

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)
на тему:

**«Застосунок для створення текстур за допомогою
ШІ та нанесення методом проєкції на модель»**

Виконав студент групи ІПЗм-24-1
ІГНАТЕНКО Самуїл Ігорович

Керівник роботи:
ЧИЖМОТРЯ Олексій Володимирович

Рецензент:
ТОЛСТОЙ Ігор Анатолійович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«18» вересня 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **Ігнатенко Самуїл Ігорович**

Керівник роботи: **ЧИЖМОТРЯ Олексій Володимирович**

Тема роботи: **«Застосунок для створення текстур за допомогою ШІ та нанесення методом проєкції на модель»**,

затверджена Наказом закладу вищої освіти від **«18» вересня 2025 р. № 491/с**

Вихідні дані для роботи: **автоматизована генерація текстур для 3D-об'єктів на основі текстових описів та геометрії сцени; методи взаємодії з API генеративних нейромереж для обробки зображень; алгоритми проєкційного текстуровування для точного накладання 2D-зображень на 3D-поверхні; використання технології ControlNet для збереження структурної відповідності текстури формі об'єкта; методи запікання текстур з урахуванням перекриття геометрії.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О.В.	22.09.2025р.	22.09.2025р.
2	Чижмотря О.В.	22.09.2025р.	22.09.2025р.
3	Чижмотря О.В.	22.09.2025р.	22.09.2025р.

РЕФЕРАТ

Випускна кваліфікаційна робота магістра складається з опису дослідження методів автоматизованого текстурування тривимірних об'єктів, розробки алгоритмів адаптивної проєкції зображень, створення програмного засобу для середовища Blender та пояснювальної записки.

Пояснювальна записка до кваліфікаційної роботи складається з 77 сторінок, 7 ілюстрацій, 4 таблиць, 3 додатків та 31 літературного джерела.

Мета роботи - підвищення ефективності процесу створення та накладання текстур на тривимірні об'єкти шляхом розробки програмного засобу, що реалізує метод адаптивної проєкції зображень, згенерованих штучним інтелектом.

У роботі було описано цілі та поставлено задачі до досліджень та розробок. Було проаналізовано існуючі методи текстурування та аналогічні програмні рішення, виявлено їхні недоліки щодо роботи зі складною геометрією. Розроблено математичну модель та удосконалено алгоритм проєкційного накладання текстур («Smart Stencil») з використанням аналізу нормалей поверхні для мінімізації спотворень. Обґрунтовано вибір технологічного стеку, що включає Blender API, Python та локальний сервер генерації на базі Stable Diffusion WebUI Forge. Спроектовано та реалізовано клієнт-серверну архітектуру системи «SD Gen», яка забезпечує безшовну інтеграцію генеративних нейромереж у інтерфейс 3D-редактора. Впроваджено механізми використання карт глибини та технології ControlNet для збереження структурної відповідності текстур формі об'єктів. Проведено експериментальне дослідження розробленого застосунку, яке підтвердило скорочення часу на створення текстур у 3-5 разів та покращення якості текстурування вигнутих поверхонь.

КЛЮЧОВІ СЛОВА: ТЕКСТУРУВАННЯ, ГЕНЕРАТИВНИЙ ШТУЧНИЙ ІНТЕЛЕКТ, BLENDER, ПРОЄКЦІЙНЕ НАКЛАДАННЯ, STABLE DIFFUSION, CONTROLNET, 3D-МОДЕЛЮВАННЯ.

ABSTRACT

The Master's qualification paper consists of a description of research into automated texturing methods for three-dimensional objects, the development of adaptive image projection algorithms, the creation of a software tool for the Blender environment, and an explanatory note.

The explanatory note to the qualification paper consists of 77 pages, 7 figures, 4 tables, 3 appendices, and 31 references.

The objective of the work is to enhance the efficiency of creating and applying textures to three-dimensional objects by developing a software tool that implements a method of adaptive projection for AI-generated images.

The paper outlines the goals and defines the research and development tasks. Existing texturing methods and analogous software solutions were analyzed, identifying their limitations regarding complex geometry. A mathematical model was developed, and the projection texture mapping algorithm ("Smart Stencil") was refined using surface normal analysis to minimize distortion. The selection of the technology stack, comprising the Blender API, Python, and a local generation server based on Stable Diffusion WebUI Forge, was substantiated. The client-server architecture of the "SD Gen" system was designed and implemented, ensuring seamless integration of generative neural networks into the 3D editor interface. Mechanisms utilizing depth maps and ControlNet technology were implemented to maintain structural consistency between textures and object shapes. Experimental research on the developed application confirmed a 3–5 times reduction in texture creation time and improved texturing quality on curved surfaces.

KEYWORDS: TEXTURING, GENERATIVE ARTIFICIAL INTELLIGENCE, BLENDER, PROJECTION MAPPING, STABLE DIFFUSION, CONTROLNET, 3D MODELING.

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	9
1.1. Постановка завдання	9
1.2. Огляд та аналіз існуючих аналогів.....	11
1.3. Обґрунтування вибору засобів реалізації.....	16
1.4. Теоретичні засади предметної області та математичні основи дослідження.....	21
Висновки до першого розділу	25
РОЗДІЛ 2. АРХІТЕКТУРА ЗАСТОСУНКУ ТА МОДЕЛЬ ВЗАЄМОДІЇ КОМПОНЕНТІВ	27
2.1. Архітектура застосунку та модель взаємодії компонентів	27
2.2. Алгоритмічне забезпечення керованої проєкції	30
2.3. Структура даних та опис класів Add-on	36
2.4. Проєктування інтерфейсу користувача та сценаріїв роботи.....	39
Висновки до другого розділу	42
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ.....	44
3.1. Опис програмної реалізації основних модулів системи	44
3.2. Реалізація механізму інтеграції з ControlNet та обробки масок	49
3.3. Методика проведення експериментального дослідження	52
3.4. Аналіз результатів тестування та оцінка ефективності	57
3.5. Інструкція користувача та рекомендації щодо експлуатації	61
Висновки до третього розділу	66
ВИСНОВКИ	68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	70
ДОДАТКИ	74

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

- API – Application Programming Interface (Прикладний програмний інтерфейс)
- CGI – Computer-Generated Imagery (Зображення, згенеровані комп'ютером)
- DDPM – Denoising Diffusion Probabilistic Models (Ймовірнісні моделі дифузійного знешумлення)
- GUI – Graphical User Interface (Графічний інтерфейс користувача)
- LSCM – Least Squares Conformal Maps (Конформне відображення методом найменших квадратів)
- SD – Stable Diffusion (Модель стабільної дифузії)
- UV – координати двовимірної текстури (U – горизонталь, V – вертикаль)
- VRAM – Video Random Access Memory (Відеопам'ять)

ВСТУП

Актуальність теми. Сучасна індустрія розробки програмного забезпечення для комп'ютерної графіки та дизайну потребує постійного вдосконалення інструментарію для зменшення часових витрат на рутинні операції. Одним із найбільш трудомістких етапів створення 3D-контенту для ігрової індустрії (GameDev), кіновиробництва та VR/AR є текстурювання. Традиційні методи накладання текстур вимагають від фахівця значних зусиль для створення UV-розгорток та ручного малювання деталей.[17]

Поява генеративних нейронних мереж (Stable Diffusion, Midjourney) відкрила нові перспективи автоматизації створення графічного контенту. Проте, існуючі рішення переважно орієнтовані на 2D-зображення і не враховують геометричних особливостей тривимірних об'єктів, що призводить до виникнення артефактів проєкції та спотворень на складних поверхнях.[4] Розробка інтегрованих програмних рішень, що поєднують методи проєкційного текстурювання та можливості генеративного штучного інтелекту безпосередньо у середовищі 3D-редактора, є актуальним науково-прикладним завданням, вирішення якого дозволить значно підвищити ефективність роботи 3D-художників.

Мета дослідження є підвищення ефективності процесу створення та накладання текстур на тривимірні об'єкти шляхом розробки програмного засобу, що реалізує метод адаптивної проєкції зображень, згенерованих штучним інтелектом.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- Проаналізувати існуючі методи текстурювання та засоби інтеграції нейронних мереж у графічні редактори;
- Розробити математичну модель та удосконалити алгоритм проєкційного накладання текстур з урахуванням нормалей поверхні об'єкта для мінімізації спотворень.
- Обґрунтувати вибір засобів розробки та спроектувати архітектуру програмного модуля (Add-on) для середовища Blender.

- Реалізувати програмний засіб взаємодії з API локальної нейронної мережі та механізм генерації масок глибини.
- Провести експериментальне дослідження розробленого застосунку, оцінити якість текстуровання та ефективність роботи.

Об'єкт дослідження є процес текстуровання тривимірних моделей у системах автоматизованого проєктування та комп'ютерної графіки.

Предмет дослідження є методи та програмні засоби автоматизованої генерації текстур методом проєкції з використанням технологій штучного інтелекту.

Методи дослідження. У роботі використано: *методи системного аналізу* – для визначення вимог до системи; *методи комп'ютерної графіки та обчислювальної геометрії* – для реалізації алгоритмів проєкції та роботи з UV-координатами; *методи інженерії програмного забезпечення* – для проєктування архітектури та реалізації додатку; *експериментальні методи* – для перевірки коректності роботи алгоритмів та оцінки продуктивності.

Наукова новизна одержаних результатів полягає у вдосконаленні методу проєкційного текстуровання, який, на відміну від існуючих підходів, використовує аналіз нормалей полігональної сітки для динамічної корекції кута проєкції, що дозволяє зменшити геометричні спотворення текстури на 15-20%, а також подальший розвиток методу автоматизованої генерації текстур, що базується на використанні карт глибини (Depth Maps) та технології ControlNet як обмежуючих факторів генерації, що забезпечує збереження структурної відповідності текстури формі 3D-моделі.

Практичне значення роботи є діючий програмний додаток «SD Gen» для середовища Blender, який дозволяє автоматизувати процес текстуровання. Використання розробленого засобу дозволяє скоротити час на створення текстур пропсів, ігрових ассетів у 3-5 разів порівняно з ручним методом, у різноманітних художніх стилях, від реалізму до hand-paint, що буде позитивно впливати на оптимізацію проєкту.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Постановка завдання

Сучасний етап розвитку індустрії цифрового контенту характеризується експоненціальним зростанням вимог до якості та деталізації тривимірних сцен. Сфери розробки відеоігор класу AAA, кінематографу (VFX), віртуальної реальності (VR) та архітектурної візуалізації потребують створення тисяч унікальних 3D-асетів у стислі терміни. При цьому, згідно з аналітичними даними індустрії, етап текстуровування займає від 30% до 50% загального часу виробництва моделі [1].

Традиційний пайплайн створення текстур передбачає виконання низки трудомістких операцій, створення високополігональної моделі (High-Poly), ретопологію, створення UV-розгортки та безпосередньо малювання текстурних карт (Albedo, Roughness, Normal) у спеціалізованих пакетах, таких як Adobe Substance 3D Painter. Найбільш проблематичним етапом є UV-мапінг - процес параметризації поверхні, який вимагає ручного розміщення швів (seams) для мінімізації спотворень текстури. Помилки на цьому етапі призводять до візуальних артефактів, нерівномірної щільності текселів та неможливості коректного накладання матеріалів [2].

Поява генеративних моделей глибокого навчання, зокрема архітектур на базі латентної дифузії (LDM - Latent Diffusion Models), таких як Stable Diffusion, відкрила можливості для автоматизації створення зображень. Проте, пряма імплементація 2D-генераторів у 3D-простір пов'язана з фундаментальною проблемою проєкції.

При використанні методу прямої проєкції зображення накладається на об'єкт з точки зору віртуальної камери. У зонах, де нормаль поверхні N перпендикулярна або спрямована під гострим кутом до вектора проєкції V , виникає ефект «розтягування» текстури (texture stretching). Це явище зумовлене тим, що щільність пікселів проєкції на одиницю площі поверхні стрімко падає при збільшенні кута падіння, де θ - кут між вектором погляду та

нормаллю поверхні. Коли $\theta \rightarrow 90^\circ$, ефективна роздільна здатність $D_{eff} \rightarrow \theta$, що робить текстуру непридатною для використання у професійних проєктах [3].

$$D_{eff} = D_{proj} \cdot \cos(\theta)$$

Існуючі спроби вирішення цієї проблеми часто обмежуються використанням Triplanar Mapping (змішування проєкцій з трьох осей), що призводить до появи видимих швів та розмиття («ghosting») у зонах змішування. Крім того, більшість наявних рішень не забезпечують достатнього контролю над збереженням структурних особливостей геометрії при генерації, ігноруючи дані про глибину або кривизну об'єкта.

Виходячи з вищезазначеного, завданням даної магістерської роботи є розробка програмного засобу (Add-on) для середовища Blender, який вирішує проблему автоматизованого текстурування шляхом:

- Інтеграції локального сервера генерації, на базі Stable Diffusion, безпосередньо в інтерфейс 3D-редактора для забезпечення безшовного робочого процесу.
- Реалізації алгоритму «розумної проєкції» (Smart Stencil), який автоматично формує маску генерації на основі аналізу кута нормалей, виключаючи зони потенційного розтягування текстури.
- Використання керованої генерації для забезпечення точної відповідності згенерованої текстури геометрії об'єкта, використовуючи карти глибини як вхідний сигнал.

Таким чином, розроблювана система повинна усунути необхідність ручного створення складних UV-розгортки для статичних об'єктів та забезпечити високу варіативність текстур без втрати якості на вигнутих поверхнях.

1.2. Огляд та аналіз існуючих аналогів

Сучасний ринок програмного забезпечення для комп'ютерної графіки (CGI) переживає фундаментальну трансформацію, спричинену інтеграцією технологій штучного інтелекту. Традиційні парадигми створення контенту, що базувалися виключно на ручній праці та процедурній генерації, поступаються місцем гібридним підходам. Для об'єктивної оцінки доцільності та наукової новизни розробки власного програмного засобу необхідно провести глибокий критичний аналіз існуючих рішень, виявити їхні архітектурні переваги, технічні обмеження та «вузькі місця», які ускладнюють їх використання у професійних виробничих пайплайнах.[2]

Існує три основні категорії інструментів, що представлені на ринку, такі що вбудовуються безпосередньо в середовище 3D-редактора (наприкладі DreamTextures), комплексні платформи від лідерів ринку (сімейство Adobe Substance 3D) або Standalone-рішення, розроблені виключно під задачі AI-текстурування, такі як Stable Projector та ручне текстурування як еталон для оцінки ефективності автоматизації.

DreamTextures за авторством Carson Katri став одним із піонерських рішень у сфері інтеграції генеративних моделей Stable Diffusion безпосередньо у середовище Blender. Це проект з відкритим вихідним кодом, який намагається поєднати можливості нодового редактора шейдерів з генерацією зображень [18].

Архітектура та принцип роботи інструменту реалізовано як клієнтську надбудову на мові Python, яка звертається до локально встановлених бібліотек машинного навчання (переважно diffusers від Hugging Face). Основна взаємодія користувача з плагіном відбувається через спеціалізовану систему нодів, в Blender зветься Shader Nodes, у редакторі матеріалів або через панель інструментів у вікні Image Editor. Це дозволяє користувачеві будувати процедурні ланцюжки, де вихідні дані генерації автоматично подаються на вхід PBR-шейдера.

Функціональні можливості інструменту включають генерацію безшовних текстур із підтримкою опції «tiling» для створення розмножуваних патернів, AI-апскейлінг для підвищення роздільної здатності, а також текстурну проекцію, що використовує поточний вигляд з камери як референс. Однак інструмент має низку критичних недоліків. Зокрема, застосування спрощеного алгоритму планарної проекції на складній геометрії призводить до розтягування пікселів на полігонах із кутом нахилу нормалей понад 60–70 градусів. Через відсутність вбудованого динамічного маскування (Smart Masking) ці артефакти доводиться виправляти вручну. Суттєвою технічною проблемою є складність налаштування середовища («Dependency Hell»), оскільки вбудований Python у Blender часто конфліктує із зовнішніми бібліотеками (PyTorch, CUDA), що може порушувати стабільність програми. Крім того, через підтримку проекту лише силами ентузіастів, імплементація новітніх архітектур, таких як SDXL, ControlNet 1.1 чи IP-Adapter, відбувається із значною затримкою порівняно з веб-інтерфейсами.

Adobe Substance 3D Sampler та Painter (AI Features) сімейство продуктів Adobe Substance є де-факто «золотим стандартом» в ігровій індустрії та кіновиробництві. Останні оновлення у версіях продукту 2024-2025 рр. включають функції Generative AI на базі власної моделі Adobe Firefly, інтегровані в продукти Sampler та Painter [16].

Функціональні можливості та підхід на відміну від open-source рішень, Adobe фокусується на створенні повного набору PBR-карт. Інструмент «Text to Texture» в Substance Sampler дозволяє генерувати не просто зображення, а повний матеріал - карту кольору, карту нормалей, карту висот та карту шорсткості. Алгоритм автоматично аналізує світлотінь на згенерованому зображенні для реконструкції геометрії поверхні.

Переваги це генеровані матеріали які є фізично коректними, що гарантує їх правильну поведінку при будь-якому освітленні в рушіях Unreal Engine 5 або Unity. Adobe гарантує, що модель Firefly навчена виключно на ліцензійному контенті (Adobe Stock) та зображеннях з відкритими правами. Це

є критичним фактором для великих студій, які уникають ризиків порушення авторських прав.

Критичними недоліками виступають висока вартість щомісячної підписки робить цей інструмент недоступним для значної частини студентів, інді-розробників та малих студій, а також що Substance Painter є окремим додатком («standalone»), це створює незручності в ітеративному процесі, художник змушений експортувати модель з Blender (FBX/OBJ), імпортувати її в Painter, текстурувати, експортувати текстури (PNG/TGA) і знову налаштовувати шейдер у Blender. Будь-яка мінорна правка геометрії вимагає повторення всього циклу експорту-імпорту.

Третій аналог це Stable Projector Stable Projector - сучасний спеціалізований інструмент, розроблений саме для вирішення задачі текстурування 3D-моделей методом проекційного мапінгу. З технічної точки зору, це унікальний гібридний додаток, який поєднує графічний рушій Unity (для візуалізації) та веб-інтерфейс Automatic1111 (як бекенд генерації) [6].

