Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Синтез зображень поверхонь методом зворотного трасування

(тема)

Виконав:
студент II курсу, групи СПм-18-3
Лунічкін О.Г.
(прізвище, ініціали)

Спеціальність
123 – Комп’ютерна інженерія
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Гусятін В.М.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп’ютерна інженерія
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові Лунічкіну Олексію Георгійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Синтез зображень поверхонь методом зворотнього трасування

затверджена наказом по університету від “ 30 ” березня 2020 р. № 478 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 18 травня 2020 р.

3. Вхідні дані до роботи 1) цільова платформа: ОС Windows;
2) мова програмування: C++;
3) підтримка графічного інтерфейсу OpenGL версії 4.3;
4) застосувати метод зворотнього для синтезу зображення;
5) проаналізувати продуктивність програмної реалізації методу зворотного трасування

4. Перелік питань, що потрібно опрацювати в роботі
1) аналіз літератури та постановка задачі
2) методи синтезу зображень поверхонь зворотним трасуванням
3) програмна реалізація
4) висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Демонстраційні матеріали. Плакати – 16 арк. ф. А4

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз літератури	31.03.20-9.04.20	
2	Розробка середовища	10.04.20-21.04.20	
3	Реалізація алгоритмів	22.04.20-29.04.20	
4	Налагодження програми	30.04.20-05.05.20	
5	Оформлення матеріалів атестаційної роботи	06.05.20-11.05.20	
6	Подання атестаційної роботи керівникові та	12.05.20-13.05.20	
	її попередній захист		
7	Подання атестаційної роботи на рецензування	14.05.20-15.05.20	

Дата видачі завдання 30 березня 2020 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Гусятін В.М.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 79 с., 25 рис., 3 табл., 1 дод., 21 джерело.

КОМП’ЮТЕРНА ГРАФІКА, ЗОБРАЖЕННЯ, RAYTRACING, OPENGL, REALTIME, ШЕЙДЕРИ.

Метою атестаційної роботи є аналіз та моделювання алгоритмів зворотнього трасування, забезпечуючих синтез зображення полігональних моделей, а також адаптація до роботи у реальному часі.

У ході виконання атестаційної роботи був проаналізований ітераційний алгоритм зворотнього трасування та адаптований до роботи з двома видами поверхонь. Для моделювання було розроблено середовище, здатне завантажувати полігональні моделі. Алгоритми були реалізовані як шейдерні програми, які здатні синтезувати зображення полігональних сцен.

ABSTRACT

Master’s thesis: 79 pages, 25 figures, 3 tables, 1 appendices, 21 sources.

COMPUTER GRAPHICS, IMAGE, RAYTRACING, OPENGL, REALTIME, SHADERS.

The major goal of this thesis is analysis and modeling of raytracing algorithms, providing image synthesis of polygonal models, and adapting them to work in realtime.

In order to analyse and model raytracing algorithm and two kinds of model surfaces program environment was built. Environment is capable of loading polygonal models and running shader programs, so raytracing algorithm was implemented as shader program, which renders loaded model.

ЗМІСТ

ВСТУП

Перші обчислювальні пристрої були механічними, та були пристосовані лише до арифметичних операцій, тому інформація з них відображалась у вигляді цифр на табло. Із появою перших електронних обчислювальних пристроїв можливості щодо обчислень значно поширилися, але результат, як і раніше, подавався майже механічно – друкувався на стрічках або кодувався на вмиканні ламп. Але невпинний розвиток техніки зробив можливим суттєво складніші обчислення, які вийшли за рамки чистої математики. Відповідно, комп’ютер здатний був оперувати не тільки операндами математичних виразів, а й багатьма іншими видами даних – текст, зображення, звуки, масиви усіляких замірів, необхідних вченим та інженерам. Як наслідок, виводити на табло або екрани лише числа було недостатньо, що призвело до створення та швидкого розповсюдження такого явища, як комп’ютерна графіка.

З початку графічно виводилась досить проста інформація – тексти, зображення. Перша цифрова двомиріна графіка була примітивна, утворена простими геомтеричними фігурами. Але згодом, із застосуванням комп’ютерів у складанні схем, креслень з’являлось більше графічних інструментів, а розробники задумались над синтезом зображень тривимірної сцени. Із появою достатньо потужних комп’ютерів з’явились системи тривимірної графіки, що дозволяли параметрично задавати сцену та отримувати її зображення. З цього моменту почався нескінченний шлях розвитоку тривимірної графіки, головною метою якої є синтез реалістичного зображення.

З розповсюдженням персональних комп’ютерів доступ до різноманітних редакторів та програмних середовищ отримала велика кількість людей, тому попит на графічні технології зріс, адже сприймати графічну інформацію звичайному користувачу значно легше, ніж текст.

Відповідно зріс попит й на тривимірну графіку, адже спеціаліст може змоделювати необхідний об’єкт, не маючи спеціальних навичок у комп’ютерній галузі, особливо знань щодо малювання та крелсення, й синтезувати будь-які необхідні двовимірні зображення.

Разом із поширенням персональних комп’ютерів почали з’являтись відеоадаптери з апаратним прискоренням деяких графічних функцій, що дозволило синтезувати тривимірні зображення у реальному часі. Тепер тривимірне моделювання та проектування доступне не тільки власникам потужних робочих станцій, а й власникам будь-якого комп’ютеру.

Сучасні системи візуалізації дозволяють зображати більшість необхідних об’єктів. Але примітиви, з яким вона працює не змінилися з моменту появи перших специфікацій для тривимірної графіки у реальному часі. Тому для складних об’єктів доводиться застосовувати багато примітивів, що потребує багато пам’яті та потужностей графічного процесора. Основа алгоритму синтезу також не змінилась, і він не здатний ефективно працювати з великими даними та реалістично розраховувати освітлення.

У даній роботі автор пропонує нові примітиви та новий алгоритм для формування тривимірної сцени для росширення функціоналу моделювання тривимірних об’єктів, а також для набагато реалістичнішого розрахунку освітлення.

1 АНАЛІЗ ЛІТЕРАТУРИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Класифікація видів комп’ютерної графіки

1.1.1 Растрова графіка

Основним елементом растрового зображення є точка (крапка). Якщо зображення екранне, то ця точка називається пікселем. Залежно від того, на яку графічну розподільчу здатність екрану налаштована операційна система комп’ютера, на екрані можуть розміщуватись зображення, які мають 640x480, 800x600, 1024x768 і більше пікселів. Приклад такого зображення наведений на рисунку 1.1.

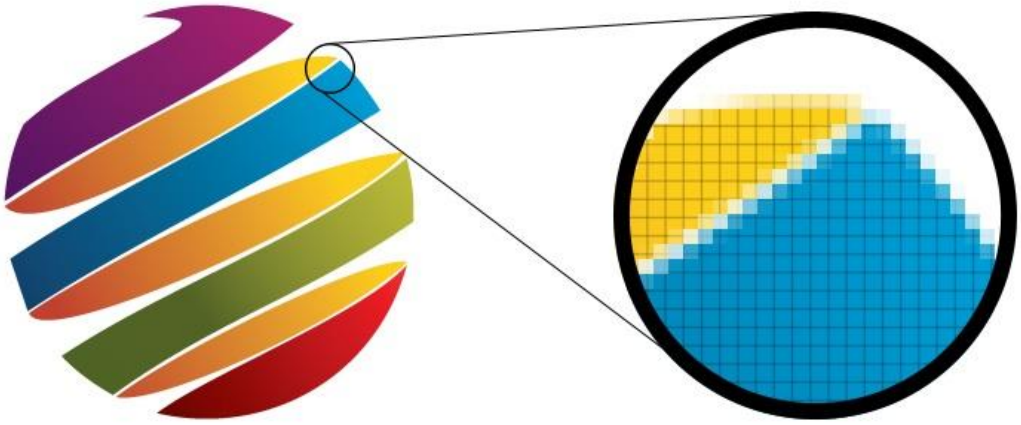

Рисунок 1.1 – Растрове зображення

З розміром зображення безпосередньо пов’язана його роздільна здатність. Цей параметр вимірюється в точках на дюйм (англ. dots per inch, dpi). У монітора з діагоналлю 15 дюймів розмір зображення на екрані становить приблизно 28x21 см. Знаючи, що в одному дюймі 25,4 мм, можна розрахувати, що при роботі монітора в режимі 800x600 пікселів роздільна здатність екранного зображення 72 dpi.

При друці розподільча здатність має бути набагато вище. Поліграфічний друк повноколірного зображення вимагає роздільної здатності 200–300 dpi. Стандартний фотознімок 10x15 см повинен мати приблизно 1000x1500 пікселів. Таке зображення буде мати 1,5 млн точок, а якщо зображення кольорове і на координування кожної точки використано три байти, то звичайній фотографії відповідатиме масив даних розміром понад 4 Мбайт.

Великий обсяг даних – головна проблема при використанні растрових зображень. Для активних робіт з великими ілюстраціями типу журнальної шпальти потрібні комп’ютери з великими обсягами оперативної пам’яті і високопродуктивними процесорами [1].

Ще один недолік, зумовлений дискретним характером растрових зображень – ефект пікселізації (у багатьох джерелах вживається англомовний термін aliasing). Приклад можна побачити на рисунку 1.1. Пікселі, розташовані у ряди та стовпчики, не можуть точно зобразити криволінійні фігури та будь-які негоризонтальні та невертикальні лінії. Для пом’якшення цього ефекту існують алгоритми, які, особливо у тривимірних візуалізаціях можуть бути досить важкими для процесорів [4]. Але, зі збільшенням роздільної здатності сучасних дисплеїв цей ефект значно менше впливає на зображення.

Так як усі сучасні монітори побудовані на основі піксельних матриць, для формування зображення на них використовується растрова графіка. Але, якщо абстрагуватися від моніторів та дивитися на алгоритми обробки інформації, то методи растрової графіки найбільше використовуються у збереженні та відображенні мультимедійних даних – фото та відео, а також для цифрового малюнку. Проте, збереження таких великих даних не є простим – винайдено багато різних алгоритмів компресії мультимедійних даних, та розробляється ще більше, адже з кожним роком роздільна здатність дисплеїв збільшується, а з ними росте й роздільна здатність зображень та відео, які хоче побачити користувач. Алгоритми компресії стають складними

та вибагливими до процесорів, тому задачі з розробки ефективніших методів є і надалі будуть актуальними.

1.1.2 Векторна графіка

У векторній графіці основним елементом зображення є лінія. В растровій графіці також існують лінії, але там вони розглядаються як комбінації точок. Відповідно, чим довша растрова лінія, тим більше пам’яті вона потребує. У векторній графіці обсяг пам’яті, для зберігання лінії, не залежить від розміру лінії, оскільки лінія представляється у вигляді формули, а точніше, у вигляді кількох параметрів [1]. Що б ми не робили з цією лінією, міняються тільки її параметри, які зберігаються в чарунках пам’яті. Кількість чарунків залишається незмінною для будь-якої лінії.

Лінія – елементарний об’єкт векторної графіки. Все, що є у векторній ілюстрації, складається з ліній. Найпростіші об’єкти об’єднуються в складніші, наприклад, чотирикутник можна розглядати як чотири взаємопов’язані лінії, а куб як дванадцять взаємопов’язаних ліній, або як шість чотирикутників. Через такий підхід векторну графіку часто називають об’єктно-орієнтованою графікою. Графічні примітиви не обмежуються лише прямими та ламаними, але й могуть бути аналітично описаними кривими другого, третього порядку, та інші. Типові примітивні об’єкти наступні:

а) лінії і ламані лінії;

б) багатокутники;

в) кола та еліпси;

г) криві Без’є;

д) безігони;

ж) текст.

Як усі об’єкти, лінії мають властивості. До цих властивостей належать: форма лінії, її товщина, колір, характер лінії (суцільна, пунктирна тощо). Замкнуті лінії мають властивість заповнення. Внутрішня область замкнутого

контуру може бути заповнена кольором, картою (заготовлені растрові зображення).

Переваги векторної графіки:

а) розмір файла, який займає описова частина, не залежить від реальної величини об’єкта, що дозволяє, використовуючи мінімальну кількість інформації, описати достатньо великий об’єкт файлом мінімального розміру;

б) у зв’язку з тим, що інформація про об’єкт зберігається в описовій формі, можна нескінченно збільшити графічний примітив, наприклад, дугу кола, і вона залишиться гладкою. З іншого боку, якщо крива представлена у вигляді ламаної лінії, збільшення покаже, що крива не є гладкою;

в) параметри об’єктів зберігаються і можуть бути легко змінені. Також це означає, що переміщення, масштабування, обертання та інше, не погіршує якості малюнка. Більш того, зазвичай вказують розміри в апаратно-незалежних одиницях (англ. device-independent unit), які ведуть до найкращої растеризації на растрових приладах;

г) при збільшенні або зменшенні об’єктів товщина ліній може бути задана постійною величиною, незалежно від реального контуру.

Недолік: майже ніякий об’єкт неможливо точно зобразити у векторному вигляді – для того, щоб зображення було подібним до оригіналу може знадобитися дуже велика кількість об’єктів з високою складністю, що негативно впливає на кількість пам’яті, яку займатиме зображення та час для його відтворення. Тому такий вид графіки використовується для схематичної інформації, без вимог до реалізму. Насамперед це звичайні текстові повідомлення, знаки, плакати, тощо. Також векторна графіка застосовується у промисловому дизайні, кресленнях. Хоча конструктори дотримуються реалізму, але він здебільшого не стосується зовнішнього виду усієї сцени – на кресленнях немає фону, немає реалістчиного зображення поверхонь, лише реалзім конструкції об’єкту. Тому застосовується спрощене зображення лініями, які не завантажують працівників зайвими подробицями.

1.1.3 Тривимірна графіка

Хоча тривимірна графіка не є самостійним видом графіки, як растрова або векторна, її методи сильно відрізняються від методів вказаних видів графіки, тому її доцільно розглядати як окреме явище.

Тривимірне зображення на площині відрізняється від двовимірного растрового або векторного тим, що включає побудову геометричної проекції тривимірної моделі(сцени) на площину (наприклад, екран комп’ютера) за допомогою спеціалізованих програм. При цьому модель може як відповідати об’єктам з реального світу (автомобілі, будівлі, ураган, астероїд), так і бути повністю абстрактною (проекція чотиривимірного фрактала) [2].

Для одержання тривимірного зображення на площині потрібні такі кроки:

а) моделювання – створення тривимірної математичної моделі сцени і об’єктів в ній;

б) рендеринг (візуалізація) – побудова проекції відповідно до обраної фізичної моделі та обчислення кольорів пікселів зображення;

в) виведення отриманого зображення на пристрій виведення – дисплей або принтер.

Вхідними даними для побудови тривимірного зображення є тривимірна сцена. Вона містить в собі декілька категорій об’єктів:

а) геометрія (побудована за допомогою різних технік модель, наприклад будівля);

б) матеріали (інформація про візуальні властивості моделі, наприклад колір стін і відбиваюча / заломлююча здатність вікон);

в) джерела світла (налаштування напрямку, потужності, спектра освітлення);

г) віртуальні камери (вибір точки та кута побудови проекції);

д) сили та дії (налаштування динамічних спотворень об’єктів, застосовується в основному в анімації);

ж) додаткові ефекти (об’єкти, що імітують атмосферні явища: світло у тумані, хмари, полум’я та ін.).

Тривимірна графіка застосовується у багатьох галузях діяльності у випадках, коли потрібно створити реалістичне зображення складного об’єкту. Особливо це стосується науковців, конструкторів та інженерів, для яких моделювати складні явища та об’єкту на двовимірних кресленнях складніше, ніж синтезувати на основі тривимірної сцени документацію. Найбільш інтенсивно графіка застосовується у тренажерах та імітаційних системах для тренування персоналу. Саме в таких системах великі вимоги і до реалізму, і до інтерактивності – користувач не тільки повинен бачити якомога реальніші об’єкти, але й мати змогу роздивлятися їх, рухати, взаємодіяти, тощо. Приклад наведено на рисунку 1.2 – тренажер пожежного літака.

