Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська дипломна робота

на тему: «Розробка методів і програмних засобів для текстурування»

Виконав: студент __4__ курсу
групи __4ПІ-20б__
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Іваха Олександр Андрійович
(прізвище та ініціали)

Керівник: д.т.н., проф. каф. ПЗ Романюк О.Н.
(прізвище та ініціали)

Рецензент: д.т.н., проф. каф. ОТ Мартинюк Т.О.
(прізвище та ініціали)

Допущено до захисту

Зав. кафедри ПЗ
д.т.н., проф. Романюк О.Н.
«10» червня 2024 р.

ВНТУ – 2024

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
О. Н. Романюк
«12» березня 2024 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Івасі Олександру Андрійовичу

1. Тема роботи – розробка методів і програмних засобів для текстурування. Керівник роботи: д.т.н., Романюк О.Н. професор кафедри ПЗ, затверджені наказом вищого навчального закладу від від 11 березня 2024 р. № 80.

2. Строк подання студентом роботи 3 червня 2024.

3. Вихідні дані до роботи: базовий метод Хекберта; метод генерації шуму – шум Перліна; графічний режим – TrueColor; дані для текстурування: координати вершин трикутника; вектори до джерела світла та спостерігача, вектор нормалі; середовище розробки – Visual Studio; мова програмування – C++.

4. Зміст текстової частини: вступ, аналіз методів накладання текстур, розробка нових методів текстурування, розробка процедурних текстур з використанням шуму Перліна, практична реалізація методів текстурування у системах комп'ютерної графіки; висновки; список використаних джерел; додатки.

5. Перелік ілюстративного матеріалу: титульний слайд; актуальність теми; мета, об'єкт, предмет, задачі дослідження; новизна і практична цінність отриманих результатів; особливості текстурування в системах комп'ютерної графіки; техніки текстурування; використання методу Хекберта для виконання перспективного перетворення; розробка методу накладання текстур з урахуванням координати глибини; алгоритм формування зображення поверхні води з використанням шуму Перліна; реалізація методів текстурування у системах комп'ютерної графіки; апробація матеріалів бакалаврської дипломної роботи; фінальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1- 4	Романюк О.Н., д.т.н., професор каф. ПЗ	13.03.24	03.06.24

7. Дата видачі завдання 13 березня 2024р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз методів накладання текстур	14.03.24 – 25.03.24	Виконано
2	Розробка нових методів текстурувань	25.03.24 – 17.04.24	Виконано
3	Розробка процедурних текстур з використанням шуму Перліна	17.04.24– 28.04.24	Виконано
4	Розробка програмного забезпечення	28.04.24 – 23.05.24	Виконано
5	Оформлення матеріалів до захисту БДР	23.05.24 – 30.05.24	Виконано

Студент О.Іваха
(підпис) **О. А. Іваха** (ініціали та прізвище)

Керівник бакалаврської дипломної роботи
(підпис) **О. Н. Романюк** (ініціали та прізвище)

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 61 сторінки формату А4, на яких є 25 рисунків, 2 таблиці, список використаних джерел містить 22 найменування.

У бакалаврській дипломній роботі були розроблені методи та програмні засоби для текстурування з використанням новітніх технік та процедур. Проведено ґрунтовний аналіз наявних підходів до текстурування, зокрема процедурного, фотографічного та проекційного. Обґрунтовано необхідність вдосконалення методів текстурування задля підвищення реалістичності візуалізацій у тривимірних моделях.

Розроблено модифіковану версію методу Хекберта, яка зменшує кількість обчислювальних операцій під час розрахунку текстурних координат, що дозволило збільшити ефективність процесу текстурування. Запропоновані нові ітераційні рівняння для точного визначення параметрів еліпса у проекціях пікселів на текстурну поверхню забезпечили зростання продуктивності.

Також було створено алгоритм текстурування з урахуванням координати глибини текселів, що значно покращує реалістичність графічних зображень. Особливу увагу приділено розробці алгоритму формування диму з використанням шумового алгоритму Перліна. Цей алгоритм дозволяє створювати реалістичні ефекти диму, використовуючи генерацію, фільтрацію та анімацію шуму Перліна.

Розроблені підходи можуть бути використані у різних галузях, таких як архітектурна візуалізація, анімація, відеоігри та віртуальна реальність, що відкриває нові можливості для подальших досліджень та вдосконалення в області комп'ютерної графіки.

Ключові слова: текстурування, комп'ютерна графіка, метод Хекберта, шум Перліна, тривимірне моделювання, алгоритм, реалістичні візуалізації, програмне забезпечення.

ABSTRACT

The bachelor’s thesis consists of 61 pages of A4 format, on which include 25 figures and 2 tables. The list of references contains 22 sources.

The bachelor's thesis developed methods and software tools for texturing using the latest techniques and procedures. A thorough analysis of existing approaches to texturing, in particular procedural, photographic and projection, was carried out. The necessity of improving texturing methods in order to increase the realism of visualizations in three-dimensional models is substantiated.

A modified version of Hackbert's method was developed, which reduces the number of computational operations during the calculation of texture coordinates, which made it possible to increase the efficiency of the texturing process. The proposed new iterative equations for the exact determination of the ellipse parameters in the projections of the pixels on the textured surface ensured an increase in productivity.

A texturing algorithm was also created, taking into account the texel depth coordinate, which significantly improves the realism of graphic images. Special attention is paid to the development of the smoke formation algorithm using Perlin's noise algorithm. This algorithm allows you to create realistic smoke effects using Perlin noise generation, filtering and animation.

The developed approaches can be used in various fields, such as architectural visualization, animation, video games and virtual reality, which opens up new opportunities for further research and improvement in the field of computer graphics.

Keywords: texturing, computer graphics, Hackbert's method, Perlin noise, three-dimensional modeling, algorithm, realistic visualizations, software.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Текстурування в комп'ютерній графіці – це метод накладення візуальних деталей на об'єкти у 3D моделях або 2D поверхнях. Це один із напрямів високопродуктивного формування реалістичного візуального зображення.

Висока продуктивність текстурування дозволяє графічним дизайнерам використовувати більш складні та деталізовані текстури, що збільшує реалістичність і візуальну привабливість сцен. Це може бути важливим для ігор та візуальних симуляцій, де якість зображення є ключовим фактором привабливості. Ефективніші методи текстурування дозволяють краще використовувати обмежені ресурси системи, наприклад, пам'ять та обчислювальні потужності GPU. Це знижує загальне навантаження на апаратне забезпечення, що є особливо важливим для мобільних пристроїв і VR, де апаратні ресурси більш обмежені. Швидше накладання текстур може знизити час завантаження гри або додатку, що покращує загальний користувацький досвід. Зокрема, це важливо для ігор та інтерактивних додатків, де швидкий відгук системи сприяє загальній привабливості продукту. Висока продуктивність накладання текстур дозволяє проектам легше масштабуватися для різних платформ і роздільних здатностей.

У відеоіграх та віртуальній реальності, реалістичні текстури сприяють створенню більш занурювального досвіду, дозволяючи гравцям відчувати себе ніби вони насправді знаходяться у грі чи віртуальному світі. Це посилює емоційний вплив і залученість користувачів. У таких областях, як архітектура та дизайн інтер'єру, реалістичні текстури дозволяють користувачам точніше уявити кінцевий продукт. Це може сприяти кращому прийняттю рішень і зменшити кількість ітерацій у дизайні, оскільки клієнти бачать реалістичне відображення матеріалів і поверхонь. Реалістичні текстури можуть допомогти краще зрозуміти складні процеси і структури, наприклад, у біології або медицині. Візуалізація

людського тіла з високим рівнем деталізації може бути використана для навчання студентів або допомоги хірургам у плануванні операцій.

З розвитком нових дисплеїв високої роздільної здатності та віртуальної реальності, реалістичність текстур стає критично важливою для забезпечення того, що контент виглядає добре на найновіших технологічних платформах. Висока деталізація текстур гарантує, що візуальні ефекти залишаються переконливими навіть на великих екранах або в інтенсивних VR-досвідах.

Важливий напрямок з підвищення реалістичності зображень пов'язаний з урахуванням перспективи зображень. Перспективно-коректне текстурування відтворює точніше, як текстури мають виглядати з різних кутів огляду, що забезпечує іммерсивність і відчуття присутності у віртуальних середовищах, наприклад, у VR (віртуальна реальність) або при створенні спецефектів у фільмах. Для галузей, де візуалізація використовується для точного представлення фізичних об'єктів або сцен, таких як архітектура, інженерія чи медицина, перспективно-коректне текстурування може значно покращити якість і достовірність зображень.

Оскільки ринок дисплеїв постійно розвивається, включаючи різноманітність роздільних здатностей і форматів, перспективно-коректне текстурування дозволяє забезпечити, що зображення виглядають коректно на будь-якому обладнанні.

З інструментами для перспективно-коректного текстурування, дизайнери можуть експериментувати з більш складними і динамічними ефектами, які виглядають природно під будь-яким кутом перегляду. Це знімає обмеження на творчість і дозволяє створювати більш залучаючі та інноваційні дизайни.

У багатьох застосуваннях, особливо в архітектурних та інтер'єрних візуалізаціях, перспективно-коректне текстурування допомагає забезпечити, що текстури не виглядають спотворено при погляді під різними кутами, що критично важливо для правильного сприйняття простору і масштабів.

З появою нових типів екранів і пристроїв відображення, таких як гнучкі дисплеї і прозорі OLED-екрани, важливо, щоб методи текстурування могли

адаптуватися до різних типів візуальних відображень. Перспективно-коректне текстурування допомагає це забезпечити, підтримуючи якість зображення на різних пристроях.

Подальша розробка і вдосконалення перспективно-коректного текстурування є не тільки необхідністю для підтримки та поліпшення існуючих технологій, але і важливим напрямком для інновацій в багатьох галузях, зокрема у візуальних медіа, ігровій індустрії, і багатьох інших.

Тому актуальними є питання підвищення продуктивності та реалістичності формування графічних сцен за рахунок розробки високопродуктивних методів і програмних засобів текстурування.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою роботи є підвищення продуктивності та реалістичності текстурування в системах комп'ютерної графіки.

Основними задачами дослідження є:

- аналіз існуючих методів і засобів текстурування;
- розробити нові:
- методи високопродуктивного накладання текстур;
- метод урахування перспективи об'єкта при накладанні текстур;
- метод формування зображення води та диму з використанням шуму Перліна;
- розробити програмні засоби для накладання текстур;
- провести експериментальні дослідження розроблених програмних засобів текстурування.

Об'єктом дослідження є процес текстурування тривимірних об'єктів у системах комп'ютерної графіки.

Предметом дослідження є методи та засоби накладання текстур.

Методи дослідження. У процесі досліджень використовувалися такі методи дослідження: теорія чисел та чисельних методів, теорія диференціально-інтегрального числення, лінійна алгебра, методи аналітичної геометрії для розробки моделей та методів текстурування тривимірних об’єктів; комп’ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Новизна отриманих результатів:

1. Модифіковано метод Хекберта, який відрізняється від відомого використанням нового ітераційного процесу для визначення координат пікселів, що дозволило зменшити кількість базових операцій і, як наслідок, підвищити продуктивність методу до 18%.

2. Модифіковано метод перспективно-коректного накладання текстур, який відрізняється від відомого використанням нових формули для спрощеного розрахунку параметрів рівняння еліпса, що дозволяє підвищити продуктивність фільтрації текстур до 15%.

3. Вперше запропоновано метод текстурування, особливість якого полягає у використанні координати глибини для уточненого визначення інтенсивностей кольору текселів, що дає можливість підвищити реалістичність формування графічних сцен.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській дипломній роботі теоретичних положень запропоновано алгоритми та розроблено програмні засоби накладання текстур.

Особистий внесок здобувача. Усі наукові результати, викладені у БДР , отримані автором особисто. У друкованих працях, опублікованих у співавторстві, автору належать такі результати: Аналіз графічних процесорів [1]; аналіз вершинних шейдерів[2]; аналіз найпотужніших відеокарт [3]; аналіз методів текстурування [4].

Апробація матеріалів бакалаврської дипломної роботи. Основні положення БДР доповідалися та обговорювалися на Міжнародних і Всеукраїнських конференціях:

- Міжнародна науково-практична конференція « Електронні інформаційні ресурси: створення, використання, доступ» (Вінниця, 2020);
- Всеукраїнська науково-технічної конференція молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» (Одеса, 2021;
- Науково-практична інтернет-конференція студентів аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (Вінниця 2020);
- LIII науково-технічна конференція підрозділів ВНТУ (Вінниця 2024).

Публікації. Основні результати досліджень опубліковано в 4 наукових працях, у матеріалах конференцій.

1 АНАЛІЗ МЕТОДІВ НАКЛАДАННЯ ТЕКСТУР

1.1 Особливості текстурування в системах комп’ютерної графіки

Текстурування матеріалів [4-9, 11, 13] є технікою у сфері комп'ютерної графіки, що допомагає надати об'єктам реалістичні відтінки. Цей процес включає застосування текстур на поверхні об'єктів для створення вражень різних матеріалів, як деревина, камінь чи метал. Проаналізуємо базові принципи та методи текстурування матеріалів, а також розглянемо приклади їх використання.

Текстурування матеріалів – це процес накладання текстур, зображень або візерунків на поверхню об'єкта у комп'ютерній графіці (див. рис. 1.1). Це дозволяє створювати більш реалістичні та деталізовані візуалізації, що імітують різноманітні типи поверхонь та матеріалів.

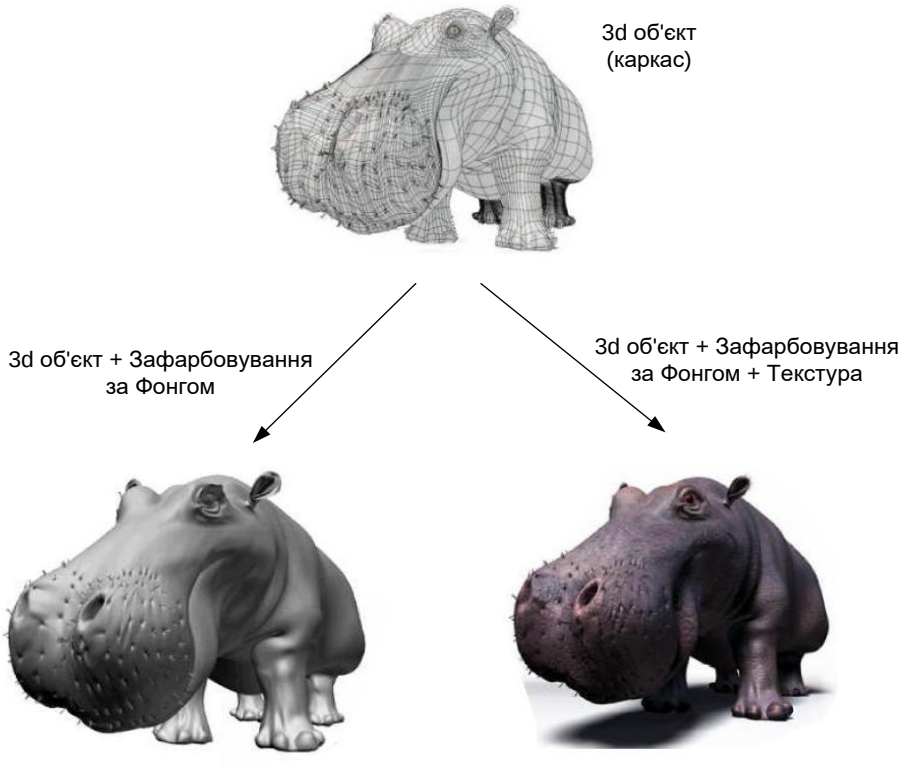


Рисунок 1.1 – Етапи накладання текстур

Першим кроком у текстуруванні матеріалів є вибір відповідних текстур. Це може бути фотографія справжнього матеріалу або текстура, створена вручну.

Важливо підібрати текстуру, яка відповідає характеристикам та зовнішньому вигляду матеріалу, який ви бажаєте відтворити.

Наступним кроком є адекватне розміщення текстур на поверхні об'єкта. Це може включати масштабування, обертання та зсув текстури, щоб вона відповідала формі та розмірам об'єкта. Також важливо враховувати напрямок освітлення та тіні, щоб текстура виглядала природно.

Освітлення та тіні відіграють ключову роль у створенні реалістичного вигляду текстурованого матеріалу. Під час текстурування необхідно враховувати напрямок та інтенсивність світла [12], щоб текстура відбивала світло та створювала ефект об'ємності. Також важливо брати до уваги тіні, які можуть впливати на візуальне сприйняття текстури.

Для створення складніших і цікавіших текстур можна використовувати кілька текстурних шарів. Це дозволяє додавати додаткові деталі, такі як подряпини, знос або візерунки, на основну текстуру. Використання різних текстурних шарів допомагає створювати більш реалістичні та унікальні матеріали.

Врахування масштабу та роздільної здатності текстур.

При текстуруванні необхідно враховувати масштаб та роздільну здатність текстур. Занадто низька роздільна здатність може призвести до пікселізації та втрати деталей, а занадто висока роздільна здатність може уповільнити процес рендерингу. Також важливо підібрати масштаб текстури, щоб вона відповідала розміру об'єкта та не виглядала спотвореною.

Дотримання цих принципів допоможе створювати якісні та реалістичні текстури для матеріалів у комп'ютерній графіці.

Процедурне текстурування [4] – це техніка, при якій текстури створюються за допомогою алгоритмів та математичних функцій. Замість використання реальних фотографій чи зображень, текстури генеруються програмно. Це дозволяє створювати унікальні та різноманітні текстури, які можна змінювати та адаптувати без втрати якості.

Фотографічне текстурування – це техніка, при якій текстури створюються на основі реальних фотографій чи зображень. Це може бути фотографія текстури дерева, каменю, трави тощо. Фотографічні текстури зазвичай мають високу роздільну здатність і деталізацію, що дозволяє створювати реалістичні матеріали.

Проекційне текстурування [19] – це техніка, при якій текстури наносяться на об'єкти з використанням проекцій. Наприклад, текстура може бути проектована на поверхню об'єкта за допомогою проектора або за допомогою математичних обчислень. Це дозволяє створювати ефекти, такі як відображення, тіні та спотворення на поверхні об'єкта.

UV-розгортка – це процес перетворення тривимірної поверхні об'єкта в двовимірний простір, щоб на нього можна було нанести текстуру. У процесі UV-розгортки створюється сітка, яка відповідає поверхні об'єкта, і кожна точка на цій сітці називається UV-координатою. Потім текстура наноситься на цю сітку, і об'єкт отримує текстурований вигляд.

Шарове текстурування – це техніка, при якій на об'єкт накладається кілька текстурних шарів. Кожен шар може мати свою власну текстуру та свої налаштування, такі як прозорість, відображення тощо. Це дозволяє створювати складні та реалістичні матеріали, такі як дерево з текстурою кори та листя.

Це лише деякі з технік текстурування матеріалів, які використовуються у комп'ютерній графіці.

Розглянемо приклади застосування текстурування матеріалів. У архітектурній візуалізації текстурування матеріалів відіграє важливу роль у створенні реалістичних сцен. Наприклад, при створенні візуалізації інтер'єру, текстури можуть бути використані для надання стінам та підлогам різних обробок, таких як цегла, дерево, камінь і т.д. Це дозволяє візуалізувати, як будуть виглядати матеріали в реальному житті та допомагає клієнтам приймати рішення щодо дизайну.

У комп'ютерних іграх текстурування матеріалів використовується для створення реалістичних та деталізованих персонажів. Наприклад, текстури

можуть бути використані для надання шкірі персонажа текстури, волоссю, одягу тощо. Це дозволяє створювати унікальні та різноманітні персонажі, які виглядають природно та привабливо для гравців.

У 3D-анімації текстурування матеріалів використовується для створення реалістичних та деталізованих об'єктів. Наприклад, текстури можуть бути використані для надання металевим об'єктам текстури металу, дерев'яним об'єктам – текстури дерева тощо. Це дозволяє створювати об'єкти, які виглядають природно та мають різні властивості матеріалів, такі як відблиск, прозорість тощо.

У кіно та на телебаченні текстурування матеріалів використовується для створення реалістичних спецефектів. Наприклад, текстури можуть бути використані для створення ілюзії вогню, диму, води та інших природних явищ. Це дозволяє створювати вражаючі візуальні ефекти, які роблять фільми та телепередачі більш захоплюючими та реалістичними.

У віртуальній реальності текстурування матеріалів використовується для створення реалістичних та деталізованих ландшафтів. Наприклад, текстури можуть бути використані для надання землі, траві, воді тощо текстур та властивостей, які відповідають реальним об'єктам. Це дозволяє створювати переконливі та захоплюючі віртуальні світи, які виглядають та відчуваються як справжні.

Це лише деякі приклади застосування текстурування матеріалів у комп'ютерній графіці. Насправді, текстурування може бути використане у багатьох інших галузях, таких як анімація фільмів, створення спецефектів, візуалізація продуктів і багато іншого. Важливо розуміти, що текстурування матеріалів є важливою частиною створення реалістичних та привабливих візуальних ефектів у комп'ютерній графіці.

Для кращого розуміння термінів з теми текстурування, найпопулярніші з них було описано у таблиці 1.1.