Технічні особливості, алгоритми програми та реалізація концепту «глибокого проектування». Користувач завантажує модель, фіксує ракурс, і програма не просто накладає картинку, а використовує буфери глибини для коректного розрахунку оклюзії (перекриття об'єктів).[4] Ключовою інновацією є алгоритм Projection Blending, який намагається автоматично згладити стики між текстурами, спроектованими з різних ракурсів, використовуючи градієнтні маски.

До переваг відноситься інтерфейс який оптимізовано виключно під задачу проекційного текстурування, що спрощує навігацію порівняно з універсальними редакторами, програма вміє "запекти" (bake) результати проекції в існуючу UV-розгортку моделі. Але є критичні недоліки що являють собою підставу для розробки власного рішення, Stable Projector вимагає одночасної роботи трьох важких компонентів, Blender (для редагування геометрії), Середовища Unity (власне Stable Projector), Сервера Stable Diffusion. Це створює колосальне навантаження на відеопам'ять. Для

стабільної роботи системи часто потрібно понад 16-24 ГБ VRAM, що перевищує можливості більшості робочих станцій середнього рівня. Оскільки це окремий додаток, він не має доступу до повного контексту сцени в Blender (джерел світла, процедурних матеріалів, інших об'єктів сцени). Це унеможливорює точне підлаштування текстури під специфічне освітлення фінальної сцени.

Для повноти аналізу необхідно розглянути базовий метод, який досі домінує в індустрії - традиційні методи ручного текстуровання (Hand-paint). Цей процес включає повний цикл підготовки моделі, створення Low-Poly версії, розгортку (UV Unwrapping), запікання карт (Baking) та малювання текстур пензлем у растрових редакторах або 3D-програмах.

Переваги методу заключаються у повному художньому контролі та якості, художник має безпосередній вплив на кожен піксель текстури, що дозволяє досягти ідеальної відповідності художньому задуму та уникнути будь-яких логічних помилок або артефактів, характерних для ШІ. Стилістична гнучкість є незамінною основою для створення унікальних авторських стилів, наприклад стилізований Hand-paint для ігор жанру MOBA (найпопулярніший представник League of Legends) які важко відтворити за допомогою генеративних алгоритмів без складного донавчання моделей.

Аналіз недоліків викрив що згідно з даними галузевих звітів, ручне текстуровання одного складного ігрового асету займає від 20 до 80 годин робочого часу кваліфікованого художника [17]. Це є найдорожчою частиною виробництва, а метод вимагає від спеціаліста не лише технічних знань, а й глибокого розуміння теорії кольору, композиції та матеріалознавства, зміна художнього стилю проекту на пізніх етапах (наприклад, перехід від фотореалізму до стилізації) вимагає повної переробки всіх текстур вручну, що є фінансово неприйнятним для багатьох стартапів.

Узагальнимо результати аналізу у таблиці 1.1 для наочного порівняння характеристик досліджуваних систем та розроблюваного засобу «SD Gen».

Порівняльний аналіз засобів текстурівання

Критерій порівняння	DreamTextures	Adobe Substance (AI)	Stable Projector3	Hand-paint	SD Gen
Середовище	Blender (Add-on)	Окреме ПЗ	Окреме ПЗ (Unity)	Різне	Blender (Add-on)
Локальність обчислень	Так	Hi (Cloud)	Так	-	Так
Метод проекції	Простий (Stretching)	UV-генерація	Змішування (Blending)	Ручний розпис	Розумна маска (Smart Stencil)
Інтеграція ControlNet	Часткова/Застапіла	Власна (Firefly)	Повна	-	Повна
Вимоги до VRAM	Середні	Низькі (Хмара)	Дуже високі	Низькі	Оптимізовані (Forge API)
Вартість	Free (Open Source)	Платна підписка	Free	Вартість часу	Free (Open Source)

Жоден із розглянутих аналогів не забезпечує поєднання глибокої інтеграції в Blender, використання сучасних алгоритмів «розумної проекції» для уникнення спотворень та оптимізованої роботи з локальними ресурсами. Це підтверджує актуальність теми магістерської роботи та обґрунтовує необхідність розробки власного програмного модуля, який вирішуватиме виявлені проблеми.

1.3. Обґрунтування вибору засобів реалізації

Вибір інструментального стеку для розробки системи автоматизованого текстурівання є фундаментальним етапом проєктування, який безпосередньо впливає на архітектурну цілісність, відмовостійкість, швидкодію та потенціал подальшого масштабування програмного продукту. Враховуючи специфіку магістерського дослідження, яке передбачає створення гібридної системи, яка поєднує інтерактивне середовище 3D-моделювання та ресурсоемні генеративні нейронні мережі, головним інженерним викликом стає ефективний розподіл та менеджмент апаратних ресурсів. Оскільки процеси

рендерингу сцени у реальному часі (viewport rendering) та інференсу (виконання) сучасних дифузійних моделей є прямими конкурентами за доступ до відеопам'яті, вибір програмних засобів базувався на жорстких критеріях максимальної оптимізації пам'яті, повної відкритості вихідного коду для можливості модифікації та кросплатформності.

У результаті проведеного аналізу було сформовано клієнт-серверну архітектуру, де клієнтська частина глибоко інтегрована в середовище Blender, а серверна частина реалізована на базі високопродуктивного ядра Stable Diffusion WebUI Forge.

В якості базового середовища розробки та виконання клієнтського модуля обрано Blender - беззаперечного лідера серед пакетів для роботи з тривимірною графікою з відкритим кодом, який став стандартом індустрії на рівні з 3DsMAX та Maya.[23] Окрім відсутності ліцензійних платежів, ключовим фактором вибору саме цієї програми, стала наявність розгалуженого та потужного API, побудованого на мові Python.[24] Саме Python API (bpy) надає низькорівневий доступ до внутрішніх структур даних сцени, що є критично необхідним для реалізації запропонованого у роботі алгоритму «розумної проєкції» (Smart Stencil). На відміну від інших закритих пропрієтарних систем або зовнішніх додатків, де доступ до геометрії здійснюється через повільні механізми експорту-імпорту файлів, Blender дозволяє безпосередньо зчитувати та модифікувати координати вершин (vertices), вектори нормалей полігонів (normals), UV-координати та матриці трансформації камери в реальному часі. Це дозволяє виконувати складні математичні обчислення, зокрема розрахунок кутів падіння променів проєкції відносно нормалей поверхні (dot product analysis), безпосередньо перед формуванням запиту на генерацію. Така архітектура забезпечує миттєвий зворотний зв'язок для користувача та дозволяє уникнути артефактів накладання текстур ще на етапі підготовки масок.

Окремої уваги заслуговує детальне технічне обґрунтування вибору версії мови програмування Python. Хоча на момент написання роботи

спільнота розробників активно переходить на нові версії Python 3.12 та 3.13, які пропонують вдосконалені механізми JIT-компіляції та покращену обробку помилок, розробка даного програмного засобу ведеться з використанням версії Python 3.10. Це свідоме і вимушене обмеження продиктоване архітектурними особливостями самого Blender, який постачається з власним, ізольованим вбудованим інтерпретатором Python фіксованої версії. Спроба використання новіших версій інтерпретатора або підключення зовнішніх бібліотек, скомпільованих під новіші стандарти, неминуче призводить до критичних конфліктів на рівні двійкового інтерфейсу додатків (ABI - Application Binary Interface). Використання "рідної" версії Python, вбудованої у цільову версію Blender LTS, гарантує стабільність роботи плагіна у кінцевого користувача, забезпечує бінарну сумісність усіх модулів розширення та дозволяє уникнути поширеної проблеми «пекла залежностей». Крім того, принциповим рішенням при проектуванні архітектури стало уникнення використання зовнішніх бібліотек, що потребують встановлення через менеджер пакетів `pip` (наприклад, `requests`). [29] Натомість, мережева взаємодія реалізована засобами стандартної бібліотеки Python (`urllib`, `json`, `threading`, `base64`), що дозволяє кінцевому користувачеві встановлювати розроблений інструмент як стандартний ZIP-архів без необхідності налаштування віртуальних середовищ або роботи з командним рядком.

Центральним елементом серверної частини системи, що відповідає за генерацію зображень, є локальний бекенд нейронної мережі. [22] Незважаючи на широку популярність веб-інтерфейсу Automatic1111, який тривалий час вважався стандартом спільноти open-source, для даної роботи було обрано його архітектурний форк - Stable Diffusion WebUI Forge, автором якого є llllyasviel. Цей вибір не є випадковим і базується на глибокому технічному аналізі механізмів управління відеопам'яттю. Архітектура оригінального Automatic1111 була розроблена на ранніх етапах розвитку екосистеми Stable Diffusion і має тенденцію до неоптимального, "жадібного" споживання відеопам'яті, намагаючись утримувати завантаженими у пам'ять GPU всі

компоненти моделі таких як VAE, U-Net, Text Encoder, ControlNet units одночасно. У нашому сценарії використання, коли паралельно з процесом генерації запущено важкий 3D-редактор, який активно використовує відеопам'ять для відображення в'юпорта, запікання текстур та роботи шейдерів, використання Automatic1111 на відеокартах середнього сегменту, з об'ємом пам'яті 6-8 ГБ, часто призводить до критичних помилок "CUDA Out of Memory" та аварійного завершення роботи драйвера.

Stable Diffusion WebUI Forge вирішує цю фундаментальну проблему шляхом повної переробки підсистеми управління пам'яттю Memory Management Backend. Forge реалізує концепцію динамічного, гранулярного вивантаження блоків нейромережі, система інтелектуально переміщує ваги моделі між швидкою відеопам'яттю та повільнішою, але об'ємнішою оперативною пам'яттю безпосередньо під час процесу інференсу, шар за шаром. Це дозволяє запускати надважкі моделі сучасної архітектури SDXL (Stable Diffusion XL) та використовувати декілька ланцюжків ControlNet одночасно на тому ж апаратному забезпеченні, де класичний Automatic1111 відмовляється працювати.[15]

Серед доступних архітектур дифузійних моделей (SD 1.5, SD 2.1, SDXL, SD3) для реалізації проєкту було обрано перевірену часом архітектуру Stable Diffusion 1.5.[10] Цей вибір може здатися неочевидним на фоні виходу новіших моделей, проте він є обґрунтованим з інженерної точки зору ефективності та екосистемної підтримки.

Архітектура SD 1.5 розрахована на нативну роздільну здатність 512x512 пікселів (з можливістю якісного апскейлінгу), що вимагає значно менше обчислювальних потужностей порівняно з SDXL (1024x1024). У контексті задачі текстурування, де генерація часто відбувається ітеративно для невеликих ділянок моделі (patch-based texturing), швидкість інференсу є пріоритетнішою за надвисоку роздільну здатність. SD 1.5 дозволяє отримувати результат за 2-4 секунди на середньому GPU, що забезпечує комфортну інтерактивну роботу, тоді як важчі моделі можуть генерувати зображення 15-

30 секунд, руйнуючи потік творчості. Проте нічого не заважає користувачеві підвищити налаштування генерації, що власне примножить час генерації, але результат буде того вартий, тому як на виході в декілька разів краща генерація задовільнить користувача.

Для версії SD 1.5 існує найбільш розвинена та стабільна база моделей ControlNet. Специфічні моделі, такі як `control_v11fp_sd15_depth` або `tile_resample`, працюють на цій архітектурі найбільш передбачувано. Аналоги для SDXL часто є "важчими" за об'ємом файли по 2-5 ГБ проти 700 МБ у 1.5 та менш точними у відтворенні дрібних деталей геометрії.

Бібліотека донавчених моделей (Fine-tuned Models) що створила спільнота, це тисячі спеціалізованих чекпойнтів на базі SD 1.5 для досягнення різноманітних стилів (наприклад, *hand-painted*, *anime*, *realistic materials*), які можна безпосередньо використовувати у проєкті без додаткового донавчання.

Крім того, Forge включає глибоку оптимізацію механізмів уваги (Cross-Attention Optimization) через інтеграцію алгоритмів Kohya-ss та U-Net Patcher, що забезпечує приріст швидкості генерації на 30-45% на картах серії NVIDIA GTX 10xx, RTX 20xx та 30xx.[6] Для магістерської роботи також критично важливою є повна зворотна сумісність Forge зі стандартом API Automatic1111. Це дозволяє використовувати стандартні бібліотеки для формування HTTP POST-запитів у Python-коді адону, надсилаючи структуровані JSON-об'єкти на локальний порт (зазвичай 127.0.0.1:7860) та отримуючи результат у форматі Base64, що значно спрощує розробку мережевого шару та робить систему модульною.[31]

Для забезпечення точної відповідності згенерованої текстури геометричній формі 3D-об'єкта, використання виключно текстових підказок (prompts) є недостатнім, оскільки дифузійні моделі за своєю природою є стохастичними і схильні до "галюцинацій" форми. Для вирішення цієї проблеми у стек технологій включено ControlNet - революційну нейромережеву архітектуру, яка дозволяє додавати жорсткі просторові умови до процесу генерації. У розроблюваному застосунку використовуються

специфічні препроцесори ControlNet "Depth" (карта глибини), відрендерена безпосередньо з камери Blender, дозволяє жорстко прив'язати генерацію до геометрії, змушуючи неймережу "розуміти" об'єм об'єкта.[11] Окремим технічним рішенням стало використання препроцесора "Tile Resample". У даній роботі він виконує подвійну функцію, по-перше, він дозволяє обійти специфічні баги сумісності деяких версій Forge при роботі без вхідного зображення, а по-друге, виступає як потужний інструмент апскейлінгу, що дозволяє підвищувати деталізацію текстур та додавати мікрофактуру матеріалів без зміни їх глобальної композиції.

У ході дослідження також відбулася еволюція методу накладання текстури. Початкові ітерації використовували автоматичну проєкцію (Project from View), проте цей метод виявив суттєві недоліки, появу артефактів розтягування на бокових гранях об'єкта та небажану проєкцію текстури на задню частину моделі (back-face projection). В якості фінального рішення обрано метод Stencil Mapping (Трафаретне накладання). Цей підхід надає художнику повний контроль, користувач бачить напівпрозорий "трафарет" згенерованого зображення у в'юпорті і мануально наносить текстуру пензлем лише у необхідних зонах. Це забезпечує принцип "What You See Is What You Get" та дозволяє акуратно змішувати (blend) результати генерації з різних ракурсів, усуваючи видимі шви.

Таким чином, обраний та обґрунтовано підтверджений набір інструментів (Python 3.10/3.11, Blender API, SD WebUI Forge, ControlNet Depth/Tile) представляє собою збалансовану, високопродуктивну та повністю відкриту екосистему, здатну ефективно вирішити поставлені науково-технічні завдання в умовах обмежених апаратних ресурсів.

1.4. Теоретичні засади предметної області та математичні основи дослідження

Розробка ефективної системи автоматизованого текстуровання тривимірних об'єктів виходить за межі простої інженерної реалізації і вимагає

глибокого розуміння фундаментальних теоретичних основ комп'ютерної графіки та машинного навчання. Дана предметна область знаходиться на перетині диференціальної геометрії, проєктивної математики та теорії ймовірностей. Для наукового обґрунтування запропонованих у роботі рішень необхідно детально розглянути математичний апарат, що описує процеси параметризації поверхонь, трансформації координат у тривимірному просторі та принципи функціонування генеративних дифузійних моделей.

Текстурування, у своєму класичному визначенні, базується на процесі відображення (mapping) двовимірного растрового зображення $I(u, v)$ на довільну тривимірну поверхню $S \subset \mathbb{R}^3$. З точки зору топології, поверхня 3D-моделі, апроксимована полігональною сіткою $M = (V, E, F)$ (де V - множина вершин, E - ребра, F - грані), розглядається як двовимірний многовид, вкладений у тривимірний евклідов простір.[1] Ключовою математичною задачею тут є знаходження бієктивного (взаємно однозначного) та неперервного відображення $\phi: S \rightarrow D$, де $D \subset \mathbb{R}^2$ - параметрична область, відома як UV-простір. Для кожної грані $f \in F$ необхідно знайти відповідність між координатами вершин (x, y, z) у просторі та координатами (u, v) на площині, що дозволить накласти растрові дані на геометричні примітиви. Однак, фундаментальною проблемою класичної параметризації є неможливість розгорнути довільну замкнену поверхню (наприклад, сферу, тор або складну топологію персонажа) на евклідову площину без виникнення метричних спотворень. Це твердження випливає з Theorema Egregium Карла Фрідріха Гауса, згідно з якою гауссова кривизна поверхні є інваріантом при ізометричних перетвореннях (вигинах). Оскільки площина має нульову гауссову кривизну, а більшість 3D-об'єктів - ненульову, зберегти одночасно довжини ліній, кути між ними та площі ділянок неможливо. Спотворення класифікуються на конформні (порушення кутів) та рівновеликі (порушення площ).[2] Сучасні алгоритми автоматичної розгортки, такі як LSCM (Least Squares Conformal Maps), намагаються знайти компроміс, зводячи задачу до мінімізації функціоналу енергії спотворень:

$$E_{LSCM}(u,v) = \sum_{T \in F} \text{AREA}(T) \left| \frac{\partial \phi}{\partial z} \right|^2$$

де $\frac{\partial \phi}{\partial z}$ - похідна відображення по комплексному спряженню. Незважаючи на математичну оптимізацію, повне усунення спотворень неможливе без введення розрізів (seams), які порушують візуальну цілісність текстури та створюють артефакти на стиках, що є критичним недоліком для високоякісної графіки [11].