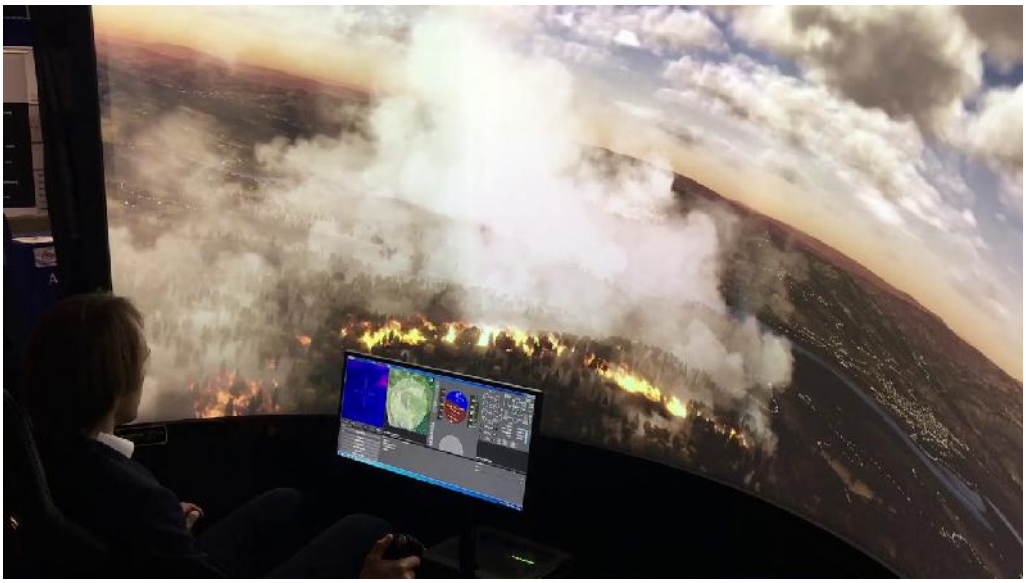

Рисунок 1.2 – Тренажер пожежного літака

У такій системі поєднується реалізм з високою інтерактивністю. Адже для майбутнього пілота важливо отримати уявлення про зовнішній вигляд погодних умов та ландшафту, запам’ятати вигляд орієнтирів для майбутніх польотів. Але, окрім реалізму, необхідно забезпечити й високу частоту

оновлення зображення, аби переміщення були гладкими та приємними зору [3]. Також моделі техніки будуть анімованими, а ландшафт будуватиметься з реальних даних. Тому для відображення необхідні якнайдетальніші моделі, але не занадто детальні, аби можна було отримати 30 кадрів у секунду. Ландшафт повинен якнайдетальніше повторювати реальний, але не занадто детально, аби його можна було швидко завантажувати під час використання тренажеру без затримок у зображенні усієї сцени. Саме таке безкомпромісне поєднання вимог потребує як спеціальні апаратні рішення, так і найефективнішого програмного забезпечення.

1.2 Процес рендерингу

1.2.1 Визначення та класифікація рендерингу

Як описувалося раніше, рендеринг являється другим етапом отримання зображення на основі тривимірної сцени – після етапу розробки тривимірної сцени та перед етапом виводу зображення на пристрій людино-машинного інтерфейсу.

Ре́ндеринг, або комп’ютерна візуалізація (англ. rendering – візуалізація, проявлення, відмальовування, подання) – це процес отримання зображення за моделлю з допомогою комп’ютерної програми. Тут модель – це заздалегідь підготовлена тривимірна сцена, яка містить геометричні дані, положення точки спостерігача, інформацію про освітлення. А зображення – це цифрове растрове зображення [2].

Рендеринг є найскладнішим етапом створення зображення як технічно, так і алгоритмічно. На даний момент не існує універсального алгоритму, дозволяючого швидко отримувати реалістичне зображення будь-яких об’єктів або явищ [3]. Насамперед це зумовлено нестачею обчислювальної потужності комп’ютерної техніки, відсутністю спеціалізованих

обчислювальних пристроїв для реалізації універсальних алгоритмів та малим числом спеціалістів, працюючих у даній сфері. Тому сьогодні методи рендерингу поділяють на дві великі групи – алгоритми офлайн рендерингу та алгоритми реального часу [2].

Методи офлайн рендерингу використовують найскладніші алгоритми та формули для обчислення освітлення та виведення форми об’єктів, гарантуючи фотореалістичний результат. Приклад наведено на рисунку 1.3. Отримане зображення є реалістичним, та у деяких випадках людина неспроможна відрізнити його від фотографії.

Рисунок 1.3 – Офлайн-рендер

Але синтез лише одного кадру потребуватиме години, або навіть дні невпинної праці потужних комп’ютерів, тому налаштування сцени та тестовий рендеринг неможливі без спрощення, яке дозволить бачити зміни у сцені одразу після їх внесення. Тому офлайн рендеринг неможливий без участі рендерингу реального часу.

Методи рендерингу у реальному часі набагато складніші у розробці та використанні, адже вони також спрямовані на досягнення реалізму, але у жорсткому обмеженні часу, наприклад, 33 мілісекунд на кадр, що дозволяє

оновлювати зображення сцени 30 разів на секунду, що сприймається людиною як неперервна анімація [4]. Приклад наведено на рисунку 1.4.

Рисунок 1.4 – Рендер у реальному часі

Для досягнення необхідної частоти оновлення зображення доводиться йти на компроміси, тому зображення не є таки деталізованим, розрахувати можна лише найпростіші матеріали поверхонь, якість освітлення нижча [4]. Хоча вже винайшли багато алгоритмів, імітуючих різні реалістичні ефекти матеріалів та освітлення у реальному часі, отримати реалістичне зображення з такими інструментами досить важко й на деяких сценах взагалі неможливо.

1.2.2 Прямий метод рендерингу

Історично першим методом рендерингу був прямий метод, коли фігури послідовно проектуються на площину екрана та малюються. Відповідно, сцена повинна складатися з примітивних фігур, які легко проектувати на площину та малювати. Такою фігурою став трикутник, з якого будують полігони, з яких складається сцена. Для трансформації використовуються матриці – вершини трикутників, описані координатою, послідовно

множаться на матриці перетворення. Для малювання найближчої до камери поверхні сцени використається механізм z-буфер. Для полігонального рендерінгу були створені дві дуже схожі специфікації – OpenGL та Direct3D, а згодом додались Vulkan та Metal. Вони реалізовані у великій кількості спеціальних пристроїв – відеоадаптерах. Тому цей метод став основним у графіці реального часу, і залишається таким й сьогодні [4]. Полігональний рендеринг здійснюється за наступною послідовністю:

а) конструювання примітивів – утворення масивів трикутників із вхідних даних, які вже можуть бути трикутниками, а можуть задаватися у іншому форматі;

б) трансформація вершин – обчислення вихідних даних кожної вершини на основі вхідних та обчислення координати вершини у просторі екрану;

в) перетворення трикутників – опціональна стадія. Дозоляє видаляти чи добавляти нові трикутники;

г) растеризація – пошук пікселів, які знаходяться всередині трикутника;

в) обчислення кольору кожного пікселю. Для кожного інтерполюються дані з етапу трансформації вершин, які застосовуються для розрахунку кольору;

г) z-тестування та інші тести для пікселю. На цьому етапі проводиться порівняння глибини розрахованого пікселю з глибиною вже намальованої сцени в цьому місці, та вирішується, можна ставити цей піксель в зображення, чи ні. Також проводиться опціональний тест на сумісність з бітовою маскою цього пікселя.

Такий метод рендерингу є достатньо простим в реалізації та обчисленні, тому він домінує сьогодні. Але в нього є істотні недоліки:

а) неможливість передавати нелінейні поверхні, лише апроксимувати полігональною сіткою;

б) неможливість намалювати прозорі об’єкти, алгоритм z-буферу неможливо застосувати до них, і, як наслідок, відсортувати по дистанції до

камери з іншими непрозорими полігонами [4];

в) неможливість малювати об’ємні явища – погідні умови, вогонь, газові явища. Адже трикутниками описуються тільки плоскі об’єкти;

г) неможливість обчислення непрямого освітлення, окрім тіней. Якщо для тіней є алгоритм, то реалізувати реалістичне дифузне освітлення та відбиття цим методом неможливо;

в) нераціональне використання обчислювальних ресурсів – піксель поверхні розраховується навіть тоді, коли його не видно.

Використання лише прямого методу зумовлює багато обмежень, тому із розвитком обчислювальної техніки розробники намагаються застосовувати алгоритми іншого типу, які здатні забезпечити реалістичне зображення об’єктів та ефектів – насамперед, це стосується атмосферних процесів, погоди та освітлення [8].

1.2.3 Зворотній метод рендерингу

Якщо у попередньому методі спочатку обчислювався піксель поверхні фігури, а потім приймалося рішення о його доцільності, то в цьому методі спочатку шукаються видимі пікселі поверхні, а тільки потім обчислюється їх колір. Це реалізовано у методах трасування променів.

Трасування променів (англ. ray tracing) у комп’ютерній графіці є способом створення зображення тривимірних об’єктів чи сцени за допомогою відстеження ходу променя світла крізь точку екрану і симуляції взаємодії цього променя з уявними об’єктами, що підлягають відображенню. Загальна схема алгоритму показана на рисунку 1.5. Цей спосіб дозволяє створювати надзвичайно реалістичні зображення, зазвичай значно вищої якості, ніж дає растеризація, проте має значно вищу обчислювальну складність [2].

Ключовим моментом трасування є пошук перетину променю з

об’єктом сцени. Саме це обчислення найтяжче та найважливіше у всій системі. Якщо реалізувати його апаратно у відеоадаптері, то можна навіть сьогодні отримати повністю трасоване тривимірне зображення у реальному часі, хоча така задача є дуже складною. Для оптимізації пошуку перетину простір може індексуватися – об’єкти простору укладаються у деяку структуру даних, що дозволяє швидко їй шукати по деякому ключу, а ключ пов’язати з просторовими координатами [3]. Це дозволяє швидко визначати для кожного променю список ймовірних об’єктів перетину, який буде значно менше, ніж увесь список об’єктів сцени. Але це ще сильніше ускладнює задачу апаратної реалізації зворотнього методу.

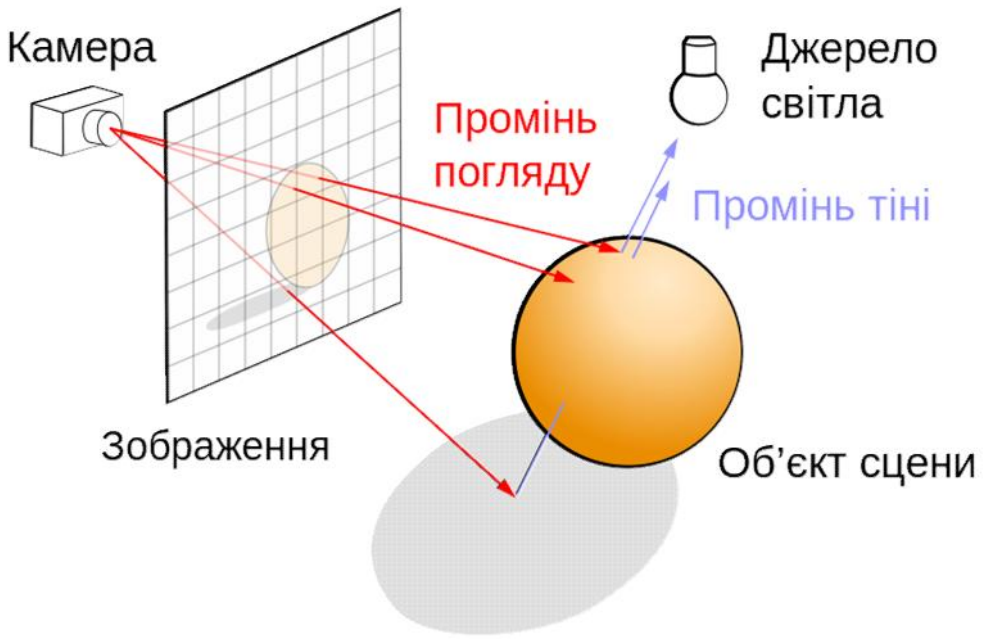

Рисунок 1.5 – Схема трасування променів

Алгоритм трасування променів дозволяє природним чином отримати такі ефекти, які для інших алгоритмів рендеринга складають значну складність. Серед них правильне затінення, дзеркальні поверхні, заломлення світла [2]. Завдяки цьому реалістичність сцен, обрахованих методом трасування променів, інколи сягає "фотографічної". Також, на відміну від

поширених полігональних алгоритмів, він дозволяє обробляти у сцені об’єкти фактично довільних геометричних форм, що можуть бути виражені математичним записом – справжні (а не апроксимованих полігонами) сфери, тори, конуси, параболоїди, еліпсоїди та т.ін. Алгоритм також дозволяє природно розпаралелити обчислення між процесорами, адже обрахування ходу кожного променя у сцені відбувається незалежно [7].

1.3 Примітиви тривимірних моделей

У сучасних системах візуалізації для моделювання поверхонь використовуються виключно полігональні моделі. При моделюванні полігони можуть мати багато кутів, але при підготовці моделі до рендеру усі многокутні полігони поділяються на трикутники, тож алгоритми синтезу зображення працюють лише з трикутниками. Саме трикутники найкраще пристосовані до растеризації, тому для графіки реального часу вони залишаються єдиним способом опису моделей [4]. Але таким простим примітивом неможливо точно передати непласкі повехрні, тож їх доводиться апроксимувати ефектами освітлення, як приведено на рисунку 1.6.

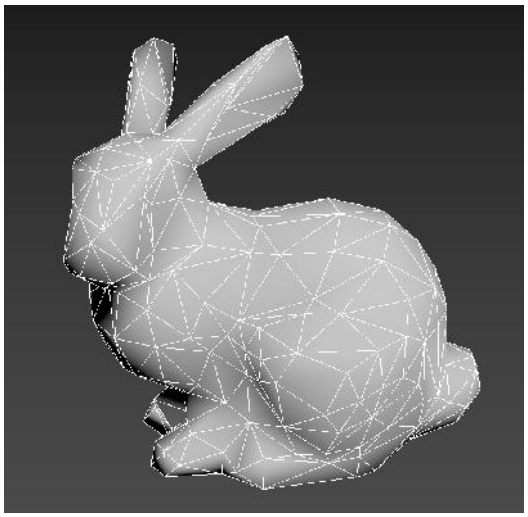

Рисунок 1.6 – Полігональна модель

Альтернативою трикутникам є нелінійні поверхні, які мають певну кривизну, або сплайни. На основі кожного трикутника можна сформувати криву поверхню, яка буде дотикатися вершин трикутника, а також буде перпендикулярна у цих вершинах нормалям трикутника [15]. У такому разі полігональна модель буде нарисована без розривів, перегибів та з цілком плавною формою, як наведено на рисунку 1.7.

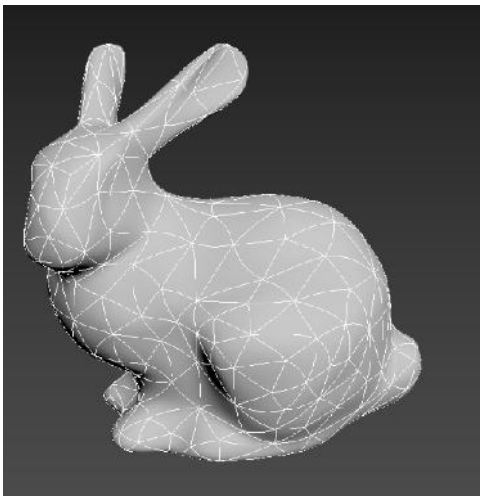

Рисунок 1.7 – Модель зі сплайнів

Але синтез зображення таких примітивів є складним обчсилювальним процесом, тому у сучасних графічних системах такі поверхні апроксимуються великою кількістю полігонів [16]. Це дозволяє отримувати досить реалістичні форми об’єктів, але потребує дуже великої кількості трикутників, що впливає на час рендеру сцени.

