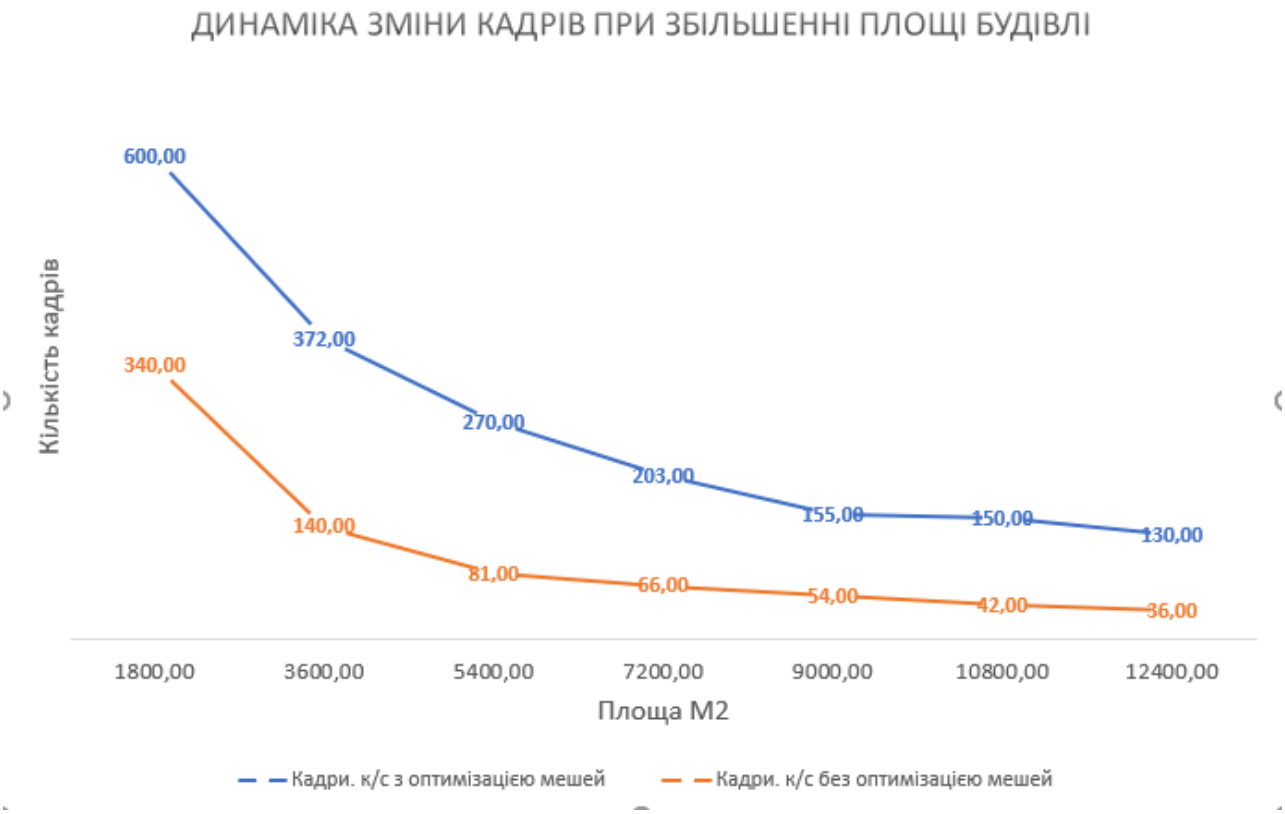

Рис. *87 Графік динаміки зміни кадрів при різній площі будівлі з об'єднаної меш-сітки та без об'єднання*

Рис. *88 Графік зміни часу створення при зміні площі будівлі*

ВИСНОВКИ

1. Досліджено та встановлено актуальність використання процедурного генерування у ігровій галузі. Проведено теоретичні та практичні дослідження щодо специфіки даної теми та її ключових понять.

2. Досліджено та проаналізовано існуючі рішення генерування будівель, а також розібрано їх недоліки та переваги.

3. На основі проведеного аналізу методів генерування будинків створено інструмент для процедурного генерування планів будинків з їх тримірним відображенням. В розробленій програмній системі використано генерування приміщення на основі вирощування кімнат, а також відтворено будівлю багатоповерхівки жилого типу з квартирами.

4. В розроблену програмну систему додана можливість корегувати різні налаштування, від таких простих, як площа будівлі, до зміни варіантів стін, дверей, дахів. В реалізованій програмній системі всі моделі будинку, такі як стіни та вікна можуть бути змінені на створені користувачем. Дана особливість системи відкриває можливості для генерування будівлі як конструктора з частин.

5. Досліджено та проаналізовано різні методи оптимізації відображення будівлі у 3D. В розробленій системі використано такі методи оптимізації: об’єднання елементів будинку в єдину меш-сітку, коректне створення ігрових модульних частин будинку та використання патерну проектування Poolobject для повторного використання у побудові будинків.

6. Розроблено програмну систему здатну генерувати будівлі у 3D з внутрішнім плануванням поверхів.

Отже, процедурне генерування – це інструмент, який може пришвидшити розробку великої кількості ігрових проектів і зменшити кількість витрачених людино-години на створення цифрового контенту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Julian Togelius, Emil Kastbjerg, David Schedl, Georgios N. *Yannakakis. What is Procedural Content Generation?: Mario on the Borderline :* Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games. New York. 2011. C. 1-6.

2. Shaker, Noor, Togelius, Julian, Nelson, Mark J. : Procedural Content Generation in Games : Springer, 2016. 218 с.

3. Хокінг, Джозеф. Unity в дії. Мультиплатформенна розробка на C# : Пітер, 2016. 336 с.