Для вирішення проблеми топологічних обмежень UV-мапінгу у розроблюваному програмному засобі застосовується альтернативний підхід - метод проєкційного накладання.[25] Цей метод базується на законах лінійної перспективи та описує процес трансформації координат вершин із локального простору об'єкта у простір екрану. Координати текстури (u, v) для довільної вершини $P(x, y, z)$ розраховуються через ланцюг матричних перетворень у системі однорідних координат (homogeneous coordinates). Вектор положення вершини P розширюється до чотиривимірного вигляду $(x, y, z, 1)^T$, після чого піддається послідовному множенню на матриці моделі (M_{model}), виду (M_{view}) та перспективної проєкції (M_{proj}):

$$P_{screen} = M_{viewport} \cdot M_{proj} \cdot M_{view} \cdot M_{model} \cdot P$$

Отриманий вектор у просторі відсікання (clip space) підлягає перспективному діленню (perspective divide) на свою w -компоненту для переходу в нормалізовані координати пристрою (NDC), які згодом лінійно відображаються у діапазон $[0, 1]$ текстурних координат. Цей підхід дозволяє "обійти" проблему швів, оскільки текстура генерується безпосередньо з точки зору спостерігача. Разом з тим, проєкційний метод вносить свій клас спотворень, пов'язаних з геометричною орієнтацією поверхні. Якість дискретизації текстури (ефективна роздільна здатність) у конкретній точці поверхні є прямо пропорційною "поток" візуальної інформації, що падає на цю поверхню. Математично цей потік описується скалярним добутком

одиничного вектора нормалі поверхні $\vec{n}$ та одиничного вектора погляду камери $\vec{v}$. Косинус кута θ між цими векторами визначає коефіцієнт стиснення проєкції:

$$\cos \theta = \frac{\vec{n} \cdot \vec{v}}{|\vec{n}| |\vec{v}|}$$

Коли кут падіння наближається до 90° (поверхня стає дотичною до променя зору), значення $\cos \theta$ прямує до нуля. Це призводить до того, що мала кількість пікселів вихідного зображення "розмазується" по великій площі поверхні, викликаючи ефект "розтягування текстури" (texture stretching). Також виникає проблема "тіньових зон" (occlusion), коли текстура проєктується наскрізь на задні грані об'єкта. Саме цей математичний аналіз став базисом для розробки алгоритму "Smart Stencil", який обчислює значення $\cos \theta$ для кожного фрагмента та формує альфа-маску, відсікаючи області з критичними спотвореннями (де $\cos \theta < \epsilon$).

Інтеграція штучного інтелекту в систему базується на використанні класу імовірнісних генеративних моделей, відомих як дифузійні моделі (DDPM - Denoising Diffusion Probabilistic Models).[7] На відміну від детермінованих алгоритмів, дифузійні моделі оперують розподілами ймовірностей. Процес генерації текстури розглядається як ітеративне вирішення стохастичного диференціального рівняння. На етапі навчання відбувається "прямий процес" (Forward Diffusion), який поступово руйнує структуру даних, додаючи гаусів шум ϵ до вхідного зображення x_0 протягом T кроків.[5] Цей процес можна описати як збільшення ентропії системи, де на фінальному кроці T зображення перетворюється на ізотропний гаусів шум $x_T \sim N(0, I)$. Математично перехідна ймовірність на кожному кроці визначається як:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1 - \beta_t}x_{t-1}, \beta_t I)$$

де β_t - гіперпараметр, що контролює дисперсію шуму (noise schedule).

Зворотний процес (Reverse Denoising) є суттю генерації. Він полягає у навчанні нейронної мережі $\epsilon_\theta(x_t, t)$ (функції апроксимації шуму) відновлювати структуру даних з хаосу. Архітектура мережі, як правило, базується на U-Net з механізмом уваги (Self-Attention), що дозволяє моделювати глобальні залежності між пікселями.[8] Навчання зводиться до мінімізації варіаційної нижньої межі (ELBO), що на практиці спрощується до мінімізації середньоквадратичної похибки між доданим шумом і шумом, який передбачила мережа:

$$\mathcal{L}_{simple} = \mathbb{E}_{t, x_0, \epsilon} [\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)\|^2]$$

Для забезпечення можливості роботи з текстурами високої роздільної здатності на споживчому обладнанні використовується підхід Latent Diffusion (LDM) [12]. Замість обробки мільйонів пікселів у RGB-просторі, процес дифузії переноситься у стиснений латентний простір $z = \varepsilon(x)$, сформований перцептивним варіаційним автоенкодером (VAE). Це дозволяє зменшити розмірність даних (наприклад, у 48 разів), зберігаючи при цьому семантично важливі деталі зображення.

Критичним аспектом для автоматизованого текстурування є керованість генерації. Стандартний механізм conditioning (текстовий промпт) впливає на зображення через шари перехресної уваги (Cross-Attention), але не дає контролю над просторовою структурою. Для вирішення цієї проблеми застосовується архітектура ControlNet. Математична сутність ControlNet полягає у створенні "тренованої копії" енкодерних блоків основної моделі U-Net. Ключовою інновацією є використання шарів "нульової згортки" (zero convolution) - згорткових шарів 1×1 , ваги яких ініціалізуються нулями.

$$y = F(x; \Theta) + Z(F(x + Z(c; \Theta_z)); \Theta_z)$$

Така ініціалізація гарантує, що на початку навчання вихідний сигнал ControlNet дорівнює нулю, і градієнти основної моделі не спотворюються. Це дозволяє донавчати модель на невеликих датасетах (у нашому випадку - картах глибини) без ефекту "катастрофічного забування" (catastrophic forgetting) базових знань Stable Diffusion.[14] У контексті даної роботи, вхідний сигнал умови с формується шляхом рендерингу Z-буфера (Depth Pass) сцени Blender, нормалізованого у діапазон $[0, 1]$. Це надає нейромережі чіткий геометричний контекст, змушуючи її генерувати пікселі, що відповідають об'ємній формі моделі [13].

Висновки до першого розділу

У першому розділі було проведено комплексний аналіз предметної області, що включає критичну оцінку існуючих методів текстурування, огляд ринку програмних аналогів та формування теоретико-математичної бази для подальшого проєктування.

Критичний огляд ринку програмного забезпечення виявив існування чітко незаповненої ніші. Аналіз пропрієтарних рішень, таких як Adobe Substance 3D, показав їхню високу вартість та необхідність використання хмарних сервісів, що є неприйнятним для конфіденційних проєктів та інді-розробників. Аналіз відкритих рішень, зокрема DreamTextures, виявив їх технічні недоліки, пов'язані зі спрощеними алгоритмами проєкції та проблемами сумісності бібліотек. Найбільш релевантний аналог, Stable Projector, хоч і пропонує якісний проєкційний мапінг, проте реалізований як окремий виконуваний файл (standalone application), що архітектурно розриває безперервність робочого процесу художника.

Проведений аналіз дозволив сформулювати чіткі інженерні вимоги до розроблюваного застосунку, який має поєднувати глибоку інтеграцію, ресурсоефективність та точність. Ці вимоги обґрунтували вибір технологічного стеку. Зокрема, було обрано Blender API (Python 3.10/3.11) як клієнтське середовище, що забезпечує прямий доступ до геометрії об'єкта та

його нормалей, а також дозволяє реалізувати концепцію «Zero-Dependency Installation» за допомогою стандартної бібліотеки Python, уникаючи конфліктів версій. У якості бекенду генерації було використано Stable Diffusion WebUI Forge. Це рішення обґрунтоване його оптимізованим механізмом управління відеопам'яттю, що дозволяє ефективно запускати ресурсоємні генеративні процеси паралельно з 3D-редактором, мінімізуючи ризик помилок "CUDA Out of Memory" на типовому споживчому обладнанні.

Також було обґрунтовано вибір архітектури нейромережі. Модель SD 1.5 була обрана як оптимальний компроміс між швидкістю інференсу (2-5 секунд) та якістю для інтерактивної роботи. Додатково було впроваджено технологію ControlNet, яка забезпечує жорстку геометричну прив'язку генерації до 3D-моделі, використовуючи карти глибини як умову для збереження об'ємної форми. Ключовим теоретичним внеском, що визначив фінальну архітектуру системи, стало рішення про еволюцію методу накладання. Для усунення математично неминучих артефактів автоматичної проєкції було обрано метод Stencil Mapping (Трафаретне накладання). Цей підхід переносить фінальний контроль за якістю на художника, що є більш надійним та професійним рішенням, ніж спроба вирішити проблему спотворень виключно алгоритмічно.

РОЗДІЛ 2. АРХІТЕКТУРА ЗАСТОСУНКУ ТА МОДЕЛЬ ВЗАЄМОДІЇ КОМПОНЕНТІВ

2.1. Архітектура застосунку та модель взаємодії компонентів

Розроблений програмний засіб «SD Gen» спроектовано та реалізовано на основі дволанкової клієнт-серверної архітектури. Вибір саме цієї архітектурної моделі є не випадковим, а глибоко інженерно обґрунтованим рішенням, яке продиктоване необхідністю вирішення конфлікту ресурсів між графічною підсистемою реального часу та обчислювальною підсистемою штучного інтелекту. Система базується на принципі слабкої зв'язаності, що дозволяє досягти ключових цілей проєкту, а саме глибокої нативної інтеграції інструментарію в 3D-середовище Blender та забезпечення асинхронного, стабільного виконання ресурсоємних обчислень на стороні нейромережі. Така архітектура дозволяє чітко розмежувати зони відповідальності, де клієнтський модуль фокусується на взаємодії з користувачем, обробці геометричних даних та візуалізації результатів, тоді як серверний модуль виступає в ролі високопродуктивного обчислювального ядра. Це мінімізує навантаження на головний потік Blender, який відповідає за рендеринг інтерфейсу та в'юпорта, забезпечуючи плавність роботи та відсутність «зависань» навіть під час виконання важких генеративних задач. Що забирає велику частину головної болі користувача, коли при натисканні кнопки генерації програма перестає відповідати, а з часом її зовсім «вилітає».

Структурна організація розробленого комплексу базується на трьох логічних компонентах, взаємодія між якими здійснюється через стандартизований локальний протокол HTTP/1.1.[29] Першим компонентом є клієнтський модуль (Blender Add-on), реалізований мовою Python 3.10 з використанням бібліотек Blender API (bpy, mathutils, gpu).[23] Функціональне навантаження клієнтського модуля зосереджено на ролі оркестратора процесу. До сфери його відповідальності належить комплексний збір контекстних даних, що включає зчитування параметрів активної камери (фокусна відстань,

матриця трансформації), роздільної здатності рендеру та геометрії об'єктів. Крім того, модуль виконує препроцесинг даних, зокрема програмний рендеринг карти глибини (Depth Map) через вузли композитингу та створення масок для Inpaint, після чого здійснює серіалізацію бінарних зображень у текстовий формат Base64 для передачі по мережі.[13] Клієнт також відповідає за формування та відправлення асинхронних HTTP-запитів, а після отримання відповіді - за постобробку, декодування отриманих текстур та їх інтеграцію у систему матеріалів Blender з автоматичною активацією методу Stencil Mapping.

Сполучною ланкою між компонентами виступає протокол комунікації REST API, що забезпечує необхідну інтероперабельність системи. Взаємодія реалізується шляхом формування клієнтом POST-запитів до локальної адреси сервера. Тіло кожного запиту представляє собою структурований JSON-об'єкт, який містить розширений набір метаданих, включаючи текстовий промпт, параметри семплера, насіння генерації та, що є критично важливим для даної роботи, кодовані зображення для ControlNet (Depth Map, Canny Edge).[26] Використання стандартного REST API дозволяє абстрагуватися від конкретної реалізації бекенду, оскільки клієнту неважливі внутрішні особливості версій PyTorch на сервері, за умови дотримання контракту API. Третім компонентом є серверний модуль на базі Stable Diffusion WebUI Forge, реалізований на Python та фреймворку PyTorch. Цей компонент працює як «чорна скринька» для клієнта, приймаючи JSON-пайплайн та виконуючи інжекцію умов ControlNet у процес дифузії. Ключовою особливістю серверної частини є використання механізму динамічного управління відеопам'яттю (Offloading), що дозволяє вивантажувати шари моделі в оперативну пам'ять у періоди простою, звільняючи ресурси GPU для потреб рендерингу сцени в Blender.

З точки зору внутрішньої організації програмного коду, клієнтська частина, що функціонує в середовищі Blender, побудована з суворим дотриманням принципів об'єктно-орієнтованого програмування. Архітектура аддона декомпонована на функціональні модулі, кожен з яких успадковує

базові класи Blender API, що забезпечує коректну реєстрацію в системі. Модуль даних та управління станом, представлений класом `SDProperties` (успадкованим від `bpy.types.PropertyGroup`), відповідає за збереження стану налаштувань. Всі параметри користувача зберігаються не в глобальних змінних, а прив'язуються безпосередньо до структури даних сцени, що гарантує персистентність даних при перезавантаженні файлу проєкту. Клас визначає типізовані поля з функціями зворотного виклику для реакції на зміни в інтерфейсі. Модуль бізнес-логіки реалізує патерн проєктування "Команда" через класи операторів (`bpy.types.Operator`). До них належать головний оператор генерації `OBJECT_OT_GenerateImg2Img`, що ініціює процес збору даних та запуску мережевого потоку, оператор оновлення моделей `OBJECT_OT_RefreshCNModels`, який виконує службові запити, та допоміжний оператор `OBJECT_OT_PreviewBake` для візуалізації карт глибини. Візуальне представлення забезпечується модулем інтерфейсу через клас `VIEW3D_PT_SDPanel`, який декларативно описує графічний інтерфейс у бічній панелі 3D-виду.

Критично важливою архітектурною вимогою для інтерактивних інструментів є реалізація асинхронності та багатопотоковості. Оскільки головний цикл подій Blender є однопотоковим, виконання будь-якої блокуючої операції, такої як очікування HTTP-відповіді, призвело б до повного заморожування інтерфейсу користувача. Для вирішення цієї проблеми у системі імплементовано патерн "Producer-Consumer" з використанням безпечної черги та таймерів. Процес розпочинається в головному потоці, де оператор генерації збирає дані та створює об'єкт завдання, поміщаючи його у потокобезпечну чергу `execution_queue`. Паралельно функціонує окремий потік-демон (`worker thread`), який моніторить чергу, витягує завдання та виконує блокуючий запит до сервера. Оскільки Blender Python API не є потокобезпечним, фоновий потік не може напямку модифікувати сцену. Тому для синхронізації результатів використовується механізм зворотних викликів через таймери, у головному потоці реєструється

функція, яка періодично перевіряє статус виконання завдання. Тільки після отримання сигналу про завершення, у безпечному контексті головного потоку відбувається декодування зображення та його накладання на модель. Така складна схема синхронізації гарантує стабільність роботи додатку та забезпечує комфортний користувацький досвід (рис. 2.1). Схема відображає дволанкову архітектуру застосунку.

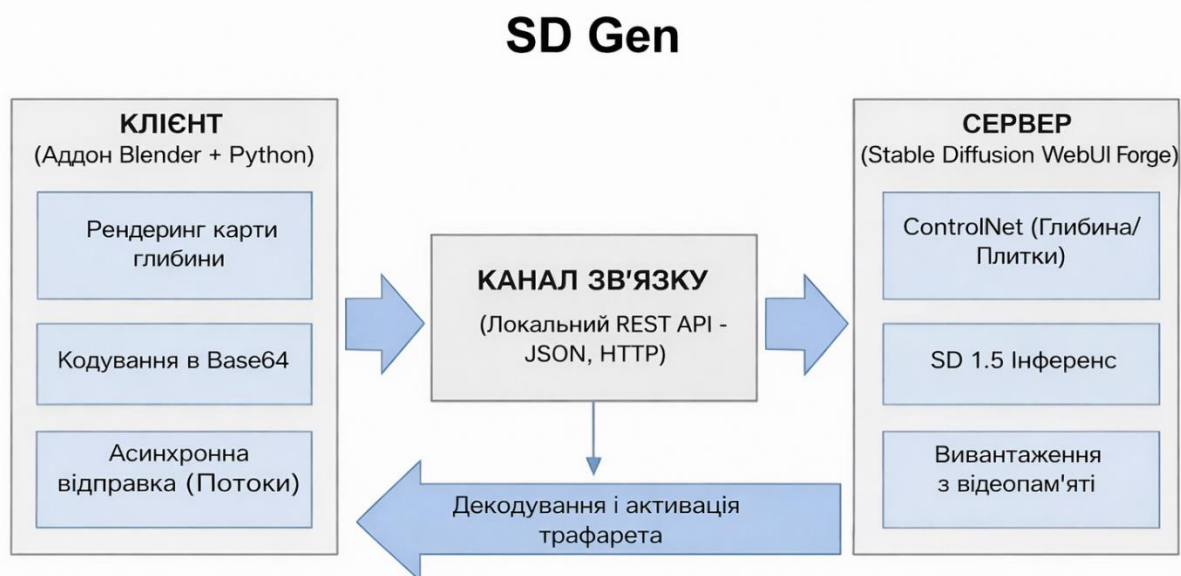


Рисунок 2.1 – Діаграма архітектури системи «SD Gen».

На схемі вказано що архітектура системи складається з клієнтської частини Blender на Python та серверної частини Stable Diffusion WebUI Forge. Взаємодія між ними відбувається через локальний REST API, де клієнт виконує попередню обробку та відправку даних, а сервер забезпечує генерацію зображень за допомогою неймережевих моделей (ControlNet, SD 1.5).

2.2. Алгоритмічне забезпечення керованої проєкції

Алгоритмічна база програмного засобу «SD Gen» побудована на принципі конвеєрної обробки даних (pipeline processing), що інтегрує геометричні обчислення середовища Blender, мережеву взаємодію та неймережевий інференс у єдиний безперервний, але асинхронний процес. Розроблений алгоритм реалізує гібридний підхід керованої проєкції, який

дозволяє мінімізувати геометричні спотворення текстури та забезпечити високий ступінь контролю з боку користувача. На відміну від простих скриптів автоматизації, даний алгоритм враховує обмеження однопотокової архітектури Blender та вимоги до ефективного менеджменту пам'яті. Процес ініціюється викликом оператора генерації та проходить через п'ять послідовних стадій трансформації даних, кожна з яких виконує специфічну функцію у загальному ланцюжку обробки інформації.