1.4 Постановка задачі

У результаті аналізу літератури не виявлено універсального методу рендерингу реалістичного зображення тривимірної сцени у реальному часі. Єдиним методом, спроможним синтезувати реалістичне зображення, є метод зворотнього трасування променю, який сьогодні не використовується у

більшості систем рендерингу реального часу [2]. Такі системи побудовані лише для рендеру прямим методом полігональних моделей, що значно обмежує як можливі форми об’єктів, так і реалістичність ефектів освітлення.

Задача даної роботи – промоделювати метод зворотнього трасування та пристосувати до роботи у реальному часі. При цьому необхідно якомога краще забезпечити сумісність з існуючими полігональними моделями. Також слід звернути увагу на нелінійні примітиви та пристосувати їх до роботи з алгоритмами зворотнього трасування.

Для моделювання необхідно створити програмне середовище, яке буде працювати на більшості персональних комп’ютерів. Програма повинна завантажувати полігональні моделі та формувати зображення на їх основі. Користувач матиме змогу рухати віртуальну камеру та роздивлятися модель з будь-яких ракурсів. Частота формування зображень зворотнім методом повинна наближатися до 30 кадрів у секунду, тому необхідно задіяти потужності графічних прискорювачів. Проте, враховуючи їх непристосованість до зворотнього методу трасування, фактична частота кадрів, особливо на великих моделях буде нижчою від 30.

2 МЕТОД СИНТЕЗУ ЗОБРАЖЕНЬ ПОВЕРХОНЬ ЗВОРОТНІМ ТРАСУВАННЯМ

2.1 Загальний опис методу зворотного трасування

Методи рендеру поділяються на два види – прямого та зворотнього трасування. Першим у системах реального часу використали метод прямого трасування, який і досі є основним у більшості випадків. Для формування зображення розраховується промінь світла зі свого початку, тобто формується зображення джерела світла. Але цей алгоритм не враховує зону видимості віртуальної камери та порядок рендеру об’єктів, тому виконується дуже багато зайвих обчислень [1]. Це, у свою чергу, робить розрахунок відбиття проміня світля майже неможливим, що є необхідним для синтезу реалістичного зображення. Схема такого алгоритму приведена на рисунку 2.1.

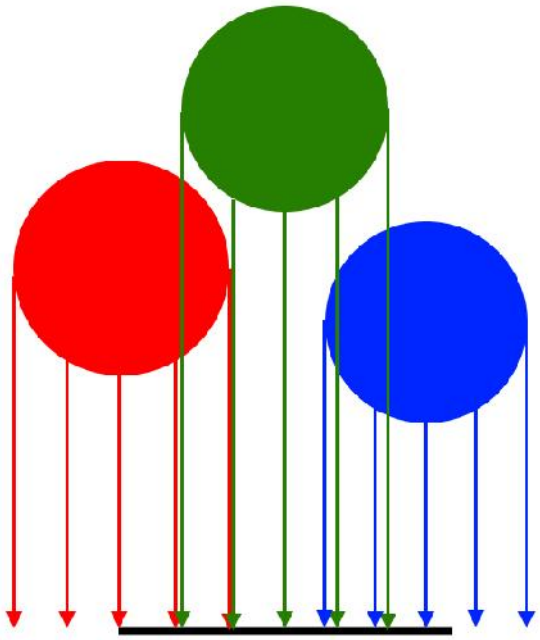

Рисунок 2.1 – Рендер прямим методом

На відміну від прямого трасування, алгоритм зворотнього трасування розраховує шлях світла від спостерігача до поверхні, з якої світло прийшло до нього. Це дозволяє відкидати більшість об’єктів, які не перетнуться з променем світла, а також дозволяє розрахувати шлях променю після відбиття від поверхні [3]. Схема роботи такого алгоритму зображена на рисунку 2.2.

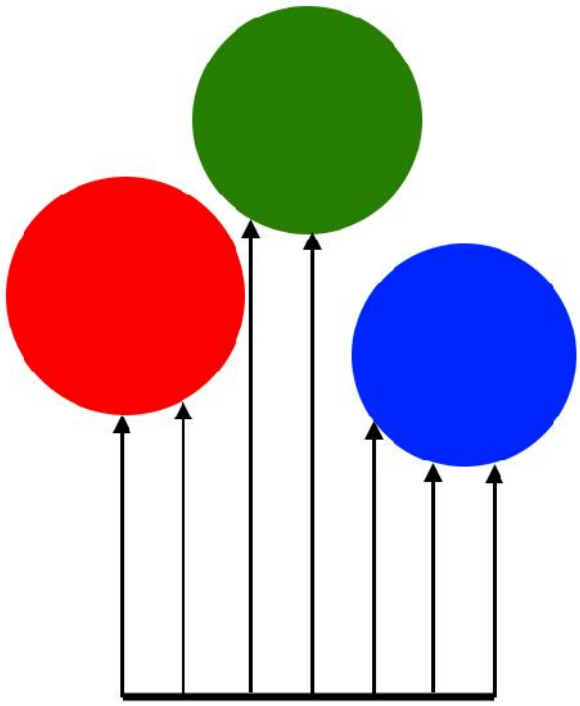

Рисунок 2.2 – Рендер зворотнім методом

Але зворотнє трасування світла, хоча і є зручнішим та логічнішим від прямого методу, воно складніше у реалізації та потребує набагато потужнішу обчислювальне техніку, ніж пряме трасування. Тому для реального часу й досі застосовують прямий метод, адже більшість відеоприскорювачів не здатні синтезувати зображення великих сцен зворотнім методом. Одначе в останні роки все більше елементів освітлення розраховуються зворотнім методом та додаються до алгоритму прямого методу, формуючи гібридні системи, які здатні синтезувати досить реалістичні зображення [4]. Але такі програми зазвичай рахують освітленні з невисокою детализацією та зі

статичними елемнтами, тобто при змінах у сцені потрібен досить важкий перерахунок даних освітлення. Система, яка рахує усе освітлення зворотнім методом, однаково добре працює як зі статичними, так і з анімованими об’єктами.

Основою алгоритму зворотнього трасування є пошук перетину проекційного променю зі сценою [1]. Проекційний промінь утворюється пікселем на екрані та вектором погляду. Вектор погляду – це вектор напрямку, який поєднує центр проекції та піксель. Тож для проекційного променю треба розрахувати точку перетину з примітивами сцени. Сцена зконструйована з великої кількості примітивів, частину яких можна відкинути простішими розрахунками, але деяка кількість все одно залишиться, і для кожного треба розрахувати перетин з промінем та найти ближчу точку перетину до початку проміню. Складність цього процесу, окрім кількості примітивів, залежить від типу примітивів. Для простих примітивів, таких, як трикутник, є аналітичний вираз пошуку перетину з променем. Для складніших примітивів немає аналітичного рішення, тому доводиться використовувати чисельні методи [10].

2.2 Ітераційні алгоритми зворотнього трасування

У загальному випадку графічний примітив не має аналітичного рішення пошуку пертину з променем. Тому за одне обчислення найти перетин неможливо. Для пошуку необхідний чисельний метод, який працює ітераціями [10].

Спочатку визначимось з графічними промітивами. Завдаються вони функцією відстані, яка має аргументом точку в просторі, а повертає вдістань від неї до найближчої точки на поверхні примітиву. Хоча така форма завдання примітиву відрізняється від математичних виразів поверхонь, але вона має просту форму для простих фігур – еліпсів, паралелепіпедів, площин,

а також такі вирази можно легко комбінувати, отримуючи комбінації фігур, та навіть інтерполювати [18].

Ітераційний процес розраховує перетин променю з поверхнею примітива на відрізку. Точність знаходженння пропорційна кількості ітерацій, і, відповідно часу розрахунку [10]. Схема розрахунку наведена на рисунку 2.3.

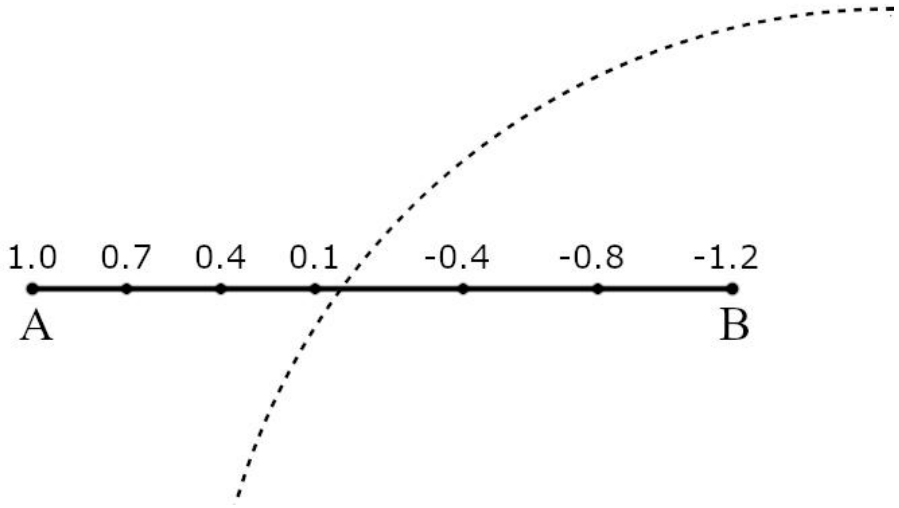

Рисунок 2.3 – Схема ітераційного процесу

Спостерігач знаходиться у точці А. Обирається точка В, до якої потребується знайти перетин. Далі починаються обиратися точки й рахуватися значення функції відстані поверхні. За одну ітерацію рахується одна або декілька точок. Після закінчення ітераційного процессу обирається точка з найменшим значенням відстані до поверхні за модулем. Якщо модуль відстані менше деякого умовно малого порогового значення, ця точка вважається належною поверхні [17]. На рисунку вище точкою з найменшою відстанню за модулем є точка зі значенням 0,1. Якщо точність 0,1 є допустимою для користувача, то цю точку можна вважати належною поверхні фігури, зображеною пунктиром.

Метод вибору точок істотньо впливає на точність та витрачений час на розрахунок. Найпростіший спосіб – послідовно обирати точки з деяким періодом, покриваючи весь відрізок від спостерігача і до кінця, або коли значення відсатні до поверхні у точці буде необхідного діапазону. Тоді вважатимемо перетин з поверхнею знайденим, і надалі рухатись немає сенсу. Але у такому методі точність прямо попорційна кількості ітерацій, і для задовільних результатів треба розрахувати дуже багато точок, що потребуватиме занадто багато часу.

Інший метод, який буде застосований у даній роботі, метод половинного ділення. У цьому методі за одну ітерацію рахується одна точка, і на основі отриманого значення функції відстані обирається напрямок, де буде розташована точка у наступній ітерації. Якщо функція відстані дала позитивний результат, то точка знаходиться перед поверхнею, наступна точка буде за напрямком проекційного променя. Якщо функція відстані негативна, то точка знаходиться всередині фігури, наступна точка буде ближче до спостерігача. Відстань від поточної точки до наступної не залежить від значень функції відстані да завжди дорівнює половині від відстані кроку попередньої ітерації. На першій ітерації відстань кроку дорівнює чверті довжини відрізку, а перша точка – середина відрізку.

Таким чином, залежність точності від кількості ітерацій є показниковою, що гораздо ефективніше лінійної залежності. Але є особливість ітераційних методів – вони нездатні з декількох точок перетину обрати найближчу до спостерігача за один ітераційний цикл. У такому випадку необхідні додаткові ітераційни цикли.

Для випадку, коли відрізок перетинає один локальний екстремум поверхні у двох місцях, тобто промінь входить у фігуру та виходить з неї, можна допрацювати умову вибору напрямку руху, щоби завжди обирати точку входу у поверхню. Для цього необхідно рахувати для кожної точки вектор градієнту, а потім скалярний добуток цього вектору та напрямку поглядовго променю. Якщо добуток позитивний, то точка ближче до

дальньої точки претину з локаьним екстремумом, якщо добуток негативний, то точка з ближчої сторони екстремума. Тож якщо добуток позитивний, напрямок кроку повинен бути до спостерігача.

Таким чином, алгоритм здатний працювати як з монотонними на відрізку функціями відстані поверхонь, так із одиничними локальними екстремумами, наприклад, перетин промінем сфери. Враховуючи, що алгоритм працюватиме з полігональними моделями, складні поверхні, які мають більше одного локального екстремуму, занадто важкі при обчисленнях та не будуть застосовані. Тому описаного алгоритму достатньо для ітераційного процесу.

2.3 Полігональна модель у ітераційному алгоритмі

У системах реального часу основним примітивом для формування сцени є трикутник [4]. Тож усі тривимірні моделі складаються з трикутників. Для того, щоби синтезувати зображення такої сцени, необхідно з усіх точок перетину проекційного променю на відрізку з полігонами обрати найближчу. Описати одним виразом функцію відстані до декількох трикутників неможливо, тому потрібно розраховувати перетин з трикутниками окремо, а потім обирати найближчу точку до спостерігача. У даній роботі на основі трикутних примітивів формуватиметься нелінійна поверхня, але необхідність перебирати трикутники та сортувати точки перетину залишається.

У найпростішому випадку, коли синтезується зображення трикутників, достатньо знайти точку перетину відрізку та площини трикутника, а потім перевірити належність точки трикутнику. Але якщо синтезована поверхня на основі полігонів нелінійна, тобто перетин з площиною застосувати не можна, необхідно на кожній ітерації знаходити точку на трикутнику, яка відповідатиме точці ітерації, та за допомогою якої можна буде обчислити нелінійну поверхню на основі даних полігона. Для вирішення цієї задачі застосуємо направляючі вектори. Вони зображені на рисунку 2.4 синім

кольором. Червоним кольором показані нормалі у вершинах полігона.

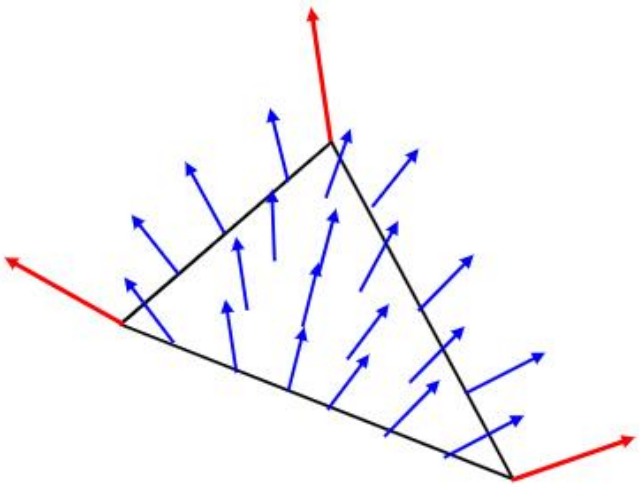

Рисунок 2.4 – Направляючі вектори

Направляючі вектори необхідні для однозначного визначення відповдіної точки на трикутнику для точки у просторі. Кожна точка трикутнику однозначно визначає відповідні точки у просторі. Справедливо й зворотнє твердження – кожна точка у просторі має однозначну відповідну точку на трикутнику. Існуюуть дві роботи [12,13], які описують побудову направляючих векторів. У роботі [12] запропонован алгоритм побудови поля направляючих на базі свойства простого відношення трьох точок афінних перетворень. У роботі [13] запропонований спрощений алгоритм, який потребує менших обчислень. Хоча такий алгоритм має недоліки, зокрема розриви на деяких моделях, але у більшості випадків ці розриви непомітні, особливо при достатній кількості ітерацій. Тож для програмної реалізації у даній роботі була обрана спрощена модель поля направляючих.