Таблиця 1.1 – Терміни текстурування

Термін	**Визначення**	**Характеристики**
Текстура	Зображення, яке прикладається до поверхні об'єкта для надання йому візуальних деталей та реалізму.	Впливає на зовнішній вигляд об'єкта. Може бути створено з фотографій, малюнків або сгенеровано за допомогою комп'ютерних алгоритмів. Може мати як повторюваний, так і унікальний характер для кожної поверхні
UV-координати	Двовимірні координати для розміщення текстури на об'єкті.	Визначають взаємозв'язок частин текстури та елементів об'єкта. Застосовуються для накладення текстур на поверхню. Можна коригувати для точного розташування текстури
Бамп-мапінг	Метод текстурування, який створює ефект висотних змін на поверхні об'єкта без зміни його геометрії.	Забезпечує додавання деталей та рельєфності на поверхню. Використовує відтінки сірого для моделювання висотних варіацій. Візуально покращує об'єкти, не ускладнюючи їхню полігональну структуру
Дифузне відбиття	Здатність матеріалу розсіювати світло у всі боки, створюючи м'яке, розсіяне світло.	Формує матову поверхню без блисків. Розподіляє світло однорідно у всі напрямки. Використовується для імітації м'якого освітлення та реалістичних матеріалів
Спекулярне відбиття	Здатність матеріалу відбивати світло в певних напрямках, що створює блиск та відображення.	Формує блиск та відображення на поверхні об'єкта. Визначає напрям і силу відбитого світла. Використовується для створення поверхонь з металічним, скляним та іншими відбиваючими ефектами
Нормал-мапінг	Метод текстурування, який використовує нормалі для створення більш детальних і складних поверхонь.	Замість того, щоб змінювати геометрію, нормал-мапи модифікують нормалі поверхні, що дозволяє досягти ефектів висотних змін і деталей. Використовує кольорові зображення для кодування нормалей поверхні. Дає більш реалістичні результати в порівнянні з бамп-мапінгом.

1.2 Техніки текстурування

Текстурування у комп'ютерній графіці є технологією, що забезпечує реалістичність та деталізацію візуальних образів (див. рис. 1.2). Цей процес включає накладання зображень, відомих як текстури, на тривимірні моделі, щоб надати їм колір, візерунки або матеріальні характеристики. Використання текстур дозволяє розробникам створювати складні сцени і об'єкти з високим рівнем деталізації, не збільшуючи при цьому значно об'єм обчислень, необхідний для відображення цих сцен в реальному часі.

Рисунок 1.2 – Текстурування

Текстурування має декілька аспектів, зокрема вибір та створення текстур, визначення способу їх накладання на моделі, а також оптимізація цих текстур для забезпечення максимальної продуктивності і якості. Техніки текстурування еволюціонували від простих методів до складних алгоритмів процедурного текстурування та бамп-мепінгу, які додають тривимірність та глибину на плоскі поверхні. Проаналізуємо техніки текстурування.

При UV Маппінг (UV Mapping) тривимірні моделі "розгортаються" в двовимірний простір, щоб на них можна було накласти текстуру. Координати UV визначають, як частини текстури відповідають різним частинам моделі. Цей метод є стандартним для більшості графічних додатків і дозволяє досягти високої точності у відтворенні деталей.

Проекційне текстурування (Projection Mapping) включає проекцію текстури на об'єкт з певного напрямку, як світло проектора. Часто використовується для швидкого текстурування складних об'єктів або для створення ефектів освітлення та тіней.

У сферичному текстуруванні [4, 16] текстура накладається на об'єкт, як ніби обгортає сферу. Циліндричне текстурування працює аналогічно, але текстура обгортає об'єкт, маючи форму циліндра. Обидва ці методи є ідеальними для текстурування круглих об'єктів.

Кубічне текстурування (Cube Mapping) використовується для накладання текстур на об'єкти, які можуть бути переглянуті з будь-якого кута, такі як небесні тіла або відображаючі поверхні. Шість квадратних текстур накладаються на кожну сторону куба, створюючи ілюзію 360-градусного перегляду.

Техніки Бамп мапінг (Bump Mapping) [17] та Нормал мапінг (Normal Mapping) використовуються для створення ілюзії рельєфності та текстури на плоских поверхнях шляхом модифікації нормалей поверхні для відтворення світлотіні. Вони дозволяють досягти великої реалістичності без додавання додаткових полігонів.

Детальне текстурування (Detail Texturing) застосовується для додавання маломасштабних деталей до основної текстури, що покращує візуальну глибину і реалістичність поверхонь при близькому огляді. Ця техніка часто використовується для доріг, стін, каміння та інших поверхонь, де детальна текстура накладається на основну для створення більш складної і переконливої візуальної структури.

При мульти-текстурування (Multi-Texturing) кілька текстур накладаються одна на одну на тій самій поверхні, дозволяючи створювати більш складні візуальні ефекти і деталізацію. Це може включати комбінацію кольорових карт, карт освітлення, карт нормалей та інших спеціальних ефектів для досягнення бажаного візуального результату.

Паралакс текстурування (Parallax Mapping) [17] є вдосконаленням техніки бамп мапінгу, яке створює ще більшу глибину і реалізм на поверхнях шляхом

зміщення текстури в залежності від кута огляду. Це додає ілюзію рельєфу без фактичного збільшення кількості полігонів у моделі.

Техніка Теселяція (Tessellation) [9] динамічно збільшує кількість полігонів моделі в реальному часі, забезпечуючи більш деталізовані і реалістичні текстурні та рельєфні ефекти на поверхнях. Використовується для покращення деталізації об'єктів при наближенні камери, забезпечуючи гладкіші і більш деталізовані візуальні враження.

1.3 Методи фільтрації текстур

Фільтрація текстур [15] – це процес обробки текстур для покращення їх якості.

Метод Nearest-Neighbor Filtering є найпростішим і найшвидшим способом фільтрації текстур. Він просто вибирає значення текселя, який геометрично найближчий до запитаної точки. Цей метод не проводить ніякої інтерполяції, тому результат може виглядати "зубчастим" або пікселізованим, особливо на великих масштабуваннях або при обертанні текстур.

Білінійна фільтрація [15, 19, 22] вдосконалює метод найближчого сусіда, використовуючи лінійну інтерполяцію між чотирма найближчими текселями (вліво, вправо, вгору, вниз) від точки запиту, щоб створити більш гладке зображення. Це зменшує пікселізацію на краях об'єктів, але може втратити деяку деталізацію у текстурі при сильному масштабуванні.

Припустимо, що ми маємо текстурні координати (u,v) в текстурному просторі, які знаходяться між чотирма сусідніми пікселями:

$$(x^0, y^0), (x^1, y^0), (x^0, y^1), (x^1, y^1)$$

Кольорове значення в точці (u,v) обчислюється наступним чином:

$$C(x0, y0) \cdot (1-\alpha) \cdot (1-\beta) + C(x1, y0) \cdot \alpha \cdot (1-\beta) + C(x0, y1) \cdot (1-\alpha) \cdot \beta + C(x1, y1) \cdot \alpha \cdot \beta$$

де: $\alpha = u - x_0$, $\beta = v - y_0$

Трилінійна фільтрація [7, 15] розширює білінійний метод, включаючи лінійну інтерполяцію між двома міпмап-рівнями. Міпмапи — це попередньо створені, масштабовані версії оригінальної текстури, що використовуються для підвищення ефективності рендеринга. Трилінійна фільтрація обирає два міпмапи, які найкраще відповідають розміру пікселів на екрані, і інтерполює між ними, щоб зменшити візуальні артефакти і покращити гладкість текстур на різних відстанях.

Припустимо, що у нас є два рівні MIP-текстур L_0 та L_1. Для кожного з них застосовується білінійна фільтрація, результатом якої є $C_0(u,v)$ та $C_1(u,v)$. Тоді трилінійна фільтрація використовує наступну формулу:

$$C(u,v) = C^0(u,v) \cdot (1-d) + C^1(u,v) \cdot d$$

де d – це фактор, що визначає рівень інтерполяції між двома MIP-текстурами.

Анізотропна фільтрація [14-16] – це найбільш продвинутий метод, який враховує кут погляду та напрямок текстури. Вона інтерполює текселі вибірково, в залежності від орієнтації текстури відносно камери, забезпечуючи значно вищу якість текстур, особливо на поверхнях, які спостерігаються під гострими кутами.

Бікубічна фільтрація використовує значення кольорів шістнадцяти сусідніх пікселів для інтерполяції кольору. Це забезпечує більш плавні переходи і кращу якість зображення, ніж білінійна фільтрація, але вимагає більше обчислювальних ресурсів.

Вибір методу фільтрації залежить від конкретного застосування та вимог до продуктивності і якості зображення. У реальних додатках часто використовують комбінації цих методів для досягнення оптимального балансу між якістю зображення і обчислювальними ресурсами.

У сучасних відеоіграх часто використовують анізотропну фільтрацію для поліпшення якості текстур на віддалених об'єктах, таких як дороги чи стіни, щоб уникнути розмиття при перегляді під гострим кутом.

У програмному забезпеченні для 3D моделювання (наприклад, Blender, 3ds Max, Maya) використовуються трілінійна та бікупічна фільтрації для досягнення високоякісних візуальних ефектів та реалістичності зображень.

У додатках VR та AR важливо використовувати високоякісні методи фільтрації текстур, такі як анізотропна фільтрація, для забезпечення реалістичності та зменшення артефактів при перегляді під різними кутами.

Ці методи фільтрації допомагають забезпечити високу якість зображення в різних сферах комп'ютерної графіки, від ігор до професійного 3D моделювання.

Фільтрація з урахуванням рівня деталізації полягає у виборі відповідного рівня MIP-текстури залежно від відстані об'єкта до камери. Це дозволяє зменшити кількість обчислень та покращити продуктивність рендерингу, особливо в сценах з великою кількістю об'єктів.

Для цього визначається відстань об'єкта до камер; Вибирається відповідний рівень деталізації текстури. Застосовується відповідний метод фільтрації (наприклад, трілінійна фільтрація) для вибраного рівня.

Префільтрація виконується на стадії створення текстур, коли текстури зберігаються з вже відфільтрованими даними. Це дозволяє зменшити артефакти при рендерингу та прискорити обчислення.

Префільтрація часто використовується для текстур оточення, таких як кубічні карти (cubemaps), щоб забезпечити якісне відображення світла та тіней.

Для нормальних карт також застосовується префільтрація для покращення вигляду поверхонь.

Фільтрація включає застосування різних фільтрів до текстур для досягнення певних візуальних ефектів, таких як розмиття, різкість, або виділення країв.

Порівняльний аналіз методів фільтрації представлено в таблиці 1.2.

Таблиця 1.2- Порівняння методів фільтрації

Метод	**Якість зображення**	**Швидкодія**	**Переваги**	**Недоліки**
Ближня фільтрація	Низька	Висока	Проста реалізація, висока продуктивність	Артефакти пікселізації
Білінійна фільтрація	Середня	Середня	Гладкі переходи між пікселями	Може викликати розмиття
Трілінійна фільтрація	Висока	Середня	Зменшує артефакти MIP-текстур	Вищі обчислювальні витрати
Анізотропна фільтрація	Дуже висока	Низька	Висока якість при гострих кутах	Високі обчислювальні витрати
Бікупічна фільтрація	Висока	Низька	Дуже гладкі переходи	Складна реалізація, високі обчислювальні витрати
LOD	Висока	Середня	Оптимізує продуктивність на великих сценах	Необхідність попередньої обробки текстур
Ближня фільтрація	Низька	Висока	Проста реалізація, висока продуктивність	Артефакти пікселізації
Білінійна фільтрація	Середня	Середня	Гладкі переходи між пікселями	Може викликати розмиття

Ці методи фільтрації текстур є важливими інструментами в арсеналі розробників комп'ютерної графіки, дозволяючи їм досягати оптимального балансу між якістю зображення та продуктивністю.

1.4 Висновки

Текстурування в комп'ютерній графіці є важливим етапом процесу створення реалістичних зображень. Текстури дозволяють накладати деталі на поверхні об'єктів, що дозволяє досягти високого рівня реалізму в комп'ютерних сценах. Використання текстур надає можливість зменшити кількість ресурсів, необхідних для відтворення складних поверхонь, зокрема, замінюючи геометричні деталі текстурами. Застосування текстур може сприяти покращенню швидкодії графічного двигуна або програми завдяки оптимізації процесу рендерингу.

Текстурування відкриває широкі можливості для творчості художників та дизайнерів, які можуть ефективно використовувати текстури для створення унікальних образів. Воно дозволяє накладати різноманітні візуальні ефекти, такі як кольори, текстури, відблиски та тіні, надаючи об'єктам більш реалістичного та привабливого вигляду.

Проте текстури можуть вимагати значних обсягів пам'яті або обчислювальних ресурсів, особливо в разі використання великої кількості високодеталізованих текстур. Це може негативно вплинути на продуктивність графічного двигуна та потребувати додаткової оптимізації.

Загалом, текстурування в комп'ютерній графіці є невід'ємною частиною процесу створення реалістичних і візуально привабливих зображень, що вимагає ретельного підходу та технічної компетентності. Воно забезпечує високий рівень деталізації та реалізму, дозволяє оптимізувати використання ресурсів та відкриває широкі можливості для художнього самовираження.

2 РОЗРОБКА НОВИХ МЕТОДІВ ТЕКСТУРУВАННЯ

2.1 Метод високопродуктивного перспективно-коректного текстурування

Для урахування перспективи використовуємо метод Хекберта [10, 21], в основі якого лежить виконання перспективного перетворення. При використанні матриць вираз має такий вигляд:

$$[uq \quad vq \quad q] = [x \quad y \quad z]\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} = [x \quad y \quad z]\begin{bmatrix} EI - FH & FG - DI & DH - EG \\ CH - BI & AI - CG & BG - AH \\ BF - CE & CD - AF & AE - BD \end{bmatrix} \tag{2.1}$$

Де u, v – текси; x, y – екранні координати; A, I – сталі перспективної проекції.

Останній вираз можна записати так:

$$u = \frac{Ax + By + C}{Gx + Hy + I}, v = \frac{Dx + Ey + F}{Gx + Hy + I} \tag{2.2}$$

Визначення текстурних координат є складним процесом, оскільки вимагає виконання детальних розрахунків на рівні пікселів за формулою (2.2).

За умови, що $x_{i+1} = x_i + 1$

$$u_{i+1} = \frac{A_1 \cdot (x_i + 1) + B_1 \cdot y_i + C_1}{G \cdot (x_i + 1) + H \cdot y_i + I} = \frac{(A_1 \cdot x_i + B_1 \cdot y_i + C_1) + A_1}{(G \cdot x_i + H \cdot y_i + I) + G}.$$

Запишемо формулу для u_0

$$u_0 = \frac{A_1 \cdot (x_0 + 0) + B_1 \cdot y_i + C_1}{G \cdot (x_0 + 0) + H \cdot y_i + I} = \frac{A_1 \cdot x_0 + B_1 \cdot y_i + C_1}{G \cdot x_0 + H \cdot y_i + I}.$$

Вираз для u_1 має такий вигляд:

$$u_1 = \frac{A_1 \cdot (x_0 + 1) + B_1 \cdot y_i + C_1}{G \cdot (x_0 + 1) + H \cdot y_i + I} = \frac{A_1 \cdot x_0 + A_1 + B_1 \cdot y_i + C_1}{G \cdot x_0 + G + H \cdot y_i + I}.$$

Запишемо, що u_2:

$$u_2 = \frac{A_1 \cdot (x_0 + 2) + B_1 \cdot y_i + C_1}{D \cdot (x_0 + 2) + E \cdot y_i + F} = \frac{A_1 \cdot x_0 + A_1 \cdot 2 + B_1 \cdot y_i + C_1}{D \cdot x_0 + D \cdot 2 + E \cdot y_i + F}.$$

Узагальнимо викладки. Формула для u_n має вигляд:

$$u_n = \frac{A_1 \cdot (x_0 + n) + B_1 \cdot y_i + C_1}{D \cdot (x_0 + n) + E \cdot y_i + F} = \frac{A_1 \cdot x_0 + A_1 \cdot n + B_1 \cdot y_i + C_1}{D \cdot x_0 + D \cdot n + E \cdot y_i + F} \quad (2.4)$$

Аналогічно для v_n:

$$v_n = \frac{D_1 \cdot x_i + E_1 \cdot y_0 + E_1 \cdot n + F_1}{G \cdot x_i + H \cdot y_0 + H \cdot n + I}.$$

З поданих формул зрозуміло, що для обчислення кожного текселя потрібно виконати дві операції ділення, вісім операцій додавання та вісім операцій множення.

Як видно із формул (2.4), для n формули для обчислення $A_1 \cdot x_0 + B_1 \cdot y_i + C_1$ та $G \cdot x_0 + H \cdot y_i + I$ є сталими. Тому їх можна обчислити один раз для кожного рядка растеризації за такими формулами:

$$u_t = A_1 \cdot x_0 + A_1 \cdot n + B_1 \cdot y_i + C_1,$$
$$u_b = G \cdot x_0 + H \cdot y_i + I.$$

де u_t і u_b константи дробу (2.4) відповідно. Отже, u_n може бути визначено за виразом:

$$u_n = \frac{u_t + A_1 \cdot n}{u_b + G \cdot n} \quad (2.5)$$

Узагальнемо результати. При знаходженні текселів параметри u_{1t} і u_{1b} є сталими, а тому їх можна знаходити один раз. Аналогічні твердження мають місце і для v_n.

Це дозволяє зменшити число мікрооперацій операцій додавання та множення до чотирьох відповідно.

На основі отриманих математичних залежностей можна розробити алгоритм паралельного обчислення координат текселів. Спочатку для кожного рядка растеризації визначаються параметри ut і ub, після чого паралельно обчислюються значення чисельника та знаменника за відповідними формулами.

$$w_n = u_t + A_1 \cdot n, v_n = u_b + G \cdot n.$$

У подальшому визначається дріб:

$$u_n = \frac{w_n}{v_n}.$$

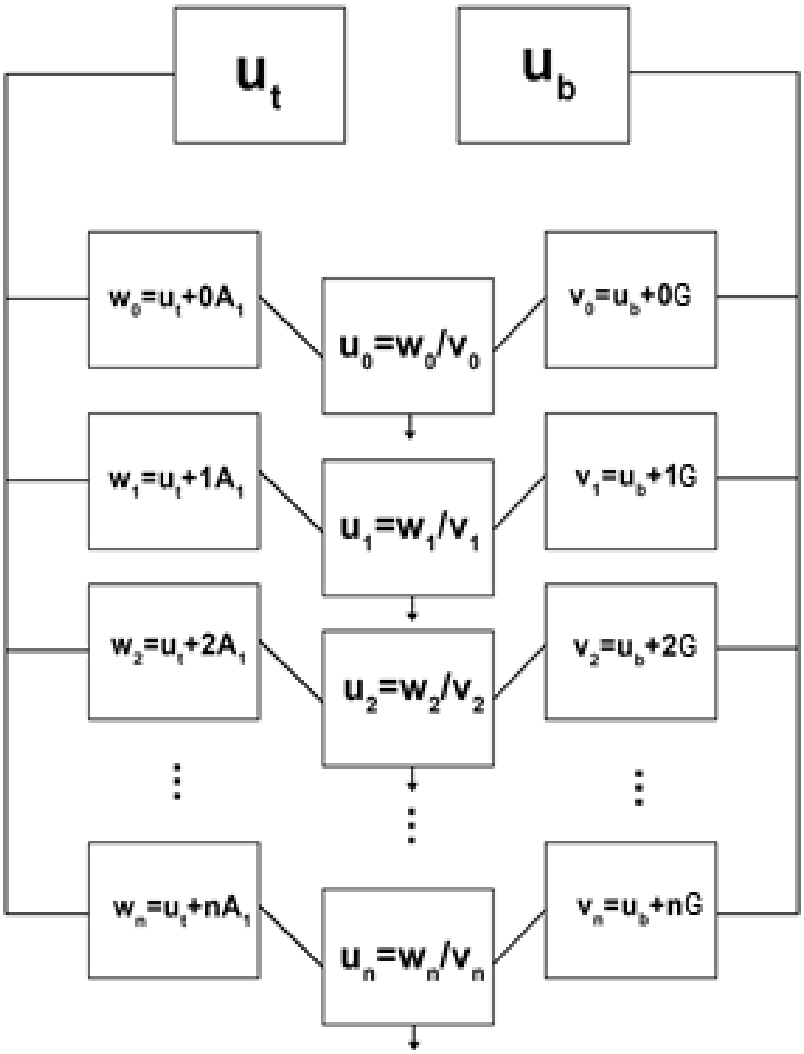


Рисунок 2.1 – Паралельне визначення текселів

Для кожного послідуючого x, n виконують інкрементування, тому можна записати:

$$w_n = w_{n-1} + A_1, v_n = v_{n-1} + G, \quad (2.6)$$

Аналізуючи вираз (2.6), можна розробити послідовний алгоритм визначення координат: wn і vn для кожного x знаходять додаванням до wn-1 і vn-1, розрахованих для x-1, A1 і D відповідно. Структуру обчислювального процесу відображено на рис. 2.2.

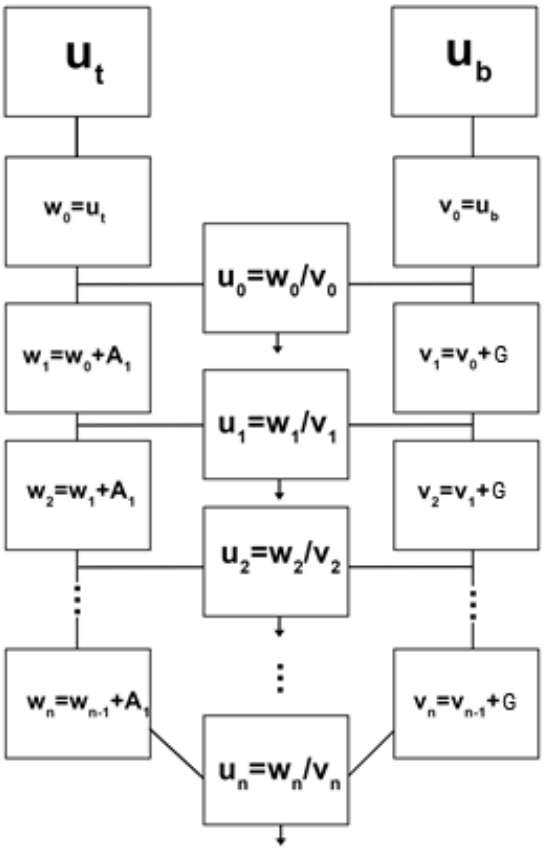


Рисунок 2.2 – Визначення текселів

Розрахунок текселів за допомогою запропонованого алгоритму можна також прискорити через паралельні обчислення. Одним із можливих способів реалізації паралелізму є паралельне проходження декількох рядків одночасно. Однак цей підхід буде недостатньо ефективним у випадках, коли кількість пікселів на рядок перевищує кількість рядків. Тому доцільно використовувати методи паралельних обчислень у межах одного рядка.