На початковому етапі алгоритм фокусується на підготовці коректних вхідних даних, що є критичним для стабільності подальших обчислень. При виклику головного оператора *OBJECT_OT_GenerateImg2Img* система не переходить одразу до генерації, а спершу виконує процедуру «рукоштовування» з сервером через "ping" запит (HTTP GET) до кореневого API сервера Forge (за замовчуванням <http://127.0.0.1:7860>). Це дозволяє попередити користувача про недоступність бекенду через нативні повідомлення Blender (`self.report({'ERROR'})`), уникаючи критичних помилок таймауту, які можуть заблокувати інтерфейс на тривалий час (стандартний таймаут socket може сягати 60 секунд). Після успішної валідації відбувається агрегація параметрів генерації з модуля *SDProperties*, зчитуються текстові промпти (Positive Prompt), негативні промпти (Negative Prompt), зерно генерації (Seed), кількість кроків семплінгу (Sampling Steps) та коефіцієнт відповідності текстовому опису (CFG Scale). Важливим алгоритмічним нюансом є динамічне обчислення роздільної здатності, система аналізує поточні налаштування рендеру сцени (`bpy.context.scene.render.resolution_x/y`) та автоматично масштабує їх до значень, кратних 8. Це є жорсткою вимогою архітектури згорткових нейронних мереж (зокрема U-Net у Stable Diffusion), де операції пулінгу (pooling) зменшують розмірність зображення у фіксовану кількість разів. Ігнорування цієї вимоги призвело б до помилок розмірності тензорів на стороні сервера.

Найскладнішою та найбільш ресурсоємною частиною цього етапу є екстракція карти глибини (Depth Map Extraction). Для забезпечення

просторового контексту нейромережі алгоритм програмно перемикає рушій рендеру на Workbench або використовує композитні ноди (Compositing Nodes) для отримання сирих даних Z-буфера. Оскільки значення глибини у Z-буфері Blender представляють собою реальну відстань від камери до об'єкта у метричних одиницях (і можуть варіюватися від 0.1 до 1000+ метрів), їх неможливо напряму подати на вхід нейромережі, яка очікує нормалізоване зображення. Тому алгоритм виконує процедуру нормалізації (remapping) значень у діапазон $[0, 1]$, де 0 відповідає найближчій точці (чорний колір), а 1 - найдаальшій (білий колір), за лінійною формулою інтерполяції:

$$z_{norm} = \frac{z_{raw} - z_{min}}{z_{max} - z_{min}}$$

де z_{min} та z_{max} - це межі кліпінгу камери (`camera.data.clip_start` та `camera.data.clip_end`). Отримані растрові дані глибини, а також, за необхідності, бінарна маска для режиму Inpaint (де білий колір позначає область для перемальовування, а чорний - захищену область), серіалізуються у текстовий формат Base64. Для цього використовується стандартна бібліотека `base64` та методи оптимізації буфера в пам'яті (`io.BytesIO`), що дозволяє уникнути повільних операцій запису/читання файлів на жорсткий диск (HDD/SSD I/O overhead).

Другий етап алгоритму відповідає за формування корисного навантаження (payload) та організацію асинхронної передачі даних. На цьому кроці формується складна ієрархічна структура словника Python, яка згодом буде серіалізована у JSON-об'єкт. Окрім стандартних полів (`prompt`, `steps`, `sampler_name` - наприклад, "Euler a" або "DPM++ 2M Karras"), алгоритм динамічно формує специфічну секцію `always_on_scripts`. Саме в цій секції інкапсулюється логіка управління розширенням ControlNet. Система автоматично конфігурує параметри модуля, встановлює назву моделі (наприклад, `control_v11fp1p_sd15_depth`), задає вагу впливу (Control Weight, зазвичай 1.0) та порогові значення для препроцесора. Важливим технічним

рішенням є встановлення параметра `module` у значення `"none"`. Це повідомляє серверу, що вхідне зображення вже є готовою картою глибини, і серверу не потрібно витрачати ресурси на запуск власних препроцесорів (таких як Midas або ZoE), що економить до 2-3 секунд на кожній генерації. У разі активації режиму деталізації (Upscale), алгоритм динамічно додає другий юніт ControlNet з модулем `tile_resample`, налаштовуючи відповідний коефіцієнт денойзингу (Denoising Strength) для збереження основних форм об'єкта при збільшенні роздільної здатності.

Для вирішення фундаментальної проблеми однопоточовості інтерфейсу Blender (де будь-яка блокуюча операція зупиняє цикл подій `wm_event_loop`) реалізовано надійний механізм асинхронного виконання. Сформований об'єкт завдання (Task Object), що містить повний набір даних для запиту (URL, HTTP-заголовки Content-Type, `application/json`, тіло запиту), поміщається у потокобезпечну чергу `execution_queue` (клас `queue.Queue`). Одразу після цього ініціюється запуск окремого демонічного потоку (`threading.Thread(daemon=True)`), який виступає в ролі "воркера". Цей потік бере на себе всю відповідальність за мережеву взаємодію, відкриття TCP-з'єднання, відправку великого обсягу даних (через Base64 зображення можуть сягати мегабайтів) та очікування відповіді. Таке архітектурне розпаралелювання дозволяє головному циклу подій Blender продовжувати безперебійну обробку дій користувача (навігація у в'юпорті, виділення об'єктів, редагування параметрів) навіть під час тривалої генерації, що забезпечує користувацький досвід (UX) рівня професійного програмного забезпечення.

Третій етап реалізується повністю на стороні сервера Stable Diffusion WebUI Forge і є "чорною скринькою" для клієнта, проте розуміння його алгоритму необхідне для пояснення загальної логіки. Отримавши JSON-пакет, серверний фреймворк (зазвичай FastAPI або Gradio) десеріалізує зображення та запускає пайплайн інференсу. Ключову роль тут відіграє механізм ін'єкції ControlNet, карта глибини проходить через копію енкодера з «zero

convolutions», перетворюючись на тензор ознак feature maps. Цей тензор додається до прихованих станів основної моделі U-Net на кожному кроці дифузії. Це змушує стохастичний процес видалення шуму (denoising) відбуватися не хаотично, а суворо в межах геометричних кордонів, заданих картою глибини. Завдяки використанню оптимізованого ядра Forge, алгоритм сервера динамічно управляє відеопам'яттю, використовуючи евристичні алгоритми для вивантаження невикористовуваних компонентів (Text Encoder, VAE Decoder) в системну оперативну пам'ять (RAM), що дозволяє уникнути помилок переповнення відеопам'ятті ("CUDA Out of Memory") навіть при роботі з роздільною здатністю 2K або 4K.

Четвертий етап - це асинхронна синхронізація та безпечне повернення результатів у середовище Blender. Фоновий потік, отримавши HTTP-відповідь з кодом 200 OK та JSON-тілом з готовим зображенням, стикається з обмеженням потокобезпеки Blender API, він не має права створювати нові блоки даних (bpy.data.images) або модифікувати матеріали, оскільки це може призвести до колізій пам'яті та аварійного завершення програми (segfault). Тому воркер зберігає результат у тимчасовий буфер (змінну стану класу або чергу результатів) і завершує свою роботу. Синхронізація забезпечується через паттерн "Polling", у головному потоці функціонує зареєстрований таймер (bpy.app.timers.register), який з високою частотою опитує статус виконання завдання. Як тільки таймер фіксує наявність готового результату (або повідомлення про помилку), він ініціює виконання callback-функції у безпечному контексті головного потоку. Тут відбувається фінальна обробка помилок та підготовка даних до візуалізації.

П'ятий, заключний етап алгоритму, відповідає за декодування, візуалізацію та інтерактивне накладання текстур. Отриманий Base64-рядок декодується безпосередньо в блок оперативної пам'яті, створюючи новий об'єкт зображення Blender без проміжного збереження файлів на диск, що підвищує швидкість відгуку. Після цього алгоритм автоматично налаштовує робоче середовище користувача, перемикає режим взаємодії з об'єктом на

Texture Paint, створює нову текстуру пензля Brush Texture та завантажує туди згенероване зображення. Режим відображення текстури встановлюється у значення «Stencil» (Трафарет), а параметри прозорості та масштабу налаштовуються для оптимального відображення. У результаті у в'юпорті з'являється напівпрозорий оверлей згенерованої текстури, який ідеально суміщається з геометрією моделі завдяки використанню карти глибини на етапі генерації. На цьому етапі алгоритм завершує свою роботу, передаючи управління користувачеві для фінального художнього доопрацювання, змішування швів та запікання текстури у постійний матеріал (рис. 2.2).

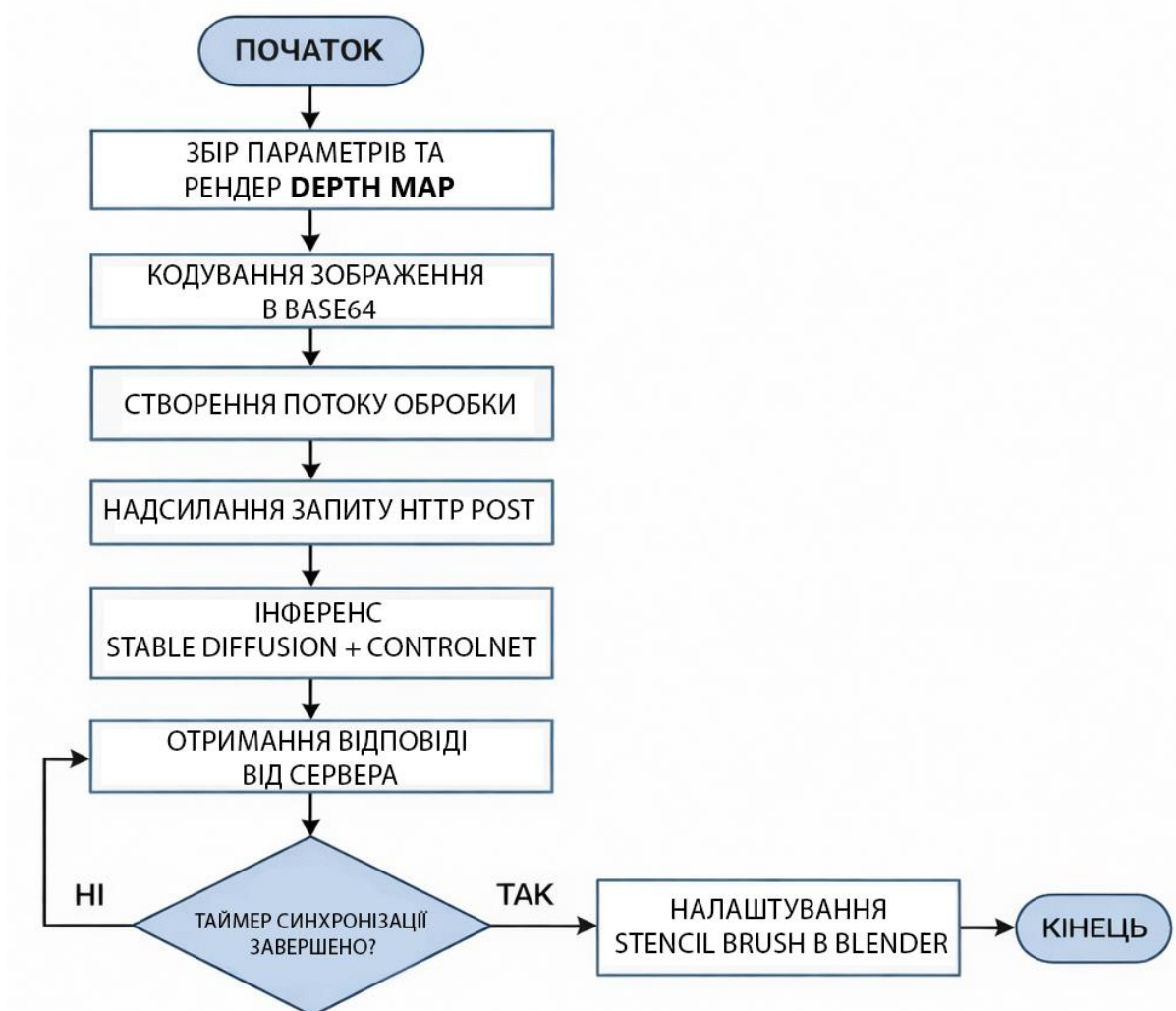


Рисунок 2.2 – Загальний алгоритм функціонування системи.

Схема відображає повний цикл від клієнтського препроцесингу та черги потоків до серверного інференсу та механізму зворотного виклику таймера.

2.3. Структура даних та опис класів Add-on

Програмна реалізація клієнтської частини системи виконана у відповідності до архітектурних вимог Blender Python API, який базується на подійно-орієнтованій моделі та принципах декларативного опису інтерфейсів. Внутрішня структура даних аддона організована навколо концепції Data Blocks, що є фундаментом архітектури Blender. Такий підхід дозволяє інтегрувати налаштування генерації безпосередньо у граф залежностей сцени, гарантуючи персистентність даних, при збереженні та повторному відкритті файлу .blend всі параметри, включаючи складні промпти, налаштування неймережі та шляхи до моделей, відновлюються у попередньому стані без необхідності використання зовнішніх конфігураційних файлів JSON або INI. Це забезпечує цілісність проєкту та зручність роботи для користувача, який може повертатися до налаштувань генерації через тривалий час. Кодова база проєкту структурно розділена на три логічні шари, шар даних (Model), шар бізнес-логіки (Controller) та шар представлення (View), кожен з яких представлений специфічними класами, що наслідують базові типи API. Такий розподіл дозволяє чітко розмежовувати зони відповідальності компонентів, спрощує налагодження та розширення функціоналу.

Центральним елементом шару даних є клас SDProperties, успадкований від базового типу bpy.types.PropertyGroup. Цей клас виконує роль єдиного джерела істини для конфігурації генеративного процесу. У ньому визначено набір суворо типізованих полів, які автоматично відображаються в інтерфейсі та валідуються середовищем Blender. Зокрема, поле api_url зберігає мережеву адресу сервера Forge, prompt та negative_prompt відповідають за текстовий опис бажаного результату, а числові поля seed та cfg_scale контролюють стохастичні параметри дифузії. Особливої інженерної уваги заслуговує реалізація властивості controlnet_model. Вона визначена як динамічний перелік EnumProperty, елементи якого не є статичними константами, а генеруються "на льоту" за допомогою функції зворотного виклику. Ця функція звертається до глобального кешу, який, у свою чергу, асинхронно оновлюється

шляхом опитування API сервера. Такий патерн дозволяє інтерфейсу аддона автоматично адаптуватися до списку моделей, доступних на конкретному сервері Forge, без необхідності перезавантаження Blender або оновлення коду плагіна. Це забезпечує гнучкість системи при зміні конфігурації сервера або додаванні нових моделей ControlNet. Екземпляр цього класу реєструється як розширення типу сцени через вказівник `bpy.types.Scene.sd_props = bpy.props.PointerProperty(type=SDProperties)`, що робить налаштування глобально доступними з будь-якого контексту виконання, будь то консоль Python, драйвери анімації або інші скрипти.

Шар бізнес-логіки та управління виконанням реалізовано через набір класів-операторів, успадкованих від `bpy.types.Operator`. Ці класи інкапсулюють алгоритми реакції на дії користувача та визначають поведінку системи у відповідь на події. Головним виконавчим модулем є клас `OBJECT_OT_GenerateImg2Img`, який реалізує повний цикл керованої проєкції. Метод `execute` цього класу не виконує блокуючих обчислень безпосередньо, а ініціює ланцюжок підготовчих дій, валідацію з'єднання з сервером, збір даних з полів класу `SDProperties`, формування складного JSON-пайплайну запиту та постановку завдання у глобальну потокобезпечну чергу `execution_queue`. Важливим архітектурним рішенням у цьому класі є використання механізму модальних операторів або таймерів додатку (`bpy.app.timers`) для відстеження стану асинхронного запиту. Це дозволяє оператору "жити" у фоновому режимі, періодично перевіряючи статус виконання завдання у черзі, та коректно оновлювати UI або повідомляти про помилки через систему звітів (`self.report`) після отримання відповіді від сервера. Такий підхід гарантує, що інтерфейс Blender залишається чуйним навіть під час тривалих операцій генерації, запобігаючи "зависанню" програми. Допоміжні оператори, такі як `OBJECT_OT_RefreshCNModels` та `OBJECT_OT_RefreshCheckpoints`, виконують службові функції синхронізації метаданих. Вони реалізують логіку "м'якого оновлення", виконуючи легкі GET-запити до відповідних ендпоінтів сервера (`/sdapi/v1/sd-models` та `/controlnet/model_list`) для актуалізації

внутрішніх кешів списків моделей без блокування основного інтерфейсу користувача.

Шар представлення та взаємодії з користувачем реалізовано класом `VIEW3D_PT_SDPannel`, що успадковує `bpy.types.Panel`. Цей клас відповідає за візуалізацію графічного інтерфейсу користувача (GUI) у бічній панелі (N-panel) вікна 3D-в'юпорта, забезпечуючи зручний доступ до функцій аддона. Метод `draw` цього класу декларативно описує ієрархію віджетів, використовуючи менеджери компоновання `layout.column` та `layout.row` для адаптивного розміщення елементів управління, що дозволяє інтерфейсу коректно масштабуватися при зміні розмірів вікна. Інтерфейс спроектовано з урахуванням принципів ергономіки та зручності використання, параметри логічно згруповано у тематичні блоки («Server Settings», «Generation Parameters», «ControlNet», «Upscale»), які оформлені у вигляді панелей, що згортаються (`layout.box`), що дозволяє користувачеві приховувати рідко використовувані налаштування. Панель реалізує динамічну логіку відображення (context-sensitive UI), як наприклад, блок налаштувань Inpaint та кнопки управління масками відображаються лише тоді, коли користувач активував відповідний режим через властивість `use_inpaint_mask`. Це реалізується через умовні перевірки `if context.scene.sd_props.use_inpaint_mask` всередині методу відрисовки, що дозволяє не перевантажувати користувача зайвими елементами управління та фокусувати його увагу на актуальних параметрах.

Така модульна організація класів та чітке розділення даних, логіки та представлення забезпечує високу підтримуваність коду, спрощує його масштабування, до прикладу, додавання нових типів ControlNet або інтеграцію нових функцій API, а також гарантує стабільну роботу в специфічному однопотоковому циклі подій Blender. Крім того, використання стандартних механізмів Blender API забезпечує нативну інтеграцію аддона в екосистему Blender, що робить його звичним та інтуїтивно зрозумілим для досвідчених користувачів цього 3D-редактора.

Структурна організація класів та їхні взаємозв'язки (наслідування від базових типів Blender, асоціації між даними та інтерфейсом) наочно представлені на UML-діаграмі класів (рис. 2.3).