Схема знаходження відповідної точки приведена на рисунку 2.5. Полігон – це трикутник ABC. Також у кожній вершині задані вектори нормалі, напрямки яких на рисунку співпдають з відрізками AA’, BB’ та CC’. Точка у просторі P, відповідна точка на трикутнику P’.

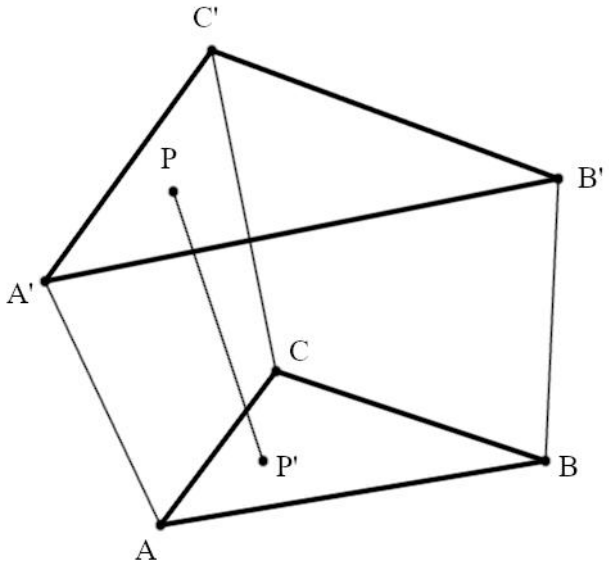

Рисунок 2.5 – Обчислення відповідної точки на трикутнику

Першим кроком необхідно обчислити відстань h від точки у просторі до трикутника. Далі необхідно обчислити скалярні добутки нормалі вершини та нормалі трикутника для кожної вершини. Після цього для кожної вершини відстань h ділиться на відповідний скалярний добуток, отримана величина буде відстанню, на яку треба підняти вершини трикутника уздовж нормалі, довжина відрізків AA’, BB’ та CC’ на малюнку. Таким чином отримуємо трикутник A’B’C’, який паралельний полігону ABC, та який знаходиться на одній площині з точкою P. Далі обчислюємо барицентричні координати за цим трикутником. Після цього за допомогою отриманих координат обчислюємо точку на полігоні ABC, яка і буде шуканою точкою P’ [13]. Якщо хоча би одна барицентрична координата менша одиниці, то точка у просторі не належить цьому полігону, і потрібно переходити до наступної точки або полігону.

Таким чином, можна визначити, для яких полігонів необхідно обчислювати повехню для даної точки. Після обчислення точок перетину відрізка з поверхнями полігонів обирається найближча до спостерігача.

2.4 Трасування поверхні Фонга

Основний примітив у тривимірній графіці реального часу – трикутник. Трикутниками важко передати складні поверхні, тому трикутників доводиться використовувати дуже багато, аби сконструювати округлі деталі. Натомість, якщо застосувати складніший примітив, кривизною якого можна керувати, кількість таких примітивів значно зменшиться, а тривимірна сцена спрощиться. Першою спробою створити подібну поверхню була теселяція Фонга. Хоча вона лише апроксимує поверхню полігонами, розрахунок точки поверхні можна застосувати у ітераційному алгоритмі й синтезувати зображення таких поверхонь без апроксимації.

Розглянемо розрахунок точки поверхні Фонга для заданої точки на полігоні. Схема зображена на рисунку 2.6.

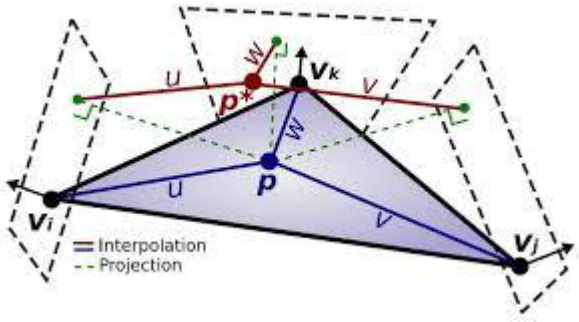

Рисунок 2.6 – Розрахунок точки поверхні Фонга

Спочатку розраховуються барицентричні координати точки на трикутнику. Потім ця точка проектується на поверхні, утворені вершинами та нормалями у них. Отримані проекції помножуються на відповідні барицентричні координати й сумуються. Далі отриману точку необхідно лінійно проінтерполювати з точкою на трикутнику з вагою отриманої точки приблизно 0,65. Формули обчислення цього коефіцієнту немає, значення підбирає користувач, дивлячись на результат. Це є головним недоліком цього

методу [16]. Проінтерпольована точка і буде точкою поверхні Фонга.

Для застосування в ітераційному алгоритмі необхідно знайти відстань від точки у просторі до поверхні [17]. Для цього знайдемо відповідну точку на трикутнику, побудуємо для неї точку поверхні Фонга. У точці поверхні будуємо площину з нормаллю, яка дорівнює направляючому векторові. Результатом функції відстані буде відстань до цієї площини. Хоча це не є абсолютно точним результатом, але точності достатньо для синтезу зображеня поверхні Фонга, аналогічному зображенню тесельованої моделі.

Головним недоліком, як зазначено вище, є неточність. Користувачеві необхідно добирати коефіцієнт, щоби повехня була максимально гладкою. Перевагами методу є простота розрахунку та відсутність додаткових даних, які потрібно було би рахувати окремо перед процесом рендеру.

2.5 Метод сферичної інтерполяції (MCI)

Даний метод [14, 15] потребує додаткових даних для кожного полігона, а саме кривизну опорних сфер. Їх потрібно розрахувати до рендеру. Опорні сфери – це сфери, центри яких знаходяться на нормалі вершини, поверхня сфери проходить через цю вершину та через одну іншу вершину. Тобто сфери дотикаються однієї вершини та проходять через іншу [15]. Для кожної точки трикутника потрібно дві такі сфери, відповідно двом іншим вершинам. На рисунку 2.7 наведено сегменти таких сфер для однієї вершини.

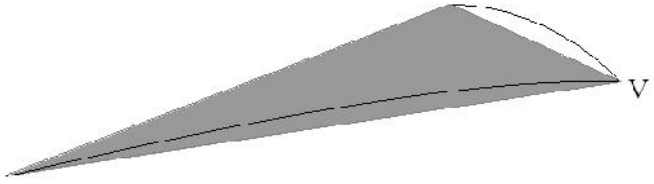

Рисунок 2.7 – Сегменти опорних сфер для однієї вершини

Після розрахунку сфер необхідно отримати кривизну цих сфер. Для цього ділимо один на радіус сфер. Таким чином, до кожного трикутника додається шість величин кривизни сфер, які знадобляться при рендері.

Далі розглянемо формування інтерпольованих сфер для кожної вершини. Для цього достатньо лінійно проінтерполювати кривизну сфер для вершини з деякими коефіцієнтами. Коефіцієнти розраховуються через кути між напрямком з вершини на точку на трикутнику та сторонами трикутника [15]. Розрахунок проілюстровано на рисунку 2.8.

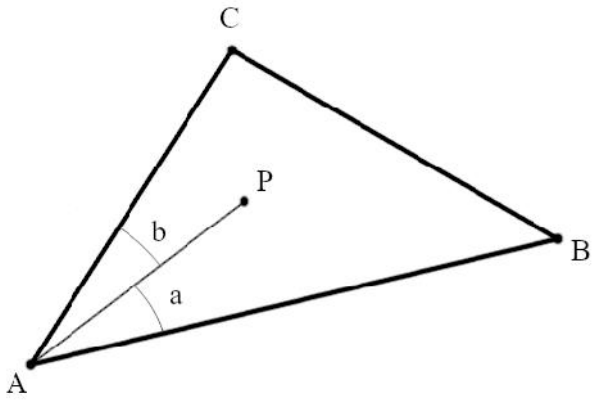

Рисунок 2.8 – Розрахунок коефіцієнтів інтерполяції

Розрахуємо коефіцієнти інтерполяції опорних сфер для вершини A. Точка на трикутнику позанчена P. Знаходимо кути між ребрами та відрізком AP a та b. Тоді коефіцієнт сфери опроної сфери на ребрі AC буде a/(a+b). Відповідно коефіцієнт опорної сфери на ребрі AB буде b/(a+b). Таким чином, сума цих коефіцієнтів завжди одиниця, якщо точка знаходиться на ребрі, то відповідний коефіцієнт буде одиниця, другий нуль [15].

Отримавши коефіцієнти та проінтерполювавши кривизну сфер. Будуємо інтерпольовану сферу. Спочатку отримуємо радіус цієї сфери, який дорівнює одиниці, поділеній на кривизену. Потім рахуємо центр сфери – для цього від радіус-вектора вершини трикутника, для якої рахується сфера, віднімаємо, або додаємо вектор нормалі, помножений на радіус. Віднімаємо у випадку, коли кривизна сфери позитивна, тобто повехня у цій вершині

опукла, в іншому випадку додаємо [15]. Таким чином отримана сфера буде дотикатися вершини та будет перпендикулярна нормалі веришини. Отримавши три таких сфери для трьох вершин, можна рахувати відстань до поверхні.

Нагадаємо, що шукається відстань до поверхні від точки у просторі. Також ми маємо відповідну точку на трикутнику, для якої розраховані інтерпольовані сфери, та барицентричні координати. Далі необхідно для кожної інтерпольованої сфери знайти перетин з відрізком, сформованим точкою у просторі та відповідною точкою на трикутнику, як показано на рисунку 2.9.

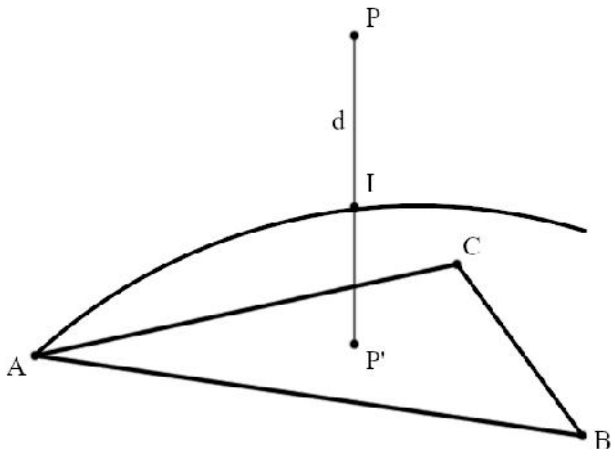

Рисунок 2.9 – Розрахунок відстані до інтерпольованої сфери

Розглянемо інтерпольовану сферу для вершини трикутника A. Точка у просторі, для якої рахується відстань до поверхні, позначена P, відповідна їй точка на трикутнику P’. Відрізок PP’ лежить на направляючій [12]. Необхідно знайти відстань d від точки у просторі P з точкою перетину відрізка PP’, яка позначена I. Аналогічно знаходимо відстані й до інших двох інтерпольованих сфер. Після цього помножуємо відстані на відповідні барицентричні координати та сумуємо, отримуючи кінцеву відстань до поверхні, яка і є нелінійною поверхнею, необхідною для формування зображення полігональної моделі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Аналіз використаних технологій

3.1.1 Специфікація OpenGL

OpenGL – (англ. Open Graphics Library – відкрита графічна бібліотека) – специфікація, що визначає незалежний від мови програмування кросс-платформовий програмний інтерфейс (API) для написання застосунків, що використовують 2D та 3D комп’ютерну графіку. Цей інтерфейс містить понад 250 функцій, які можуть використовуватися для малювання складних тривимірних сцен з простих примітивів. Широко застосовується індустрією комп’ютерних ігор і віртуальної реальності, у графічних інтерфейсах (Compiz, Clutter), при візуалізації наукових даних, в системах автоматизованого проектування тощо [5].

Стандарт OpenGL був розроблений і затверджений у 1992 році передовими компаніями в області розробки апаратного та програмного забезпечення для роботи з комп’ютерною графікою. Метою співпраці було створення апаратно-незалежного інтерфейсу придатного для реалізації на різних платформах. Основою стандарту стала бібліотека IRIS GL, розроблена компанією Silicon Graphics.

Курт Акелей (Kurt Akeley) і Марк Сіґал (Mark Segal) створили оригінальну специфікацію OpenGL. Кріс Фразаєр (Chris Frazier) редагував версію 1.1, а Джон Ліч (Jon Leech) версії з 1.2 по 2.0.

31 липня 2006 року на SIGGRAPH було анонсовано, що контроль за специфікацією OpenGL буде переданий Khronos Group. У серпні 2008 Khronos Group опублікував специфікацію OpenGL 3.0.

На базовому рівні, OpenGL – це всього лише специфікація, тобто, – просто документ, який описує набір функцій та їх точну поведінку. На основі

цих специфікацій виробники апаратного забезпечення створюють реалізації – бібліотеки функцій, які відповідають заявленій в OpenGL специфікації. Ці реалізації проектуються для того, щоби за можливістю забезпечити апаратне прискорення [5]. Коли апаратне прискорення не допускається, виконання функцій здійснюється за допомогою програмного забезпечення. Виробники повинні пройти спеціальні тести на відповідність, перш, ніж їхню реалізацію класифікуватимуть, як реалізацію OpenGL. Таким чином, розробникам програмного забезпечення необхідно всього лиш навчитися використовувати описані у специфікації функції, і лишити їхню реалізацію за розробниками апаратного забезпечення.

Ефективні реалізації OpenGL існують для операційних систем Linux, MacOS X, Microsoft Windows та багатьох UNIX-подібних ОС, а також для таких ігрових боксів, як Sony PlayStation 3. Різні програмні реалізації OpenGL існують для платформ, виробники яких не підтримують дану специфікацію.

OpenGL орієнтується на такі два завдання:

а) сховати складності адаптації різних 3D-прискорювачів, надаючи розробнику єдиний API;

б) приховати відмінності в можливостях апаратних платформ, вимагаючи реалізації відсутньої функціональності за допомогою програмної емуляції.

Основним принципом роботи OpenGL є отримання наборів векторних графічних примітивів у вигляді точок, ліній та багатокутників з наступною математичною обробкою отриманих даних та побудовою растрової картинки на екрані і/або в пам’яті. Векторні трансформації та растеризація виконуються графічним конвеєром (graphics pipeline), який власне являє собою дискретний автомат. Абсолютна більшість команд OpenGL потрапляють в одну з двох груп: або вони додають графічні примітиви на вхід в конвеєр, або конфігурують конвеєр на різне виконання трансформацій [8].

OpenGL є низькорівневим процедурним API, що змушує програміста

диктувати точну послідовність кроків, щоб побудувати результуючу растрову графіку (імперативний підхід). Це є основною відмінністю від дескрипторних підходів, коли вся сцена передається у вигляді структури даних (найчастіше дерева), яке обробляється і будується на екрані. З одного боку, імперативний підхід вимагає від програміста глибокого знання законів тривимірної графіки та математичних моделей, з іншого боку – дає свободу впровадження різних інновацій [5].

Для підтвердження незалежності від мови програмування були розроблені різні варіанти прив’язки (binding) функцій OpenGL або повністю перенесені на інші мови. Одним із прикладів може служити бібліотека Java 3D, яка може використовувати апаратне прискорення OpenGL. Пряма прив’язка функцій реалізована в Lightweight Java Game Library, яка має пряму прив’язку OpenGL для Java. Sun також випустила версію Java OpenGL (JOGL), яка надає пряму прив’язку до Сі-функцій OpenGL, на відміну від Java 3D, яка не має настільки низькорівневої підтримки. Офіційний сайт OpenGL має посилання на прив’язки для мов Java, Фортран 90, Perl, Pike, Python, Ada, Visual Basic і Pascal. Є також варіанти прив’язки OpenGL для мов C++ і C#.