Можна запропонувати паралельний розрахунок текселів у парних і непарних точок рядка пастеризація (див. рис. 2.3).

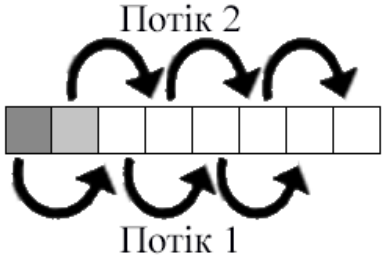


Рисунок 2.3 – Паралельне обчислення текселів

За умови, що $w_n = w_{n-1} + A_1$, а $v_n = v_{n-1} + D$

$$\begin{matrix} w_{n+1} & (w_{n-1} + A_1) + A_1 \\ v_{n+1} & (v_{n-1} + G) + G \end{matrix}$$

Незалежний розрахунок текселів в парних і непарних точках рядка растеризації виконують за виразами:

$$w_n = w_{n-2} + 2A_1$$
$$v_n = v_{n-2} + 2G$$
$$(2.7)$$

В циклі підготування необхідно знайти перші два текселя за виразом (2.7).

З урахуванням двох останніх виразів можна реалізувати паралельну растеризацію рядка з довільною кількістю потоків з використанням виразів:

$$\begin{matrix} w_n & w_{n-k} + kA_1 \\ v_n & v_{n-k} + kG, \end{matrix}$$

де k – кількість паралельних потоків.

Суттєво підвищити продуктивність текстурування шляхом зустрічного текстурування : справа наліво за виразом (2.6), а зліва направо за таким виразом:

$$w_n = w_{n+1} - A_1, v_n = v_{n+1} - G$$

.

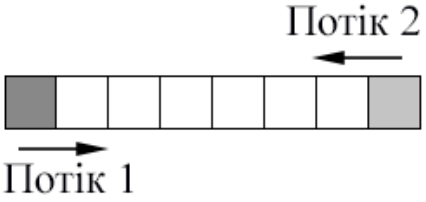


Рисунок 2.4 – Зустрічне текстурування

У вдосконаленій версії методу Хекберта [7, 21] використовуються ітераційні формули для визначення тексельних координат, що допомагає

скоротити кількість арифметичних операцій і збільшити ефективність на 26%. Також було розроблено методики для розпаралелювання процесу обрахунку текстурних координат на основі аналітичних залежностей, що значно прискорює текстурування.

2.2 Метод підвищення продуктивності при перспективно-коректному текстуруванні

Складемо рівняння еліпса за алгоритмом [5, 18]:

1. Знаходження осей:

$$(U_x, V_x) = \left(\frac{du}{dx}, \frac{dv}{dx}\right), \left(U_y, V_y\right) = \left(\frac{du}{dy}, \frac{dv}{dy}\right) \quad (2.8)$$

2. Розрахунок коефіцієнтів

$$\begin{gathered} A = V_x^2 + V_y^2, \\ B = -2\left(U_x V_x + U_y V_y\right), \\ C = U_x^2 + U_y^2, \\ F = \left(U_x V_y + U_y V_x\right)^2. \end{gathered} \quad (2.9)$$

3. Визначення рівняння:

$$AU^2 + BUV + CV^2 = F, \quad (2.10)$$

де $U = u - u_0, V = v - v_0$.

Для визначення необхідних для фільтрації текселів складають рівняння еліпса за формулами (2.8), (2.9), (2.10).

Знайдемо похідні, що мають такий вигляд:

$$\frac{\partial u}{\partial x} = u(x+1,y) - u(x,y) = \frac{A(x+1)+By+C}{G(x+1)+Hy+I} - \frac{Ax+By+C}{Gx+Hy+I},$$

$$\frac{\partial v}{\partial x} = v(x+1,y) - u(x,y) = \frac{D(x+1)+Ey+F}{G(x+1)+Hy+I} - \frac{Dx+Ey+F}{Gx+Hy+I}$$

$$\frac{\partial u}{\partial y} = u(x,y+1) - u(x,y) = \frac{Ax+B(y+1)+C}{Gx+H(y+1)+I} - \frac{Ax+By+C}{Gx+Hy+I} \quad (2.11)$$

$$\frac{\partial v}{\partial y} = v(x,y+1) - u(x,y) = \frac{Dx+E(y+1)+F}{Gx+H(y+1)+I} - \frac{Dx+Ey+F}{Gx+Hy+I}$$

Як видно з формул обчислення похідних для визначення коефіцієнтів рівняння еліпса є трудомісткою процедурою. На кожен піксель у рядку растеризації необхідно виконати 10 операцій множення, 10 операцій додавання, 4 операції ділення та 1 операцію віднімання. Тому існує потреба у розробці методів, які б мали меншу обчислювальну складність.

Процес визначення коефіцієнтів для рівняння еліпса через обчислення похідних є витратним за часом. Для кожного пікселя в лінії растеризації потрібно здійснити десять множень, десять додавань, чотири ділення та одне віднімання. Це підкреслює необхідність створення більш ефективних методів, що вимагають менших обчислювальних витрат.

Визначимо похідні для виразу (2.8), використавши формулу (2.11):

$$\frac{A(x+1)+By+C}{G(x+1)+Hy+I} - \frac{Ax+By+C}{Gx+Hy+I} = \frac{AI-CG+AHy-BGy}{(I+Gx+Hy)(G+I+Gx+Hy)} \quad (2.12)$$

$$\frac{D(x+1)+Ey+F}{G(x+1)+Hy+I} - \frac{Dx+Ey+F}{Gx+Hy+I} = \frac{DI-FG-DHy+EGy}{(I+Gx+Hy)(G+I+Gx+Hy)} \quad (2.13)$$

$$\frac{Ax + B(y+1) + C}{Gx + H(y+1) + I} - \frac{Ax + By + C}{Gx + Hy + I} = \frac{BI - CH - AHx + BGx}{(I + Gx + Hy)(G + I + Gx + Hy)}$$
(2.14)

$$\frac{Dx + E(y+1) + F}{Gx + H(y+1) + I} - \frac{Dx + Ey + F}{Gx + Hy + I} = \frac{EI - FH - DHx + EGx}{(I + Gx + Hy)(G + I + Gx + Hy)}$$
(2.15)

Як випливає з формул (2.12) і (2.13), чисельники не залежать від значень і, отже, можуть бути визначені лише один раз для кожного . Додатково, функції і змінюються лінійно, що дозволяє розраховувати їх за допомогою ітерацій. Розглянемо обчислення зміни чисельників для цих формул за допомогою методу кінцевих різниць [11]:

$$M = \big(AI - CG + AH(y+1) - BG(y+1)\big) - (AI - CG + AHy - BGy) = AH - BG$$
(2.16)

$$N = \big(DI - FG - DH(y+1) + EG(y+1)\big) - (DI - FG - DHy + EGy) = DH - EG$$
(2.17)

З аналізу формул (2.16) і (2.17) стає зрозуміло, що прирости функцій не пов'язані з координатами точок, а залежать лише від коефіцієнтів перетворення полігона, тому їх можна розрахувати один раз для кожного полігону. Так, чисельники n для формул (2.12) і (2.13) для мінімального n можна обчислити використовуючи наступні формули:

$$\text{num} = AI - CG + AHy - BGy,$$
$$\text{num} = DI - FG - DHy + EGy,$$

а для кожного наступного за виразами:

$$\text{num}_i = \text{num}_{i-1} + M,\ \text{num}_i = \text{num}_{i-1} + N.$$

Тому, обсяг арифметичних операцій на один піксель скорочується.

З виразів (2.14) і (2.15) видно що чисельники не є змінними від y і можуть були розраховані один раз на рядок растеризації. Знайдемо приріст чисельників для виразів (2.14) і (2.15):

$$O = \big(BI - CH - AH(x+1) + BG(x+1)\big) - (BI - CH - AHx + BGx) = BG - AH \quad (2.18)$$

$$P = \big(EI - FH - DH(x+1) + EG(x+1)\big) - (EI - FH - DHx + EGx) = EG - DH \quad (2.19)$$

Таким чином, чисельники для останніх двох виразів для початкового рядка растеризації розраховуються так:

$$\text{num} = BI - CH - AHx + BGx$$
$$\text{num} = EI - FH - DHx + EGx,$$

а для наступного за виразами:

$$num_i = num_{i-1} + O,$$
$$\text{num}_i = \text{num}_{i-1} + P.$$

Відповідно до формул (2.18) і (2.19) прирости чисельників для формул (2.14) і (2.15) також не залежать від координат точок, а лише від коефіцієнтів перетворення полігона, а тому можуть бути обчислені один раз на полігон.

Згідно з формулами (2.18) і (2.19), прирости значень чисельників, використовуваних у формулах (2.14) і (2.15), не залежать від положення точок, а тільки від коефіцієнтів перетворення полігона. Таким чином, вони можуть бути визначені один раз для кожного полігону. Оскільки чисельники зазначених формул залежать від певних значень, при горизонтальній послідовності растеризації необхідно інкрементувати чисельник кожного пікселя у лінії растеризації. Кожен новий рядок у процесі растеризації має точки з

координатами, що дорівнюють координатам точок у попередньому рядку, тому розрахунок значень чисельників для формул (2.14) і (2.15) ефективно проводити заздалегідь для всього полігону перед початком растеризації, щоб уникнути повторення розрахунків.

Розрахунок знаменника (*denom*) для виразів (2.12) – (2.15) можна скоротити. Нехай $G + I$ позначимо як J, $I + Hy$ як K, а $G + I + Hy = J + Hy$ як L. Знаменник виразів (2.12) – (2.15) має такий вигляд:

$$\text{denom} = (I + Gx + Hy)(G + I + Gx + Hy) = (K + Gx)(L + Gx). \quad (2.20)$$

Використаємо отримані вирази у формули (2.11), отримуємо, що:

$$\frac{\partial u}{\partial x} = \frac{num_{-1} + M}{(K + Gx)(L + Gx)}, \frac{\partial v}{\partial x} = \frac{num_{-1} + N}{(K + Gx)(L + Gx)},$$

$$\frac{\partial u}{\partial y} = \frac{num_{-1} + O}{(K + Gx)(L + Gx)}, \frac{\partial v}{\partial y} = \frac{num_{-1} + P}{(K + Gx)(L + Gx)} \quad (2.21)$$

Відповідно до формул (3.19) для обчислення часткових похідних достатньо 6 операцій додавання, 2 операції множення та 4 операції ділення на кожен піксель. Порівняно з базовим методом, кількість операцій додавання зменшено на 4, операцій множення зменшено на 8, а операцій віднімання зменшено на 1.

Запропонований метод дозволяє знизити обчислювальну складність визначення параметрів рівняння еліпса при обчисленні положення проекції пікселя на текстурну площину за методом Хекберта, що дозволяє зменшити кількість операцій додавання та множення шляхом використання ітераційних формул і, як наслідок, підвищити продуктивність до 18%.

2.3 Розробка методу накладання текстур з урахуванням координати глибини

Цей метод заснований на принципі, що кожен крок по рядку растеризації виконується за нелінійною траєкторією, яка визначається залежно від значень z-координат у просторі для даного рядка.

На рисунок 2.5 зображено Q_1Q_2 – рядок растеризації в об’єктній системі координат, AB – в екранній системі координат.

Треба встановити, як змінюються координати j-тої точки у рядку растеризації на поверхні об'єкта

Точка O, яка є центром перспективи, розташована на відстані d від площини екрана, як показано на рисунку 2.5. Для відрізка прямої Q_1Q_2 існує така залежність [2].

$$\frac{x(j)-x_1}{x_2-x_1}=\frac{z(j)-z_1}{z_2-z_1}=w \tag{2.22}$$

Для *x(j)* і *z(j* можна записати такі вирази:

$$x(j)=x_1+(x_2-x_1)\cdot\frac{z(j)-z_1}{z_2-z_1} \tag{2.23}$$

$$z(j)=z_1+(z_2-z_1)\cdot\frac{x(j)-x_1}{x_2-x_1} \tag{2.24}$$

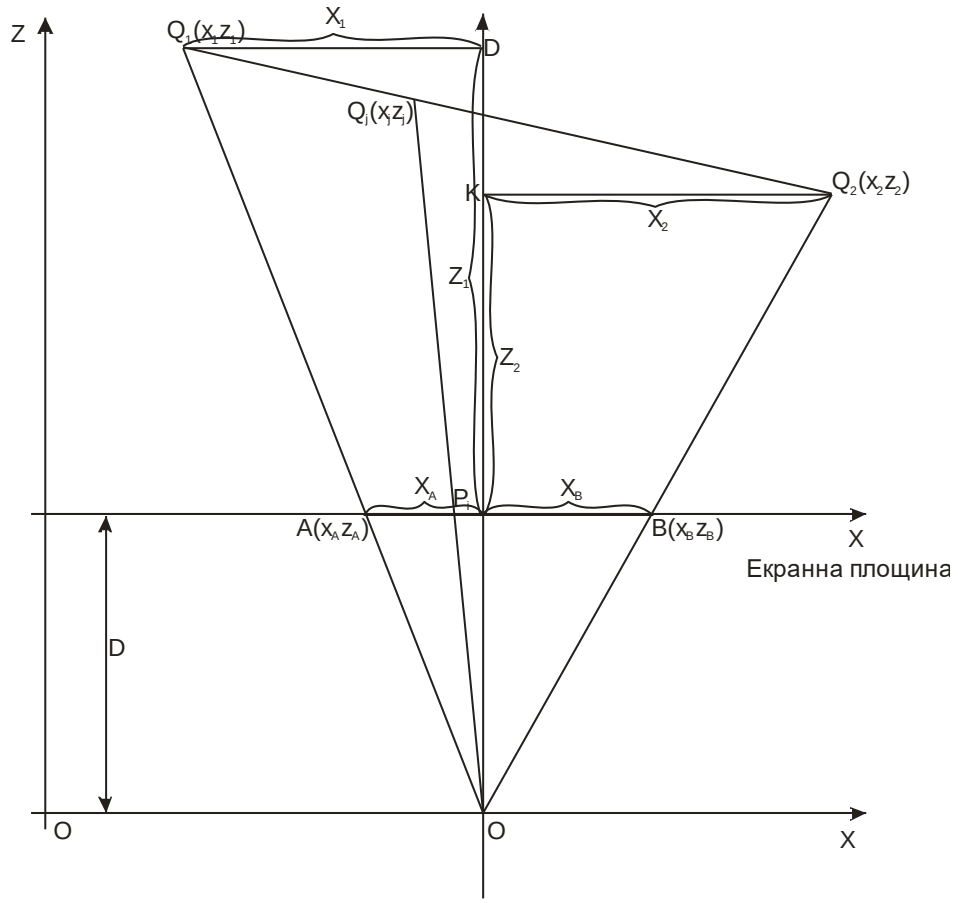


Рисунок 2.5 – Розміщення відрізка Q_1Q_2 в екранній системі координат

Врахуємо (2.22) в рівняннях (2.23) і (2.24)

$$x(j) = x_1 + (x_2 - x_1) \cdot w. \tag{2.26}$$

$$z(j) = z_1 + (z_2 - z_1) \cdot w. \tag{2.27}$$

Трикутник $\triangle OP_iB$ подібний $\triangle OKQ_2$, оскільки в них кути однакові. Аналогічно $\triangle OP_iA$ подібний трикутнику $\triangle ODQ$. Тому, маємо:

$$\frac{d}{z_2} = \frac{x_B}{x_2};$$
$$\frac{d}{z_1} = \frac{x_A}{x_1};$$
$$\frac{d}{z(j)} = \frac{x(i)}{x(j)}. \tag{2.28}$$

З формули (2.28) можна записати, що:

$$x_1 = \frac{z_1 \cdot x_A}{d}$$
$$x_2 = \frac{z_2 \cdot x_B}{d}$$
$$z(j) = \frac{x(j)}{x(i)} \cdot d.$$
(2.29)

З урахуванням формул (2.27), (2.28) та (2.29) знаходимо:

$$z(j) = \frac{x(j)}{x(i)} \cdot d = \frac{x_1 + w(x_2 - x_1)}{x(i)} \cdot d = \frac{\frac{z_1 \cdot x_A}{d} + w\left(\frac{z_2 \cdot x_B}{d} - \frac{z_1 \cdot x_A}{d}\right)}{x(i)} \cdot d == \frac{z_1 \cdot x_A + w(z_2 \cdot x_B - z_1 \cdot x_A)}{x(i)}.$$
(2.30)

$x(i) = x_A + (x_B - x_{A1}) \cdot u -$ рівняння прямої в екранній системі координат

$$u = \frac{x(i) - x_A}{x_B - x_A}$$
(2.31)

Рівняння (2.30) матиме вигляд:

$$z_1 + w(z_2 - z_1) = \frac{z_1 \cdot x_A + w(z_2 \cdot x_B - z_1 \cdot x_A)}{x_A + (x_B - x_{A1}) \cdot u};$$
$$z_1 + w(z_2 - z_1) = \frac{z_1 \cdot x_A}{x_A + (x_B - x_{A1}) \cdot u} + \frac{w(z_2 \cdot x_B - z_1 \cdot x_A)}{x_A + (x_B - x_{A1}) \cdot u};$$
$$z_1 - \frac{z_1 \cdot x_A}{x_A + (x_B - x_{A1}) \cdot u} = \frac{w(z_2 \cdot x_B - z_1 \cdot x_A)}{x_A + (x_B - x_{A1}) \cdot u} - w(z_2 - z_1);$$
$$\frac{z_1 \cdot x_A + z_1 \cdot (x_B - x_{A1}) \cdot u - z_1 \cdot x_A}{x_A + (x_B - x_{A1}) \cdot u} = \left(\frac{(z_2 \cdot x_B - z_1 \cdot x_A) - (x_A + (x_B - x_A) \cdot u) \cdot (z_2 - z_1)}{x_A + (x_B - x_{A1}) \cdot u}\right);$$
$$z_1 \cdot (x_B - x_{A1}) \cdot u = w\big(z_2 \cdot x_B - z_1 \cdot x_A - z_2 \cdot x_A + z_1 \cdot x_A - z_2 \cdot u \cdot (x_B - x_A) + z_1 \cdot u \cdot (x_B - x_A)\big);$$
$$z_1 \cdot (x_B - x_{A1}) \cdot u = w\big(z_2 \cdot (x_B - x_A) - z_2 \cdot u \cdot (x_B - x_A) + z_1 \cdot u \cdot (x_B - x_A)\big);$$
$$z_1 \cdot u = w(z_2 - z_2 \cdot u + z_1 \cdot u);$$
$$w = \frac{z_1 \cdot u}{z_2 - u \cdot (z_2 - z_1)}$$

Підставимо отримане значення в (2.26) та (2.27):

$$x(j) = x_1 + (x_2 - x_1) \cdot \frac{z_1 \cdot u}{z_2 - u \cdot (z_2 - z_1)} \quad (2.32)$$

$$z(j) = z_1 + (z_2 - z_1) \cdot \frac{z_1 \cdot u}{z_2 - u \cdot (z_2 - z_1)} \quad (2.33)$$

На рисунку 2.6 зображено приклад зображення, сформованого запропонованим методом.

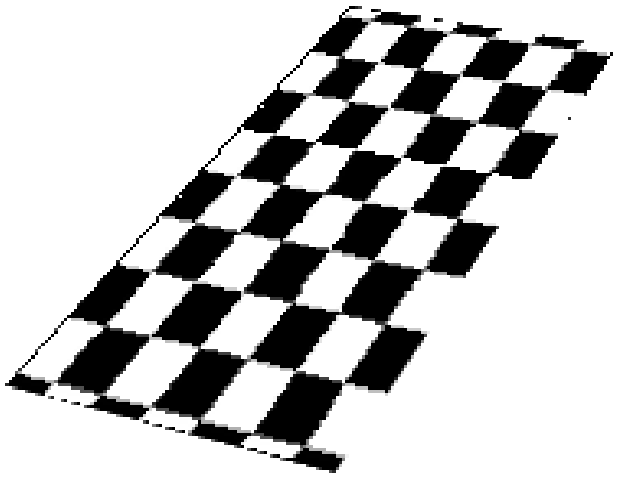


Рисунок 2.6 – Приклад текстурування за розробленим методом

Таким чином, розроблені аналітичні формули для обрахунку текстурних координат, виходячи з z-координат об'єкта. Це забезпечило можливість врахування перспективи об'єкта і визначення нелінійного зв'язку між координатами на екрані та текстурними координатами.

2.4 Висновки

У розділі було запропоновано модифікацію методу Хекберта, яка включає використання нових ітераційних розрахунків для визначення позицій текселів, зменшуючи тим самим загальну кількість арифметичних операцій додавання та множення при встановленні текстурних координат для кожного пікселя, що сприяло підвищенню ефективності на 26%.

Розроблено нові ітераційні рівняння для точного визначення параметрів еліпса у процесі розрахунку проекції пікселя на текстурну поверхню згідно методу Хекберта, забезпечуючи зниження обсягу арифметичних операцій і зростання продуктивності на 18%.