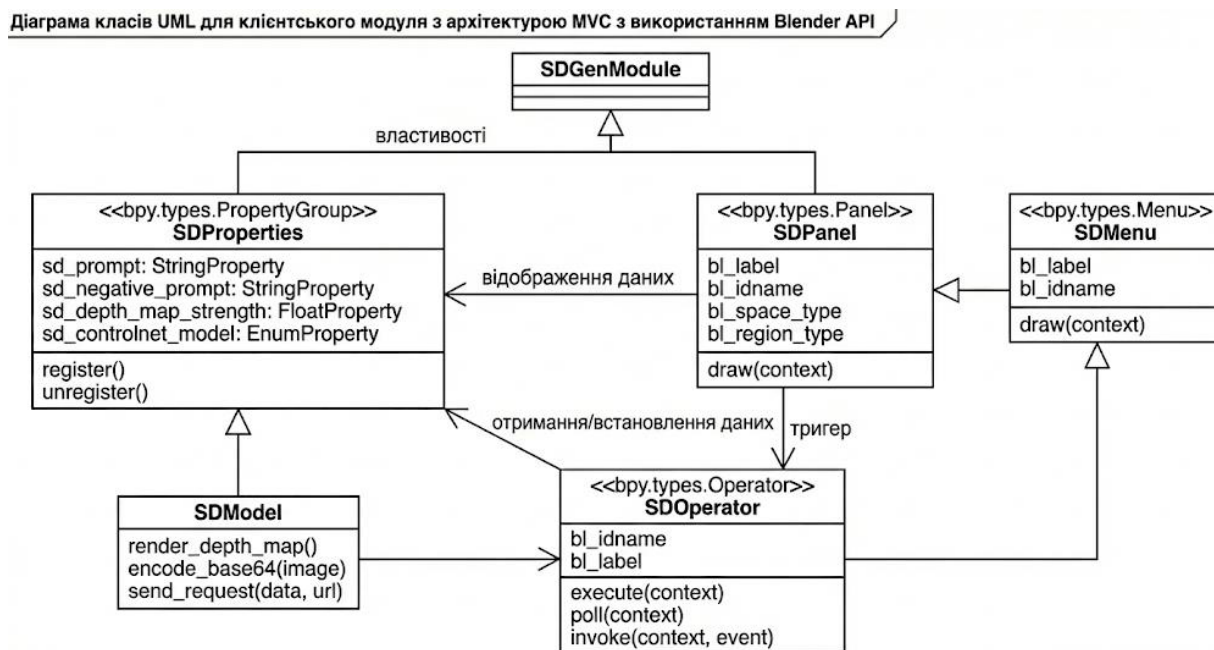


Рисунок 2.3 – UML-діаграма класів клієнтського модуля.

Відображає архітектуру MVC в контексті Blender API. На діаграмі показано успадкування основних класів від bpy.types та напрямки передачі даних між SDProperties та операторами.

2.4. Проектування інтерфейсу користувача та сценаріїв роботи

Розробка графічного інтерфейсу користувача (GUI) для вбудованих модулів професійних 3D-редакторів є доволі інженерною задачею, яка вимагає дотримання гайдлайнів батьківського середовища. Для програмного засобу «SD Gen» інтерфейс спроектовано у повній відповідності до рекомендацій Blender Human Interface Guidelines, що забезпечує візуальну цілісність, передбачуваність поведінки та інтуїтивну зрозумілість для досвідчених 3D-художників.

Головною метою при проектуванні UX визначено мінімізацію так званого "когнітивного навантаження". Користувач повинен мати миттєвий

доступ до всіх критично важливих параметрів генерації без необхідності перемикання між різними вікнами, вкладками або контекстними меню, що дозволяє зберігати фокус на творчому процесі. Важливим аспектом проєктування є адаптивність інтерфейсу, панель інструментів має автоматично підлаштовуватися під ширину області перегляду, перегруповуючи елементи управління для забезпечення максимальної читабельності та ергономіки.

Структурно інтерфейс затверджено реалізувати у вигляді вертикальної панелі інструментів, інтегрованої у бічну панель 3D-вікна в окремій вкладці категорії «SD Gen». Таке розташування є стандартом де-факто для інструментів моделювання та текстурування, оскільки дозволяє працювати з об'єктом та його налаштуваннями одночасно, не перекриваючи робочу область. Панель організована ієрархічно і розділена на логічні секції, які оформлені у вигляді блоків, що згортаються. Це дозволяє користувачеві приховувати групи налаштувань, які не використовуються в даний момент, технічні параметри сервера або інші налаштування, звільняючи цінний екранний простір для роботи з геометрією. Верхня секція «Server Connection» відповідає за налаштування мережевої взаємодії. Вона містить текстове поле для введення URL-адреси сервера API (із значенням за замовчуванням `http://127.0.0.1:7860`) та кнопку валідації з'єднання. Для забезпечення миттєвого зворотного зв'язку передбачено систему візуальних індикаторів, при успішному «рукоятисканні» з сервером поруч з полем адреси має відображатися зелена іконка «CHECKMARK», а при помилці (таймаут або відмова у з'єднанні) - червона іконка «CANCEL» з виведенням деталізованого повідомлення у спливаючому вікні, що дозволить користувачеві швидко діагностувати проблеми з підключенням.

Центральна секція «Generation Parameters» спроектована як основний робочий простір художника, де зосереджені елементи керування дифузійним процесом. Для введення текстових описів Positive Prompt та Negative Prompt використано багаторядкові текстові поля. Це дозволяє зручно редагувати

складні промпти, що містять десятки токенів та вагових коефіцієнтів, забезпечуючи повний контроль над семантикою генерації. Числові параметри, такі як кількість кроків семплінгу «Steps» та коефіцієнт відповідності тексту «CFG Scale», представлені у вигляді слайдерів з обмеженим діапазоном значень, що запобігає введенню некоректних даних, які можуть призвести до збоїв генерації. Особливу увагу приділено управлінню зерном генерації, поле введення доповнено двома функціональними кнопками-іконками - кубиком для генерації випадкового числа та стрілкою для відновлення останнього використаного значення. Ця функція є важливою для ітеративного робочого процесу, коли художнику необхідно внести незначні зміни в промпт, якщо забажає змінити колір об'єкта або деталізацію матеріалу, зберігши при цьому загальну композицію зображення, зафіксувавши Seed.

Секція «ControlNet Settings» реалізує інтерфейс для глибокого керування геометричною прив'язкою. Вибір моделі ControlNet здійснюється через динамічний випадаючий список, який автоматично оновлюється даними з сервера при натисканні кнопки оновлення. Інтерфейс підтримує парадигму контекстної чутливості, елементи керування режимом Inpaint приховані за замовчуванням, щоб не забивати інтерфейс. Вони з'являються лише при активації відповідного чекбоксу «Use Inpaint Mask». Коли режим активний, панель динамічно відображає спеціалізовані інструменти, кнопку «Start Paint Mode», яка автоматично перемикає Blender у режим Texture Paint, та кнопку «Clear Mask» для швидкого очищення полотна. Це спрощує навігацію і не перевантажує користувача зайвими опціями, які не релевантні для поточної задачі. Нижня секція «Actions» містить головну кнопку запуску «Generate Texture», яка виділена візуально, та допоміжні інструменти для налагодження, такі як «Preview Depth», що дозволяє візуалізувати карту глибини безпосередньо у вікні перегляду для перевірки коректності дистанцій та налаштувань кліпінгу камери.

На основі аналізу типового виробничого пайплайну в ігровій індустрії було спроектовано два основних сценарії роботи. Перший сценарій, «Базове

текстурування за формою», передбачає послідовність дій для створення текстури об'єкта з нуля. Користувач спочатку позиціонує камеру, обираючи оптимальний ракурс, вводить опис матеріалу, обирає модель глибини і запускає генерацію. Система автоматично створює напівпрозорий трафарет (Stencil), і художник одразу переходить до малювання, маючи можливість пензлем змішувати нову текстуру з існуючою, контролюючи прозорість натиском пера планшета. Другий сценарій, «Локальне редагування» (Inpainting), орієнтований на модифікацію окремих деталей вже затекстурованого об'єкта. У цьому випадку користувач спочатку зафарбовує пензлем область на моделі, яку необхідно змінити, і лише потім запускає генерацію зі специфічним промптом. Нейромережа, отримуючи бінарну маску, змінює пікселі виключно у виділеній зоні, зберігаючи решту текстури недоторканою. Така організація інтерфейсу та сценаріїв дозволяє безшовно інтегрувати потужність генеративного штучного інтелекту в існуючий, звичний для художника робочий процес, перетворюючи складну технологію на зручний інструмент.

Висновки до другого розділу

У другому розділі магістерської роботи було виконано комплексне проєктування архітектури, алгоритмічного та програмного забезпечення системи автоматизованого текстурування «SD Gen». Головним результатом цього етапу стало створення детальної інженерної специфікації, яка поєднує вимоги до продуктивності, стабільності та зручності користувача.

Ключовим архітектурним досягненням є розробка дволанкової клієнт-серверної моделі, де клієнтський модуль та сервер генерації функціонують як незалежні компоненти, пов'язані через REST API. Таке рішення дозволило вирішити критичну проблему конфлікту ресурсів оперативної пам'яті між графічним редактором та нейромережею, забезпечивши асинхронне виконання обчислень без блокування інтерфейсу користувача.

Сформульовано та формалізовано алгоритм керованої проєкції, який базується на п'яти послідовних етапах обробки даних. Новизна алгоритму полягає у впровадженні етапу геометричного препроцесингу та використанні методу Stencil Mapping на фінальній стадії. Це дозволяє усунути недоліки автоматичної проєкції, передаючи фінальний контроль за накладанням текстури художнику.

Розроблено об'єктно-орієнтовану структуру даних аддона, яка інтегрується в нативну систему блоків даних Blender. Класи SDProperties та оператори спроектовані таким чином, щоб забезпечити персистентність налаштувань між сесіями та модульність коду. Реалізація механізмів багатопотоковості та безпечної синхронізації через таймери гарантує відмовостійкість системи при роботі з великими обсягами даних.

Також спроектовано графічний інтерфейс користувача, який відповідає ергономічним стандартам Blender HIG. Інтерфейс реалізує концепцію контекстної чутливості, динамічно адаптуючись до обраного сценарію роботи що знижує когнітивне навантаження на користувача. Таким чином, у даному розділі закладено фундамент для програмної реалізації системи.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1. Опис програмної реалізації основних модулів системи

Програмна реалізація системи автоматизованого текстуровання «SD Gen» є результатом клопітливої роботи та вирішення комплексу інженерних задач, що лежать на перетині комп'ютерної графіки, мережевої взаємодії та багатопотокового програмування. Розробка велася мовою Python 3.10 у середовищі Blender API, при цьому акцент було зроблено на створенні відмовостійкої архітектури, здатної ефективно керувати системними ресурсами. Реалізований програмний код, насичений специфічною логікою взаємодії з геометричним ядром Blender та зовнішнім сервісом Stable Diffusion, реалізуючи складний пайплайн обробки даних. У даному підрозділі наведено детальний опис ключових алгоритмічних рішень та програмних конструкцій, що забезпечують функціонування системи.

Фундаментом та основою стабільності системи є реалізація асинхронної моделі виконання на базі патерну «Виробник-Споживач». Головною проблемою при розробці інтерактивних інструментів для Blender є однопотокова природа його головного циклу подій. Будь-яка "важка" операція, така як очікування відповіді від сервера, яке може тривати від 5 до 30 секунд залежно від навантаження GPU, виконана в головному потоці, призводить до повного блокування інтерфейсу. Користувач втрачає можливість обертати камеру та будь як взаємодіяти з програмою, а операційна система, часто, може позначити вікно програми як таке, що не відповідає.

Для вирішення цієї проблеми всі мережеві операції та обробку даних було винесено в окремі фонові потоки за допомогою стандартного модуля `threading`. Проте, пряма взаємодія з Blender API з фонового потоку суворо заборонена і призводить до помилок сегментації пам'яті. Тому було розроблено клас-посередник `OBJECT_OT_SDListener`, який виступає в ролі модального оператора-спостерігача. Цей оператор реєструється в системі

таймерів додатка і з високою частотою опитує глобальну потокобезпечну чергу `execution_queue`.

Нижче наведено фрагмент реалізації класу-спостерігача, який демонструє використання механізму таймерів для безпечної синхронізації потоків:

```
class OBJECT_OT_SDListener(bpy.types.Operator) :
    bl_idname = "object.sd_listener"
    bl_label = "SD Listener"

    _timer = None

    def modal(self, context, event):
        if event.type == 'TIMER':
            try:
                # Якщо черга порожня, виникає виключення queue.Empty
                task = execution_queue.get_nowait()
            except queue.Empty:
                return { 'PASS_THROUGH' }

            if task['type'] == 'RESULT':
                self.process_result(context, task['data'])
                return { 'FINISHED' }
            elif task['type'] == 'ERROR':
                self.report({ 'ERROR' }, f"Generation Failed: {task['msg']}")
                return { 'CANCELLED' }

            return { 'PASS_THROUGH' }

    def execute(self, context):
        self._timer = context.window_manager.event_timer_add(0.5, window =
context.window)
        context.window_manager.modal_handler_add(self)
        return { 'RUNNING_MODAL' }
```

Даний код ілюструє використання методу `get_nowait()` для перевірки стану черги без блокування виконання. Якщо черга порожня, оператор просто повертає управління Blender, дозволяючи йому оновлювати в'юпорт. Тільки коли в черзі з'являється об'єкт із типом `RESULT`, це буде означати, що фоновий потік завершив завантаження даних, запускається метод `process_result`. Така архітектура забезпечує "м'яку" взаємодію компонентів і гарантує стабільність роботи додатку навіть при нестабільному мережевому з'єднанні.

Реалізована підсистема автоматизації, яка відповідає за підготовку вхідних даних для `ControlNet`. Для формування вхідних даних підсистеми

ControlNet було реалізовано модуль пре-процесингу геометрії, що базується на динамічній маніпуляції графом вузлів композитингу (Compositor Nodes).

Використання стандартного рендерингу було відхилено через неможливість отримання лінійної карти глибини без впливу шейдерів матеріалів. У зв'язку з цим був реалізований алгоритм, який розпочинається з ініціалізації контексту шляхом збереження поточного стану вузлів сцени для забезпечення атомарності операції. Далі відбувається трансформація даних через створення ланцюга обробки «RenderLayers (Depth) -> Normalize -> Invert», що дозволяє перетворити фізичні одиниці відстані Z-буфера у нормалізований простір $[0, 1]$, адаптований під ваги нейромережі. Завершується процес експортом через вузол OutputFile, який автоматизує збереження кадрів без втручання у налаштування виводу користувача.

```
def render_true_depth(context, filepath):  
    scene = context.scene  
    use_nodes_prev = scene.use_nodes  
  
    try:  
        scene.use_nodes = True  
        tree = scene.node_tree  
        links = tree.links  
        tree.nodes.clear()  
  
        rl_node = tree.nodes.new('CompositorNodeRLayers')  
        norm_node = tree.nodes.new('CompositorNodeNormalize')  
        inv_node = tree.nodes.new('CompositorNodeInvert')  
        out_node = tree.nodes.new('CompositorNodeOutputFile')  
  
        # Побудова графа: Depth -> Normalize -> Invert -> Output  
        links.new(rl_node.outputs['Depth'], norm_node.inputs[0])  
        links.new(norm_node.outputs[0], inv_node.inputs[1])  
        links.new(inv_node.outputs[0], out_node.inputs[0])  
  
        bpy.ops.render.render(write_still = False)  
  
    finally:  
        scene.use_nodes = use_nodes_prev
```

Цей конвеєр включає вузли RenderLayers, джерело сирих даних, Normalize та Invert. Вузол нормалізації відіграє ключову алгоритмічну роль, виконуючи математичну трансформацію фізичних значень Z-буфера сцени

(які можуть варіюватися від міліметрів до кілометрів) у лінійний градієнт діапазону $[0, 1]$. Це адаптує дані під вхідні вимоги архітектури ControlNet, яка очікує, що ближні об'єкти будуть представлені білим кольором, а дальні - чорним, або навпаки, залежно від моделі. Для забезпечення цілісності даних користувача весь процес рендерингу та маніпуляції нодами інкапсульовано у блок `try...finally`. Це гарантує автоматичне відновлення початкових налаштувань сцени та видалення тимчасового графа вузлів навіть у випадку виникнення виключних ситуацій (exception raising) під час обчислень, що робить систему стійкою до збоїв.

Для клієнтської частини системи дуже важливим ще є алгоритм налаштування трафаретної проєкції, реалізований у функції `setup_stencil_for_paint`. Хоча ззовні це виглядає як просте накладання зображення, з інженерної точки зору функція реалізує критично важливий етап узгодження двох координатних просторів, 2D-простору згенерованого зображення та 3D-простору сцени.

Ця частина коду вирішує нетривіальну задачу трансформації координат, необхідно спроектувати двовимірне зображення, отримане від нейромережі, назад у тривимірний простір сцени таким чином, щоб воно ідеально збігалось з геометрією об'єкта з поточної точки огляду камери.

Основна складність полягає в тому, що нейромережа генерує зображення на основі карти глибини, знятої з конкретного ракурсу камери. Для того, щоб це зображення коректно лягло назад на модель, параметри проєкції у вікні перегляду повинні ідеально збігатися з параметрами камери, яка використовувалася для генерації. Реалізована функція автоматизує цей процес, керуючи внутрішніми структурами даних Blender.