Схема роботи програми, побудованої на основі OpenGL, можна представити конвеєєром [6], я копказано на рисунку 3.1. Він складається з етапів:

а) будування масиву трикутників із вхідних даних (input assembler);

б) вершинний шейдер (vertex shader) – розраховує координати вершин трикутників у просторі екрану, а також інші параметри, які передадуться наступним стадіям;

в) теселяція (tesselation) – розділяє трикутники на велике число малих трикутників, робляси модель деталізованою;

г) геометричний шейдер (geometry shader) – видаляє або добавляє примітиви, також дозволяє збільшити деталізацію;

д) растеризація (rasterization) – знаходить внутрішні до примітивів пікселі та інтерполює повершинні данні;

ж) фрагментний шейдер (fragment shader) – на основі повершиних та глобальних даних обчислює колір пікселю;

з) фінальні піксельні операції (color blending) – виконує кероване змішування розрахованого кольору зі старим занченням кольору пікселю, також виконуються перевірки зед-буферу та бітової маски на сумісність.

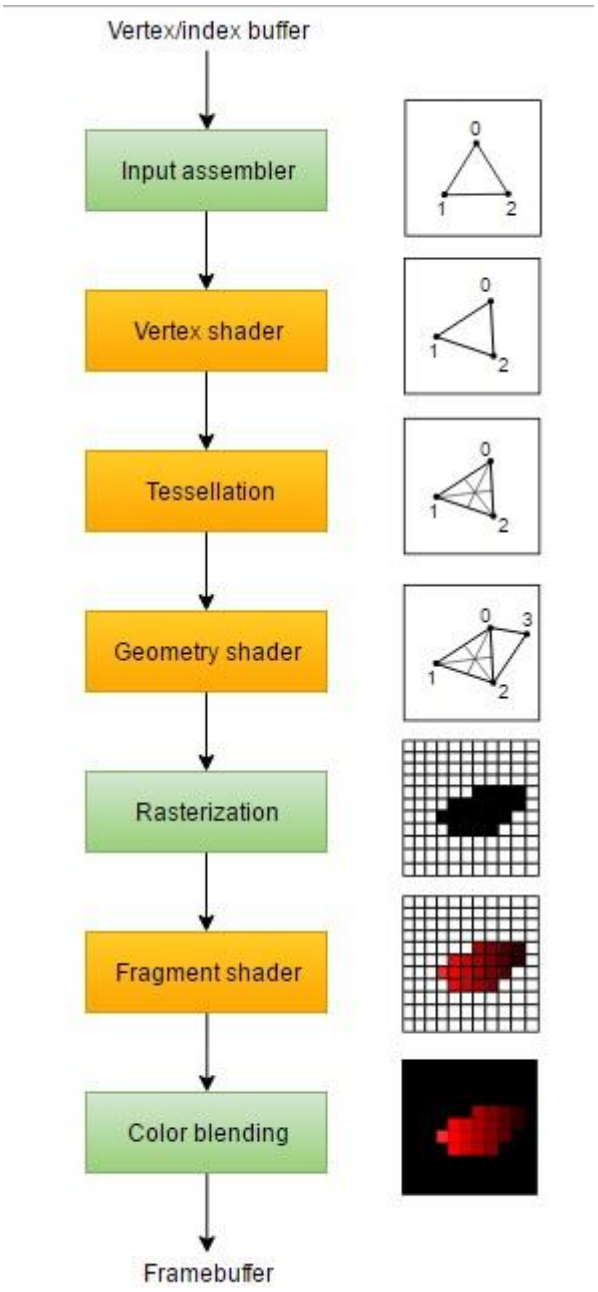

Рисунок 3.1 – Конвеєр OpenGL

3.1.2 Бібліотека WinAPI

Операційною системою для проекту була вибрана Windows, тому для створення додатку необхідно використати бібліотеку WinAPI.

Windows Api (application programming interfaces) – загальне найменування для цілого набору базових функцій інтерфейсів програмування застосунків операційних систем сімейства Windows корпорації Майкрософт [21]. Є найпрямішим способом взаємодії застосунків з Windows. Для створення програм, що використовують Windows API, Майкрософт випускає SDK, який називається Platform SDK і містить документацію, набір бібліотек, утиліт і інших інструментальних засобів.

Windows API був спочатку спроектований для використання в програмах, написаних на мові C (або C++). Робота через Windows API – це найближчий до системи спосіб взаємодії з нею прикладних програм.

Існує чотири версії Windows API:

а) Win16 – перша версія Windows API для 16-розрядних версій Windows. Спочатку називався просто Windows API, потім почав називатися Win16 для розрізнення із Win32;

б) Win32s – підмножина Win32, що встановлюється на сімейство 16-розрядних систем Windows 3.x і реалізовує обмежений набір функцій Win32 API для цих систем;

в) Win32 – 32-розрядний API для сучасних версій Windows. Найпопулярніша нині версія. Базові функції цього API реалізовані в DLL kernel32.dll і advapi32.dll; базові модулі GUI – в user32.dll і gdi32.dll. Win32 з’явився разом з Windows NT і потім був перенесений (у декілька обмеженому вигляді) в системи серії Windows 9x. У сучасних версіях Windows, що походять від Windows NT, роботу Win32 забезпечують два модулі: csrss.exe (client/server Runtime Server Subsystem), що працює в призначеному для користувача режимі, і win32k.sys у режимі ядра;

г) win64 – 64-розрядна версія Win32, що містить додаткові функції для

використання на 64-розрядних комп’ютерах. Win64 API можна знайти тільки в 64-розрядних версіях Windows XP і Windows Server 2003.

У проекті використовується версія Win32.

3.1.3 Мова програмування C++

C++ (Сі-плюс-плюс) – мова програмування високого рівня з підтримкою кількох парадигм програмування: об’єктно-орієнтованої, узагальненої та процедурної. Розроблена Б’ярном Страуструпом (англ. Bjarne Stroustrup) в AT&T Bell Laboratories (Мюррей-Хілл, Нью-Джерсі) 1979 року та початково отримала назву «Сі з класами». Згодом Страуструп перейменував мову на C++ у 1983 р. Базується на мові C. Вперше описана стандартом ISO/IEC 14882:1998, найбільш актуальним же є стандарт ISO/IEC 14882:2017.

У 1990-х роках C++ стала однією з найуживаніших мов програмування загального призначення. Мову використовують для системного програмування, розробки програмного забезпечення, написання драйверів, потужних серверних та клієнтських програм, а також для розробки розважальних програм, наприклад, відеоігор. C++ суттєво вплинула на інші популярні сьогодні мови програмування: C# та Java.

При створенні C++ прагнули зберегти сумісність з мовою C. Більшість програм на C справно працюватимуть і з компілятором C++. C++ має синтаксис, заснований на синтаксисі C [20].

Нововведеннями C++ порівняно з C є:

а) підтримка об’єктно-орієнтованого програмування через класи;

б) підтримка узагальненого програмування через шаблони;

в) доповнення до стандартної бібліотеки;

г) додаткові типи даних;

д) обробка винятків;

ж) простори імен;

з) вбудовані функції;

и) перевантаження операторів;

і) перевантаження імен функцій;

ї) посилання і оператори управління вільно розподіленою пам’яттю.

У 1998 році ратифіковано міжнародний стандарт мови С++: ISO/IEC 14882 «Standard for the C++ Programming Language». Поточна версія цього стандарту – ISO/IEC 14882:2017.

Переваги мови С++:

а) швидкодія. Швидкість роботи програм на С++ практично не поступається програмам на С, хоча програмісти отримали в свої руки нові можливості і нові засоби [20];

б) масштабованість. На мові С++ розробляють програми для найрізноманітніших платформ і систем;

в) можливість роботи на низькому рівні з пам’яттю, адресами, портами. (Що, при необережному використанні, може легко перетворитися на недолік.);

г) можливість створення узагальнених алгоритмів для різних типів даних, їхня спеціалізація, і обчислення на етапі компіляції, з використанням шаблонів;

д) підтримуються різні стилі та технології програмування, включаючи традиційне директивне програмування, ООП, узагальнене програмування, метапрограмування (шаблони, макроси).

Недоліки мови С++:

а) наявність безліч можливостей, що порушують принципи типобезпеки приводить до того, що в С++ програми може легко закрастися важковловима помилка. Замість контролю з боку компілятора розробники вимушені дотримуватися вельми нетривіальних правил кодування. По суті, ці правила обмежують С++ рамками якоїсь безпечнішої підмови. Більшість проблем типобезпеки С++ успадкована від С, але важливу роль в цьому питанні грає і відмова автора мови від ідеї використовувати автоматичне

управління пам’яттю (наприклад, збірку сміття). Так візитною карткою C++ стали вразливості типу «переповнювання буфера» [20];

б) погана підтримка модульності. Підключення інтерфейсу зовнішнього модуля через препроцесорну вставку заголовного файлу (#include) серйозно уповільнює компіляцію, при підключенні великої кількості модулів. Для усунення цього недоліку, багато компіляторів реалізують механізм прекомпіляції заголовних файлів (англ. Precompiled Headers);

в) недостача інформації про типи даних під час компіляції (CTTI);

г) мова C++ є складною для вивчення і для компіляції;

д) деякі перетворення типів неінтуїтивні. Зокрема, операція над беззнаковим і знаковим числами видає беззнаковий результат;

е) препроцесор C++ (успадкований від C) дуже примітивний. Це приводить з одного боку до того, що з його допомогою не можна (або важко) здійснювати деякі завдання метапрограмування, а з іншого, в наслідок своєї примітивності, він часто приводить до помилок і вимагає багато дій з обходу потенційних проблем. Деякі мови програмування (наприклад, Scheme і Nemerle) мають набагато могутніші і безпечніші системи метапрограмування (також звані макросами, але макроси C/C++ вони мало нагадують);

є) з кінця 1990-х в спільноті C++ набуло поширення так зване метапрограмування на базі шаблонів. По суті, воно використовує особливості шаблонів C++ в цілях реалізації на їхній базі інтерпретатора примітивної функціональної мови програмування, що виконується під час компіляції. Сама по собі ця можливість вельми приваблива, але, внаслідк вище згаданого, такий код вельми важко сприймати і зневаджувати. Мови Lisp/Scheme, Nemerle і деякі інші мають могутніші і водночас простіші для сприйняття підсистеми метапрограмування. Крім того, в мові D реалізована порівнянна за потужністю, але значно простіша в застосуванні підсистема шаблонного метапрограмування;

ж) хоча декларується, що C++ мультипарадигмена мова, реально в мові

відсутня підтримка функціонального програмування. Частково, даний пропуск усувається різними бібліотеками (Loki, Boost) що використовують засоби метапрограмування для розширення мови функціональними конструкціями (наприклад, підтримкою лямбд/анонімних методів), але якість подібних рішень значно поступається якості вбудованих у функціональні мови рішень. Такі можливості функціональних мов, як зіставлення зі зразком взагалі украй складно емулювати засобами метапрограмування.

Більшість недоліків мови усувається глибоким освоєнням мови. Також слід враховувати, що усі системні бібліотеки, в тому числі WinAPI та реалізація OpenGL написані на мові C, тому C++ є єдиною високорівневою мовою, яка дозволяє користуватися цими бібліотеками. В інших мовах програмування розробнику доведеться використовувати прив’язки, або біндинги, щоби звертатися до цих бібліотек. Це ускладнює програмування, адже ці біндінги можуть не підтримувати останні версії бібліотек, або не реалізовувати повністю увесь функціонал.

3.1.4 Середовище розробки Microsoft Visual Studio

Microsoft Visual Studio – серія продуктів фірми Майкрософт, які включають інтегроване середовище розробки програмного забезпечення та ряд інших інструментальних засобів. Ці продукти дозволяють розробляти як консольні програми, так і програми з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-застосунки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight.

Visual Studio включає один або декілька з наступних компонентів:

а) Visual Basic .NET, а до його появи – Visual Basic;

б) Visual C++;

в) Visual C#;

г) Visual F# (входить до складу Visual Studio 2010);

д) Visual Studio Debugger.

Багато варіантів постачання також включають:

а) Microsoft SQL Server;

б) MSDE Visual Source Safe – файл-серверна система управління версіями.

Це середовище розробки є комфортним та функціональним. Visual Studio підтримує розфарбовування тексту відповідно мові програмування та спеціальну систему IntelliSense, яка намагається вгадати, шо користувач печатає у документі та надає варіанти автозаповнення слова. Вона розбирає написаний код, і віходячи з нього, пропонує назви доступних змінних та методів, що значно спрощує написання програм. Ще одна корисна функція середовища – постійна фонова компіляція коду, яка дозволяє одразу помітити синтаксичні помилки.

Крім цього, Visual Studio має багато корисних розширень, зокрема, підтримку шейдерної мови GLSL, добавляючи розфарбування та фонову компіляцію шейдерної програми. Також в даному середовищі спрощено керування бібліотеками, і замість скриптових файлів достатньо налаштувати проект через графічний інтерфейс.

Visual Studio Community edition безкоштовна, тому добре підходить для навчальних проектів та проектів, яким не потрібні інтегровані у середовище спеціалізовані сервиси командної розробки й адміністрування від Microsoft.

3.2 Бібліотека об’єктно-орієнтованого графічного API

Обраним графічним програмним інтерфейсом для паралельних обчислень та синтезу зображення є бібліотека OpenGL. Але вона постачається для мови C у структурно-функціональному стилі [5]. У компіляторі мови C++ також можна використовувати цю бібліотеку, але її

стиль написання є незручним для застосування у додатках. Тому першим кроком програмної реалізаціїї у даній роботі є написання об’єктно-орієнтовної бібліотеки для мови C++ на основі бібліотеки OpenGL мови C. У коді вона називається “core”.

Незважаючи на стиль бібліотеки OpenGL, в ній присутня чітка об’єктна ієрархія [5]. Її вирішено перенести у бібліотеку C++, щоби зберегти сумісність з оригінальним інтерфейсом OpenGL на мові C, а також для легкого засвоєння іншими розробниками, які знайомі із OpenGL. Одначе деякі специфічні системні об’єкти скомбіновані для зменшення коду.

Головною метою створення бібліотеки “core” у даній роботі є зменшення коду додатку. Для цього більшість операцій автоматизовано, тобто виклики багатьох функцій OpenGL знаходяться всередині декількох методів класів, що дозволяє користувачу бібліотеки описувати лише ту операцію, яку він хоче виконати над об’єктом, без повторення багатьох однакових системних викликів. Також до скалду бібліотеки додано окрім інтерфейсу OpenGL бібліотека побудови графічного інтерфейсу та модуль для автоматичної загрузки ресурсів програми із файлової системи через пошук файлів ресурсів.

Першим об’єктом, який необхідний розробнику, є контекст OpenGL. Саме цей об’єкт описує сесію використання графічних потужностей комп’ютера та містить у собі усі інші об’єкти та загальну інформацію про графічну систему. У об’єктно-орієнтовному інтерфейсі він поєднаний з контекстом вікна операційної системи, тому користувач, створюючи об’єкт вікна одразу отримує й графічний контекст, готовий до малювання. Клас об’єкту має назву Window, та має бути успадкований користувачем для реалізаціїї реакцій на деякі події. Клас містить наступні методи:

- MakeContextCurrent – обирає вікно для малювання;
- SwapBuffers – завершує формування кадру та настроює другий буфер для малювання у нього наступного, буфер з першим кадром починає вивдоитися на екран;

- PollEvents – оброблює усі системні повідомлення вікна, фактично, це єдина функція роботи вікна;
- ShouldClose – повідомляє користувачу, що можна завершувати головний цикл додатку та виконання програми взагалі;
- SetShouldClose – дозволяє користувачу запустити вихід з програми зазначеним вище механізмом;
- GetKey – повідомляє статус кнопки клавіатури – натиснута чи ні;
- GetMouseButton – повідомляє статус кнопки миші – натиснута чи ні;
- GetResolution – повідомляє роздільну здатність області вікна, у якій довзволено малювати.

Клас має наступні події, реакцію на які користувач може програмувати:

- OnKey – повідомляє про натискання на кнопку клавіатури;
- OnChar – також виконується при натисканні на кнопку клавіатури, але передає користувачу літеру для набирання тексту;
- OnMouseBtn – повідомляє про натискання на кнопку миші;
- OnCursorPos – повідомляє про переміщення курсору;
- OnCursorEnter – повідомляє про надходження курсору до вікна або покидання;
- OnScroll – виконується при прокрутці, зазвичай, це прокрутка колеса миші;
- OnResize – виконується при зміні розмурі вікна.