Запропоновано метод текстурування, який враховує для текселів координату глибини, що підвищує реалістичність формування графічних зображень.

3 РОЗРОБКА ПРОЦЕДУРНИХ ТЕКСТУР З ВИКОРИСТАННЯМ ШУМУ ПЕРЛІНА

3.1 Формування шуму Перліна

Шум Перліна [22] представляє собою візуальну структуру, яка характеризується як випадковими властивостями, так і певною впорядкованістю, ступінь якої можна змінювати за допомогою спеціальних програмних засобів. Такі штучно створені структури знайшли широке застосування в комп'ютерній графіці для імітації реалістичних природних текстур, наприклад, формувань хмар, вогню, гірських та лісових масивів. Також можлива генерація турбулентних потоків і сильно неупорядкованих структур, наприклад, броунівського руху частинок. Можна стверджувати, що шум Перліна широко використовується у візуальних сценах, генерованих комп'ютером.

Цей тип текстури генерується за псевдовипадковим алгоритмом шляхом застосування спеціального виду інтерполяції. Одним із найпоширеніших алгоритмів при створенні візуальних ефектів є так званий класичний шум Перліна, створений ще у 1983 р. [22]. З тих пір алгоритм отримав реалізацію на різних мовах програмування і розвивався з метою створення все більш складних і максимально реалістичних об'єктів. Були вигадані алгоритми створення функції шуму будь-якої необхідної розмірності.

Формування класичного шуму Перліна в 1-D включає в себе три етапи:

- створення прямої, на якій через визначений крок будуються вектори, що мають випадкове напрямок;

- розрахунок за алгоритмом Перліна координат точок на площині або просторі між створеними векторами;

- інтерполяція координат точок.

Найсучаснішою бібліотекою для обчислення координати шуму Перліна в 1-D на мові програмування C++ є модуль perline-noise, який дозволяє обчислювати зміну значення функції шуму залежно від координати.

Як було сказано вище, для отримання шуму Перліна в 1-D створюється пряма, поділена псевдовипадково спрямованими векторами (див. рис. 3.1). Напрямок векторів залежить від заданих початкових умов, які використовуються у вигляді цілого числа.

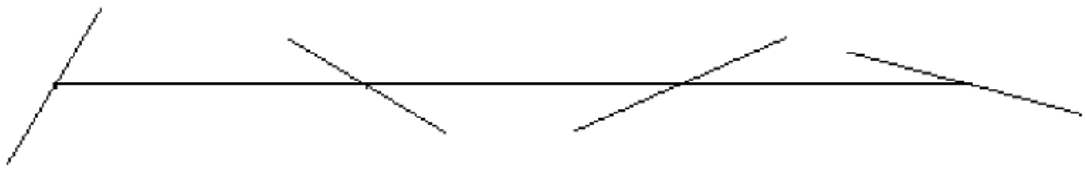


Рисунок 3.1 – Розбиття векторами

На наступному етапі створюється (подібно до алгоритму кривих Безьє) хвиляста лінія, яка дотикається послідовно всіх векторів. Проте, дана хвиляста лінія є занадто гладкою, щоб нагадувати структуру природних об'єктів. Далі необхідно виконати наступну операцію: підвищується роздільна здатність початкової картини (дане розділення називають октавою — якщо крок між псевдовипадковими векторами зменшується у 2 рази, то це друга октава). Якщо присутні 3 відрізки, то на другій октаві число відрізків збільшується до 6. Згодом для всіх точок картини підвищеної роздільної здатності повторюється операція будівництва векторів і дотичної до них кривої, і дана крива складається з первісною.

Якщо діяти за наведеним алгоритмом і далі, то на третій октаві складаються уже 3 кривих. При складанні 4 і більше октавних кривих результуюча картина набуває дещо «ступінчастого» вигляду і нагадує характеристику природних об'єктів у розподілі параметрів.

Описаний алгоритм генерації координат 1-D шуму Перліна реалізований у бібліотеці perline-noise для мови C++. За допомогою застосування різної кількості октав та початкових умов для кожної осі координат, можна також створити 2-D або 3-D картину шуму Перліна, а комбінуванням вкладу кількох октав в одну координату отримують складні картини шуму. Для вирішення цієї

задачі було розроблено програму, яка виконує наступну послідовність дій (на прикладі генерації 3-D зображення):

1. Імпорт функції PerlinNoise з бібліотеки perlin_noise.

2. Імпорт відповідних функцій для створення тривимірного графіка [7, 8].

3. Створення функцій генерування координат точок шуму Перліна для кожної осі координат, кожна з яких дозволяє вибрати кількість використовуваних октав та початкові умови для генерації.

4. Створення змінної, що означає кількість точок, на яку розбивається відрізок по кожній осі координат.

5. Створення трьох порожніх масивів, в які будуть записуватися згенеровані координати шуму Перліна по кожній з осей x, y, z.

6. Перебір у циклі послідовно кожної точки осей координат, генерація координат точок шуму Перліна за допомогою функцій п. 3 і запис кожної координати у відповідний масив п. 5.

7. Створення фігури і тривимірної області графіка.

8. Рисування точкового графіка за масивами координат із можливістю вибору розміру і кольору точок, типу бажаного маркера.

Наведемо результати, що виходять при розрахунках прикладів із різним набором початкових умов для визначення координат шуму Перліна і числом використовуваних октав. Спочатку використовуємо однакову кількість октав (наприклад, 10) і початкові умови (вказується цілим числом, наприклад, 100) для всіх осей координат. 3-D графік, який виводиться програмою при цьому, представлений на рис. 3.2. Умовний розмір точок прийнято рівним 0.01, кількість точок — 1000.

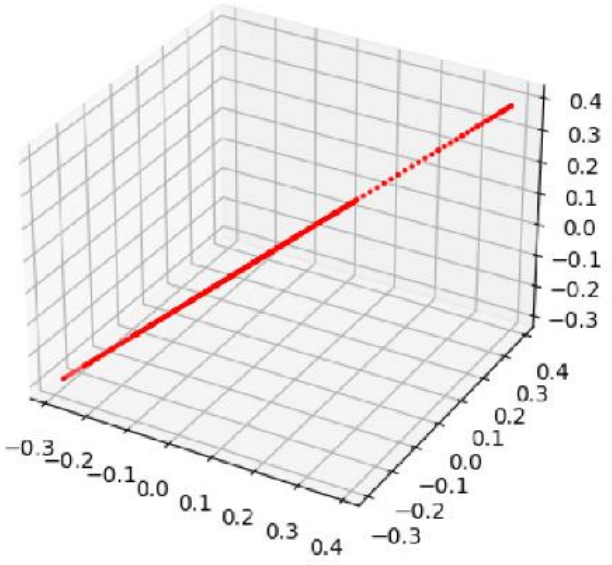


Рисунок 3.2 – Шум Перліна, генерований при однакових апріорних

З графіку видно, що він має прямий характер, тобто всі координати змінюються за лінійним законом.

Навіть при незначному зміні початкових умов (при значеннях 100, 101, 102) графік істотно змінюється (див. рис. 3. 3).

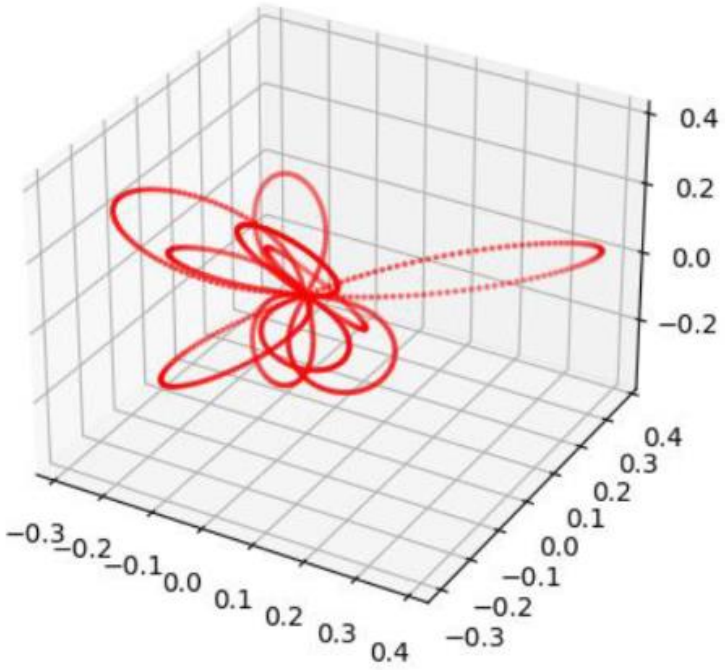


Рисунок 3.3 – Шум Перліна, генорований за різних початкових умовах

Отримуємо складну траєкторію, яка змінюється в тих же межах, що і рис. 3.2. Тепер спробуємо більший розмір генерованих точок - 1 (для збільшення виразності виведеної картини) та одночасно задати невелику, але різну кількість октав для осей координат (1, 2, 3) і різні початкові умови (100, 200, 300). При цьому програмою виводиться рисунок, що відрізняється від попередніх (див. рис. 3.4).

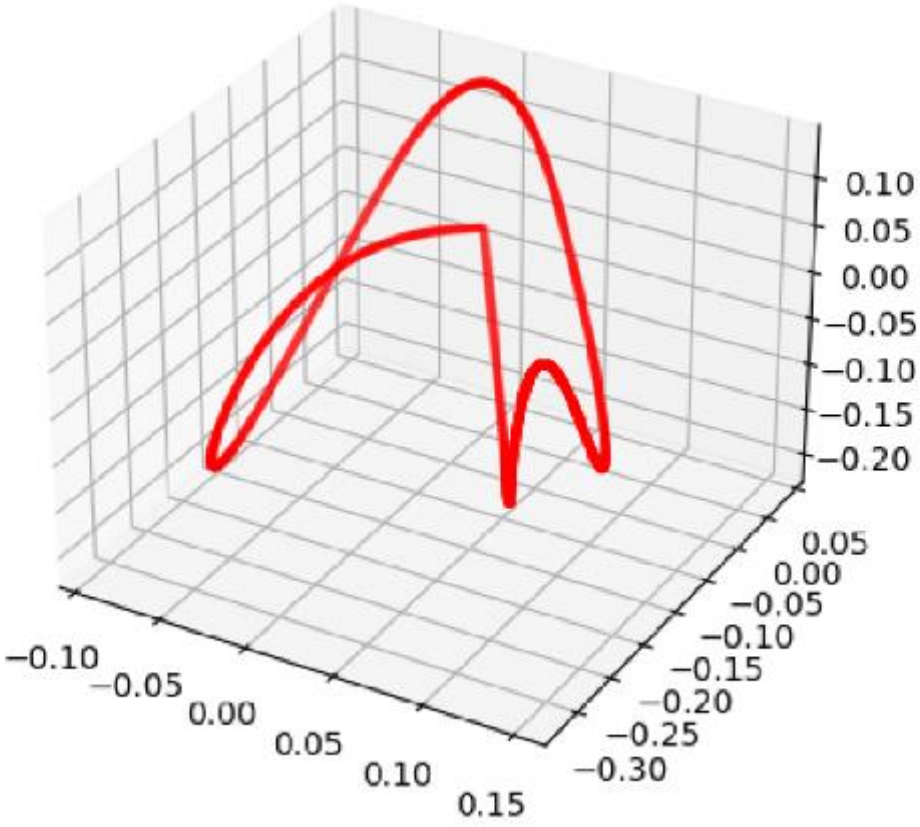


Рисунок 3.4 – Шум Перліна при різній кількості октав

Можливі й інші варіанти штучно згенерованих за допомогою функції Шум Перліна картин. Таким чином, варіюючи початкові умови та кількість октав, можна отримати дуже цікаві для різних галузей науки та техніки масиви даних (наприклад, імітуючи різні види течій – ламінарних, турбулентних) [9].

Наприклад, використовуючи ту ж графічну бібліотеку Matplotlib і чотирикратне застосування функції PerlinNoise з різною кількістю октав, можна отримати картину в 2-D, що нагадує хмари (див. рис. 3. 5). Використовуючи

бібліотеки виводу графіки, відмінні від Matplotlib, можна отримати і інші різні цікаві візуальні ефекти [10].

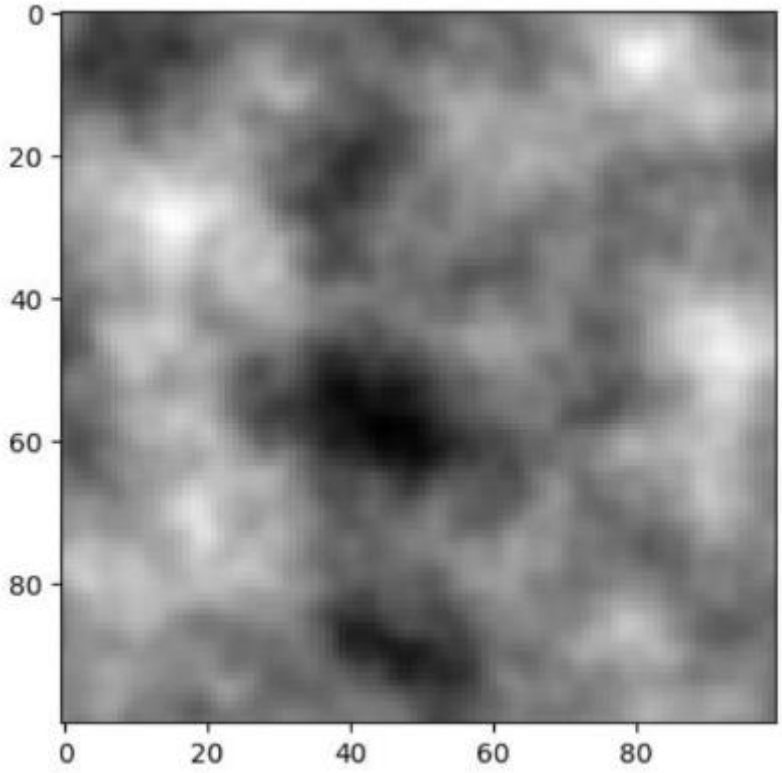


Рисунок 3.5 – Шум Перліна, що імітує вид хмар у 2-D

Таким чином, у БДР описано методику генерації шуму Перліна і сферу його практичного застосування.

Розроблено програму на мові C++, яка дозволяє виводити візуальну тривимірну картину формованого шуму Перліна за різними поєднаннями аргументів функції з бібліотеки perlin_noise для кожної з осей координат. Також представлена повна послідовність дій, які виконує програма, описані використовувані команди та підключаємі бібліотеки функцій.

У БДР розглянуто тільки некомерційні, вільно розповсюджувані програмні засоби, що є важливим з огляду на те, що вони доступні всім зацікавленим дослідникам, і їх освоєння не вимагає якоїсь складної підготовки. Зазвичай, засоби мови програмування C++ є інтуїтивно зрозумілими.

3.2 Алгоритм формування зображення поверхні води з використанням шуму Перліна

Шум Перліна – це тип градієнтного шуму, який часто використовується в комп'ютерній графіці для створення натуралістичних текстур та форм, включаючи воду, хмари, вогонь, та інше. Кен Перлін розробив цей алгоритм у 1983 році, і він виявився надзвичайно корисним у графіці завдяки своїй здатності генерувати візуально приємні, органічні візерунки.

Метод використовування шуму Перліна для формування води включає такі етапи:

1. Генерація базового шуму: спочатку генерується базовий шум Перліна. Це здійснюється шляхом інтерполяції між псевдовипадковими значеннями, що заздалегідь розподілені в просторі. Кожній точці в просторі присвоюється псевдовипадкове градієнтне векторне значення, а потім для точки, яка має бути оцінена, визначаються вектори до її найближчих градієнтів і знаходиться їх дотична.

2. Покращення шуму: часто використовують кілька шарів шуму з різним масштабом і амплітудою, щоб створити більш складну і реалістичну текстуру. Це називається фрактальним шумом або шумом Перліна октав. Шари комбінуються, щоб додати деталі на різних рівнях масштабу.

3. Анімація води: для анімації води за допомогою шуму Перліна можна змінювати параметри шуму з часом, наприклад, зміщуючи координати шуму або змінюючи його частоту. Це створює враження руху та хвиль на водній поверхні.

4. Візуалізація: кінцеве візуальне представлення залежить від способу відображення шуму на водну поверхню. Зазвичай це включає застосування шуму як дисплейсмент-мапу, що змінює висоту водної поверхні відповідно до значення шуму в кожній точці.

Використання шуму Перліна для створення текстури води є ефективним способом досягнення реалістичних візуальних ефектів в 3D моделюванні та анімації, забезпечуючи органічну, непередбачувану поверхню, яка відображає реальні характеристики води. Шум Перліна дозволяє створювати водні поверхні,

що виглядають природньо, з варіаціями, схожими на натуральні хвилі та рух води.

Анімація води з використанням шуму Перліна може включати ряд технік:

1. Зміщення координат. Шум Перліна часто використовується з динамічним зміщенням координат, що дозволяє "пересувати" шумовий патерн, створюючи ілюзію потоку води. Це можна зробити шляхом постійного збільшення або зменшення значень координат шуму на невелику величину за кожен кадр анімації.

2. Зміна частоти та амплітуди. Для додання більше реалізму можна змінювати частоту та амплітуду шуму Перліна. Зменшення частоти може імітувати більші хвилі, тоді як збільшення частоти допомагає створити враження більш дрібних хвиль або водної турбулентності.

Візуалізація води з шумом Перліна може включати:

1. Дисплейсмент-мапинг [22]. шум Перліна часто застосовують як дисплейсмент-мапу для модифікації висоти вершин водної поверхні в 3D середовищі. Це дозволяє створити візуальні хвилі, які фізично змінюють форму поверхні води.

2. Рендеринг освітлення та відблисків [18, 19]. правильне освітлення та відображення відблисків важливі для досягнення реалістичного вигляду води. Реалістичні водні поверхні часто відображають світло і тіні, що робить їх більш тривимірними і візуально привабливими.

Застосування шуму Перліна у візуалізації води не тільки підвищує реалізм, але й дає розробникам і дизайнерам засіб для створення більш динамічних та захоплюючих візуальних ефектів у різноманітних проектах, від відеоігор до анімаційних фільмів.

Алгоритм генерації шуму Перліна включає кілька кроків, які спільно створюють випадкову, але згладжену текстуру. Ось детальний поетапний опис алгоритму з використанням формул:

Крок 1: Ініціалізація градієнтів

Спершу потрібно створити градієнтний вектор для кожної точки у сітці. Ці вектори можуть бути випадково обрані з набору попередньо визначених векторів, наприклад, векторів, які вказують у різних напрямках на одиничному колі або сфері.

Крок 2: Визначення осередків

Для кожної точки x = (x, y), яку хочемо оцінити, ідентифікуємо чотири найближчі осередки сітки (верхній лівий, верхній правий, нижній лівий, нижній правий), які називаємо g00, g10, g01, g11.

Крок 3: Обчислення векторів відстаней

Обчислюємо вектори від кожного осередку сітки до точки x, наприклад, d00 = x − g00 та аналогічно для інших трьох осередків.

Крок 4: Скалярні добутки

Обчислюємо скалярні добутки між градієнтними векторами кожного осередку та відповідними векторами відстаней:

$$S = g00 \cdot d00,\ T = g10 \cdot d10\ ,U = g01 \cdot d01,\ V = g11 \cdot d11$$

Крок 5: Інтерполяція

Інтерполюємо між отриманими скалярними добутками, використовуючи згладжувальну функцію f(t), наприклад, $f(t)=6t^5-15t^4+10t^3$ яка забезпечує плавний перехід між значеннями:

1. Інтерполюємо в горизонтальному напрямку: $Sx = S + f(x)(T-S)$, $Ux = U + f(x)(V-U)$.
2. Інтерполюємо в вертикальному напрямку: $P = Sx + f(y)(Ux-Sx)$.
3. Де x і y — це локальні координати точки x відносно найближчого лівого нижнього осередку.

Крок 6: Повторення та Накладання

Для досягнення більшої деталізації та реалістичності текстури можна використовувати метод повторення та накладання декількох шарів шуму Перліна, відомий як фрактальний шум або шум Перліна октав:

Накладання октав: Шум Перліна генерується кілька разів на різних рівнях частоти та амплітуди. Кожен новий шар (октава) має вдвічі вищу частоту та вдвічі нижчу амплітуду. Результати кожної октави додаються разом, що забезпечує більш складну та деталізовану текстуру.

Формула октав: Кожна октава розраховується як:

$Pi = Pi - 1 + PerlinNoise(x2^i, y2^i)(1/2^i)$ попередній розрахунок шуму, i — номер октави, x і y — координати точки.

Крок 7: Нормалізація

Оскільки сума шумів різних октав може виходити за межі звичайного діапазону значень, результат часто потрібно нормалізувати, щоб він лежав у певному відповідному діапазоні, наприклад, від 0 до 1 або від -1 до 1:

Формула нормалізації:

$$P_{final} = P - Min(P)/(Max(P) - Min(P)),$$

де P — це кінцевий шум після додавання всіх октав, а Min(P), Max(P) — мінімальне та максимальне значення, які були отримані в ході генерації.

Шум Перліна може застосовуватися у різноманітних сценаріях, таких як створення текстур (вода, хмари, земля), моделювання теренів, генерація випадкових ландшафтів у відеоіграх, і т.д. Залежно від конкретного застосування, шум може бути додатково модифікований або поєднаний з іншими ефектами.

Приклад зображення сформованого з використанням шуму Перліна зображено на рисунку 3.6.

Рисунок 3.6 – Приклад зображення води, сформованого з використанням шуму Перліна

3.3 Алгоритм формування диму з використанням шуму Перліна

Алгоритм формування диму з використанням шуму Перліна – це спосіб створення реалістичного вигляду диму в комп'ютерній графіці.