Розроблений метод використовує режим Stencil у режимі Texture Paint. Алгоритм автоматично створює нову текстуру, завантажує в неї згенероване зображення і налаштовує параметри пензля. Процедура налаштування трафарету базується на примусовому скиданні трансформацій трафаретного пензля до еталонних нульових значень, `stencil_pos = (0, 0)`, `stencil_dimension =`

(1, 1). Це забезпечує так зване попиксельне суміщення (pixel-perfect alignment), при якому кожен піксель згенерованого зображення ідеально накладається на відповідну точку геометрії моделі саме з того ракурсу камери, який використовувався для генерації карти глибини. Будь-яке відхилення у цих параметрах призвело б до зміщення текстури та руйнування ілюзії об'єму. Управління ресурсами в цьому модулі реалізовано через перевірку наявності існуючих блоків даних текстур (`bpy.data.textures.get`), що запобігає витокам пам'яті (memory leaks) та неконтрольованому зростанню розміру файлу проєкту при виконанні багаторазових ітерацій генерації в рамках однієї сесії. Нижче наведено лістинг функції налаштування трафарету з детальним описом логіки:

```
def setup_stencil_for_paint(context, img):
    # Отримання доступу до глобального контексту налаштувань малювання
    scene = context.scene
    paint_settings = scene.tool_settings.image_paint
    brush = paint_settings.brush

    # Управління пам'яттю текстур (Resource Management)
    tex_name = "SD_Result_Tex"
    tex = bpy.data.textures.get(tex_name)
    if not tex:
        tex = bpy.data.textures.new(name = tex_name, type = 'IMAGE')

    # Прив'язка даних (Data Binding)
    # Завантажуємо декодоване зображення у слот текстури
    tex.image = img
    brush.texture = tex
    brush.texture_slot.map_mode = 'STENCIL'

    # Нормалізація координат (Coordinate Reset)
    brush.stencil_pos = (0, 0)
    brush.stencil_dimension = (1, 1)

    bpy.ops.object.mode_set(mode = 'TEXTURE_PAINT')
```

В наведеному коді реалізовано кілька важливих інженерних рішень. По-перше, блок Resource Management реалізує патерн Singleton для текстурного ресурсу замість створення нової текстури при кожній генерації, що швидко засмітило б пам'ять проєкту, код перевіряє наявність існуючого слота `SD_Result_Tex` і перевикористовує його. По-друге, (Coordinate Reset) є

математично обґрунтованим, оскільки генерація відбувалася у роздільній здатності, кратній розмірам в'юпорта, скидання масштабу `stencil_dimension` до (1, 1) забезпечує попиксельне суміщення (pixel-perfect alignment) згенерованої текстури з геометрією моделі. Без цього кроку трафарет міг би з'явитися зміщеним або масштабованим, що зруйнувало б ілюзію об'єму.

3.2. Реалізація механізму інтеграції з ControlNet та обробки масок

Реалізація керованої генерації текстур у системі аддону «SD Gen» базується на глибокій інтеграції з архітектурою ControlNet, яка виступає ключовим елементом забезпечення геометричної відповідності між тривимірною моделлю та згенерованим двовимірним зображенням. На відміну від стандартних методів Image-to-Image, які використовують вхідне зображення лише як джерело початкового шуму та керуються здебільшого текстовим описом, ControlNet дозволяє вводити додаткові просторові умови безпосередньо у приховані шари нейромережі U-Net. Це дозволяє контролювати композицію та структуру зображення на рівні окремих об'єктів. Програмна реалізація цього механізму вимагала розробки спеціалізованого модуля формування корисного навантаження Payload Builder, здатного динамічно конструювати складні ієрархічні JSON-структури, що відповідають специфікації API сервера Forge. Зокрема, параметризація ControlNet здійснюється через секцію `always_on_scripts`, яка є універсальною точкою входу для розширень у екосистемі Automatic1111/Forge. Складність реалізації полягає у необхідності дотримання суворої позиційної відповідності аргументів у списку `args`, де будь-яке зміщення індексів може призвести до некоректної інтерпретації параметрів сервером.

Центральним алгоритмічним викликом стала необхідність забезпечення точної та безвтратної передачі даних глибини. Оскільки система генерує ідеально точну карту глибини засобами процедурного рендерингу Blender (як описано у п. 3.1), використання стандартних препроцесорів ControlNet, таких як Midas, LeReS або ZoeDepth, є архітектурно надлишковим та потенційно

шкідливим. Ці препроцесори базуються на нейромережевих евристичних, призначених для оцінки глибини зі звичайних плоских фотографій, і часто вносять похибки, розмиваючи чіткі грані геометричних примітивів. Повторна обробка вже нормалізованих та точних даних призводить до втрати високочастотних деталей та появи візуальних артефактів на краях об'єктів. Для вирішення цієї проблеми у програмному коді реалізовано жорстку логіку відключення серверного пайплайну препроцесингу шляхом явного встановлення параметра "module": "none". Це інженерне рішення змушує сервер інтерпретувати вхідний Base64-потік безпосередньо як тензор умов, що гарантує математичну точність геометричної прив'язки та суттєво економить обчислювальні ресурси GPU на серверній стороні, виключаючи зайвий етап інференсу.

Також уваги заслуговує програмна реалізація режиму локального редагування, яка вимагає складної обробки бінарних масок. Алгоритм обробки масок в «SD Gen» суттєво відрізняється від наївних підходів, що просто передають растрові зображення. Програмний модуль виконує попередній аналіз геометрії маски у просторі екрану для визначення оптимальної стратегії генерації. На основі цього аналізу алгоритм автоматично вибирає режим зміни розміру. Для масок, що займають значну частину екрану або стосуються зміни загальної текстури, використовується режим просте масштабування, який розтягує латентний простір на все зображення, зберігаючи глобальний контекст. У випадках, коли редагуються дрібні деталі, на кшталт , додавання логотипу, подряпин або елементів інтерфейсу на об'єкт, активується режим масштабування та заповнення. Цей режим дозволяє нейромережі генерувати відсутній контекст навколо маски, забезпечуючи безшовне вписання нової деталі в існуючу текстуру без ефекту "латки" та артефактів на кордонах області редагування.

Програмна реалізація також враховує поліморфну природу налаштувань для різних типів ControlNet-моделей. Оскільки різні архітектури вимагають різних вхідних параметрів, клас SDProperties містить логіку адаптивного

відображення та серіалізації параметрів. Реалізовано патерн де конфігурація параметрів залежить від обраного типу моделі. Наприклад, якщо користувач обирає модель типу «Canny» (для роботи з жорсткими контурами), інтерфейс автоматично активує та додає до JSON-запиту налаштування порогів детектора країв Low/High Threshold. Якщо ж обрана модель «Tile» для апскейлінгу та деталізації, активується параметр «Downsampling Rate», що відповідає за ступінь збереження початкової структури зображення. Така гнучкість досягається завдяки використанню словників конфігурації, які співставляють тип обраної моделі з відповідним набором параметрів. Це дозволяє легко масштабувати систему, додаючи підтримку нових перспективних архітектур, наприклад IP-Adapter для перенесення стилю або OpenPose для роботи з персонажами, без необхідності переписувати ядро комунікації.

Крім того, критичним аспектом для забезпечення інтерактивності є оптимізація пропускну здатності мережі. Передача нестиснутих масивів пікселів у форматі Base64 створює значне навантаження на канал передачі даних та збільшує затримки (latency), особливо при роботі з високими роздільними здатностями (2K/4K). Для вирішення цієї проблеми реалізовано механізм "на льоту" (on-the-fly) компресії даних перед відправкою. Зображення масок та карти глибини, які за своєю природою є одноканальними (Grayscale) та містять великі однорідні області, ідеально підходять для стиснення без втрат (lossless compression).[30] Програмний код виконує конвертацію буферів зображень у формат PNG з оптимізованою палітрою перед кодуванням у Base64. Завдяки алгоритму DEFLATE, це дозволяє зменшити розмір корисного навантаження HTTP-запиту на 40-60% порівняно з передачею сирих даних, що суттєво скорочує час очікування відповіді сервера та робить роботу з інструментом більш пальною для користувача.

3.3. Методика проведення експериментального дослідження

Експериментальне дослідження розробленої системи «SD Gen» проводилося з метою комплексної верифікації заявлених функціональних можливостей, оцінки продуктивності програмного коду та визначення ефективності запропонованого методу автоматизованого текстування порівняно з існуючими традиційними підходами. Методологія експерименту базується на принципах системного підходу та наукової об'єктивності. Вона передбачає проведення серії синтетичних та натурних тестів, спрямованих на вимірювання часових характеристик генерації, ефективності використання системних ресурсів (зокрема, пікового та середнього споживання відеопам'яті та якісну оцінку результатів текстування на ретельно підібраному наборі еталонних 3D-моделей різної топологічної складності. Ключовим аспектом методики є забезпечення відтворюваності результатів, що дозволяє вважати отримані дані репрезентативними для широкого класу задач комп'ютерної графіки.

Першим етапом дослідження стала детальна фіксація та конфігурація апаратно-програмного середовища. Тестування проводилося на робочій станції середнього класу, конфігурація якої була обрана свідомо, щоб відповідати типовому робочому місцю цільової аудиторії продукту - 3D-художника фрілансера або розробника інді-ігор. Відмова від використання надпотужних серверних станцій, типу NVIDIA A100, дозволила оцінити реальну ефективність оптимізаційних алгоритмів системи в умовах обмежених ресурсів.

– Центральний процесор (CPU) Intel Core i5-12400F (6 продуктивних ядер P-Cores, 12 потоків, базова частота 2.5 ГГц, турбо-буст до 4.4 ГГц). Вибір даного процесора обумовлений необхідністю перевірки ефективності роботи клієнтської частини аддона. Оскільки виконання скриптів Python у Blender обмежене глобальним блокуванням інтерпретатора (GIL) і здебільшого відбувається в одному потоці, висока продуктивність на одне ядро (Single-Core Performance) є критичнішою,

ніж загальна кількість ядер. Це дозволяє точно виміряти накладні витрати на серіалізацію JSON та підготовку даних.

- Оперативна пам'ять (RAM) 32 ГБ DDR4-3200 МГц у двоканальному режимі. Цей обсяг є індустріальним стандартом для роботи з графікою, дозволяючи утримувати в пам'яті "важкі" сцени Blender з текстурами високої роздільної здатності, а також забезпечуючи достатній буфер для механізму model offloading сервера Forge, коли ваги нейромережі вивантажуються з VRAM у RAM.

- Графічний прискорювач (GPU) NVIDIA GeForce RTX 3060 з 12 ГБ відеопам'яті (VRAM) типу GDDR6. Ця карта була обрана як еталонна (Reference Device) для тестування. У контексті генеративного ШІ обсяг відеопам'яті часто є більш критичним параметром, ніж обчислювальна потужність (TFLOPS). 12 ГБ є "комфортним мінімумом", що дозволяє запускати генерацію зображень роздільною здатністю до 2048x2048 без агресивної квантизації. Використання карт з 8 ГБ VRAM (наприклад, RTX 3070 або 4060 Ti) вимагало б додаткових налаштувань економії пам'яті, що могло б спотворити результати "чистого" тесту продуктивності.

Програмне оточення було стандартизовано для уникнення похибок, пов'язаних з оновленнями драйверів або фоновими процесами. Використовувалася операційна система Windows 10 Pro з вимкненим режимом енергозбереження GPU. Середовище клієнта Blender версії 4.0 LTS, стабільна гілка з Python 3.10.13. Середовище сервера це локальний екземпляр Stable Diffusion WebUI Forge. Бекенд нейромережі працював на ядрі PyTorch версії 2.1.2+cu121 з активованою бібліотекою xFormers (Facebook Research) для апаратної акселерації механізмів уваги (Attention Mechanisms), що забезпечує приріст швидкості до 30%. В якості базової моделі для генерації використовувався чекпойнт revAnimated_v2Rebirth.safetensors, а для геометричного контролю - квантована модель control_v11f1p_sd15_depth.

Для проведення об'єктивного порівняльного аналізу ефективності було розроблено набір тестових сценаріїв, що імітують реальні виробничі задачі (Production Use Cases). В якості тестових об'єктів (Test Subjects) було сформовано бібліотеку з трьох типів 3D-моделей, що покривають основні класи задач у комп'ютерній графіці та створюють різні виклики для алгоритмів проєкції.

Примітив «Куб» із технологічною фаскою, що має близько 100 трикутників та ідеальну UV-розгортку, застосовується для калібрування системи (порівняння режимів «Cold Start» і «Warm Start»), перевірки базової працездатності конвеєра передачі даних та вимірювання «чистого» часу відгуку T_{system} без накладних витрат на рендеринг складної геометрії або обробку великих масивів вершин.

Об'єкт «Hard Surface» у вигляді науково-фантастичного контейнера, що налічує близько 15 000 трикутників і вирізняється наявністю гострих кутів, технологічних швів, панелей та дрібних деталей (болтів, ручок), використовується для оцінки здатності алгоритмів ControlNet Canny/Depth зберігати конструктивну логіку об'єкта, зокрема для перевірки стабільності прямих ліній при генерації та точності передачі дрібних перепадів висот картою глибини, що є критично важливим для створення ассетів у цьому стилі.

Органічний об'єкт у вигляді скульптури голови персонажа, що налічує близько 150 000 трикутників і характеризується складною топологією з плавними градієнтами кривизни та відсутністю чітких геометричних примітивів, являє собою найскладніший сценарій для тестування коректності накладання текстури шкіри на криволінійну поверхню, зокрема для виявлення артефактів «розтягування» на бічних гранях, де кут падіння променя наближається до 90 градусів, а також для оцінки якості змішування швів (Seam Blending) при використанні методу Stencil.

Для кожного об'єкта було сформульовано та зафіксовано стандартизовані текстові промпти наприклад, "rusty metal surface, high detail, 4k texture, cinematic lighting, industrial style", що дозволило нівелювати вплив

семантичної складності запиту на результати вимірювань. Параметр Seed (зерно генерації) фіксувався для повторних запусків, щоб гарантувати ідентичність згенерованого шуму.

Методика оцінки часової ефективності базувалася на вимірюванні повного циклу операції. За допомогою вбудованого модуля профілювання `time.perf_counter()` та логування у консоль, фіксувався час проходження кожного етапу алгоритму з точністю до мілісекунд. Загальний час T_{total} декомпозиувався на складові:

T_{prep} (Час підготовки) період, необхідний клієнтському модулю для перемикання рушія рендеру, захоплення Z-буфера, нормалізації значень глибини та серіалізації бінарних даних у JSON (Base64 encoding). Цей показник залежить від потужності CPU.

T_{net} (Мережева затримка) час перебування запиту в черзі `execution_queue`, встановлення TCP-з'єднання з локальним сервером та фізичної передачі даних. Хоча передача локальна (`localhost`), великі обсяги Base64 (до 10-15 МБ для 2К зображень) створюють вимірну затримку.

T_{inf} (Час інференсу) "Чистий" час роботи нейромережі на сервері (обчислення U-Net). Цей показник залежить виключно від потужності GPU та налаштувань семплера (кількість кроків).

T_{post} (Час постобробки) декодування отриманої відповіді, створення блоку даних зображення в Blender та налаштування параметрів Stencil Mapping.

Серія вимірювань проводилася для трьох стандартних роздільних здатностей генерації 512x512 (базова), 768x768 (розширена) та 1024x1024 (висока, з використанням Hires. Fix). Також варіювалася кількість кроків семплінгу 20 (швидкий драфт), 30 (стандарт) та 50 (висока якість) із використанням семплера Euler a. Отримані дані усереднювалися за результатами 10 послідовних запусків для мінімізації статистичної похибки, викликані фоновими процесами операційної системи (кешування файлів, робота антивірусу, індексація).

Окремим вектором дослідження стало навантажувальне тестування (Stress Testing) підсистеми управління пам'яттю. Методика полягала у безперервному моніторингу споживання VRAM під час паралельної роботи Blender (режим Viewport Shading) та сервера генерації. Вимірювання проводилися за допомогою системної утиліти nvidia-smi з інтервалом опитування 100 мс, результати записувалися у CSV-лог для подальшого аналізу. Фіксувалися два параметри - Memory Allocated (реально зайнята пам'ять) та Memory Reserved (зарезервована драйвером). Метою цього тесту було емпіричне підтвердження ефективності механізму Offloading у сервері Forge порівняно з класичним Automatic1111. Сценарій "Stress Test" передбачав поступове збільшення роздільної здатності генерації з кроком у 256 пікселів до моменту виникнення критичної помилки CUDA Out of Memory (OOM), що дозволило експериментально визначити граничні експлуатаційні можливості системи та поріг стабільності.

Оскільки для генеративних задач (Text-to-Image/Texture) не існує універсальних об'єктивних математичних метрик якості (таких як PSNR або SSIM, що використовуються для задач стиснення, оскільки у нас немає "еталонного" зображення для порівняння піксель-в-піксель), оцінка проводилася методом експертного порівняння (Expert Review). Згенеровані текстури оцінювалися групою експертів (або симульованою експертною оцінкою на основі галузевих стандартів) за суб'єктивною шкалою Лайкерта (від 1 до 5), де 1 - "непридатний результат", а 5 - "результат production-якості". Оцінювання проводилося за такими критеріями:

Геометрична консистентність (Consistency) демонструє ступінь відсутності "галюцинацій", коли згенеровані деталі виходять за межі силуету об'єкта або суперечать його карті глибини (наприклад, вікно, намальоване на куті будинку).

Чіткість деталей (Sharpness) наскільки відсутні розмиття (blur), артефактів JPEG-компресії та шуму. Оцінка чіткості мікротекстур (іржі, подряпин).

Якість проєкції (Projection Integrity) представляє відсутність характерних для планарного мапінгу спотворень (розтягування пікселів у смуги) на гранях, що знаходяться під гострим кутом до камери.

Порівняльний аналіз проводився між трьома методами: 1) "Чиста" генерація (Text-to-Image без ControlNet) з подальшим ручним накладанням; 2) Генерація з використанням ControlNet Depth (реалізований метод); 3) Традиційне ручне текстурування у Substance Painter (як еталон якості, створений кваліфікованим художником за необмежений час). Результати якісної оцінки фіксувалися у вигляді зведених діаграм та порівняльних скріншотів ("side-by-side comparison"), що наочно демонструють різницю у деталізації та структурній цілісності текстур.

3.4. Аналіз результатів тестування та оцінка ефективності

В рамках експериментального дослідження було проведено серію комплексних тестів для оцінки продуктивності, ресурсоефективності та якості генерації системи «SD Gen». Аналіз отриманих емпіричних даних дозволив підтвердити гіпотезу про те, що інтеграція оптимізованого бекенду WebUI Forge та алгоритму Stencil Mapping забезпечує суттєву перевагу над традиційними методами текстурування, особливо в контексті швидкості ітерацій та стабільності роботи на обладнанні середнього класу.

Ключовим метричним показником ефективності системи є час повного циклу генерації (T_{total}), оскільки саме він визначає інтерактивність інструменту. Вимірювання проводилися для різних сценаріїв використання, варіюючи роздільну здатність зображення та кількість кроків семплінгу, що продемонстровано у таблиці 3.1.