Наступний об’єкт, необхідний користувачу – буфер даних. Цей об’єкт відповідає за зберігання даних, які будуть використовуватися графічним конвеєром. Реалізован за допомогою декількох класів.

Абстрактний клас BufferBase описує будь-який буфер даних. Він дозволяє передавати буфер до шейдерних програм, але не дозволяє створювати та заповнювати інформацією. Містить наступні методи:

- Bind – обрати буфер для застосування;
- Unbind – відмінити вибір буферу;
- BindBase – обрати буфер для загального використання у шейдерній

програмі;

- GetElementSize – розмір одного елемента даних у байтах;
- GetStructSize – розмір структури даних буферу;
- GetSize – кількість структур даних у буфері.

Наступний клас – Buffer<T>. Він будується шаблоном та успадковує клас BufferBase, доповнюючи його операціями створення буферу та оновлення інформацією. Містить наступні методи:

- bMap – відображає усі дані буферу на вказівник в оперативній пам’яті центрального процесору;
- bUnmap – закунчує відображення, оновлюючи дані буферу даними зі вказівника;
- bMapRange – аналогічний метод до bMap, але відображає деякий проміжок даних;
- bFlushRange – дозволяє штучно оновлювати деякий діапазон відображеного буферу зі вказівника;
- bSetData – завантажує дані зі вказівника до буферу;
- bGetData – завантажує дані з буферу до вказівника;
- bClear – заповнює весь буфер однаковими даними;
- bClearRange – заповнює діапазон буферу однаковими даними.

Наступний об’єкт – вертексний масив. Він поєднує у собі всі буфери з вертексними даними, формуючи усю необхідну інформацію для рендеру однієї тривимірної моделі. Клас називається VertexArray та містить наступні методи:

- Bind, Unbind – обирає та відміняє вибір масиву;
- IsIndexed – повідмоляє, чи формуються примітиви за допомогою заданих користувачем індексів;
- bEnableVertexAttribArray – вмикає використання буфера вертексних даних за вказаним номером;
- bDisableVertexAttribArray – вимикає використання буфера вертексних даних за вказаним номером;

- bDraw – малює фігуру, описану цим масивом даних;
- bAttachBuffer – додає до масиву буфер вертексних даних за вказаним номером;
- bAttachElementBuffer – додає до масиву буфер з індексами.

Користувачеві доступні декілька специфічних класів моделей, які спрощують створення фігур:

- MeshBase – базовий клас моделі, який містить вертексний масив та метод для малювання;
- ModelMesh – клас статичної моделі, що містить буфери для координат точок, нормалей, тангент, кольорів та текстурних координат;
- ScreenQuadMesh – клас чотирикутника, що описує площину екрану. Містить буфери для текстурних координат, координат кутів проєктивного кубу.

Для створення та використання шейдерних програм існують класи ShaderStage, ShaderProgram та Material. Клас ShaderStage містить скомпільований код однієї стадії шейдерної програми. Він при створенні компілює вказаний користувачем код та не містить ніяких методів чи полів.

Клас ShaderProgram описує одну шейдерну програму, яку можна використовувати при рендері. Він містить наступні методи:

- Bind, Unbind – обирає та відміняє вибір шейдерної програми;
- AttachStage – приєднує скомпільовану стадію;
- DetachStage – від’єднує стадію;
- Link – компонує програму з приєднаних стадій;
- Set – установлює значення глобальної змінної;
- SetTexture – установлює текстуру;
- SetBuffer – установлює буфер загального використання;
- CreateMaterial – створює матеріал зі схемою, відовідною до параметрів шейдерної програми.

Клас Material є допоміжним до шейдерної програми та дозволяє зберігати значення глобальних змінних по текстовому імені змінної.

Окрім функціоналу OpenGL, у бібліотеку інтегровані деякі інші допоміжні бібліотеки, які застосовуються майже у будь-яких додатках з тривимірною графікою.

GLFW – це бібліотека для стфорення вікна з контекстом OpenGL. Окрім створення вона дозволяє керувати вікном та отримувати системні повідомлення щодо вводу даних – натиснуті кнопки, перміщення курсору миші, осі джойстиків.

Assimp – бібліотека дозволяє імпортувати велику кількість файлів з тривимірними моделями. Вона підтримує усі популярні формати, довзоляючи обробляти їх через єдиний програмний інтерфейс.

stb_image – дозволяє завантажувати зображення із поширених форматів файлів зображень, зокрема png, jpeg, bmp.

3.3 Середовище виконання шейдерних програм

Для максимального прискорення алгоритми зворотнього трасування реалізовані як шейдерні програми. Для віконання таикх програм потрібно середовище, яке завантажуватиме код програм, компілюватиме його та виконуватиме. Також середовище відвповідає за завантаження та підготовку полігональних моделей, які будуть зображені. Окрім того, середовище обчислює дані щодо віртуальний камери, якою керує користувач та має змогу таким чином роздивлятися модель.

Основою для створення середовиша є ооб’єктно-орієнтована бібліотека «core». Для роботи середовища знадобиться базовий клас застосунка Application, об’єкт Camera, завантежені шейдери (клас ShaderProgram) й тривмірні моделі (Model).

При старті додатку біліотека автоматично знайде усі ресурси, файли яких підтримуються. Тож необхідно знайти необхідні шейдерні програми за ім’ям (Bake та Raytrace) та усі моделі, які знайдені (зберігаються у списку models). Також створюється об’єкт камери, який знаходиться у змінній cam.

Данні полігональної моделі для застосування у алгоритмах вдірзіняються від даних, які зберігаються. Структура, яка описує трикутник для алгоритмів зворотнього трасування, приведена у прикладі 3.1.

```
struct Triangle {
      vec4 Position[3];
      vec4 Normal[3];
      vec4 TriNormal;
      vec4 NormDot;
      vec4 Domain;
      vec4 VertexData[3];
};
```

Приклад 3.1 – Структура трикутника

Поле Poisitoin містить координати вершин. Поле Normal містить нормалі у вершинах. TriNormal – це нормаль площини трикутника. NormDot– це скалярні добутки номалі у вершинах та нормалі трикутника, зберігаються відповідно у компонентах xyz вектору. Doamin – це сфера максимальної кривизни, центр якої знаходиться на площні трикутника у центрі описуючого кола. Радіус цієї сфери дорівню радіусу описуючого кола. Необхідна для попереднього тесту на точку перетину з проекційним променем. VertexData – це додаткові дані для кожної вершини: x та y – кривизна опороних сфер, z – кут трикутника у радіанах. Розраховуються трикутника у такий формат у шейдерній програмі Bake. Розрахунок відбувається один раз при виборі полігональної моделі.

Функція Update викликається кожен кадр. У ній реалізовано інтерфейс користувача, керування камерою та рендер зображення (виконання шейдерної програми Raytrace).

Керування камерою виконується кнопками та мишкою. Кнопками камера переміщується, мишкою повертається. Переміщення вприв’язано до горизонтальнох площини та напрямку погляду, як це працює у більшості тривимріних редакторів. Поворот камери відбувається по двох кутах, також аналогічно іншим тривимірним редакторам.

3.4 Реалізація алгоритмів зворотнього трасування

3.4.1 Ітераційний процес

Перший крок алгоритму – обчислення проекційного променю для кожного пікселю. Обчислення зображення на всьому екрані виконуються за допомогою малювання двох трикутників, вершини яких розташовані у кутах екрану. Таким чином можна обчислити у вертексній стадії дані для кожній вершині, після чого вони будуть проінтерпольовані для кожного пікселю. Для цього спочатку розраховуємо обратну матрицю трансформації камери. Матриця трансформації камери ViewProjection перераховує координати з глобальних у нормалізовані координати пристрою. Нормалізованя координати пристрою – це координатна система, прив’язана до екрану. Кути екрану мають координати -1 та 1 по осях x та y, також по глибині координата осі z может приймати значення від -1 до 1, як показано на рисунку 3.2.

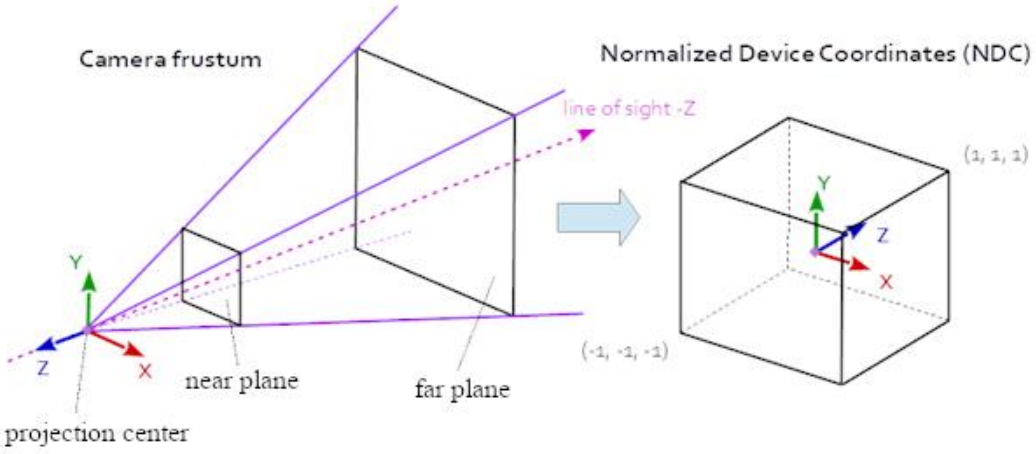

Рисунок 3.2 – Нормалізовані координати пристрою

Нормалізований простій діапазоном (-1; 1) по всіх осях покриває увесь простір, у якому об’єкти будуть нарисовані камерою. Таким чином, проекційний промінь – це напрям від пікселю на ближній площині (z = -1) до пікселю на дальній площині (z = 1), x та z – координати пікселю на екрані у

діапазоні (-1; 1). Нормалізовані координати для кожної вершини складються у вертексний буфер, з додаванням текстурних координат (анаолгічні координатам xy у нормалізованому просторі, але мають діапазон (0; 1)), як наведено на прикладі 3.2.

```
Vertex data[4]{
                {vec2(0, 0), vec3(-1, -1, -1), vec3(-1, -1, 1)},
                {vec2(0, 1), vec3(-1, 1, -1),  vec3(-1, 1, 1)},
                {vec2(1, 0), vec3(1, -1, -1),  vec3(1, -1, 1)},
                {vec2(1, 1), vec3(1, 1, -1),   vec3(1, 1, 1)}
        };
```

Приклад 3.2 – Вертексні дані

У прикладі перший вектор – текстурна координата, далі координата пікселю у ближній площині нормалізованого простору, остання координата пікселю у дальній площині нормалізованого простору.

Наступний крок – розрахунок цих координат у глобальному просторі. Для цього передаємо у шейдерну програму обратну матрицю трансформації камери ViewProjection. В коді вона називається IVP. Далі помножуємо цю матрицю на координати нормалізованого простору, як приведено у прикладі 3.3.

```
layout(location = 0) in vec2 uv;
layout(location = 1) in vec3 nearPos;
layout(location = 2) in vec3 farPos;
out vec3 NearPos;
out vec3 FarPos;
out vec3 LookDir;
out vec2 UV;
uniform dmat4 IVP;
void main()
{
gl_Position = vec4(nearPos.xy,0,1);
dvec4 fPos=IVP * dvec4(farPos,1);
fPos.xyz/=fPos.w;
dvec4 nPos=IVP * dvec4(nearPos,1);
nPos.xyz/=nPos.w;
LookDir=vec3(normalize(fPos.xyz-nPos.xyz));
FarPos=vec3(fPos.xyz);
NearPos=vec3(nPos.xyz);
    UV=uv;
}
```

Приклад 3.3 – Розразунок координат вершин

Окрім координат вершин у ближній да дальній площині (NearPos та FarPos відповідно) також отриман вектор LookDir, який є напрямом проекційного променю. Ці дані будуть проінтерполоьовані, і у наступні стадії кожен піксель отримає свої координати у змінних з такими ж іменами.

У наступній стадії, фрагментній, за допомогою отриманих координат та буферу за даними трикутників, приведених у прикладі 3.1, обчилюється кожен піксель. Саме тут реалізован алгоритм зворотнього трасування. Код ітераційного процесу наведно у прикладі 3.4.

```
      vec3 view = normalize(LookDir);
      Result=mix(vec3(1,0.7,0.5),vec3(0.8,0.9,1),pow(abs(view.y),0.5));
          int density=int(pow(2,Rank)); float cdepth = length(FarPos-NearPos);
          for(int t=0; t<TriCount; ++t)
          {
            Triangle tri = Tris[t];
              vec2 bounds = rsi(NearPos-tri.Domain.xyz,view,tri.Domain.w);
            if(bounds.y < 0 || bounds.x - bounds.y < 0) continue;
            vec3 near=NearPos+view*(bounds.x-bounds.y);
            vec3 far=NearPos+view*(bounds.x+bounds.y);
            float totalLength=length(far-near);
            float deltaLength=max(totalLength/density,minDelta);
            vec3 curDelta=(far-near)/2;
            vec3 cpos=(near+far)/2;
              bool hit=false;
              float curScene=1000;
            vec3 curNorm=vec3(0);
            vec3 weights=vec3(0);
              float dir=0;
              for(int i=1; i<Rank; i++)
              {
                  vec4 t=Field(tri, cpos);
                  curScene=t.a;
                  weights=t.xyz;
      curNorm=tri.Normal[0].xyz*weights.r+tri.Normal[1].xyz*weights.g+tri.Nor
mal[2].xyz*weights.b;
                  curDelta/=2;
                  dir=sign(curScene);
                  dir=min(-sign(dot(curNorm,view)),dir);
                  if(dir==0) dir=1;
                  cpos+=dir*curDelta;
              }
            float depth = distance(NearPos, cpos);
              hit=abs(curScene)<=deltaLength;
              float b = min3(weights);
              if(hit && b>=0 && depth < cdepth && distance(cpos,far) > 1e-3) {
                  float f=Lambert(curNorm,ldir);
                  cdepth=depth;
                  float grid = ShadeGrid(cpos);
                  Result=vec3((0.5+grid*0.25)*f+0.25);
            }}
```

Приклад 3.4 – Ітераційний процес

Напрям проекційного променю нормалізується у змінну view. У колір пікселю Result записується колір фону. У змінну density рахується кількість можливих значень глибини на відрізку ітераційного ходу. Далі йде тіло циклу, де перебираються трикутники. Змінна cdepth зберігатиме глибину точки перетину, тому вона ініціалізована максимальним значенням глибини.

Спочатку рахується перетин проекційного променю зі сферою максимальної кривизни Domain. Якщо перетину немає, або він за зоною видимості камери, пропускаємо ітерацію циклу та йдемо до наступного трикутника. Якщо перетин є, то з кооринат перетину формується відрізок ітераційного ходу (змінні near та far), на якому працюватиме алгоритм. Трикутник вписаний у сферу максимальної кривизни, тому за сферою шукати перетину з поверхнею немає сенсу. Отриманий відрізок ітераційного ходу значно менший від видимого діапазону проекційного променю, тому для однакового результату потребує значно менше ітерацій.

Після формування відрізку ітераційного процесу заповнюються стратові дані. cpos – поточна координата ітераційного процесу, середина відрізку. curDelta – довжина кроку, у коді вона спочатку половина довжини відрізку, але у ітерації вона буде одразу поділена ще на 2, тож загалом вона дорівнює чверті довжини відрізку при старті.