Розроблений в БДР алгоритм використовує шум Перліна для створення різноманітних випадкових структур, які подібні до форми і руху диму. Шум Перліна – це вид шуму, який використовується для створення неперіодичних, нерегулярних паттернів, які при цьому виглядають природно.

Основний алгоритм формування диму зазвичай виглядає приблизно так:

1. Генерація шуму Перліна: спочатку створюється шум Перліна в тривимірному просторі. Це може бути зроблено за допомогою генерації випадкових значень на сітці або за допомогою градієнтів.

2. Фільтрація шуму: отриманий шум може бути підданий фільтрації для створення більш гладкої форми, яка більше нагадуватиме рух диму. Це може

включати в себе використання фільтрів, які розгладжують або розмивають різні частини шуму.

3. Приведення до відповідного масштабу: отримані значення шуму можуть бути масштабовані таким чином, щоб вони були відповідні для відображення як дим або хмара.

4. Анімація руху: для створення ефекту руху диму, значення шуму можуть бути анімовані, змінюючи їх в часі. Це може включати в себе зсув частоти шуму, зміну амплітуди або зміну їх положення в просторі.

5. Візуалізація: остаточні значення шуму візуалізуються як текстура або модель, що представляє дим.

Цей алгоритм може бути розширений або модифікований залежно від конкретних потреб індивідуального проєкту.

Покроковий алгоритм з формулами для формування диму з використанням шуму Перліна:

1. Генерація шуму Перліна:

Для кожної точки (x, y, z) в тривимірному просторі генеруємо випадковий градієнт gx , gy, gz.

Розраховуємо градієнтний шум для кожної точки за допомогою функції шуму Перліна noise(x, y, z). Використовуючи інтерполяцію, знаходимо значення шуму для кожної точки сітки.

2. Фільтрація шуму:

Застосовуємо фільтри, такі як фільтр Гауса або медіанний фільтр, для згладжування значень шуму.

3. Масштабування значень шуму:

Налаштовуємо масштаб значень шуму, використовуючи функцію масштабування scale(noise), щоб вони були відповідні для сцени.

4. Анімація руху:

Модифікуємо значення шуму згідно з часом t, використовуючи функцію animate(noise,t), щоб створити ефект руху диму.

Візуалізуємо значення шуму як текстуру або модель, що представляє дим.

Нижче наведено загальні формули, які були використані:

- Функція шуму Перліна: noise(x, y, z) = Perlin(x, y, z).

- Функція масштабування: scale(noise) = noise×scale_factor.

- Функція анімації: animate(noise, t) = noise + animation_factor×t.

Де Perlin(x, y, z) – функція шуму Перліна, scale_factorscale_factor – коефіцієнт масштабування, animation factoranimation_factor – коефіцієнт анімації, t – час.

3.4 Висновки

У розділі було запропоновано новий алгоритм формування шуму Перліна. Розроблено програму на мові програмування C++, яка за допомогою користувацьких функцій та модуля Perline-Noise дозволяє створювати тривимірний образ шуму Перліна, візуалізація якого залежить від початкових умов та аргументів функцій. Розроблено алгоритми формування зображень води та диму на основі шуму Перліна.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ МЕТОДІВ ТЕКСТУРУВАННЯ У СИСТЕМАХ КОМП'ЮТЕРНОЇ ГРАФІКИ

4.1 Основні функції програмного модуля для текстурування

У бакалаврськый дипломній роботі розроблено програмний модуль для текстурування тривимірних графічних об'єктів та їх порівняння. Розроблений програмний модуль працює лише з низькополігональними простими тривимірними об'єктами, які розміщені на невеликій відстані від спостерігача, для кращого відтворення перспективи при текстуруванні.

Основними функціями програмного модуля є:

1. Введення вихідних текстур та формування їх архіву.

2. Вибір простого низькополігонального об'єкту з списку доступних. В програмі закладено декілька об'єктів, для яких уже видалено невидимі поверхні.

3. Текстурування згідно з вибраним методом.

4. Зміна положення тривимірного об'єкта у просторі.

5. Визначення загальної кількості тактів роботи мікропроцесора для базових операцій вибраного методу текстурування.

У рамках бакалаврської роботи було створено програмний модуль для накладання текстур на 3D графічні об'єкти та проведено їх аналіз. Цей модуль адаптовано для роботи з простими об'єктами з невеликою кількістю полігонів, розташованими на оптимальній відстані від користувача, що сприяє кращій реалізації перспективи при текстуруванні.

Основні можливості цього програмного модуля включають:

- завантаження існуючих текстур та створення їх колекції;

- вибір простого об'єкта з низькою кількістю полігонів із переліку вже підготовлених об'єктів, з яких видалено невидимі частини;

- накладання текстур за допомогою обраної методики;

- зміну позиції 3D об'єкта у тривимірному просторі;

- корекцію точки перспективи;

- розрахунок загального часу обробки мікропроцесором основних операцій обраного методу текстурування.

У програмі впроваджено значно спрощений графічний конвеєр порівняно з тими, що використовуються в системах комп'ютерної графіки. Він охоплює етап геометричних перетворень, які включають афінні та перспективні перетворення для переміщення об'єктів у просторі.

Графічний конвеєр включає етап геометричних перетворень, де над об’єктом виконуються афінні та видові перетворення, що дозволяє здійснювати переміщення об’єкта в просторі, Фаза рендерингу в створеному програмному забезпеченні є досить простою і не включає процес видалення невидимих елементів за допомогою z-буферу, адже передбачається, що такі елементи не будуть показані. Програмний компонент дозволяє інтегрувати додаткові методи текстурування та розширити функціональність інтерфейсу для користувачів, з метою вивчення як наявних, так і новітніх стратегій текстурування 3D графіки. Розроблений графічний інтерфейс містить п’ять ключових розділів (див. рисунок 4.1).

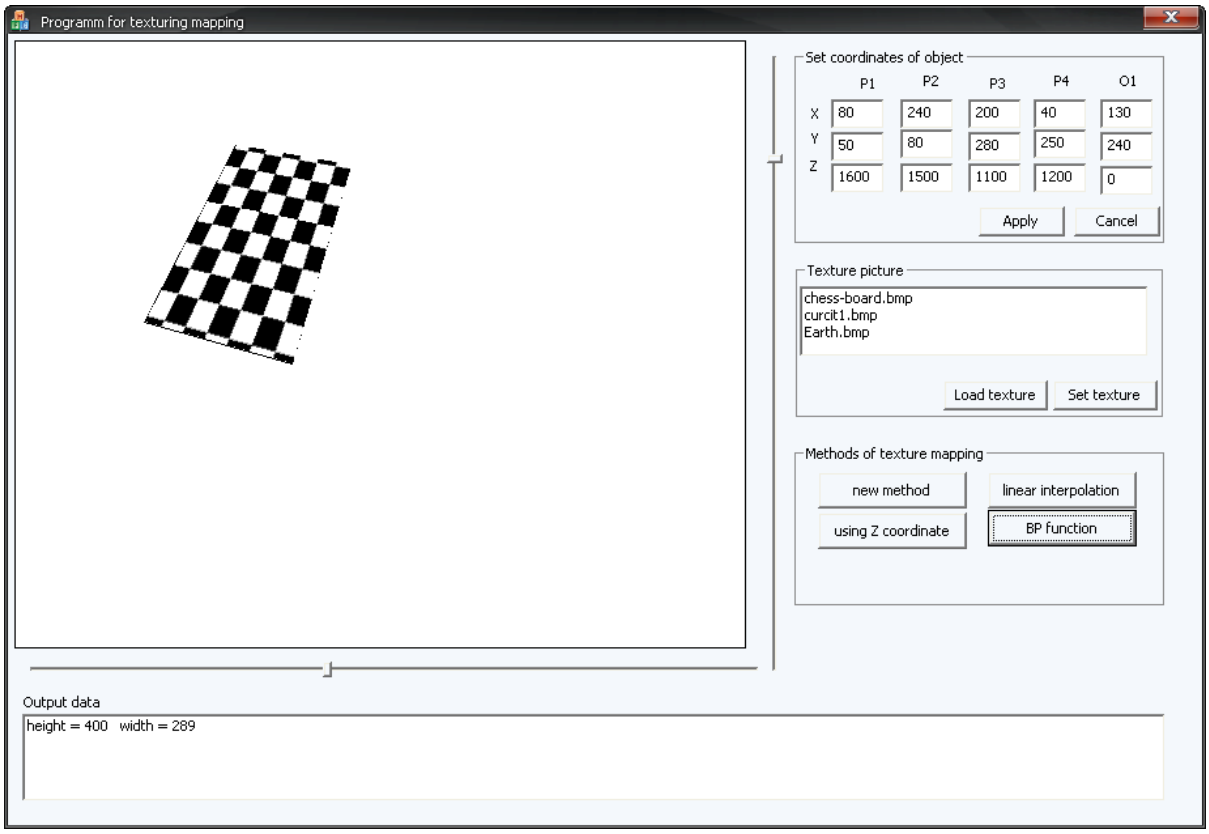


Рисунок 4.1 – Інтерфейс користувача програмного модуля

Перша секція включає в себе ряд полів введення CEdit, які дозволяють користувачеві адаптувати координати об'єкта та точки перспективи в тривимірному просторі, як показано на рисунку 4.2. В даний момент програмний компонент підтримує відображення тільки однієї площини в 3D просторі. Змінити координати вершин можливо за допомогою визначеного набору полів введення.

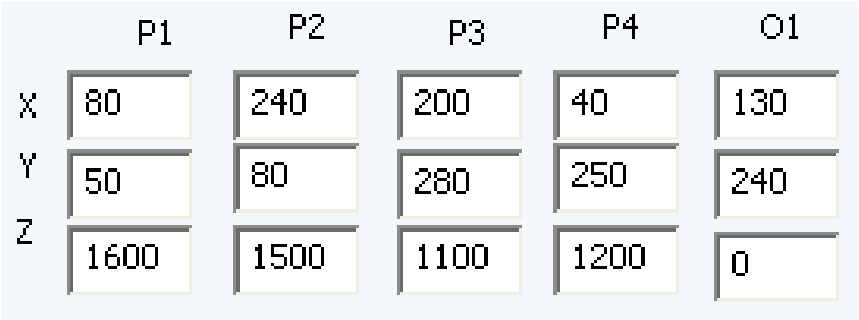


Рисунок 4.2 – Поля редагування для зміни параметрів розміщення об’єкта

Друга частина інтерфейсу користувача включає набір кнопок для керування процесом текстурування (див. рис. 4.3).

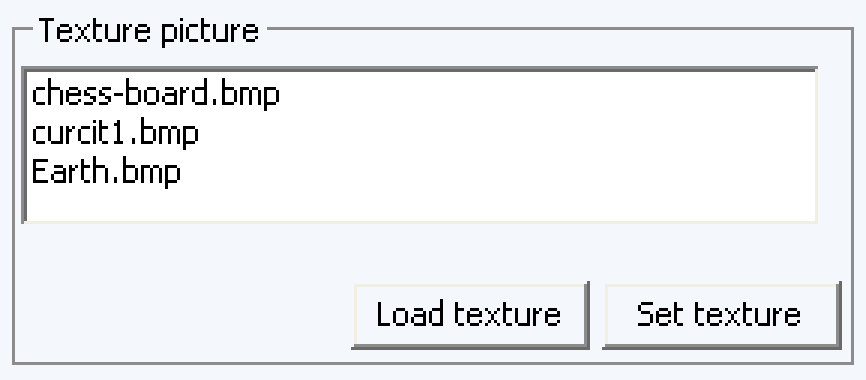


Рис. 4.3. Поля керування процесом текстурування

Вибір текстури для накладання її на тривимірну площину здійснюється з використанням кнопки “Load texture”. З’являється діалогове вікно, де слід обрати *.BMP файл (див. рис. 4.4).

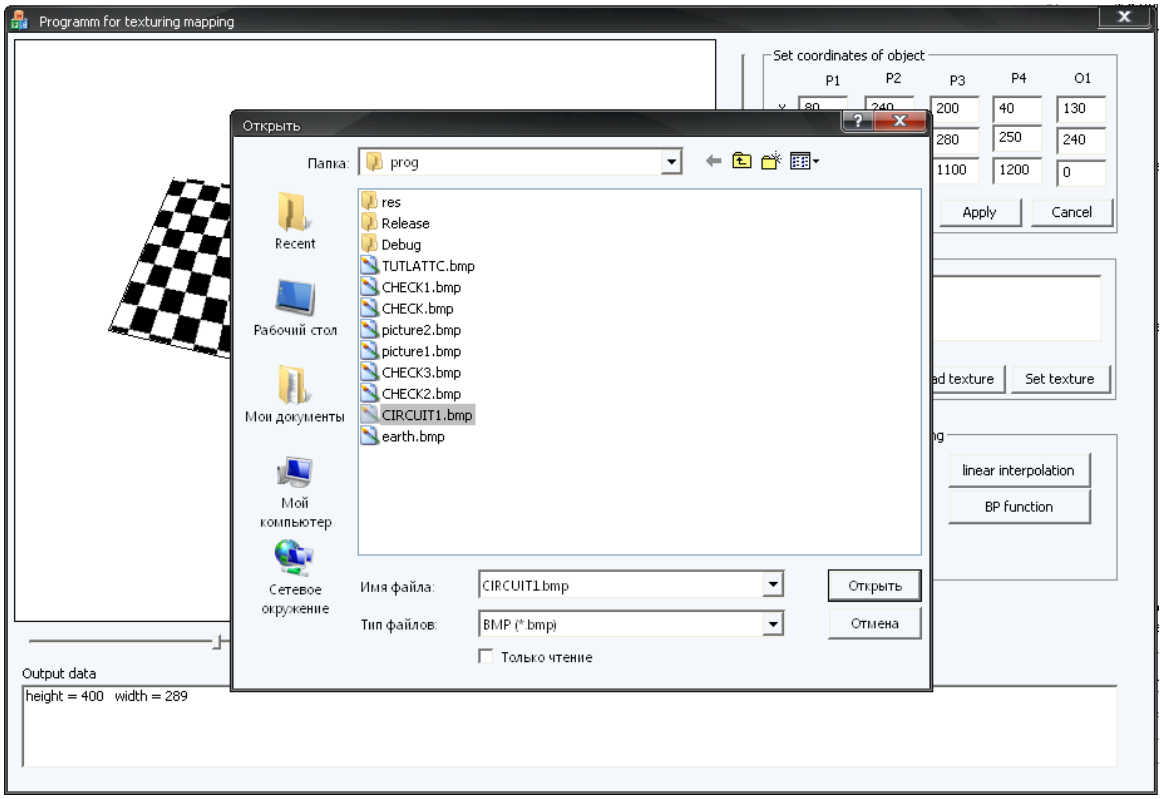


Рисунок 4.4 – Вибір текстури для накладання на тривимірну площину

Коли користувач натискає кнопку "Відкрити", відбувається завантаження текстури до переліку доступних текстур, дозволяючи її використовувати для накладення на 3D-поверхню. Кнопка "Set texture" призначена для призначення обраної текстури як активної, щоб її можна було застосовувати в обраному способі текстурування. Третій сегмент інтерфейсу користувача створено для показу створених 3D-об'єктів (див. рис. 4.5). Частина інтерфейсу номер чотири показує загальну кількість циклів роботи мікропроцесора, необхідних для базових операцій у використовуваному способі текстурування. П’ятий розділ інтерфейсу дозволяє вибрати метод текстурування для 3D-об'єктів (див. рис. 4.6).

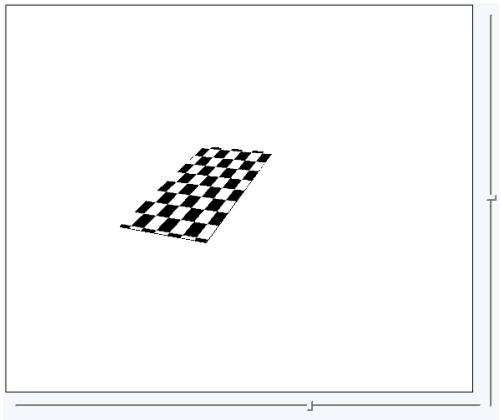


Рисунок 4.5 – Поле для виводу згенерованих тривимірних об’єктів

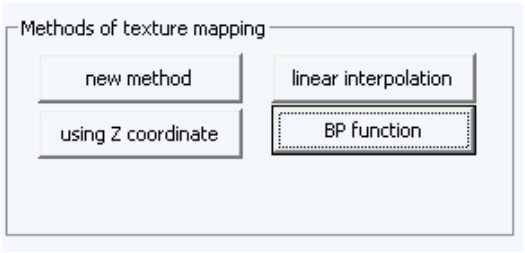


Рисунок 4.6 – Поля для вибору методу текстурування

Додатковий компонент був створений з метою надання текстур тривимірним моделям, включаючи врахування відблисків на поверхнях. Він виконує наступні ключові завдання: а) імпорт 3D-моделей з файлів у форматі *.3DS; б) завантаження текстурних файлів у форматі *.JPG; в) використання специфічної моделі освітлення для обраного способу текстурування, яка враховує z-координату при проекції моделі. Цей компонент побудований на основі бібліотеки idx3d, що надає фундаментальні інструменти для роботи з 3D-моделями і тривимірною геометрією. Структура компонента представлена у вигляді об'єктно-орієнтованої моделі на рисунку 4.7. Програма містить графічний конвеєр, що охоплює етапи геометричних перетворень для афінних та проекційних змін об'єктів, дозволяючи їх переміщення у просторі, а також етап рендерингу. На останньому відбувається видалення невидимих елементів за допомогою z-буфера та застосування текстурування згідно обраної методики. Розроблений компонент дозволяє інтеграцію нових моделей освітлення та технік текстурування, а також розширення користувацького інтерфейсу для вивчення

як наявних, так і інноваційних стратегій текстурування 3D-об'єктів. Інтерфейс програми розділено на три функціональні блоки (див. рис. 4.8). Перший блок інтерфейсу включає набір елементів керування процесом текстурування (див. рис. 4.9). Для вибору моделей, що будуть текстуровані, використовується кнопка “Вибрати об’єкт…”, яка відкриває діалогове вікно для вибору необхідного файлу у форматі *.3DS. Кнопка “Пауза” тимчасово зупиняє динамічне відображення моделей.

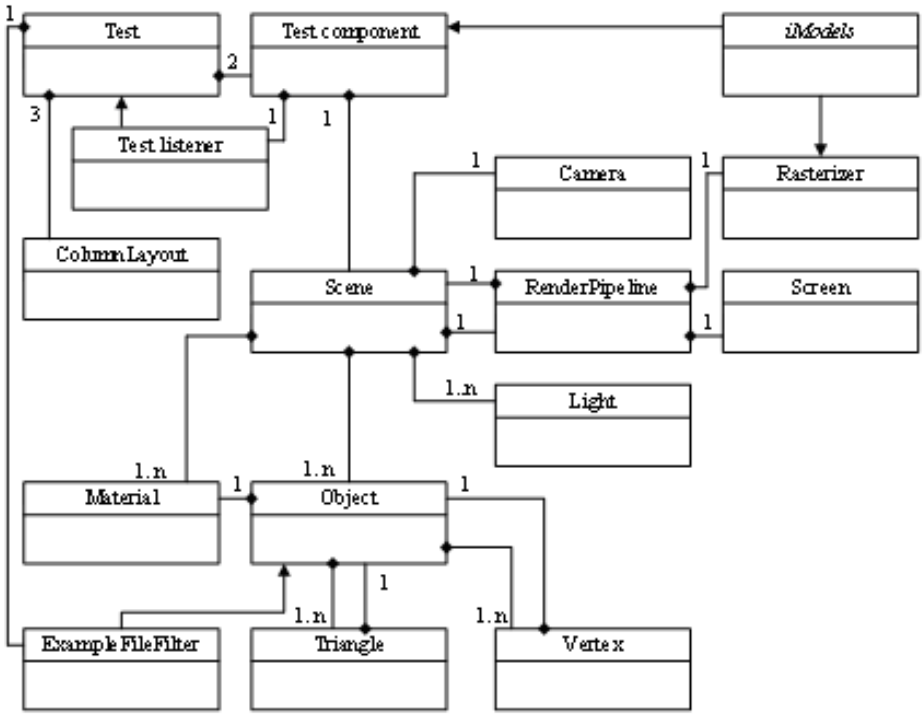


Рисунок 4.7 – Об’єктно-орієнтована модель програмного модуля

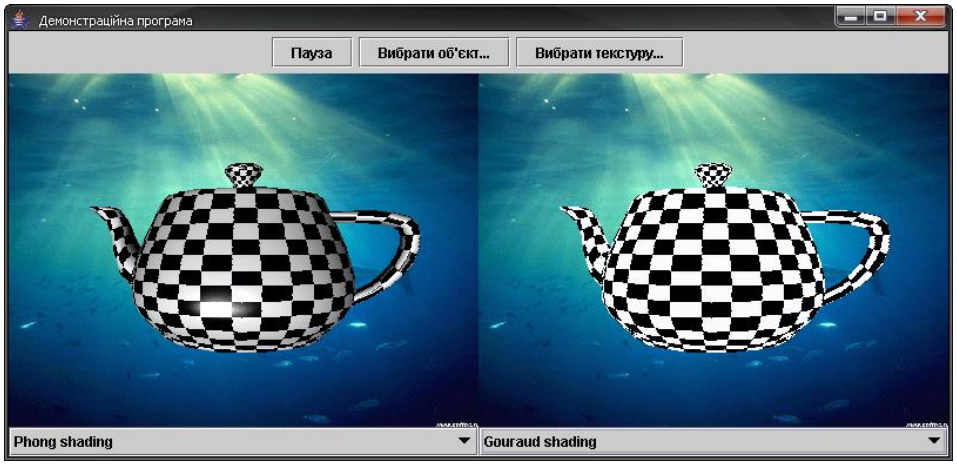


Рисунок 4.8 – Інтерфейс користувача програмного модуля

Друга частина інтерфейсу користувача призначена для виводу згенерованих тривимірних об’єктів (див. рис. 4.10).