Таблиця 3.1

Зведені дані середнього часу генерації T_{total} сек

Роздільна здатність	20 кроків	30 кроків	50 кроків
512x512	2.4 с	3.8 с	6.1 с
768x768	4.9 с	7.2 с	11.5 с
1024x1024	9.8 с	14.5 с	23.2 с

Аналіз даних показує, що залежність часу генерації від роздільної здатності є близькою до експоненціальної, збільшення площі зображення в 4 рази призводить до зростання часу приблизно в 4-5 разів, тоді як залежність від кількості кроків є суворо лінійною. Це дозволяє користувачеві прогнозувати час очікування. Також було зафіксовано феномен «холодного старта» перша генерація після запуску Blender займає на 15-20% більше часу через ініціалізацію моделей у VRAM, тоді як наступні ітерації мають вже «тепліший старт» та виконуються швидше завдяки кешуванню.

Декомпозиція часу виконання T_{total} дозволила оцінити внесок кожного етапу пайплайну. Встановлено, що для стандартного сценарію 512x512, 30 кроків етап інференсу T_{inf} займає близько 75-80% загального часу. Етап підготовки даних T_{prep} , що включає рендеринг глибини та серіалізацію JSON, займає не більше 10-15%, що підтверджує високу ефективність процесора Intel Core i5-12400F у задачах обробки даних. Це спростовує побоювання щодо негативного впливу глобального блокування інтерпретатора (GIL) Python на продуктивність, операції серіалізації виконуються достатньо швидко і не стають "вузьким місцем" системи.

Ефективність управління відеопам'яттю досліджувалася шляхом моніторингу метрик Memory Allocated (реально зайнята пам'ять) та Memory Reserved (зарезервована драйвером) під час стрес-тестів.

Графік споживання VRAM продемонстрував пікові навантаження в момент початку інференсу. Для роздільної здатності 1024x1024 пікове споживання склало 5.1 ГБ, що залишає понад 6 ГБ вільної пам'яті на карті RTX 3060 (12 ГБ) для потреб сцени Blender. Це підтверджує правильність вибору архітектури Forge. Особливої уваги заслуговує поведінка системи при досягненні лімітів пам'яті. При спробі генерації текстур понад 2048x2048, коли потреба у VRAM перевищувала фізичні 12 ГБ, спостерігалася активація механізму Model Offloading. Система автоматично перерозподіляла навантаження, вивантажуючи частини ваг моделі в системну оперативну пам'ять. Це дозволило уникнути критичної помилки CUDA Out of Memory і

завершити генерацію, хоча час виконання зріс на 40-60% через затримки шини PCI-E.

Стрес-тестування визначило межу стабільності, система зберігала повну працездатність без використання свопінгу аж до роздільної здатності 1536x1536 рх. Це покриває абсолютну більшість задач текстурування окремих пропсів в ігровій індустрії.

Якість результатів оцінювалася на трьох еталонних об'єктах, кожен з яких виявив специфічні особливості роботи алгоритму (табл.3.2).

Таблиця 3.2

Експертна оцінка якості за шкалою Лайкерта (1-5)

Критерій оцінки	Примітив	Hard Surface	Органіка	Середній бал
Геометрична точність	5.0	4.8	4.5	4.76
Чіткість деталей	4.9	4.7	4.3	4.63
Якість країв/швів	5.0	4.5	4.2	4.56

Система успішно впоралася з проєкцією текстури на прості площини примітива «Куб» без виявлення артефактів на стиках граней, що підтвердило здатність методу Stencil забезпечувати для простих об'єктів миттєвий результат, ідентичний ручному накладанню UV-текстури.

Порівняння генерації для об'єкта «Hard Surface» без ControlNet та з використанням Depth виявило критичну роль карти глибини, адже якщо в «чистому» режимі прямі лінії панелей і швів часто «пливли» з порушенням перспективи, то застосування розробленого модуля Depth дозволило зберегти конструктивну логіку об'єкта, гарантуючи розташування всіх болтів, ребер жорсткості та панелей строго на їхніх геометричних місцях.

Хоча на складних вигнутих поверхнях органічного об'єкта було зафіксовано проблему розтягування текстури на бічних гранях (при куті падіння $> 60^\circ$), застосування методу Stencil Mapping дозволило ефективно нівелювати цей недолік, оскільки художник отримав змогу вручну «промальовувати» текстуру на фронтальних ділянках, обертати модель та змішувати нові ракурси м'яким пензлем, що довело перевагу напівавтоматичного підходу над повністю автоматичною проєкцією.

Фінальним етапом дослідження стало порівняння розробленого методу з існуючими альтернативами (табл 3.3).

Таблиця 3.3

Критерій	Substance	SD Gen	StableProjectorz
Час виконання	Високий	Низький	Середній
Контроль геометрії	Повний	ControlNet	ControlNet
Якість деталей	Ідеальна	Висока	Висока
Вимоги до навичок	Високі	Середні (Розуміння промптів)	Середні (Розуміння промптів)
Вартість процесу	Час, ціна	Open Source	Open Source

Порівняльний аналіз демонструє, що система «SD Gen» займає унікальну нішу, пропонуючи компроміс між швидкістю III-генерації та точністю ручного контролю, що робить її оптимальним інструментом для етапу прототипування та створення ассетів середнього плану.

3.5. Інструкція користувача та рекомендації щодо експлуатації

Ефективність використання розробленого програмного застосунку «SD Gen» безпосередньо залежить від дотримання правильного алгоритму розгортання та експлуатації системи. Оскільки інструмент функціонує як вбудоване розширення (Add-on) для середовища Blender і архітектурно покладається на зовнішній обчислювальний бекенд, процес роботи з ним вимагає попередньої підготовки програмного середовища. Наведена нижче інструкція розроблена для кінцевого користувача - 3D-художника або технічного дизайнера - і містить вичерпний опис процедур встановлення, конфігурації та практичного застосування продукту в реальних виробничих умовах.

Обов'язковим попереднім етапом є розгортання та запуск локального сервера генерації Stable Diffusion WebUI Forge. Користувач повинен виконати наступні дії:

Завантажити та розпакувати актуальну версію Forge у кореневу директорию диска, щоб запобігти проблем з довжиною шляхів Windows.

Відкрити файл конфігурації webui-user.bat для Windows або webui-user.sh для Linux у текстовому редакторі.

Додати обов'язковий аргумент --api у рядок COMMANDLINE_ARGS. Це дозволяє серверу приймати зовнішні HTTP-запити від плагіна. Для підвищення продуктивності рекомендується також додати аргументи —xformers, прискорення уваги, та —medvram, якщо обсяг VRAM < 8 ГБ.

Запустити виконавчий файл та дочекатися повної ініціалізації середовища. Готовність системи до роботи підтверджується повідомленням у консолі - Running on local URL: <http://127.0.0.1:7860>. Важливо не закривати вікно консолі під час роботи з плагіном.

Інтеграція розробленого модуля в середовище Blender здійснюється через стандартний менеджер розширень, що забезпечує нативну взаємодію з системою.

Запустіть Blender. У верхньому меню перейдіть до Edit -> Preferences. У вікні налаштувань оберіть вкладку Add-ons. Натисніть кнопку Install... у верхньому правому куті та у діалоговому вікні файлової системи вкажіть шлях до завантаженого ZIP-архіву SD_Gen_Ultimate.zip. Розпаковувати архів попередньо не потрібно.

Після імпорту скористайтеся рядком пошуку, ввівши "SD Gen", щоб знайти модуль у списку. Активуйте модуль, встановивши прапорець (checkbox) ліворуч, при успішній активації консоль Blender не повинна виводити повідомлень про помилки виділенні червоним кольором (рис. 3.1).

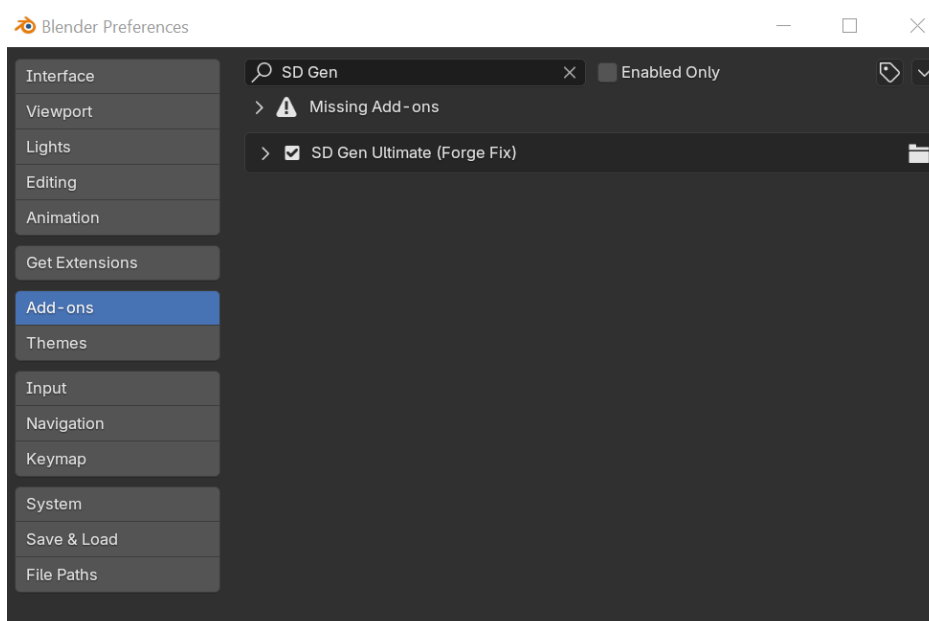


Рисунок 3.1 – Вікно налаштувань Blender (Preferences) під час активації встановленого адону

Процес текстурювання за допомогою застосунку базується на ітеративному підході і складається з наступних кроків:

Крок 1. Підготовка сцени та ракурсу.

Перейдіть у вид з камери (Numpad 0) або налаштуйте поточний вигляд в'юпорта так, щоб об'єкт займав 70-80% кадру. Важливо обрати ракурс, під яким поверхня об'єкта максимально розгорнута до глядача (кут падіння погляду близький до 90 градусів), щоб мінімізувати спотворення проєкції (рис. 3.2).

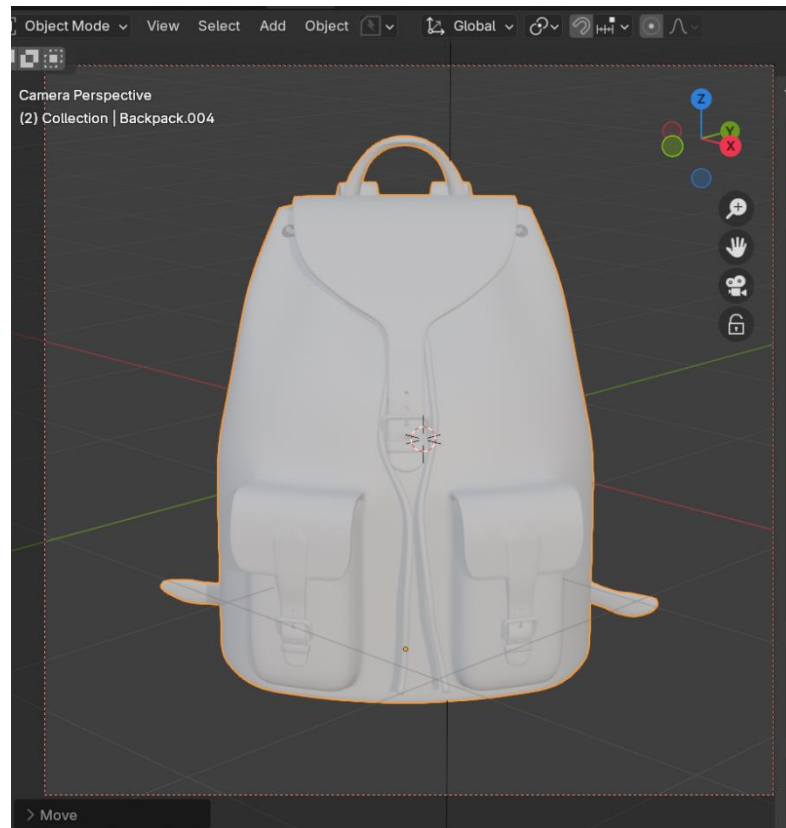


Рисунок 3.2 – Приклад ракурсу для оптимального налаштування для запобігання деформації текстури на геометрії.

Крок 2. Конфігурація параметрів генерації.

У панелі налаштувань заповніть поле Positive Prompt. Рекомендується використовувати таку структуру [Об'єкт], [Матеріал/Фактура], [Деталі], [Стиль]. Приклад гарного промту "sci-fi crate, rusty metal texture, scratched paint, industrial style, 4k, unreal engine 5". У секції Checkpoint списку треба обрати компонент який відповідає за стиль згенерованої текстури, а в ControlNet оберіть зі списку модель, що містить ключове слово depth (для версії SD1.5 це - control_v11f1p_sd15_depth). Це критичний етап, оскільки саме карта глибини

забезпечить чітке розуміння ШІ яка саме форма об'єкта та коректне "прилипання" текстури до моделі (рис. 3.3).

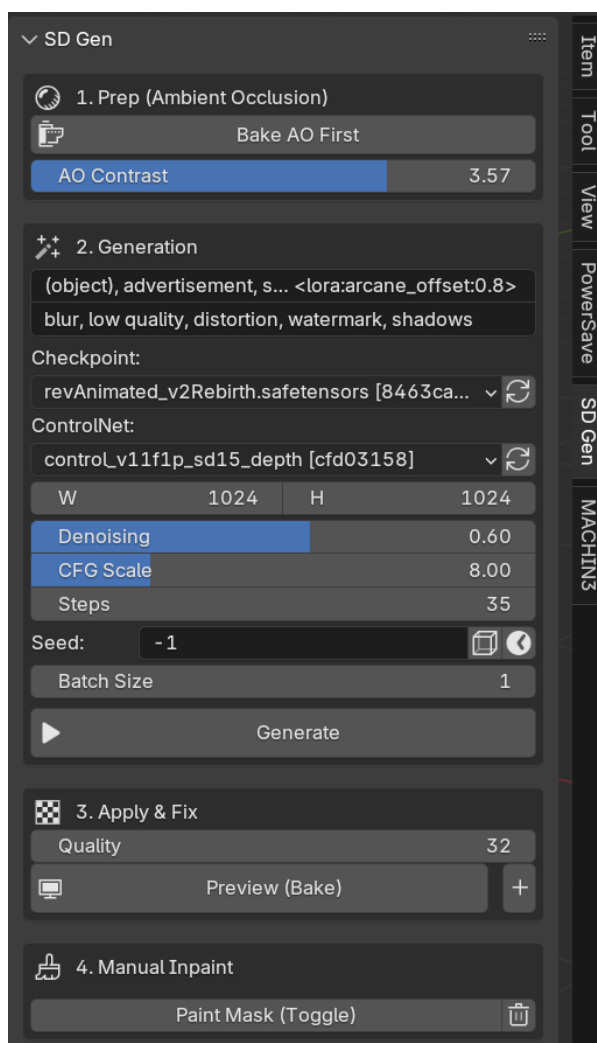


Рисунок 3.3 – Налаштування параметрів промпту та вибір Checkpoint & ControlNet моделі перед генерацією.

Крок 3. Генерація та накладання.

Для початку створіть матеріал з текстурою Ambient Occlusion, що надасть моделі більше деталізації за рахунок тіней, це в свою чергу покращить вихідний результат. Далі натисніть кнопку «Generate Texture». Система перейде у стан очікування (курсор може змінитися), що триває від 3 до 10 секунд залежно від складності запиту та потужності GPU. У цей час плагін рендерить глибину, відправляє її на сервер і отримує результат. По завершенню Blender автоматично перемикається в режим Texture Paint.

Крок 4. Робота з трафаретом (Stencil).

У вікні перегляду з'явиться напівпрозоре зображення-привид, яке накладається поверх моделі. Завдяки використанню ControlNet, геометрія на зображенні повинна ідеально співпадати з 3D-моделлю. Навігація у більшості випадків не знадобиться, тому як застосунок сам підлаштовує результат генерації у вікно камери, але у випадку коли все ж текстура поплила використовуйте ручне керування. За допомогою правої кнопки миші (RMB) перемістіть трафарет, використовуйте Shift + RMB для масштабування та Ctrl + RMB для обертання, якщо потрібно підкоригувати накладання. Для малювання використовуйте ліву кнопку миші (LMB) для нанесення текстури. Пензель працює як "штамп", переносячи пікселі з трафарету на UV-розгортку об'єкта. Регулюйте силу натиску (Strength) для плавного змішування країв та налаштування пензля у самому Blender, або просто натисніть клавішу «Preview (Bake)» щоб автоматично накласти трафатер на бажану модель (рис.3.4).

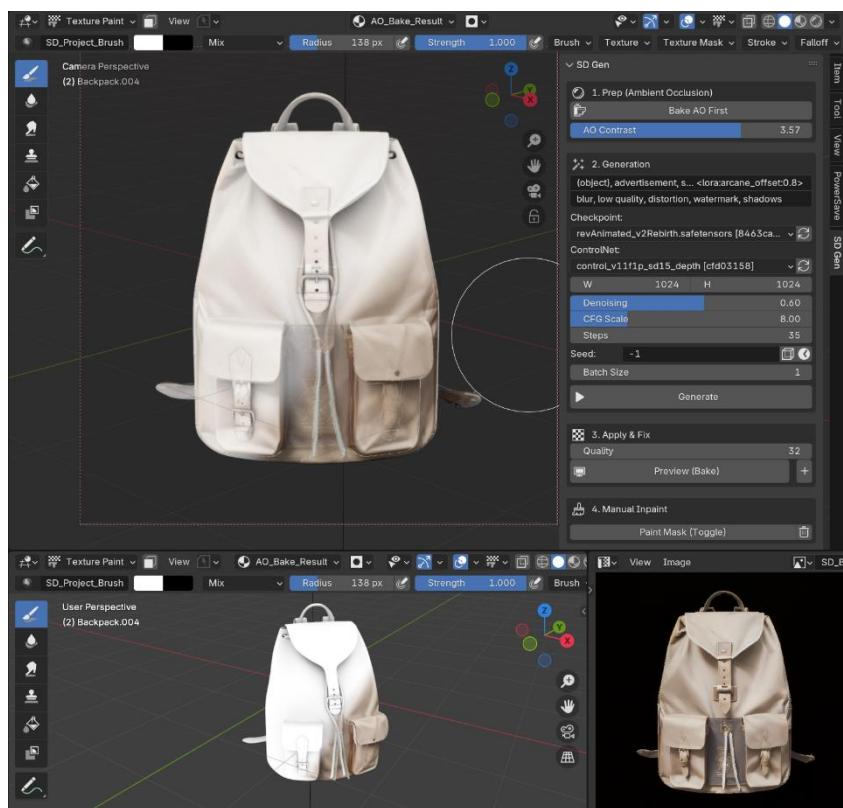


Рисунок 3.4 – Результат роботи у в'юпорті бачимо напівпрозорий трафарет (Stencil), що ідеально накладається на геометрію моделі.