Далі починається цикл ітераційного процесу. Спочатку розраховується функція відстані до поверхні Field(tri, cpos), яка повертає у векторі барицентричні координати трикутника та відстань. За знаком функції відстані визначаэмо напрямок кроку, змінна dir. Далі визначається вектор градієнту. Якщо рахувати градієнт чисельно, небхідно ще 6 разів рахувати відстань до поверхні, що є абсолютно недопустимим. Тому, пам’ятаючи, що вектор градієнту – це вектор нормалі до ізоповерхні, розраховуємо нормаль, інтерполюючи нормалі вершин барицентричними координатами. Результат знаходиться у змінній curNorm.

Наступним кроком рахуємо скалярний добуток отриманої нормалі та напрямку проекційного променю. Якщо він позитивний, то напрямок кроку

стає негативним, аби йти до ближчої поверхні локального екстремуму. В результаті усіх обчислень напрямок може стати нулем, якщо не вистачить точності тридцятидвохбітного числа, тому такий випадок необхідно перевірити та задати позитивний напрямок.

Після визначення напрямку робимо крок – до поточної координати додаємо вектор кроку помножений на напрямок. Вектор кроку ділиться на два.

Після завершення ітераційного циклу перевіряються критерії перетину з поверхнею. Перший критерій – відстань до поверхні менша за модулем від порогового значення. Наступний критерій – кожна барицентрична координата повинна бути позитивна. Також перевіряється глибина, якщо вона більша від попереднього значення, піксель не буде змінено. Останній крок після перевірок – колір фігури, який складається із сітки з простим освітленням на основі нормалей.

3.4.2 Поверхня Фонга

Код функції відстані до поверхні Фонга наведено у прикладі 3.5.

```
vec4 PhongField(Triangle tri, vec3 pos) {
vec3 ppos[3];
    vec3 weights;
float height=Height(pos, tri.Position[0].xyz, tri.TriNormal.xyz);
for(int i=0; i<3; i++)
ppos[i]=tri.Position[i].xyz+tri.Normal[i].xyz/tri.NormDot[i]*height;
    weights=Barycentric(pos,ppos[0],ppos[1],ppos[2]);
vec3 triPos =
tri.Position[0].xyz*weights.x+tri.Position[1].xyz*weights.y+tri.Position[2].x
yz*weights.z;
vec3 dir = normalize(triPos-pos)*sign(height);
for (int i = 0; i < 3; i++) ppos[i] = Project(triPos,
tri.Position[i].xyz, tri.Normal[i].xyz);
vec3 phPos = ppos[0] * weights[0] + ppos[1] * weights[1] + ppos[2] *
weights[2];
phPos = mix(triPos, phPos, 0.65);
return vec4(weights, PlaneDist(pos, dir, phPos, -dir));
}
```

Приклад 3.5 – функція поверхні Фонга

Спочатку розраховується відстань від точки у просторі до площини трикутника (змінна height). Далі кожна вершина трикутника підіймається уздовж нормалі на висоту height, поділену на склаярний добуток нормалі та нормалі трикутника, який було розраховано раніше, при підготовці моделі до рендеру. Підняті вершини знаходяться у змінних ppos. За ними розраховуєтються барицентричні координати визовом функції Barycentric. Далі, вершини трикутника сумуються, помножуючись на барицентричні координати, отримуючи видповідну точку на трикутнику triPos. Далі отримується напрямок dir від точки у просторі до відповідної точки на трикутнику, а також помножується на знак height, аби точки під трикутником мали той же напрямок, що і точки над трикутником.

Отримавши точку на трикутнику, розраховуєму відповідну точку на поверхні Фонга. Проектуємо точку на трикутнику triPos на площині, які візначені вершинами та нормалями у них. Результат складається у змінні ppos. Отримані проекції помножуємо на барицентричні координати та сумуємо. Таким чином отримуємо точку поверхні Фонга (змінна phPos). Її необхідно проінтерполювати з точною на трикутнику з коефіцієнтом 0.65. Останнім кроком будет розрахунок відстані від точки у просторі до площини, утвореною точкою поверхні Фонга та вектором, зворотнім до вектору dir.

Час рендеру одного кадру наведений у таблиці 3.1. Розмір зображення 800 пікслеів у висоту та ширину. На рис. 3.3 зображено модель SphereHigh, побудовану методом Фонга, на рис. 3.4 модель Fresko, на рис. 3.5 модель Teapot.

Таблиця 3.1 – Середній час рендеру зображення поверхні Фонга

Назва моделі	Кількість полігонів	Час, мс
SphereHigh	20	15
Fresko	80	25
Teapot	300	51

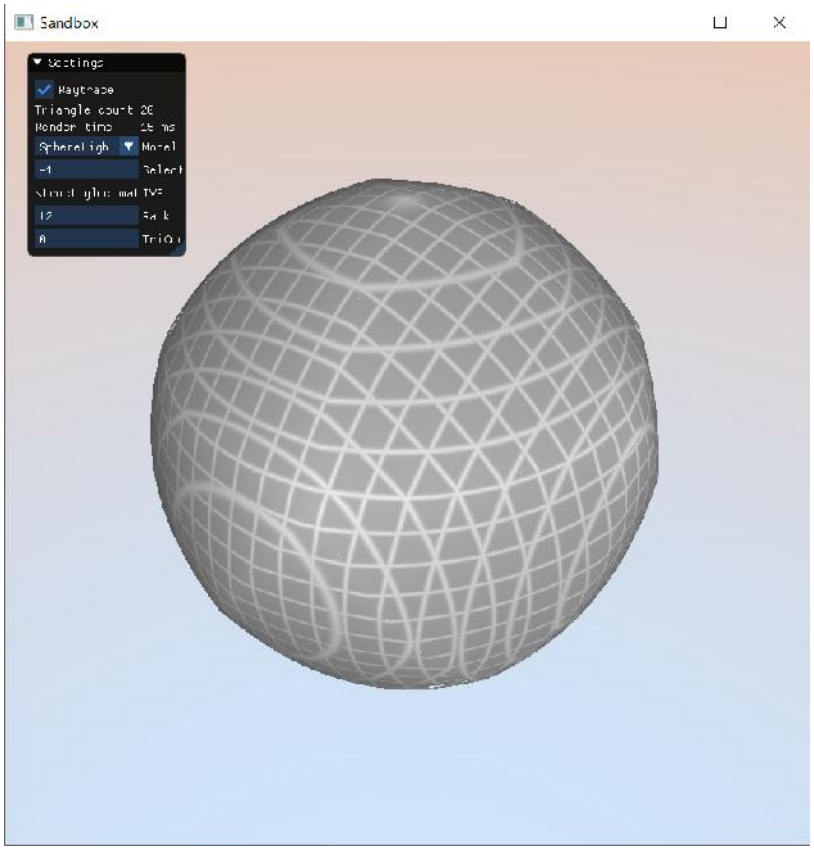

Рисунок 3.3 – Модель SphereHigh

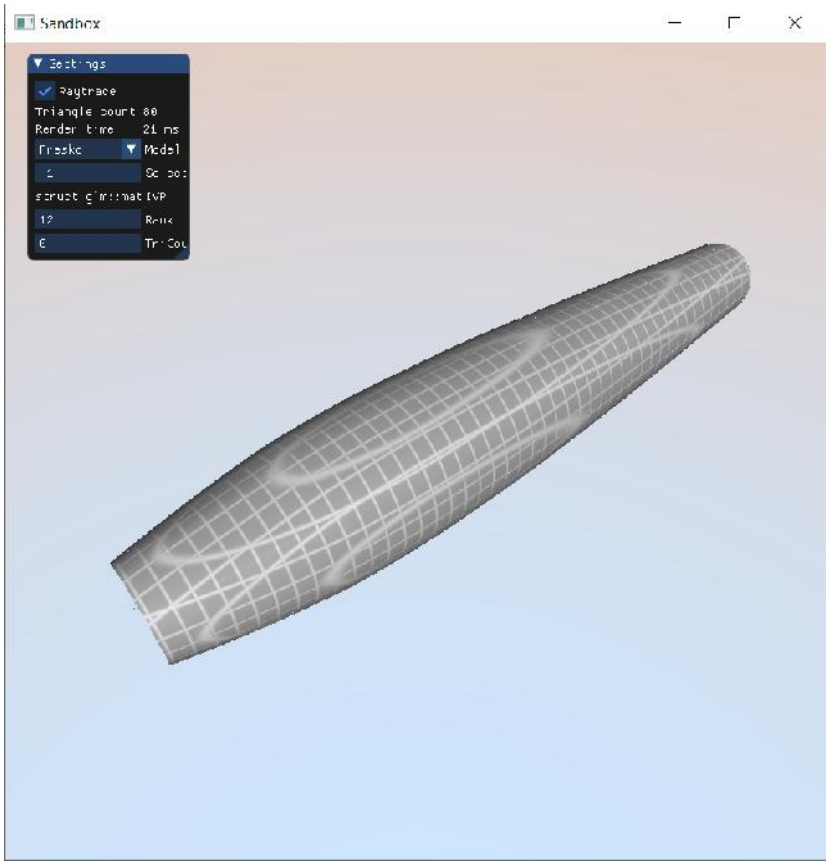

Рисунок 3.4 – Модель Fresko

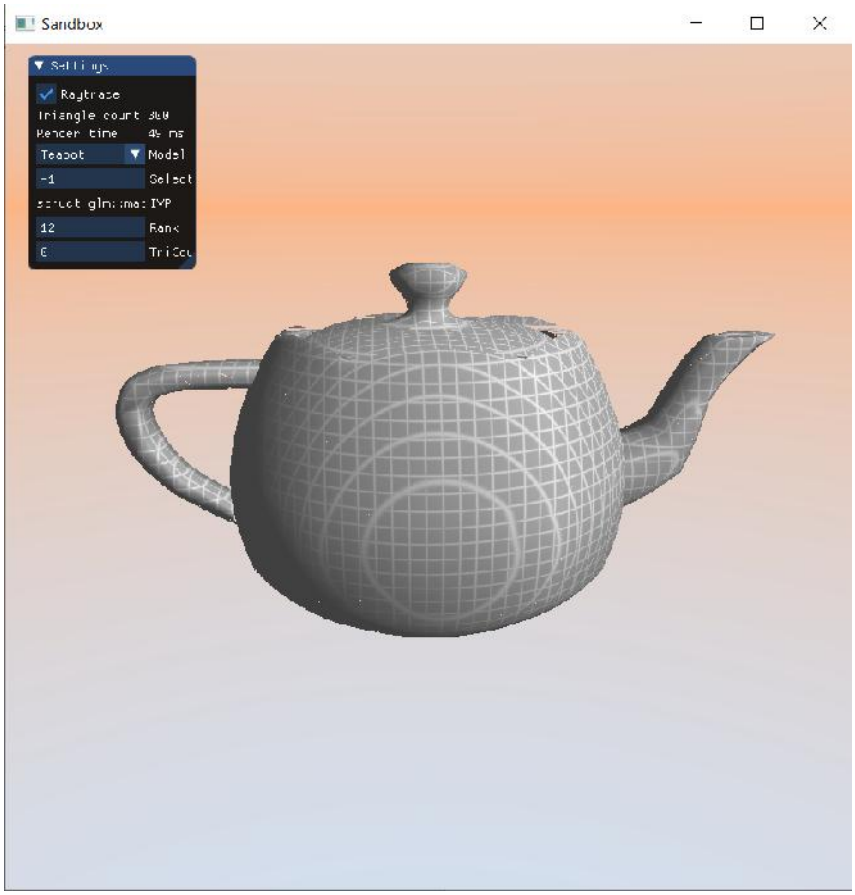

Рисунок 3.5 – Модель Teapot

Побудована поверхня Фонга у деяких випадках порушує плавність поверхні. Це можно виправитти, підбираючи коефіцієнт інтерполяції між поверхнею фонга та площиною трикутника, але це необхідно робити для кожної моделі, що є головним недоліком методу. Проте час рендеру навіть для трьохсот полігонів довзоляє працювати майже у реальному часі.

3.4.3 Поверхня MCI

Код функції відстані до поверхні MCI наведено у прикладі 3.6.

Початок функції аналогічний функції поверхні Фонга – знаходиться відповідна точка на поверхні (triPos) та напрям до неї (dir). Але при розрахунку точки на поверхні барицентричні коорінати обрізаються по одиниці та нормалізуються, щоби ця точка не знаходилася за трикутником. Далі, рахуються коефіцієнти інтерполяції опорних сфер.

```
      vec2 rsi(vec3 r0, vec3 rd, float sr) {
          float a = dot(rd, rd);
          float b = 2.0 * dot(rd, r0);
          float c = dot(r0, r0) - (sr * sr);
          float d = (b*b) - 4.0*a*c;
          if (d < 0.0) return vec2(1e5,-1e5);
      return vec2(-b/(2.0*a), sqrt(d)/(2.0*a));
      }
      float SphereDist(vec3 pos, vec3 dir, vec4 sphere) {
      vec2 r = rsi(pos-sphere.xyz, dir, abs(sphere.w));
      return r.x-sign(sphere.w)*r.y;
      }
      vec4 BlendSpheres(vec3 pos, vec3 norm, float c0, float c1, vec2 w) {
      float c = c0*w.x+c1*w.y;
      c = c>0 ? max(c,1e-6) : min(c,-1e-6);
      float r = 1/c;
      return vec4(pos-norm*r,r);
      }
      float  AngleBetween(vec3  a,  vec3  b)  {  return  acos(dot(normalize(a),
normalize(b))); }
      vec4 PhongField(Triangle tri, vec3 pos) {
      vec3 ppos[3];
          vec3 weights;
      float height=Height(pos, tri.Position[0].xyz, tri.TriNormal.xyz);
      for(int                  i=0;                  i<3;                  i++)
ppos[i]=tri.Position[i].xyz+tri.Normal[i].xyz/tri.NormDot[i]*height;
          weights=Barycentric(pos,ppos[0],ppos[1],ppos[2]);
      vec3 cw = clamp(weights, 0, 1);
      cw /= cw.x + cw.y + cw.z;
      vec3                                triPos                                =
tri.Position[0].xyz*cw.x+tri.Position[1].xyz*cw.y+tri.Position[2].xyz*cw.z;
      vec3 dir = normalize(triPos - pos)*sign(height);
            vec2 w01;
      vec2 w12;
      vec2 w02;
      w01.x     =     clamp(AngleBetween(triPos     -     tri.Position[2].xyz,
tri.Position[1].xyz - tri.Position[2].xyz) / tri.VertexData[2].z,0,1);
      w12.x     =     clamp(AngleBetween(triPos     -     tri.Position[0].xyz,
tri.Position[2].xyz - tri.Position[0].xyz) / tri.VertexData[0].z,0,1);
      w02.x     =     clamp(AngleBetween(triPos     -     tri.Position[1].xyz,
tri.Position[2].xyz - tri.Position[1].xyz) / tri.VertexData[1].z,0,1);
      w01.y = 1 - w01.x;
      w12.y = 1 - w12.x;
      w02.y = 1 - w02.x;
      vec4                                  s0                                  =
BlendSpheres(tri.Position[0].xyz,tri.Normal[0].xyz,tri.VertexData[0].x,tri.Ve
rtexData[0].y,w12);
      vec4                                  s1                                  =
BlendSpheres(tri.Position[1].xyz,tri.Normal[1].xyz,tri.VertexData[1].x,tri.Ve
rtexData[1].y,w02);
      vec4                                  s2                                  =
BlendSpheres(tri.Position[2].xyz,tri.Normal[2].xyz,tri.VertexData[2].x,tri.Ve
rtexData[2].y,w01);
            vec3 dist;
      dist.x = SphereDist(pos, dir, s0);
      dist.y = SphereDist(pos, dir, s1);
      dist.z = SphereDist(pos, dir, s2);
      return vec4(weights, dot(dist,weights));
      }
```

Приклад 3.6 – функція поверхні MCI

Для цього обчилюється відношення кутів між точкою на трикутнику та ребром для однієї сфери, для іншої сфери можна від одиниці відняти коефіцієнт першой сфери, адже їх сума завжди дорівнює одиниці за визначенням.