4. Карачун О. О., Гульчук Г. Г. Російсько-український математичний словник / за ред. В. Я. Карачун. Київ : Вища школа, 1995. 40 с.

5. Dahl A.. Procedural Generation of Indoor Environments [master's work]. Göteborg: Chalmer University of Technology; 2008.

6. Camazzato D. A.. Method for growth-based procedural floor plan generation [master's work]. Rio Grande: Pontifical Catholic University of Rio Grande do Sul; 2015.

7. Офіціальний сайт розробників Blender. URL: https://www.blender.org/ (Дата звернення 27.11.2020)

8. Офіціальний сайт 3dsMax. URL: https://www.autodesk.com/products/3ds-max/overview (Дата звернення 27.11.2020).

9. Реалізація комбінування мешів. URL: https://assetstore.unity.com/packages/tools/modeling/mesh-combiner-157192#reviews (Дата звернення 27.11.2020).

10. Huber, Stefan *The Topology of Skeletons and Offsets* : Proceedings of the 34th European Workshop on Computational Geometry (EuroCG'18). Berlin, Germany, March 21-23, 2018. P. 115-120.

11. Tarjan, Robert E., Van Wyk, Christopher J. An O(n log log n)-time algorithm for triangulating a simple polygon, *SIAM Journal on Computing.* 1988. Vol. 17, No 1. P. 143–178.

12. Офіційний сайт розробників Unity3D. URL: https://unity.com/ru (Дата звернення 27.11.2020).

13. Shaker, Noor, Togelius, Julian, Nelson, Mark J. Procedural Content Generation in Games : monograph Springer : 2018. 218 c.

14. Відкрита реалізація алгоритму Straight skeleton: сайт GitHub. URL: https://github.com/reinterpretcat/csharp-libs/tree/master/straight_skeleton (Дата звернення 27.11.2020).

15. Aichholzer Oswin, Aurenhammer Franz, Alberts David, Gärtner, Bernd. A novel type of skeleton for polygons. *Journal of Universal Computer Science*. 1995. №1. С. 752–761.

16. Aichholzer Oswin, Aurenhammer Franz. *Straight skeletons for general polygonal figures in the plane* : Proc. 2nd Ann. Int. Conf. Computing and Combinatorics. Springer-Verlag, 1996. С. 117–126.

17. Доповідь розробників The Witcher 3 про використання процедурної генерації в розробці гри : презентація. URL: https://www.gdcvault.com/play/1020197/Landscape-Creation-and-Rendering-in (Дата звернення 27.11.2020).

18. Доповідь розробників Horizon Zero Down та їх процедурна генерація: презентація. URL: https://www.gdcvault.com/play/1020197/Landscape-Creation-and-Rendering-in (Дата звернення 27.11.2020).

19. Доповідь розробників Marvel’s Spider-Man та їх процедурна генерація: презентація. URL: https://www.youtube.com/watch?v=4aw9uyj9MAE&ab_channel=GDC (Дата звернення 27.11.2020).

20. Andersson S.. Detailed Procedurally Generated Buildings [master's work]. Sweden: Linköping University; 2019.

21. Pádua L.. Procedural Modeling of Buildings Composed of Arbitrarily-Shaped Floor-Plans: Background, Progress, Contributions and Challenges of a Methodology Oriented to Cultural Heritage. *Journals Computers*. Volume 8, Issue 2. 2019. С. 1-10.

22. Kužel V.. Procedural building reconstruction from building outlines [master's work]. Prague: Charles University; 2018.

23. I. da Silveira, D. Camozzato, F. Marson and other. *Real-time Procedural Generation of Personalized Façade and Interior Appearances Based on Semantics*. Conference 14th Brazilian Symposium on Computer Games and Digital Entertainment. Brazilian, 2015. С 1-8.

24. Bradley B. Towards the Procedural Generation of Urban Building Interiors [master's work]. England: The University of Hull; 2005.

25. T.Tutenel, R. Bidarra, R. M. Smelik, Klaas Jan de Kraker. *Rule-based layout solving and its application to procedural interior generation*. Conference: Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation. The Netherlands, 2009 С. 1–11.

26. T.Adão, F. Pereiraa, E. Peresb, L. Magalhãesb. *Procedural Generation of Traversable Buildings Outlined by Arbitrary Convex Shapes*. Procedia Technology. 2014. С. 310–321.

27. Стаття порівняння інді та ААА розробників ігор. URL: https://www.windowscentral.com/indie-vs-aaa-which-type-game-you (Дата звернення 27.11.2020).

28. Волік С.О., студент магістратури ФЕЕІТ ІІ ЗНУ. Наук. кер.: к.ф.-м.н., доц. Міхайлуца О.М. «Проблеми процедурного проектування приміщень». Збірник наукових праць студентів, аспірантів і молодих вчених «Молода наука-2020» : у 5 т. Запорізький національний університет. Запоріжжя: ЗНУ, 2020. Т.5. С. 159.

29. Волік С.О., студент магістратури ФЕЕІТ ІІ ЗНУ. Наук. кер.: к.ф.-м.н., доц. Міхайлуца О.М. «Комп’ютерна система генерації тривимірного ігрового світу». Збірник наукових праць студентів, аспірантів і молодих вчених «Молода наука-2020» : у 5 т. Запорізький національний університет. Запоріжжя: ЗНУ, 2020. Т.5. С. 79-80.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Волік Сергій Олександрович , студент _2_ курсу, форми навчання денної, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти sp115-05@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «Комп'ютерна система генерації тривимірного ігрового світу» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений/ознайомлена;

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 30.11.20 Підпис _____ ПІБ (студент) Волік Сергій Олександрович

Дата 30.11.20 Підпис _____ ПІБ (науковий керівник) Олена Миколаївна Міхайлуца