Рисунок 4.9 – Кнопки для керування процесом текстурування

Рисунок 4.10 – Поля для вибору методу текстурування тривимірного об’єкта

У програмі реалізовано багатопотокову модель класів, яку схематично зображено на рисунку 4.11.

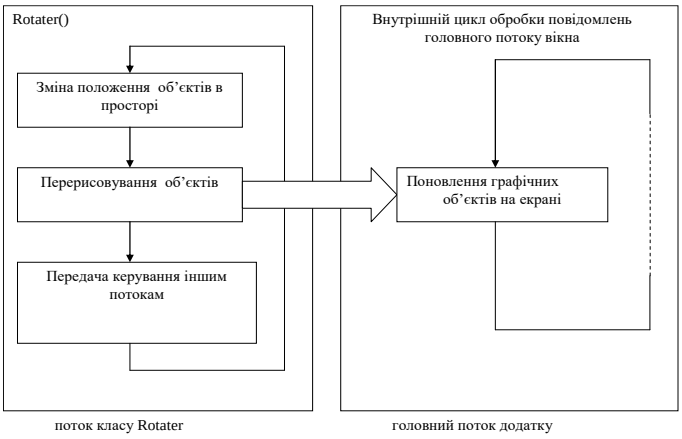


Рисунок 4.11 – Багатопотокова модель, реалізована в програмному модулі

4.2 Системні вимоги

Визначення системних вимог до програми є ключовим етапом у розробці програмного забезпечення. Це процес документування функціональних і нефункціональних вимог, які повинна виконувати система, щоб задовольнити потреби користувачів і досягти поставлених цілей.

Для моделювання та проведення тестових перевірок методів заповнення кольором використовувався ПК з такою конфігурацією: ОС Windows 7; CPU –

Intel Pentium 2400; відеоадаптер – інтегрований Intel(R) 82865G Graphics Controller, що є частиною материнської плати; RAM – 512 Mb. Розроблений програмний компонент надає можливість проведення моделювання та тестування різноманітних технік текстурування та моделей освітлення.

4.3 Висновки

У четвертому розділі було розглянуто та детально описано основні функціональні можливості розробленого програмного модуля для текстурування тривимірних графічних об’єктів. Програмний модуль, створений у рамках бакалаврської дипломної роботи, орієнтований на роботу з простими низькополігональними об'єктами, що знаходяться на оптимальній відстані від спостерігача. Завдяки цьому забезпечується краще відтворення перспективи при текстуруванні. Також було описано системні вимоги до застосунку.

ВИСНОВКИ

У бакалаврській дипломній роботі було розроблено методи та програмні засоби для текстурування з використанням сучасних технік та процедур. Проведено детальний аналіз існуючих методів текстурування, зокрема процедурного, фотографічного та проекційного підходів. Обґрунтовано необхідність удосконалення методів текстурування для підвищення реалістичності візуалізацій у тривимірних моделях.

Розроблено модифіковану версію методу Хекберта, яка зменшує кількість арифметичних операцій під час розрахунку текстурних координат, що дозволило підвищити ефективність процесу текстурування на 26%. Запропоновані нові ітераційні рівняння для точного визначення параметрів еліпса у проекціях пікселів на текстурну поверхню забезпечили зростання продуктивності на 18%.

Також було створено алгоритм текстурування з урахуванням координати глибини текселів, що значно підвищує реалістичність графічних зображень. Особливу увагу приділено розробці алгоритму формування диму з використанням шуму Перліна. Цей алгоритм дозволяє створювати реалістичні ефекти диму, використовуючи генерацію, фільтрацію та анімацію шуму Перліна.

Розроблені підходи можуть бути використані у різних галузях, таких як архітектурна візуалізація, анімація, відеоігри та віртуальна реальність, що відкриває нові можливості для подальших досліджень та вдосконалення в області комп'ютерної графіки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] О. А Іваха, О. В. Романюк, та О.Н Романюк, “Графічні процесори у вирішенні сучасних ІТ-задач“, *Електронні інформаційні ресурси: створення, використання, доступ:3 бірник матеріалів Міжнародної науково-практичної Інтернет конференції* 9-10 листопада. – Суми/Вінниця : НІКО/ВНТУ, 2020. – С. 90-93.

[2] О. Н. РОМАНЮК., О. А., ІВАХА, та ДУДНИК О.О, “Аналіз шейдерів“ , *Стан, досягнення та перспективи інформаційних систем і технологій / Матеріали XXI Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів*. Одеса, 22-23 квітня 2021 р. - Одеса, Видавництво ОНАХТ, 2021 р. – с.223-234

[3] О. А. Іваха О. Н. Романюк, та І. В. Хом’юк, “Аналіз найпотужніших відеокарт“. *Матеріали молодіжної науково-практичної інтернет-конференції студентів аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2020)» : збірник матеріалів*. – Вінниця: ВНТУ, 2021. – 2 с.

[4] О. А Іваха, та О. Н. Романюк Аналіз методів текстурування. *Матеріали LIII науково-технічної конференції підрозділів ВНТУ, Вінниця*, 20-22 березня 2024 р. Електрон. текст. дані. 2024. URI: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2024/paper/view/20153.

[5] Naty Hoffman, Tomas Akenine-Moller, Real-Time Rendering. Publisher : A K Peters/CRC Press; 4th edition (August 6, 2018), 1178 pages.

[6] P. Godse, Dr. D. A. Godse. Computer Graphics and Multimedia Publisher : 9789333223362 (November 3, 2020), 686 pages.

[7] R. Stuart Ferguson. Practical Algorithms for 3D Computer Graphics, Publisher : A K Peters/CRC Press; 2nd edition (October 6, 2019), 520 pages.

[8] М.Ф. Пічугін, І.О. Канкін та В.В. Воротніков, *Комп'ютерна графіка. Навчальний посібник*. Центр навчальної літератури. 2019. 346 с.

[9] О. Н Романюк, О. В. Романюк та Р. Ю, Чехместрук, *Комп`ютерна графіка : навчальний посібник* – Вінниця : ВНТУ, 2022. – 141 с.

[10] N. Tatarchuk, “Evolution of Programmable Models for Graphics Engines”, *High Performance Graphics*, 2017.

[11] І. В. Ільїна, та О. В. Біжко, “Аналіз особливостей візуалізації тривимірних об'єктів”, *Системи управління, навігації та зв'язку,* вип. 2, с. 88-92, 2019.

[12] О. Н. Романюк, та О. О. Дудник, “Підвищення реалістичності зафарбовування тривимірних графічних об'єктів”, *Вісник ХНТУ,* № 3 (58), с. 269-272, 2016.

[13] О. Н. Романюк, та О. О. Дудник, “Анізотропна фільтрація з використанням текстурних карт вагових коефіцієнтів”, *Реєстрація, зберігання і обробка даних*, № 3, с. 34-41, 2017.

[14] О. Н. Романюк та О. О. Дудник, “Анізотропна фільтрація текстур з використанням методів кешування”, *Вісник Вінницького політехнічного інституту,* 2016, № 6, с. 59-64.

[15] О. О. Дудник, О. Н. та Романюк, “Аналіз методів фільтрації текстур”, *Матеріали міжнародної науково-практичної Інтернет-конференції "Молодь в технічних науках:дослідження, проблеми, перспективи (МТН-2015)*. Вінниця, 2015, [Електронний ресурс]. Доступно: http://conf.inmad.vntu.edu.ua/fm/index.php?page=materials&line=11&mat=115.

[16] О. Н. Романюк, О. О. Дудник, “Аналіз методів анізотропної фільтрації текстур”, Збірник матеріалів *Всеукраїнської науково-практичної конференції "Електронні інформаційні ресурси в освіті і науці: створення, використання, доступ*", грудень 2014 р., Вінниця, 2014., с. 3-8.

[17] О. О. Дудник, “Аналіз методів рельєфного текстурування”, *Електронні інформаційні ресурси: створення, використання, доступ: Збірник матеріалів Міжнародної науково-практичної Інтернет- конференці*ї., Вінниця, 2017.

[18] B. G. Blundell, “An Introduction to Computer Graphics and Creative 3-D Environments”, Springer – Verlag London Limited, 2008.

[19] C. Doppioslash, “The Graphics Pipeline”, Physically Based Shader Development for Unity 2017, pp. 33–42, Jul. 2017.LAAKSONEN, Jarno, et al. OpenGL rendering pipeline. 2017.

[20] S. Laine and T. Karras, “High-performance software rasterization on GPUs*” Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphic*s - HPG 11, 2011.

[21] P. S. Heckbert, “Survey of Texture Mapping*”, IEEE Computer Graphics and Applications*, vol. 6, no. 11, pp. 56–67, 1986.

[22] P. Godse, Dr. D. A. Godse. Computer Graphics and MultimediaP. Godse, Dr. D. A. Godse. Computer Graphics and Multimedia Publisher : (November 3, 2020), 686 pages.

ДОДАТКИ

Додаток А. Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., професор
_____ О. Н. Романюк
«12» березня 2024 р.

Технічне завдання
на бакалаврську дипломну роботу «Розробка методів і програмних засобів для текстурування»
за спеціальністю 121 – Інженерія програмного забезпечення

Керівник бакалаврської дипломної роботи:
_____ д.т.н., професор Романюк О. Н.
«12» березня 2024 р.
Виконав:
_____ студент гр. 4ПІ-20б О. А. Іваха
«12» березня 2024 р.

Вінниця – 2024 року

1. Найменування та галузь застосування

Бакалаврська дипломна робота: «Розробка методів і програмних засобів для текстурування».

Галузь застосування – комп’ютерна графіка.

2. Підстава для розробки.

Підставою для виконання бакалаврської дипломної роботи (БДР) є індивідуальне завдання на БДР та наказ №80 від «11» березня 2024 р. ректора ВНТУ про закріплення тем БДР.

3. Мета та призначення розробки.

Метою бакалаврської дипломної роботи є підвищення продуктивності та реалістичності текстурування в системах комп’ютерної графіки.

Призначення роботи – розробка та реалізація програмного забезпечення для текстурування.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БДР:

1. Naty Hoffman, Tomas Akenine-Moller, Real-Time Rendering. Publisher A K Peters/CRC Press; 4th edition (August 6, 2018), 1178 pages.

2. P. Godse, Dr. D. A. Godse. Computer Graphics and Multimedia Publisher : 9789333223362 (November 3, 2020), 686 pages.

3. О. О. Дудник, О. Н. та Романюк, “Аналіз методів фільтрації текстур”, *Матеріали міжнародної науково-практичної Інтернет-конференції "Молодь в технічних науках:дослідження, проблеми, перспективи (МТН-2015)*. Вінниця, 2015, [Електронний ресурс]. Доступно: http://conf.inmad.vntu.edu.ua/fm/index.php?page=materials&line=11&mat=115.

5. Технічні вимоги

Вихідні дані до роботи: базовий метод Хекберта; метод генерації шуму – шум Перліна; графічний режим – TrueColor; дані для текстурування: координати вершин трикутника; вектори до джерела світла та спостерігача, вектор нормалі; середовище розробки – Visual Studio; мова програмування: C++.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

1. Пояснювальна записка до БДР.
2. Технічне завдання.
3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

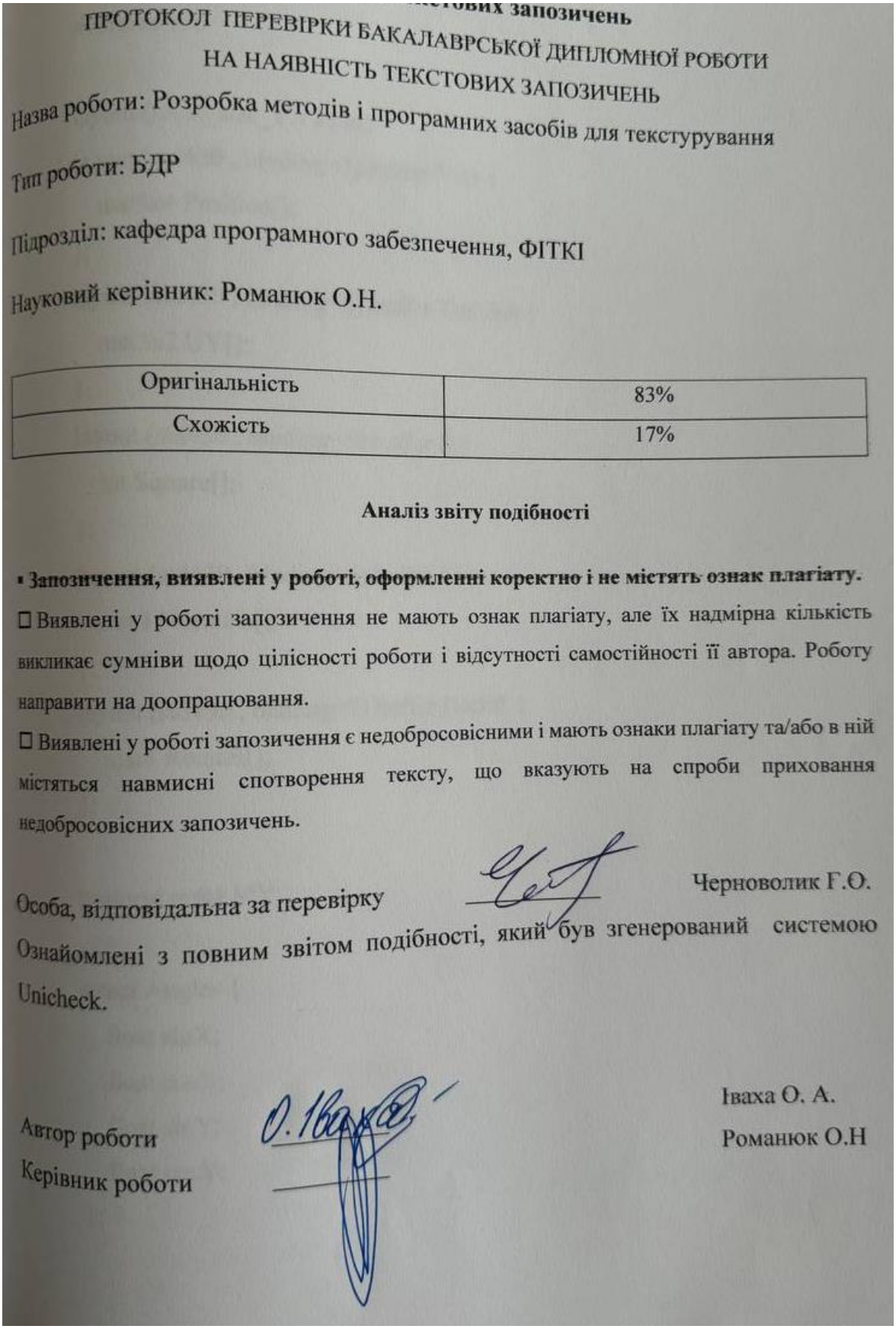
9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи
1	Аналіз методів накладання текстур	14.03.24 – 25.03.24
2	Розробка нових методів текстурувань	25.03.24 – 17.04.24
3	Розробка процедурних текстур з використанням шуму Перліна	17.04.24– 28.04.24
4	Розробка програмного забезпечення	28.04.24– 23.05.24
5	Оформлення матеріалів до захисту БДР	23.05.24– 30.05.24

10. Порядок контролю та прийняття

Виконання етапів бакалаврської дипломної роботи контролюється керівником згідно з графіком виконання роботи. Прийняття бакалаврської дипломної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б. Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень



ПРОТОКОЛ ПЕРЕВІРКИ БАКАЛАВРСЬКОЇ ДИПЛОМНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Розробка методів і програмних засобів для текстурування

Тип роботи: БДР

Підрозділ: кафедра програмного забезпечення, ФІТКІ

Науковий керівник: Романюк О.Н.

Оригінальність	83%
Схожість	17%

Аналіз звіту подібності

▪ **Запозичення, виявлені у роботі, оформленні коректно і не містять ознак плагіату.**

☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цілісності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховання недобросовісних запозичень.

Особа, відповідальна за перевірку Черноволик Г.О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck.

Автор роботи Іваха О. А.

Керівник роботи Романюк О.Н

Додаток В. Лістинг програми

```
#version 430

layout (local_size_x = 1) in;
layout (std430 , binding=3) buffer Vert {
    mat3x4 Position[];
};
layout (std430 , binding=4) buffer TexVert {
    mat3x2 UV[];
};
layout (std430 , binding=5) buffer S {
    int Square[];
};
layout (std430 , binding=6) buffer Norm {
    vec4 Normals[];
};
layout (std430 , binding=7) buffer BackR {
    mat4 Rotates[];
};

uniform mat4 MV;

struct Angles {
    float sinX;
    float cosX;
    float sinY;
    float cosY;
};
```

```
vec3 getPlaneNormal(vec4 v1, vec4 v2, vec4 v3){
    float v1x = v2.x-v1.x,
          v1y = v2.y-v1.y,
          v1z = v2.z-v1.z,
          v2x = v3.x-v1.x,
          v2y = v3.y-v1.y,
          v2z = v3.z-v1.z;
    return vec3(v1y*v2z-v1z*v2y, v1z*v2x-v1x*v2z, v1x*v2y-v1y*v2x);
}

Angles getAngles(vec3 normal){
    float d = sqrt(normal.y*normal.y + normal.z*normal.z);
    return Angles(normal.y/d, normal.z/d, normal.x, d);
}

vec3 mul(vec3 v, mat4 m){
    vec4 r = m * vec4(v,   0);
    if (r.w != 0.0)
        return vec3(r.x/r.w, r.y/r.w, r.z/r.w);
    return r.xyz;
}

void turnPoygon(uint idx){
    vec3 normal = getPlaneNormal(Position[idx][0], Position[idx][1],
Position[idx][2]);
    normal = normalize(normal);
    Normals[idx] = vec4(normal, 0.0);
    Angles ang = getAngles(normal);
    Normals[idx] = vec4(ang.sinX, ang.cosX, ang.sinY, ang.cosY);
    mat4 mTX = mat4(
```

```
        vec4(1,         0,        0, 0),
        vec4(0,  ang.cosX, ang.sinX, 0),
        vec4(0, -ang.sinX, ang.cosX, 0),
        vec4(0,         0,        0, 1)
    );
    mat4 mTY = mat4(
        vec4( ang.cosY,         0,        ang.sinY, 0),
        vec4(        0,         1,                0, 0),
        vec4(-ang.sinY,         0,        ang.cosY, 0),
        vec4(        0,         0,                0, 1)
    );
    mat4 mT = mTY * mTX;
    Position[idx] = mT * Position[idx];
    mTX = mat4(
        vec4(1,         0,        0, 0),
        vec4(0,  ang.cosX,-ang.sinX, 0),
        vec4(0,  ang.sinX, ang.cosX, 0),
        vec4(0,         0,        0, 1)
    );
    mTY = mat4(
        vec4( ang.cosY,         0,       -ang.sinY, 0),
        vec4(        0,         1,                0, 0),
        vec4( ang.sinY,         0,        ang.cosY, 0),
        vec4(        0,         0,                0, 1)
    );
    Rotates[idx] = mTX * mTY;
}

void swapVertexes(uint idx){
    vec4 buff;
```

```
    vec2 tbuf;
    if (Position[idx][1].y < Position[idx][0].y) {
        buff = Position[idx][1];
        Position[idx][1] = Position[idx][0];
        Position[idx][0] = buff;

        tbuf = UV[idx][1];
        UV[idx][1] = UV[idx][0];
        UV[idx][0] = tbuf;
    }
    if (Position[idx][2].y < Position[idx][0].y) {
        buff = Position[idx][2];
        Position[idx][2] = Position[idx][0];
        Position[idx][0] = buff;

        tbuf = UV[idx][2];
        UV[idx][2] = UV[idx][0];
        UV[idx][0] = tbuf;
    }
    if (Position[idx][1].y > Position[idx][2].y) {
        buff = Position[idx][2];
        Position[idx][2] = Position[idx][1];
        Position[idx][1] = buff;

        tbuf = UV[idx][2];
        UV[idx][2] = UV[idx][1];
        UV[idx][1] = tbuf;
    }
}
```

```
void roundPositions(uint idx) {
    for (uint i = 0; i < 3; ++i){
        for (uint j = 0; j < 3 ; j ++){
            Position[idx][i][j] = round(Position[idx][i][j]);
        }
    }
}

void getSquare2(uint idx) {
    float dx13 = 0, dx12 = 0, dx23 = 0;
    float x1 = round(Position[idx][0].x);
    float x2 = round(Position[idx][1].x);
    float x3 = round(Position[idx][2].x);

    float y1 = round(Position[idx][0].y);
    float y2 = round(Position[idx][1].y);
    float y3 = round(Position[idx][2].y);
    float b;
    if (y2 < y1) {
        b = y1; y1 = y2; y2 = b;
        b = x1; x1 = x2; x2 = b;
    }
    if (y3 < y1) {
        b = y1; y1 = y3; y3 = b;
        b = x1; x1 = x3; x3 = b;
    }
    if (y2 > y3) {
        b = y2; y2 = y3; y3 = b;
        b = x2; x2 = x3; x3 = b;
    }
```

```
if (y3 != y1) {
    dx13  = x3 - x1;
    dx13 /= y3 - y1;
}
if (y2 != y1) {
    dx12  = x2 - x1;
    dx12 /= y2 - y1;
}
if (y3 != y2) {
    dx23  = x3 - x2;
    dx23 /= y3 - y2;
}

float wx1 = x1;
float wx2 = wx1;

float _dx13 = dx13;

if (dx13 > dx12) {
    float b = dx13;
    dx13 = dx12;
    dx12 = b;
}
int square = 0;
for (int i = int(y1); i < y2; i++){
    square += int(round(wx2) - round(wx1));

    wx1 += dx13;
    wx2 += dx12;
}
```

```
    if (y1 == y2){
        wx1 = x1;
        wx2 = x2;
    }

    if (_dx13 < dx23) {
         float b = _dx13;
         _dx13 = dx23;
         dx23 = b;
    }

    for (int i = int(y2); i <= y3; i++){
        square += int(round(wx2) - round(wx1));

        wx1 += _dx13;
        wx2 += dx23;
    }
    Square[idx] = square;
}

void getSquare(uint idx){
    float dx13 = 0, dx12 = 0, dx23 = 0;

    if (Position[idx][2].y != Position[idx][0].y) {
        dx13  = Position[idx][2].x - Position[idx][0].x;
        dx13 /= Position[idx][2].y - Position[idx][0].y;
    }
    if (Position[idx][1].y != Position[idx][0].y) {
        dx12  = Position[idx][1].x - Position[idx][0].x;
```

```
    dx12 /= Position[idx][1].y - Position[idx][0].y;
}
if (Position[idx][2].y != Position[idx][1].y) {
    dx23  = Position[idx][2].x - Position[idx][1].x;
    dx23 /= Position[idx][2].y - Position[idx][1].y;
}

float wx1 = Position[idx][0].x;
float wx2 = wx1;

float _dx13 = dx13;

if (dx13 > dx12) {
    float b = dx13;
    dx13 = dx12;
    dx12 = b;
}

int square = 0;
for (int i = int(Position[idx][0].y); i < Position[idx][1].y; i++){
    square += int(round(wx2) - round(wx1));

    wx1 += dx13;
    wx2 += dx12;
}

if (Position[idx][0].y == Position[idx][1].y){
    wx1 = Position[idx][0].x;
    wx2 = Position[idx][1].x;
}
```

```
    if (_dx13 < dx23) {
        float b = _dx13;
        _dx13 = dx23;
        dx23 = b;
    }

    for (int i = int(Position[idx][1].y); i <= Position[idx][2].y; i++){
        square += int(round(wx2) - round(wx1));

        wx1 += _dx13;
        wx2 += dx23;
    }
    Square[idx] = square;
}

void main() {
    uint idx = gl_GlobalInvocationID.x;

    turnPoygon(idx);
    getSquare2(idx);
}
```