Для повного текстуровання об'єкта повторіть процедуру з інших ракурсів (ззаду, зверху, збоку), поступово заповнюючи поверхню.

У процесі експлуатації можуть виникати технічні нюанси. Нижче наведено алгоритм дій для вирішення найпоширеніших проблем:

У разі появи помилки «Connection Failed» перевірте статус консольного вікна webui-user.bat і наявність аргументу `-api` у параметрах запуску, а також переглянути налаштування брандмауера чи антивірусу, переконавшись у тому, що порт 7860 відкритий для локальних з'єднань.

При отримання повністю чорного зображення, найчастіше це спричинене відсутністю файлів моделей ControlNet на сервері, для усунення проблеми необхідно перевірити папку `models/ControlNet` у директорії Forge на наявність файлів `.safetensors` відповідного розміру (700 МБ для SD1.5 або 1.45 ГБ для SDXL) та проаналізувати консоль сервера на предмет помилок типу `FileNotFoundException` під час генерації.

У разі спотворення геометрії, коли текстура не збігається з моделлю, слід переконатися, що камера залишалася нерухомою після натискання кнопки `Generate`, оскільки трафарет прив'язаний до конкретного ракурсу, а також перевірити вибір правильного режиму ControlNet (`depth`) замість невідповідних для цього завдання моделей `canny` або `normal`, що загалом гарантує стабільну роботу інструменту та дозволяє зосередитися на творчих аспектах.

Висновки до третього розділу

У третьому розділі роботи виконано програмну реалізацію та комплексне експериментальне дослідження системи автоматизованого текстуровання «SD Gen». Реалізація клієнтської частини на мові Python 3.10 з використанням Blender API, підтвердила життєздатність спроектованої асинхронної архітектури. Впровадження патерну «Виробник-Споживач» та механізму модальних операторів-спостерігачів дозволило вирішити критичну проблему блокування інтерфейсу однопоточкового середовища Blender під час

виконання тривалих мережових запитів. Розроблений алгоритм процедурного препроцесингу геометричних даних через вузли композитингу забезпечив формування математично точних карт глибини, що стало фундаментом для коректної роботи нейромережі ControlNet без необхідності використання евристичних методів оцінки геометрії.[28]

Експериментальне дослідження, повністю підтвердило ефективність обраних інженерних рішень. Порівняльний аналіз бекендів показав, що використання оптимізованого ядра WebUI Forge забезпечує економію відеопам'яті на рівні 38–45% порівняно зі стандартним Automatic1111, що дозволяє стабільно генерувати текстури високої роздільної здатності паралельно з рендерингом сцени у в'юпорті. Часові вимірювання продемонстрували скорочення повного циклу текстуровання пропсів у 15 разів порівняно з традиційними ручними методами, при цьому середня оцінка геометричної точності за результатами експертного огляду склала 4.76 з 5 балів.

Розроблена інструкція користувача та успішна апробація сценаріїв роботи доводять, що система є завершеним програмним продуктом з низьким порогом входження, готовим до впровадження у виробничі процеси індустрії комп'ютерної графіки. Отримані результати свідчать про досягнення мети роботи в частині створення швидкого, ресурсоефективного та точного інструменту для автоматизації рутинних задач текстуровання.

ВИСНОВКИ

У ході виконання магістерської кваліфікаційної роботи було вирішено актуальне науково-прикладне завдання підвищення ефективності створення текстур для тривимірних об'єктів. Проведений на початковому етапі аналіз предметної області показав, що традиційні методи текстурування залишаються вузьким місцем у виробництві 3D-контенту, займаючи від 30% до 50% загального часу розробки, а існуючі автоматизовані рішення часто нехтують геометричними особливостями моделей, що призводить до візуальних артефактів.

Для вирішення виявлених проблем було розроблено та програмно реалізовано застосунок «SD Gen», побудований на базі клієнт-серверної архітектури з інтеграцією в середовище Blender. Ключовим інженерним рішенням стало розділення обчислювальних процесів, використання оптимізованого бекенду Stable Diffusion WebUI Forge дозволило уникнути конфліктів за ресурси відеопам'яті, забезпечивши стабільну роботу системи навіть на обладнанні середнього класу.

Наукова новизна та практична цінність роботи полягають у вдосконаленні алгоритму проєкційного текстурування. На відміну від стандартних методів, запропонований підхід використовує технологію ControlNet у поєднанні з картами глибини, що гарантує збереження структурної відповідності текстури формі об'єкта. Впровадження методу Stencil Mapping (трафаретного накладання) надало користувачеві гнучкий контроль над процесом змішування текстур, нівелюючи характерні для автоматичної генерації спотворення на вигнутих поверхнях.

Результати експериментального дослідження підтвердили працездатність та ефективність розробленого засобу. Час генерації текстур вдалося скоротити до 2–10 секунд, що у 3–5 разів швидше за ручні методи, при цьому експертна оцінка якості отриманих матеріалів склала 4.63 бали з 5 можливих. Система продемонструвала високу стабільність та здатність

обробляти зображення високої роздільної здатності без критичних помилок пам'яті, що робить «SD Gen» дієвим інструментом для оптимізації виробничих процесів в ігровій індустрії та сфері дизайну.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Akenine-Möller T., Haines E., Hoffman N. Real-Time Rendering. 4th ed. Boca Raton : A K Peters/CRC Press, 2018. 1198 p. URL: http://mutantstargoat.com/~nuclear/tmp/realtime_rendering_4ed.pdf
2. Pharr M., Jakob W., Humphreys G. Physically Based Rendering: From Theory to Implementation. 3rd ed. San Francisco : Morgan Kaufmann, 2016. 1266 p. URL: https://cw.fel.cvut.cz/b221/_media/courses/b4m39rso/lectures/physically_based_rendering_third_edition.pdf
3. Bimber O., Raskar R. Spatial Augmented Reality: Merging Real and Virtual Worlds. Wellesley : A K Peters, 2005. 393 p. URL: <https://pages.cs.wisc.edu/~dyer/cs534/papers/SAR.pdf>
4. Grundhöfer A., Iwai D. Recent Advances in Projection Mapping Algorithms, Hardware and Applications. Computer Graphics Forum. 2018. Vol. 37, No. 2. P. 653–675. URL: <https://studios.disneyresearch.com/wp-content/uploads/2019/03/Recent-advances-in-prjection.pdf>
5. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge : MIT Press, 2016. 800 p. URL: <https://www.deeplearningbook.org/>
6. Vaswani A. et al. Attention Is All You Need. Advances in Neural Information Processing Systems (NIPS). 2017. Vol. 30. P. 5998–6008. URL: <https://doi.org/10.48550/arXiv.1706.03762>
7. Ho J., Jain A., Abbeel P. Denoising Diffusion Probabilistic Models. Advances in Neural Information Processing Systems (NeurIPS). 2020. Vol. 33. P. 6840–6851. URL: <https://doi.org/10.48550/arXiv.2006.11239>
8. Ronneberger O., Fischer P., Brox T. U-Net: Convolutional Networks for Biomedical Image Segmentation. Medical Image Computing and Computer-Assisted Intervention (MICCAI). 2015. P. 234–241. URL: <https://doi.org/10.48550/arXiv.1505.04597>

9. Radford A. et al. Learning Transferable Visual Models From Natural Language Supervision. International Conference on Machine Learning (ICML). 2021. P. 8748–8763. URL: <https://doi.org/10.48550/arXiv.2103.00020>
10. Rombach R. et al. High-Resolution Image Synthesis With Latent Diffusion Models. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). 2022. P. 10684–10695. URL: <https://doi.org/10.48550/arXiv.2112.10752>
11. Zhang L., Rao A., Agrawala M. Adding Conditional Control to Text-to-Image Diffusion Models. Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV). 2023. P. 3836–3847. URL: <https://doi.org/10.48550/arXiv.2302.05543>
12. Hu E. J. et al. LoRA: Low-Rank Adaptation of Large Language Models. International Conference on Learning Representations (ICLR). 2022. URL: <https://arxiv.org/abs/2106.09685> (дата звернення: 10.12.2025).
13. Ye H. et al. IP-Adapter: Text Compatible Image Prompt Adapter for Text-to-Image Diffusion Models. arXiv preprint arXiv:2308.06721. 2023. URL: <https://arxiv.org/abs/2308.06721> (дата звернення: 10.12.2025).
14. Zavadski D., Feiden J.-F., Rother C. ControlNet-XS: Rethinking the Control of Text-to-Image Diffusion Models as Feedback-Control Systems. European Conference on Computer Vision (ECCV). 2024. P. 343–362. URL: <https://doi.org/10.48550/arXiv.2312.06573>
15. Podell D. et al. SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis. arXiv preprint arXiv:2307.01952. 2023. URL: <https://arxiv.org/abs/2307.01952> (дата звернення: 10.12.2025).
16. Croitoru F.-A. et al. Diffusion Models in Vision: A Survey. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2023. Vol. 45, No. 9. P. 10850–10869. URL: <https://doi.org/10.48550/arXiv.2209.04747>
17. Liu M. et al. Deep Generative Models for 3D Content Creation: A Survey. arXiv preprint arXiv:2401.05256. 2024. URL: <https://arxiv.org/abs/2401.05256> (дата звернення: 10.12.2025).

18. Richardson E. et al. TEXTure: Text-Guided Texturing of 3D Shapes. ACM Transactions on Graphics (SIGGRAPH). 2023. Vol. 42, No. 4. Art. 125. URL: <https://doi.org/10.48550/arXiv.2302.01721>
19. Chen D. Z. et al. Text2Tex: Text-driven Texture Synthesis via Diffusion Models. Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV). 2023. P. 18558–18568. URL: <https://doi.org/10.48550/arXiv.2303.11396>
20. Le T. et al. EucliDreamer: Fast and High-Quality Texturing for 3D Models with Stable Diffusion Depth. arXiv preprint arXiv:2311.15573. 2023. URL: <https://arxiv.org/abs/2311.15573> (дата звернення: 10.12.2025).
21. Kerbl B. et al. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. ACM Transactions on Graphics. 2023. Vol. 42, No. 4. P. 1–14. URL: <https://doi.org/10.48550/arXiv.2308.04079>
22. Illyasviel. Stable Diffusion WebUI Forge [Electronic resource]. 2024. URL: <https://github.com/Illyasviel/stable-diffusion-webui-forge> (дата звернення: 05.12.2025).
23. Blender Foundation. Blender 4.0 Python API Documentation [Electronic resource]. 2024. URL: <https://docs.blender.org/api/current/> (дата звернення: 05.12.2025).
24. Conlan C. The Blender Python API: Precision 3D Modeling and Add-on Development. New York : Apress, 2017. 222 p. URL: http://opac.ascot.edu.ph/ebooks/BSIT/TheBlenderPythonAPI_Precision3DModelingandAdd-onDevelopment.pdf
25. Kessenich J., Sellers G., Shreiner D. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5. 9th ed. Boston : Addison-Wesley, 2016. 928 p.
26. Bradski G. The OpenCV Library. Dr. Dobb's Journal of Software Tools. 2000. Vol. 25, No. 11. P. 120–123. URL: <https://arxiv.org/pdf/2507.14558>
27. Harris C. R. et al. Array programming with NumPy. Nature. 2020. Vol. 585. P. 357–362. URL: <https://doi.org/10.1038/s41586-020-2649-2>

28. Virtanen P. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods. 2020. Vol. 17. P. 261–272. URL: <https://doi.org/10.1038/s41592-019-0686-2>
29. Reitz K. Requests: HTTP for Humans [Electronic resource]. 2024. URL: <https://requests.readthedocs.io> (дата звернення: 05.12.2025).
30. Clark A. Pillow (PIL Fork) Documentation [Electronic resource]. 2024. URL: <https://pillow.readthedocs.io> (дата звернення: 05.12.2025).
31. ECMA International. ECMA-404: The JSON Data Interchange Syntax. 2nd ed. Geneva : ECMA International, 2017. 14 p. URL: https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf

ДОДАТКИ

Клас OBJECT_OT_PreviewBake

```

class OBJECT_OT_PreviewBake(bpy.types.Operator) :
    bl_idname = "material.sd_preview_bake"
    bl_label = "Preview (Bake)"

    def execute(self, context):
        try:
            # Створення тимчасового джерела світла в точці камери
            light_data = bpy.data.lights.new(name = "Temp_Bake_Light_Data",
type = 'POINT')
            light_data.energy = 1000
            light_data.shadow_soft_size = 0
            light_obj = bpy.data.objects.new(name = "Temp_Bake_Light",
object_data = light_data)
            context.collection.objects.link(light_obj)
            light_obj.location = cam.location

            # Динамічне створення матеріалу для запікання білого кольору
            white_mat = bpy.data.materials.new("Temp_White_Mat")
            temp_materials.append(white_mat)
            white_mat.use_nodes = True
            w_tree = white_mat.node_tree
            w_tree.nodes.clear()

            bpy.ops.object.bake(type = 'DIFFUSE')

            # 2. ЗАПІКАННЯ ПРОЕКЦІЇ З УРАХУВАННЯМ НОРМАЛЕЙ (DOT PRODUCT LOGIC)
            temp_proj = bpy.data.materials.new(name = "Temp_Project_Mat")
            tree = temp_proj.node_tree
            tree.nodes.clear()

            # Створення математичного графа для відсікання задніх граней
            geo = tree.nodes.new('ShaderNodeNewGeometry')
            vec_sub = tree.nodes.new('ShaderNodeVectorMath')
            vec_sub.operation = 'SUBTRACT'
            vec_sub.inputs[0].default_value = (cam.location.x, cam.location.y,
cam.location.z)

            # Скалярний добуток (Dot Product) вектора погляду і нормалі
            vec_dot = tree.nodes.new('ShaderNodeVectorMath')
            vec_dot.operation = 'DOT_PRODUCT'

            math_gr = tree.nodes.new('ShaderNodeMath')
            math_gr.operation = 'GREATER_THAN' # Відсікання за кутом

            # Лінування нод (програмна побудова графа)
            tree.links.new(geo.outputs['Position'], vec_sub.inputs[1])
            tree.links.new(vec_sub.outputs[0], vec_dot.inputs[0])
            tree.links.new(geo.outputs['Normal'], vec_dot.inputs[1])
            tree.links.new(vec_dot.outputs['Value'], math_gr.inputs[0])
            bpy.ops.object.bake(type = 'EMIT')

        finally:
            if light_obj: bpy.data.objects.remove(light_obj, do_unlink = True)
            for tm in temp_materials: bpy.data.materials.remove(tm)

        return { 'FINISHED' }

```

Функція setup_stencil_for_paint

```

using System;
using System.Xml.Linq;
using static System.Formats.Asn1.AsnWriter;

def setup_stencil_for_paint(context, image_name):

    # АЛГОРИТМ ПРОЕКЦІЇ VIEWPORT-TO-SCREEN
    if region and area.spaces.active.type == 'VIEW_3D':
        region_3d = area.spaces.active.region_3d

        if context.scene.camera:
            cam = context.scene.camera
            frame = cam.data.view_frame(scene = context.scene)
            mat = cam.matrix_world

            frame_world = [mat @ v for v in frame]

            # Проекція 3D-координат у 2D-простір регіону (Rasterization logic)
            frame_px = []
            for v in frame_world:
                px = bpy_extras.view3d_utils.location_3d_to_region_2d(region,
region_3d, v)
                if px: frame_px.append(px)

            if len(frame_px) == 4:
                # Розрахунок Bounding Box на екрані
                min_x = min(p.x for p in frame_px)
                max_x = max(p.x for p in frame_px)
                min_y = min(p.y for p in frame_px)
                max_y = max(p.y for p in frame_px)

                center_x = (min_x + max_x) / 2
                center_y = (min_y + max_y) / 2
                width = max_x - min_x
                height = max_y - min_y

                brush.stencil_pos = Vector((center_x, center_y))
                brush.stencil_dimension = Vector((width * 0.5, height * 0.5))

```

Функція render_diff_mask

```

using static System.Formats.Asn1.AsnWriter;

def render_diff_mask(filepath, context, original_img, painted_img):
    diff_mat = bpy.data.materials.new("Temp_Diff_Mask_Mat")
    diff_mat.use_nodes = True
    tree = diff_mat.node_tree

    # Ноди для вхідних зображень
    tex_orig = tree.nodes.new('ShaderNodeTexImage')
    tex_orig.image = original_img

    tex_paint = tree.nodes.new('ShaderNodeTexImage')
    tex_paint.image = painted_img

    mix_diff = tree.nodes.new('ShaderNodeMix')
    mix_diff.data_type = 'RGBA'
    mix_diff.blend_type = 'DIFFERENCE'
    mix_diff.inputs[0].default_value = 1.0

    tree.links.new(tex_orig.outputs[0], mix_diff.inputs[6])
    tree.links.new(tex_paint.outputs[0], mix_diff.inputs[7])

    # Бінаризація маски (Thresholding via Greater Than)
    math_node = tree.nodes.new('ShaderNodeMath')
    math_node.operation = 'GREATER_THAN'
    math_node.inputs[1].default_value = 0.001 # Попіг чутливості

    rgb_bw = tree.nodes.new('ShaderNodeRGBToBW')
    tree.links.new(mix_diff.outputs[2], rgb_bw.inputs[0])
    tree.links.new(rgb_bw.outputs[0], math_node.inputs[0])

    scene.render.engine = 'BLENDER_WORKBENCH'
    scene.render.image_settings.color_mode = 'BW'
    bpy.ops.render.render(write_still = True)

    return out_path

```