Шейдер растеризації

```
#version 430

layout (local_size_x = 1) in;

layout (std430 , binding=3) buffer Vert {
    mat3x4 Position[];
};
```

```
layout (std430 , binding=5) readonly buffer S {
   uint Square[];
};

layout (std430 , binding=8) buffer Pix {
   vec4 Pixel[];
};

layout (std430 , binding=9) buffer PIndex {
   uint PolyIndex[];
};

layout (std430 , binding=10) buffer Offs {
   uint Offset[];
};

void rasterize2(uint idx) {
   uint offset = Offset[idx];
   float dx13 = 0, dx12 = 0, dx23 = 0;
   float x1 = round(Position[idx][0].x);
   float x2 = round(Position[idx][1].x);
   float x3 = round(Position[idx][2].x);

   float y1 = round(Position[idx][0].y);
   float y2 = round(Position[idx][1].y);
   float y3 = round(Position[idx][2].y);
   float b;
   if (y2 < y1) {
      b = y1; y1 = y2; y2 = b;
```

```
    b = x1; x1 = x2; x2 = b;
}
if (y3 < y1) {
    b = y1; y1 = y3; y3 = b;
    b = x1; x1 = x3; x3 = b;
}
if (y2 > y3) {
    b = y2; y2 = y3; y3 = b;
    b = x2; x2 = x3; x3 = b;
}
if (y3 != y1) {
    dx13  = x3 - x1;
    dx13 /= y3 - y1;
}
if (y2 != y1) {
    dx12  = x2 - x1;
    dx12 /= y2 - y1;
}
if (y3 != y2) {
    dx23  = x3 - x2;
    dx23 /= y3 - y2;
}

float wx1 = x1;
float wx2 = wx1;

float _dx13 = dx13;

if (dx13 > dx12) {
    float b = dx13;
```

```
    dx13 = dx12;
    dx12 = b;
}

for (int i = int(y1); i < y2; i++){
    for (int j = int(round(wx1)); j <= int(round(wx2)); j++){
        Pixel[offset] = vec4(j, i,  Position[idx][0].z, 1.0);
        PolyIndex[offset] = idx;
        offset++;
    }

    wx1 += dx13;
    wx2 += dx12;
}

if (y1 == y2){
    wx1 = x1;
    wx2 = x2;
}

if (_dx13 < dx23) {
     float b = _dx13;
     _dx13 = dx23;
     dx23 = b;
}

for (int i = int(y2); i <= y3; i++){
    for (int j = int(round(wx1)); j <= int(round(wx2)); j++){
        Pixel[offset] = vec4(j, i,  Position[idx][0].z, 1.0);
        PolyIndex[offset] = idx;
```

```
            offset++;
        }

        wx1 += _dx13;
        wx2 += dx23;
    }
}
void rasterize(uint idx){
    uint offset = Offset[idx];

    float dx13 = 0, dx12 = 0, dx23 = 0;
    if (Position[idx][2].y != Position[idx][0].y) {
        dx13  = Position[idx][2].x - Position[idx][0].x;
        dx13 /= Position[idx][2].y - Position[idx][0].y;
    }
    if (Position[idx][1].y != Position[idx][0].y) {
        dx12  = Position[idx][1].x - Position[idx][0].x;
        dx12 /= Position[idx][1].y - Position[idx][0].y;
    }
    if (Position[idx][2].y != Position[idx][1].y) {
        dx23  = Position[idx][2].x - Position[idx][1].x;
        dx23 /= Position[idx][2].y - Position[idx][1].y;
    }

    float wx1 = Position[idx][0].x;
    float wx2 = wx1;

    float _dx13 = dx13;

    if (dx13 > dx12) {
```

```
    float b = dx13;
    dx13 = dx12;
    dx12 = b;
}

int square = 0;
for (int i = int(Position[idx][0].y); i < Position[idx][1].y; i++){
    for (int j = int(round(wx1)); j <= int(round(wx2)); j++){
        Pixel[offset] = vec4(j, i,  Position[idx][0].z, 1.0);
        PolyIndex[offset] = idx;
        offset++;
    }
    wx1 += dx13;
    wx2 += dx12;
}

if (Position[idx][0].y == Position[idx][1].y){
    wx1 = Position[idx][0].x;
    wx2 = Position[idx][1].x;
}

if (_dx13 < dx23) {
     float b = _dx13;
     _dx13 = dx23;
     dx23 = b;
}

for (int i = int(Position[idx][1].y); i <= Position[idx][2].y; i++){
    for (int j = int(round(wx1)); j <= int(round(wx2)); j++){
        Pixel[offset] = vec4(j, i,  Position[idx][0].z, 1.0);
```

```
            PolyIndex[offset] = idx;
            offset++;
        }
        wx1 += _dx13;
        wx2 += dx23;
    }
}

void main() {
    rasterize2(gl_GlobalInvocationID.x);
}
#version 430

layout (local_size_x = 22) in;
layout (std430 , binding=3) buffer Vert {
    mat3x4 Position[];
};

layout (std430 , binding=8) buffer Pix {
    vec4 Pixel[];
};
layout (std430 , binding=9) buffer PIndex {
    uint PolyIndex[];
};

layout (std430 , binding=11) buffer Col {
    vec4 Color[];
};
layout (std430 , binding=4) buffer TexVert {
    mat3x2 UV[];
```

```
};
layout (std430 , binding=12) buffer DTV {
   vec4 DUV[];
};
layout (std430 , binding=13) buffer B {
   vec4 BR[];
};
layout (std430 , binding=0) buffer Scr {
   vec4 ScreenBuffer[];
};
uniform sampler2D Tex1;
uniform uint Square;
ivec2 Size = ivec2(700, 700);
double square(float x1, float y1, float x2, float y2, float x3, float y3) {
      return abs(((x1-x3)*(y2-y3) - (x2-x3)*(y1-y3))) / 2.0;
}

vec3 toBarycentric(vec4 decart, uint pidx) {
   mat3x4 poly = Position[pidx];
   double S =  square(poly[0].x, poly[0].y, poly[1].x, poly[1].y, poly[2].x,
poly[2].y);
   double Su = square(poly[0].x, poly[0].y, poly[1].x, poly[1].y, decart.x,
decart.y);
   double Sv = square(poly[0].x, poly[0].y, poly[2].x, poly[2].y, decart.x,
decart.y);
   double Sw = square(decart.x, decart.y, poly[1].x, poly[1].y, poly[2].x,
poly[2].y);
   double u = Su / S;
   double v = Sv / S;
   double w = 1.0 - u - v;
```

```
    return vec3(u,v,w);
}
vec2 toUV(vec4 decart, uint idx) {
    vec3 bar = toBarycentric(decart, PolyIndex[idx]);
    mat3x2 uv = UV[PolyIndex[idx]];

    float b2 = bar.x, b1 = bar.y, b0 = bar.z;
    float u = b0*uv[0].x + b1*uv[1].x + b2*uv[2].x;
    float v = b0*uv[0].y + b1*uv[1].y + b2*uv[2].y;
    return vec2(u,v);
}
void getColor(uint idx) {
    vec2 uv = toUV(Pixel[idx], idx);
    Color[idx] = texture(Tex1, uv);
}

void main(){
    uint idx = gl_GlobalInvocationID.x;
    BR[idx] = vec4(0,2147483647,0,0);
    if (idx < Size.x*Size.y) ScreenBuffer[idx] = vec4(0,0,0,0);
//    DUV[idx] = vec4(1, 1, 1, 1);
    if (idx < Square)
    getColor(gl_GlobalInvocationID.x);
}
```

Шейдер зворотнього повороту

```
#version 430

layout (local_size_x = 22) in;
layout (std430 , binding=8) buffer Pix {
    vec4 Pixel[];
```

```
};
layout (std430 , binding=11) buffer Col {
   vec4 Color[];
};
layout (std430 , binding=9) buffer PIndex {
   uint PolyIndex[];
};
layout (std430 , binding=0) buffer Scr {
   vec4 ScreenBuffer[];
};
layout (std430 , binding=1) buffer ZBf {
   vec4 DBuff[];
};
layout (std430 , binding=2) buffer S {
   uint SpinBuffer[];
};

layout (std430 , binding=7) buffer BackR {
   mat4 Rotates[];
};
layout (std430 , binding=13) buffer B {
   vec4 BR[];
};
layout (std430 , binding=6) buffer Norm {
   vec4 Normals[];
};
uniform mat4 P;
uniform uint Square;
ivec2 Size = ivec2(700, 700);
mat4x4 spaseToScreen(vec3 space) {
```

```
        vec3 fspace = vec3(floor(space.xy), space.z);
        vec2 weight = space.xy - fspace.xy;
        float one = 1.0;
        return mat4x4(
            vec4(fspace, (one-weight.x)*(one-weight.y)),
            vec4(fspace.x+1, fspace.yz, weight.x*(one-weight.y)),
            vec4(fspace.x,   fspace.y-1, fspace.z, weight.x*(one-weight.y)),
            vec4(fspace.x+1, fspace.y-1, fspace.z, weight.x*weight.y)
        );
    }

    int getBufIndex(vec2 coords) {
        return int(coords.y) * Size.x + int(coords.x);
    }
    void setColor(vec4 point, vec4 color, float z) {
        if (point.x < 0 || point.y < 0 || point.x > Size.x || point.y > Size.y) return;
        int id = getBufIndex(point.xy);
        bool written = false;
            do {
                    bool can_write = (atomicExchange(SpinBuffer[id], 1) != 1);
                    if (can_write) {
                            float old_poly = BR[id].y;
                            float new_poly =
float(PolyIndex[gl_GlobalInvocationID.x]);
                            if (z > BR[id].x) {
                                BR[id].x = z;
                                if (BR[id].y != new_poly) {
                                    BR[id].y  = new_poly;
                                    ScreenBuffer[id] = vec4(color.xyz*point.w, point.w);
                                }
```

```
                        }
                        if (new_poly == old_poly){
                            vec4 buff = ScreenBuffer[id];
                            ScreenBuffer[id] = vec4(buff.xyz + (color.xyz*point.w),
buff.w + point.w);
                    }
                        memoryBarrier();
                        atomicExchange(SpinBuffer[id], 0);
                        written = true;
                    }
                } while (!written);

        }
        void turnPoint(uint idx) {
            vec4 space_pos_homo = Rotates[PolyIndex[idx]] * Pixel[idx];
            float z = space_pos_homo.z;
            space_pos_homo  = P * space_pos_homo;
            vec3 space_pos = space_pos_homo.xyz / space_pos_homo.w;
            mat4x4 screen_pos = spaseToScreen(space_pos);
            setColor(screen_pos[0], Color[idx], z);
            setColor(screen_pos[1], Color[idx], z);
            setColor(screen_pos[2], Color[idx], z);
            setColor(screen_pos[3], Color[idx], z);
        }
        void main() {
            uint idx = gl_GlobalInvocationID.x;
            if (idx < Square)
            turnPoint(idx);
        }
```

Фрагментний шейдер

```
#version 430

in vec3 Color;
layout (location=0) out vec4 FragColor;
layout (binding=0) buffer readonly Scr {
   vec4 ScreenBuffer[];
};
ivec2 Size = ivec2(700, 700);
int getBufIndex(vec2 coords) {
   return int(coords.y) * Size.x + int(coords.x);
}
void main() {
   int id =  getBufIndex(gl_FragCoord.xy);
   float w =  ScreenBuffer[id].w;
   if (w == 0.0) discard;
   FragColor = vec4(ScreenBuffer[id].xyz / w, 1.0);
}
void CprogDlg::drawtrianglereal(TPolytri *poly)
{
  _3D_POINT_KOORDINATE P1, P2, P3, S1, S2, S3;
  P1.Set(poly->x1, poly->y1, poly->z1);
  P2.Set(poly->x2, poly->y2, poly->z2);
  P3.Set(poly->x3, poly->y3, poly->z3);

  //define region on screen plane (projection of polygon from object space)
  float x, y;
  y = O1.y + (z - O1.z)*(P1.y - O1.y)/(P1.z - O1.z);
  if (P1.y - O1.y != 0)
    x = O1.x + (y - O1.y)*(P1.x - O1.x)/(P1.y - O1.y);
  else
```

```
{
  y = O1.y;
  x = O1.x + (z - O1.z)*(P1.x - O1.x)/(P1.z - O1.z);
}

S1.Set(x, y, z);
y = O1.y + (z - O1.z)*(P2.y - O1.y)/(P2.z - O1.z);
if (P2.y - O1.y != 0)
  x = O1.x + (y - O1.y)*(P2.x - O1.x)/(P2.y - O1.y);
else
{
  y = O1.y;
  x = O1.x + (z - O1.z)*(P2.x - O1.x)/(P2.z - O1.z);
}
S2.Set(x, y, z);

y = O1.y + (z - O1.z)*(P3.y - O1.y)/(P3.z - O1.z);
if (P3.y - O1.y != 0)
  x = O1.x + (y - O1.y)*(P3.x - O1.x)/(P3.y - O1.y);
else
{
  y = O1.y;
  x = O1.x + (z - O1.z)*(P3.x - O1.x)/(P3.z - O1.z);
}
S3.Set(x, y, z);

poly->x1 = (int)S1.x;
poly->y1 = (int)S1.y;
poly->x2 = (int)S2.x;
poly->y2 = (int)S2.y;
```

```
  poly->x3 = (int)S3.x;
  poly->y3 = (int)S3.y;

  float x1, y1, z1, x2, y2, z2, x3, y3, z3;
  float dxdy1, dxdy2, dxdy3;
  float tempf;
  float denom;
  float dy;
  int y1i, y2i, y3i;
  int side;
// Shift XY coordinate system (+0.5, +0.5) to match the subpixeling
//  technique

  x1 = poly->x1 + 0.5;
  y1 = poly->y1 + 0.5;
  z1 = poly->z1 + 0.5;
  x2 = poly->x2 + 0.5;
  y2 = poly->y2 + 0.5;
  z2 = poly->z2 + 0.5;
  x3 = poly->x3 + 0.5;
  y3 = poly->y3 + 0.5;
  z3 = poly->z3 + 0.5;
// Sort the vertices in ascending Y order

  #define swapfloat(x, y) tempf = x; x = y; y = tempf;

  if (y1 > y2)
  {
     swapfloat(x1, x2);
     swapfloat(y1, y2);
```

```
    swapfloat(z1, z2);
    swapfloat(P1.x, P2.x);
    swapfloat(P1.y, P2.y);
    swapfloat(P1.z, P2.z);
  }
  if (y1 > y3)
  {
    swapfloat(x1, x3);
    swapfloat(y1, y3);
    swapfloat(z1, z3);
    swapfloat(P1.x, P3.x);
    swapfloat(P1.y, P3.y);
    swapfloat(P1.z, P3.z);
  }
  if (y2 > y3)
  {
     swapfloat(x2, x3);
     swapfloat(y2, y3);
     swapfloat(z2, z3);
     swapfloat(P2.x, P3.x);
     swapfloat(P2.y, P3.y);
     swapfloat(P2.z, P3.z);
   }
#undef swapfloat

  y1i = y1;
  y2i = y2;
  y3i = y3;

 // Skip poly if it's too thin to cover any pixels at all
```

```
if ((y1i == y2i && y1i == y3i) || ((int) x1 == (int) x2 && (int) x1 == (int) x3))  return;

denom = ((x3 - x1) * (y2 - y1) - (x2 - x1) * (y3 - y1));

 if (!denom) return; // Skip poly if it's an infinitely thin line

// Calculate X-slopes along the edges
if (y2 > y1)
   dxdy1 = (x2 - x1) / (y2 - y1);
if (y3 > y1)
   dxdy2 = (x3 - x1) / (y3 - y1);
if (y3 > y2)
   dxdy3 = (x3 - x2) / (y3 - y2);

// Determine which side of the poly the longer edge is on
side = dxdy2 > dxdy1;
if (y1 == y2)
   side = x1 > x2;
if (y2 == y3)
   side = x3 > x2;

if (!side)  // Longer edge is on the left side
{
    _3D_LINE l1, l2, l_scrn1, l_scrn2;
    l1.Set(P1, P3);   //longer side
    l_scrn1.Set(x1, y1, z1, x3, y3, z3);

  // Calculate slopes along left edge
```

```
if (y1i < y2i)  // Draw upper segment if possibly visible
{
  l2.Set(P1, P2);
  l_scrn2.Set(x1, y1, z1, x2, y2, z2);
  drawtrianglerealseg(l1, l2, l_scrn1, l_scrn2, y1i, y2i);
 }
 if (y2i < y3i)          // Draw lower segment if possibly visible
 {
   l2.Set(P2, P3);
   l_scrn2.Set(x2, y2, z2, x3, y3, z3);
   drawtrianglerealseg(l1, l2, l_scrn1, l_scrn2, y2i, y3i);
 }
}
else      // Longer edge is on the right side
{
 _3D_LINE l1, l2, l_scrn1, l_scrn2;
 l1.Set(P1, P3);   //longer side
 l_scrn1.Set(x1, y1, z1, x3, y3, z3);

 if (y1i < y2i)          // Draw upper segment if possibly visible
 {
    l2.Set(P1, P2);
    l_scrn2.Set(x1, y1, z1, x2, y2, z2);
    drawtrianglerealseg(l2, l1, l_scrn2, l_scrn1, y1i, y2i);
 }
 if (y2i < y3i)          // Draw lower segment if possibly visible
 {
    l2.Set(P2, P3);
    l_scrn2.Set(x2, y2, z2, x3, y3, z3);
    drawtrianglerealseg(l2, l1, l_scrn2, l_scrn1, y2i, y3i);
```

```
            }
          }
        }

    void CprogDlg::drawtrianglerealseg(_3D_LINE line1, _3D_LINE line2,
_3D_LINE line_screen1, _3D_LINE line_screen2, int y1, int y2)
    {
        ls1.Set(line_screen1.A, line_screen1.B);
        ls2.Set(line_screen2.A, line_screen2.B);
        float t_x1, t_x2;
        for (int y_scanline=y1; y_scanline<y2; y_scanline++)
        {
          t_x1 = ls1.A.x + (y_scanline - ls1.A.y) * (float)(ls1.B.x -
ls1.A.x)/(float)(ls1.B.y - ls1.A.y);
          t_x2 = ls2.A.x + (y_scanline - ls2.A.y) * (float)(ls2.B.x -
ls2.A.x)/(float)(ls2.B.y - ls2.A.y);

          lOPi1.Set(t_x1, y_scanline, z, O1.x, O1.y, O1.z);
          lOPi2.Set(t_x2, y_scanline, z, O1.x, O1.y, O1.z);

          Si1 = GetPoint(lOPi1, line1);   //point on left edge
          Si2 = GetPoint(lOPi2, line2);   //point on right edge

        //set two point to create line on texture plane
          _3D_POINT_KOORDINATE Ti1, Ti2, Q0, Ri1, Ri2;

          Ti1.Set(Si1.x, Si1.y, Si1.z + dist);
          Ti2.Set(Si2.x, Si2.y, Si2.z + dist);

          line1_l_t.Set(Si1, O2);
```

```
line2_l_t.Set(Ti1, Ti2);

line1_r_t.Set(Si2, O2);
line2_r_t.Set(Ti1, Ti2);

Ri1 = GetPoint(line1_l_t, line2_l_t);
Ri2 = GetPoint(line1_r_t, line2_r_t);

float drx, drz;
drx = abs(Ri1.x - Ri2.x);
drz = abs(Ri1.z - Ri2.z);

line1_l.Set(Si1, Si2);
line2_l.Set(t_x1, y_scanline, z, t_x2, y_scanline, z);

Q0 = GetPoint(line1_l, line2_l);

f loat tx, ty, tz, x1, x2, q0x;
x1 = t_x1;
x2 = t_x2;
q0x = Q0.x;

float Lq1q2, Lq0q1;
tx = abs(Si1.x - Si2.x);
ty = abs(Si1.y - Si2.y);
tz = abs(Si1.z - Si2.z);
Lq1q2 = sqrt((double)(tx*tx + ty*ty + tz*tz));

if (Si2.x > Si1.x)
{
```

```
   tx = abs(Si2.x - Q0.x);
   ty = abs(Si2.y - Q0.y);
   tz = abs(Si2.z - Q0.z);
}
else
{
   tx = abs(Si1.x - Q0.x);
   ty = abs(Si1.y - Q0.y);
   tz = abs(Si1.z - Q0.z);
}

Lq0q1 = sqrt((double)(tx*tx + ty*ty + tz*tz));
float n, alfa, betta, K, K1, K2, kci1, kci2;

if (Q0.x > x1)
{
  n = (float)(Ri2.x - Ri1.x)/Lq1q2;
  alfa = (float)(Ri2.y - Ri1.y)/(float)(Ri2.x - Ri1.x);
  betta = (float)(Ri2.x*Ri1.y - Ri1.x*Ri2.y)/(float)(Ri2.x-Ri1.x);

   K = (x2-x1)*(Lq0q1+Lq1q2)/(float)((Q0.x-x1)*Lq1q2);
   K2 = K - 1;
   K1 = (K - 1)*Ri1.x + n*Lq0q1;
   kci2 = K*(q0x-x2);
   kci1 = kci2 * Ri1.x;
}
else
{
  n = (float)(Ri2.x - Ri1.x)/Lq1q2;
  alfa = (float)(Ri2.y - Ri1.y)/(float)(Ri2.x - Ri1.x);
```

```
            betta = (float)(Ri2.x*Ri1.y - Ri1.x*Ri2.y)/(float)(Ri2.x-Ri1.x);

            K = (x2-x1)*(Lq0q1+Lq1q2)/(float)((x2-Q0.x)*Lq1q2);
            K2 = K - 1;
            K1 = (K - 1)*Ri1.x + n*Lq0q1;
            kci2 = K*(x1-q0x);
            kci1 = kci2 * Ri1.x;
          }

          for (int xi=x1; xi<=x2; xi++)
          {
            int u, v;
            u = kci1/kci2;
            v = abs(alfa*u + betta);
            if (u >= texture_width)
              u = u - texture_width;
            if (v >= texture_height)
              v = v - texture_height;

            cdc->SetPixel(xi, y_scanline, RGB(myrgb[v][u].r, myrgb[v][u].g,
myrgb[v][u].b));
            kci1 += K1;
            kci2 += K2;
          }
        }
      }
```

Додаток Г. Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА МЕТОДІВ І ПРОГРАМНИХ ЗАСОБІВ
ДЛЯ ТЕКСТУРУВАННЯ

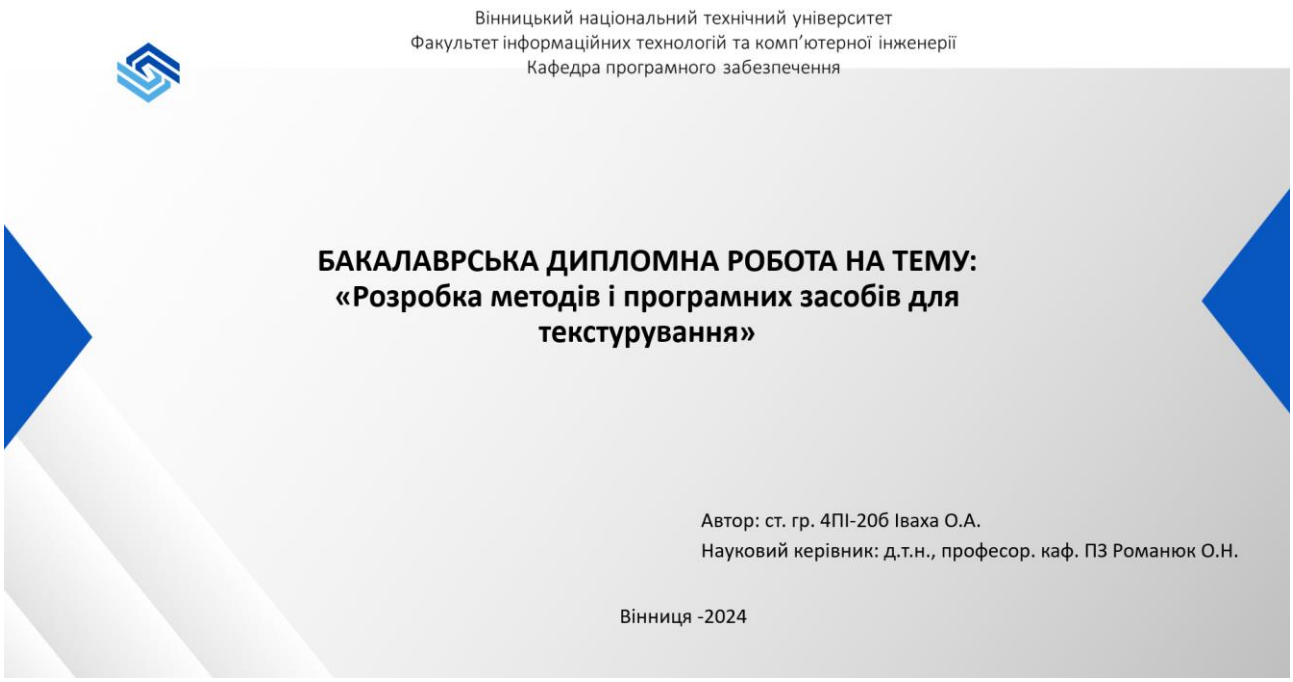


Рисунок Г.1 – Титульний слайд

Рисунок Г.2 – Актуальність теми

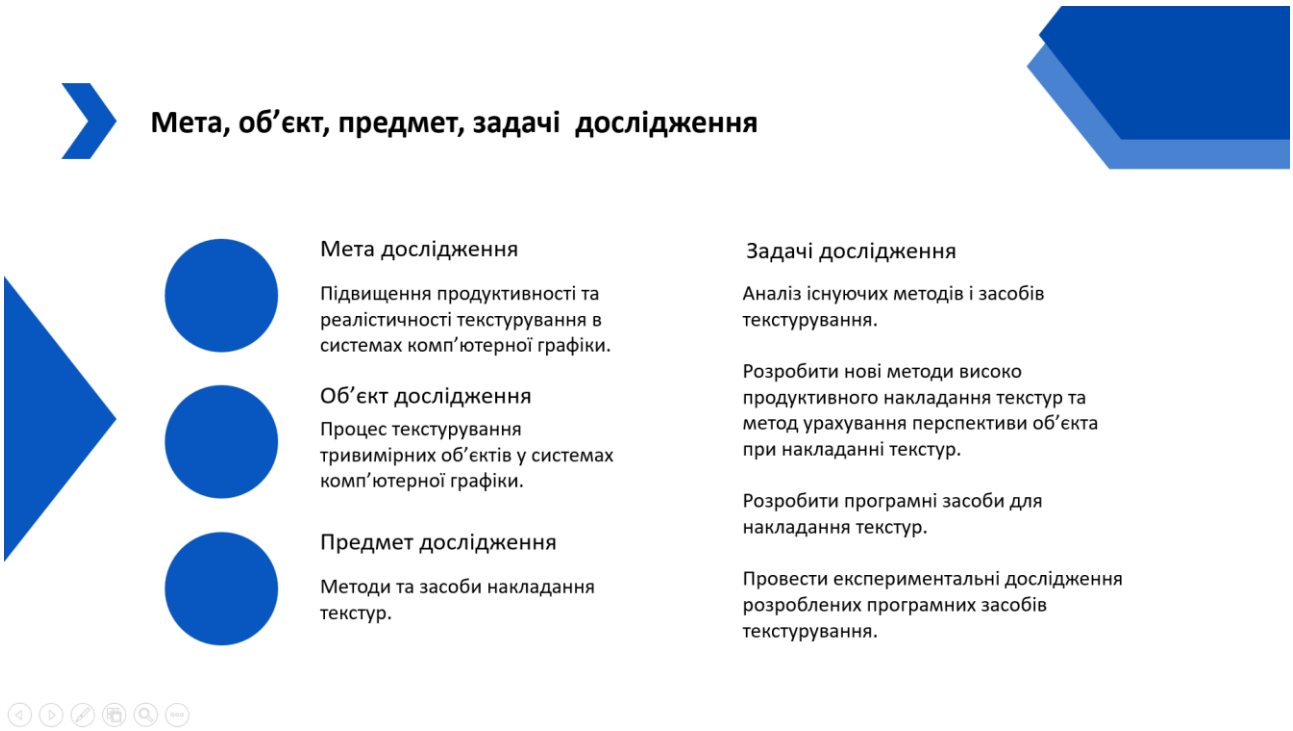


Рисунок Г.3 – Мета, об’єкт та предмет дослідження

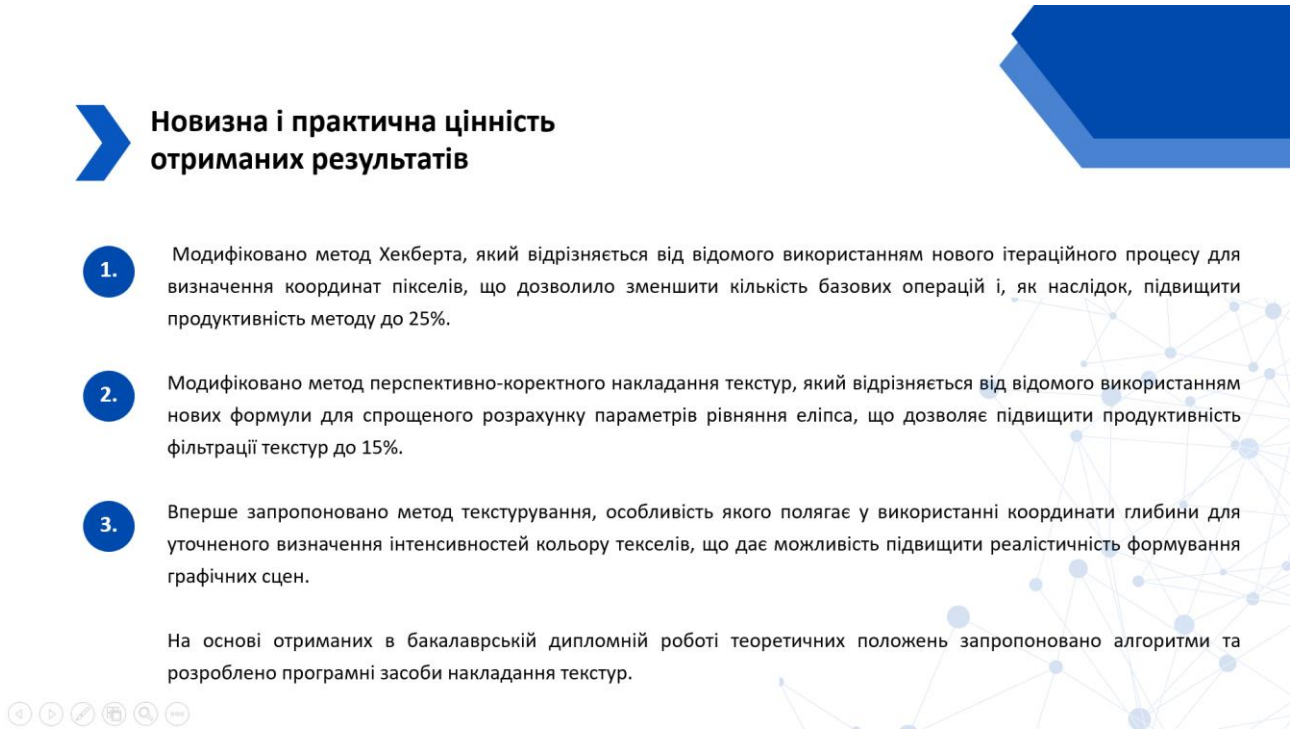
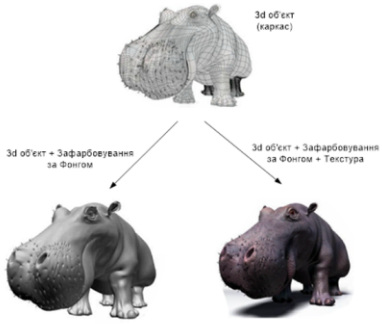


Рисунок Г.4 – Новизна і практична цінність отриманих результатів

Особливості текстурування в системах комп’ютерної графіки

Текстурування матеріалів – це процес накладання текстур, зображень або візерунків на поверхню об'єкта у комп'ютерній графіці.



Етапи накладання текстур

Техніки текстурування матеріалів

- Процедурне текстурування
- Фотографічне текстурування
- Проекційне текстурування
- UV-розгортка
- Шарове текстурування

Рисунок Г.5 – Особливості текстурування в системах комп’ютерної графіки

Техніки текстурування

Текстурування

- UV Маппінг (UV Mapping)
- Проекційне текстурування (Projection Mapping)
- Сферичне текстурування
- Циліндричне текстурування
- Кубічне текстурування (Cube Mapping)
- Бамп мапінг (Bump Mapping)
- Нормал мапінг (Normal Mapping)
- Детальне текстурування (Detail Texturing)
- Мульти-текстурування (Multi-Texturing)
- Паралакс текстурування (Parallax Mapping)
- Техніка Теселяція (Tessellation)

Рисунок Г.6 – Техніки текстурування

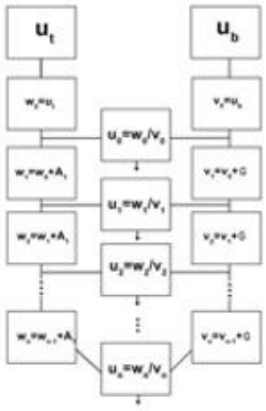
Використання методу Хекберта для виконання перспективного перетворення

1. **Ітераційні формули для обчислення тексельних координат u та v :**

$$u_n = \frac{A_1 \cdot (x_0 + n) + B_1 \cdot y_i + C_1}{D \cdot (x_0 + n) + E \cdot y_i + F} \qquad v_n = \frac{D_1 \cdot x_i + E_1 \cdot y_0 + E_1 \cdot n + F_1}{G \cdot x_i + H \cdot y_0 + H \cdot n + I}$$

У вдосконаленій версії методу Хекберта використовуються ітераційні формули для визначення тексельних координат, що допомагає зменшити кількість арифметичних операцій і збільшити ефективність.

Основна ідея полягає у тому, що замість повторного обчислення параметрів для кожного текселя, використовуються ітераційні формули, які дозволяють швидше обчислити координати.



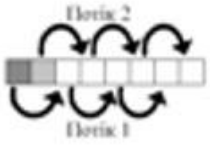
Визначення текселів

Рисунок Г.7 – Використання методу Хекберта для виконання перспективного перетворення

Використання методу Хекберта для виконання перспективного перетворення

2. **Паралельне обчислення текселів**

$$w_n = w_{n-k} + kA_1 \qquad v_n = v_{n-k} + kG$$

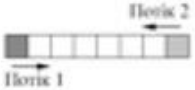


Паралельне обчислення текселів

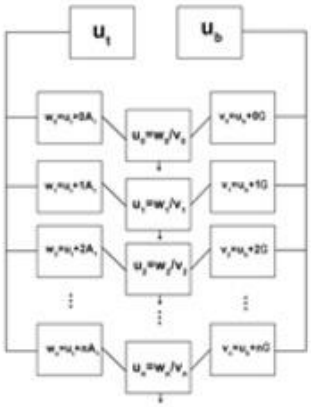
3. **Зустрічне текстурування**

$$w_n = w_{n+1} - A_1 \qquad v_n = v_{n+1} - G$$

Формула зустрічного текстурування використовується для зменшення кількості обчислень, використовуючи результати обчислення з одного текселя для обчислення наступного.



Зустрічне текстурування



Паралельне визначення текселів

Рисунок Г.8 – Використання методу Хекберта для виконання перспективного перетворення №2

Розробка методу накладання текстур з урахуванням координати глибини

Метод накладання текстур враховує координати глибини (z-координати) для правильного відображення перспективи об'єкта. Основою методу є співвідношення координат точок на поверхні об'єкта та екрану

Використані формули:

$$\frac{x(j) - x_1}{x_2 - x_1} = \frac{z(j) - z_1}{z_2 - z_1} = w$$

$$x(j) = x_1 + (x_2 - x_1) \cdot \frac{z_1 \cdot u}{z_2 - u \cdot (z_2 - z_1)}$$

$$z(j) = z_1 + (z_2 - z_1) \cdot \frac{z_1 \cdot u}{z_2 - u \cdot (z_2 - z_1)}$$

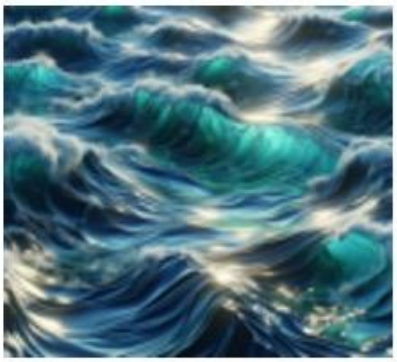
Приклад текстурування за розробленим методом

Таким чином, розроблені аналітичні формули для обрахунку текстурних координат, виходячи з z-координат об'єкта. Це забезпечило можливість врахування перспективи об'єкта і визначення нелінійного зв'язку між координатами на екрані та текстурними координатами

Рисунок Г.9 – Розробка методу накладання текстур з урахуванням координати глибини

Алгоритм формування зображення поверхні води з використанням шуму Перліна

1. **Генерація базового шуму**
   Створюється базовий шум Перліна шляхом інтерполяції між псевдовипадковими градієнтними значеннями в просторі.
2. **Покращення шуму**
   Використовуються кілька шарів шуму з різним масштабом і амплітудою (фрактальний шум), щоб додати деталі на різних рівнях.
3. **Анімація води**
   Змінюються параметри шуму з часом (зміщення координат, зміна частоти) для створення ефекту руху та хвиль.
4. **Візуалізація**
   Шум використовується для зміни висоти водної поверхні, створюючи реалістичні візуальні ефекти.



Приклад зображення води, сформованого з використанням шуму Перліна

Рисунок Г.10 – Алгоритм формування зображення поверхні води з використанням шуму Перліна

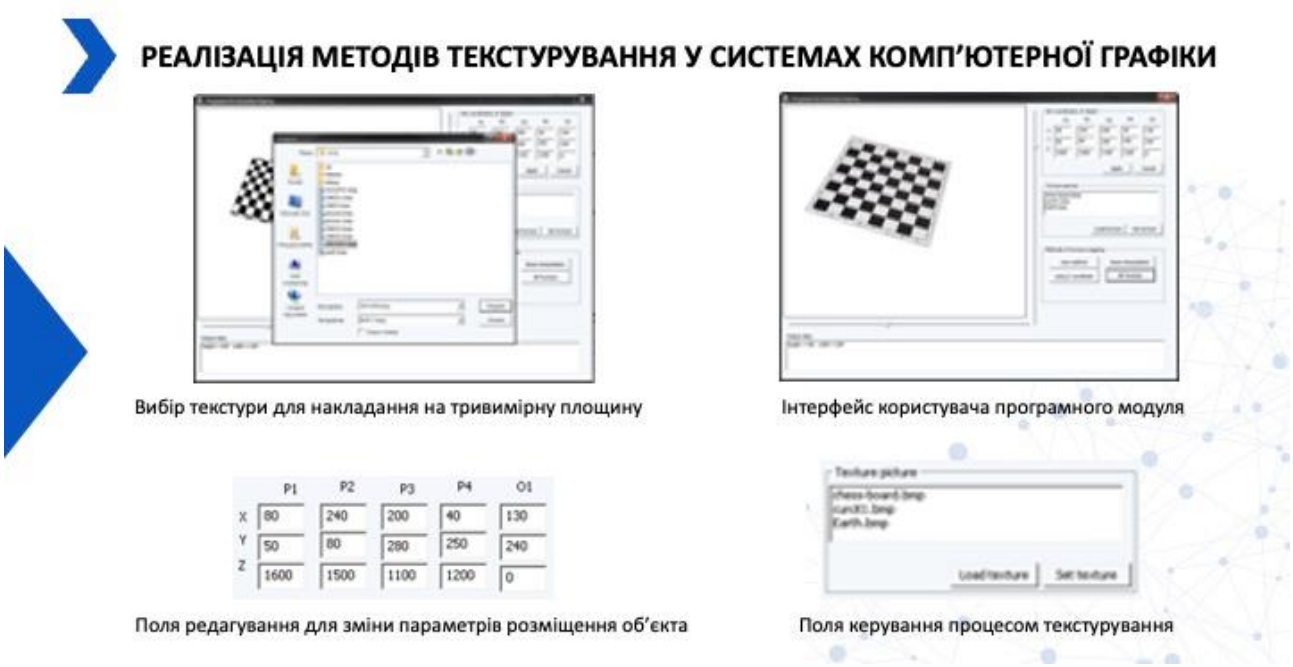


Рисунок Г.11 – Реалізація методів текстурування у системах комп'ютерної графіки

Рисунок Г.12 – Реалізація методів текстурування у системах комп'ютерної графіки №2

Рисунок Г.13 – Реалізація методів текстурування у системах комп'ютерної графіки №3

Рисунок Г.14 – Апробація матеріалів бакалаврської дипломної роботи

Рисунок Г.14 – Реалізація методів текстурування у системах комп'ютерної графіки №4

Рисунок Г.15 – Фінальний слайд