Коефіцієнти передаються у функцю BlendSpheres, яка повертає центр та радіус інтерпольованої сфери. Окрім коефіцієнтів у функцію передаються позиція, нормаль вершини та дві кривизни опорних сфер. Кривизни опорних сфер помножуються на коефіцієнти та сумуються. Далі одиниця ділиться на утворену кривизну, отримуючи радіус інтерпольованої сфери. Центр інтерпольованої сфери отримується відніманням від віершини вектора нормалі помноженого на радіус.

Отримавши три інтерпольовані сфери для кожної вершини, шукається відстань від точки у просторі, для якої рахується відстань, до кодної сфери уздовж напрямку dir. Для цього написана функція SphereDist. Знайдені відстані помножуються на відповідні барицентричні координати та сумуються, утворюючи відстань до поверхні MCI.

Час рендеру одного кадру наведений у таблиці 3.2. Розмір зображення також 800 пікслеів у висоту та ширину. На рис. 3.6 зображено модель SphereHigh, побудовану методом Фонга, на рис. 3.7 модель Fresko, на рис. 3.8 модель Teapot.

Таблиця 3.2 – Середній час рендеру зображення поверхні MCI

Назва моделі	Кількість полігонів	Час, мс
SphereHigh	20	29
Fresko	80	39
Teapot	300	61

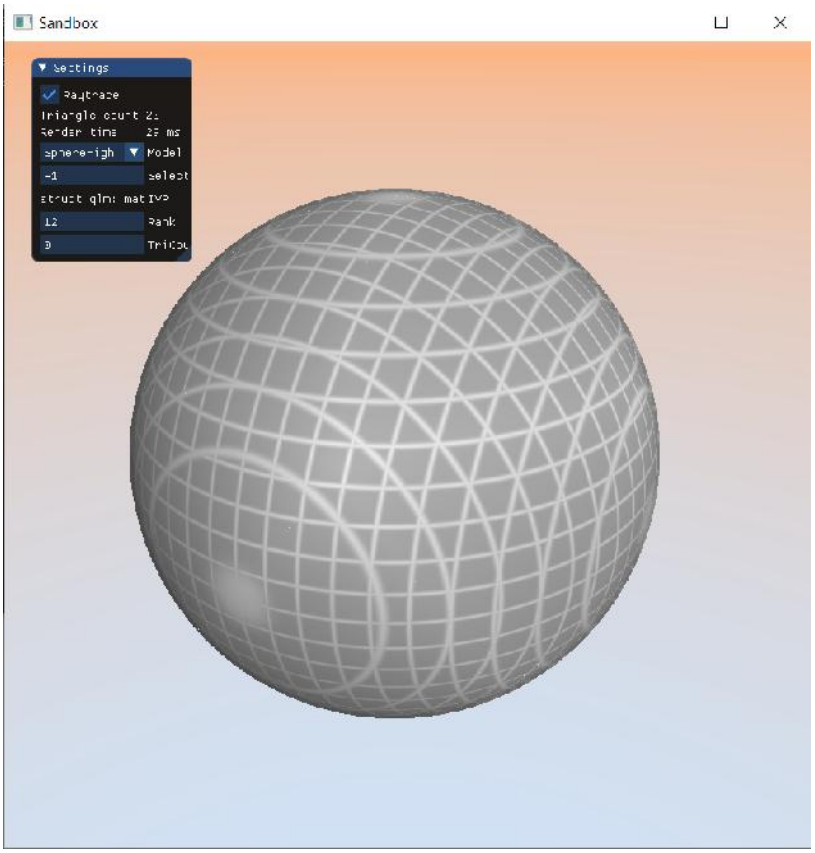

Рисунок 3.6 – Модель SphereHigh

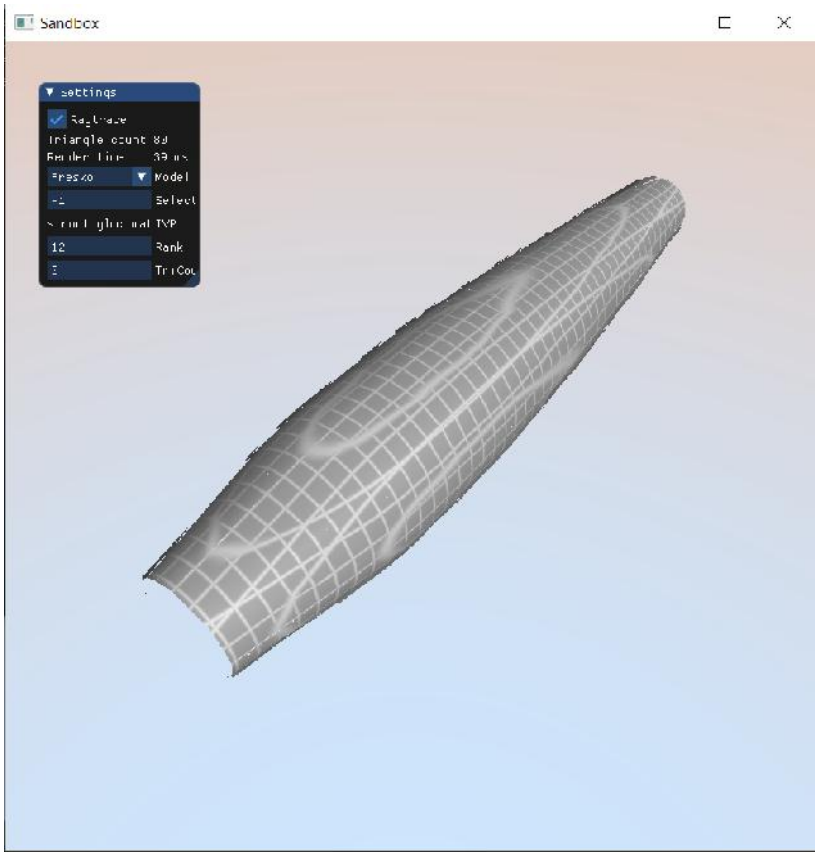

Рисунок 3.7 – Модель Fresko

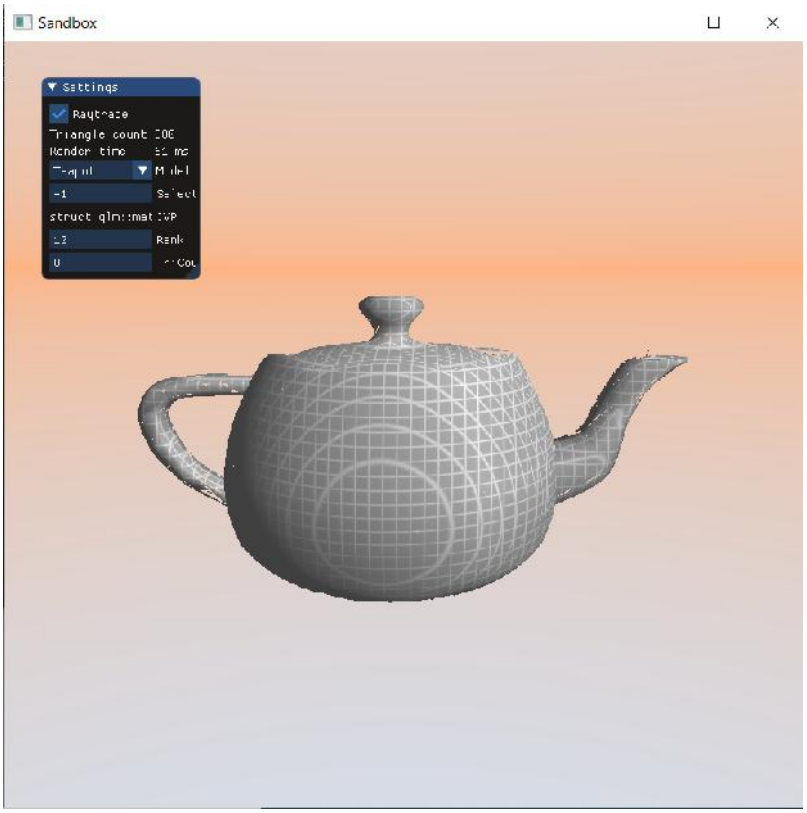

Рисунок 3.8 – Модель Teapot

Поверхня MCI значно краще передає плавність поверхні, особливо у дтелаях зі сферичною кривизною. Проте час рендеру кадру також збільшився.

3.5 Інструкція користувачу

Для роботи програма потребує операційну систему Windows та підтримку у системі специфікації OpenGL 3.4.

Програма є вікном. Розмір вікна змінюється аналогічно усім вікнам системи Windows, відповідно змінюється й навантаження при ренері сцени. Програма шукає та завантажує усі моделі у локальній дирекотрії та вкладених папках. Тож, щоби додати модель, небхідно експортувати її у поширений формат (найпоширеніші fbx та obj) та покласти поряд з самою програмою.

Після старту програма знаходиться у полігональному режимі, коли

модель рисується прямим методом через традиційний графічний конвеєр. Цей режим довзволяє перемістити камеру у зручну позицію, а також має багадо інформації для налагодження.

Щоби рухати камеру, треба натискати кнопки WASD для пермеіщення у горизонтальній площині та Shift і Space для руху по вертикалі. Для повороту камери треба натиснути праву кнопку миші та рухати курсор, тоді камера буде повертатися по двом кутам.

Усі елементи графічного інтерфейсу розташовані у вкладеному вікні Settings, яке наведено на рисунку 3.9.

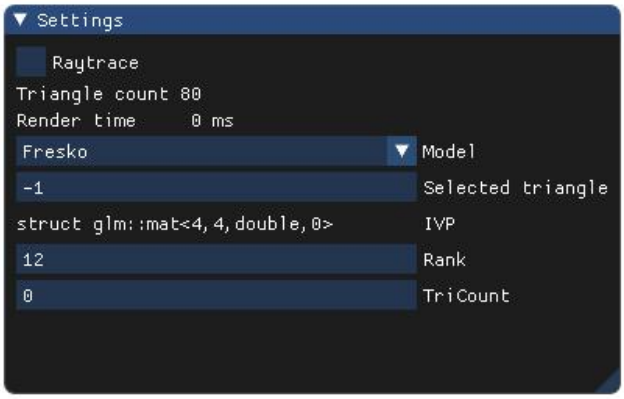

Рисунок 3.9 – Графічний інтерфейс користувача

Вікно інтерфейсу можна сгортати та змінювати розмір, аналогічно системним вікнам. Першим елементом інтерфейсу є кнопка Raytrace. Вона вмикає та вимикає синтез зображення зворотнім методом. Далі йдуть написи з кількістю полігонів у поточній моделі та часу рендеру кадру. Поле Model дозволяє зі списку обрати модель.Поле Select при вводі числа більшого або рівному нулю ізолює в полігональному режимі трикутник з таким номером, що спрощує пошук та налагодження візуальних артефактів. Поле Rank задає кількість ітерацій алгоритму. Інші поля задаються програмно та користувач не може з ними взаємодіяти.

ВИСНОВКИ

Були проаналізовані інсуючі методи формування зображення у комп’ютерній графіці. Розглянені методи офлайн графіки та методи для роботи у реальному часі. Також розглянені алгоритми прямого та зворотнього трасування променю.

Найкраще адаптованим до сучасних обчислювальних пристроїв виявився метод прямого трасування поверхонь, описаних трикутниками. Цей алгоритм має апаратну підтримку у кожному сучасному графічному процесорі та здатний зображувати досить складні полігональні сцени у реальному часі. Проте таким чином неможливо зобразити об’ємні явища та нелінійні поверхні, також розрахунок реалсітичної освітленості потребує багато обчислювальних реусрсів й працює на невеликих сценах.

Найреалістичнішим є метод зворотнього трасування, який дозволяє працювати зі значно ширшим асортиментом примітивів. Розрахунок освітленості у цьому алгоритмі ефективніший та однаково добре працює з будь-якими поверхнями, дозволяючи природним чином отримати багато реалістичних ефектів: тінь, відбиття світла, фонове освітлення із затіненням. Також даний метод добре паралелиться, що дозволяє значно прискорити його роботу за допомогою відеодаптерів. Одначе велика кількість розрахунків, навіть із прискоренням, не дозволяли застосувати подібний алгоритм у системах реального часу.

Враховуючи реалізм методу зворотнього трасування, та обмеженість поверхонь, описаних трикутними примітивами, сформулювано задачу по адаптації методу зворотнього трасування для праці у реальному часі та формування зображень нелінійних поверхонь, якими значно проще описати органічні об’єкти.

Для роботи у реальному часі був обраний ітераційний метод зворотнього трасування, який може працювати зі складними примітивами. У

ролі примітивів реалізовано два види поверхонь – поверхня Фонга та поверхня MCI (методу сферичної інтерполяції).

Реалізовані алгоритми при маленьких полігональних моделях можуть працювати у реальному часі, проте на великих моделях, в яких є тисячі і більше трикутників, частота рендеру може падати менше одного кадру в секунду, тому конкурувати з традиційними методами рендеру нові алгоримти на сучасному обладнанні ще не спроможні. Нелінійна поверхня примітивів дозволяє спростити моделювання об’єктів та використовувати загалом менше трикутників. Хоча сам процес моделювання також змінився й потребує моделювати нормалі, на відміну від звичайних моделей, де нормалі автоматично генеруються.

Поверхня Фонга дозволяє формувати плавну поверхню у більшості випадків, проте на деяких полігонах плавність порушується. Також у алгоритмі цієї поверхні є коефіцієнт, який потрібно добирати, що також зменшує універсальність методу. Але поверхня досить проста в обчисленнях та дає змоуг працювати у реальномі часі досить досить складним сценам. Натомість поверхня MCI значно краще забезпечує плавність поверхні, без необхідності добирати будь-яку коефіцієнти, але й обчислення складніші, тому на складних сценах достатньої частоти кадрів не буде досягнуто.

Подальшим розвитком проекту є позбавлення від візуальних аретфактів, які можна зустріти у поточній версії програми. Також важливо додати окремий алгоритм класифікації сцени, який значно ефективніше дозволить відкидати для кожного пікселю примітиви, які не потрібно рендерити. Окрім рисування поверхонь, слід приділити увагу розрахунку освітлення, зокрема, таким ефектам, як відбиття світла, затінення, розсіяння та інші ефекти. Також потрібна реалізація матеріалів, які зможуть досить реалістично передавити характеристики поверхонь.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Computer Graphics: Principles and Practice (3rd Edition) [Текст] / John F. Hughes, Andries van Dam, Morgan McGuire та ін. – Boston. : Addison-Wesley Professional, 2013. – 1264с. – ISBN 0321399528.

2. Matt Pharr. Physically Based Rendering, Third Edition: From Theory to Implementation 3rd Edition [Текст] / Matt Pharr, Wenzel Jakob, Greg Humphreys. – Burlington. : Morgan Kaufmann, 2016. – 1266 с. – ISBN 0128006455.

3. Jeremy Birn. Digital Lighting and Rendering (3rd Edition) (Voices That Matter) [Текст] / Jeremy Birn. – San Francisco. : New Riders, 2013. – 464 с. – ISBN 0321928989.

4. Tomas Akenine-Moller. Real-Time Rendering, Third Edition [Текст] / Tomas Akenine-Moller, Eric Haines, Naty Hoffman. – Boca Raton. : CRC Press, 2008. – 1045 с. – ISBN 1568814240.

5. John M. Kessenich. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 (9th Edition) 9/E [Текст] / John M. Kessenich, Graham Sellers, Dave Shreiner. – Boston. : Addison-Wesley Professional, 2016. – 976 с. – ISBN 0134495497.

6. David Wolff. OpenGL 4 Shading Language Cookbook - Second Edition [Текст] / David Wolff. – Birmingham. : Packt Publishing, 2013. – 394 с. – ISBN 1782167021.

7. Steve Marschner. Fundamentals of Computer Graphics, Fourth Edition [Текст] : учеб. / Steve Marschner, Peter Shirley. – Boca Raton. : CRC Press, 2015. – 748 с. – ISBN 1482229390.

8. Hubert Nguyen. GPU Gems 3 [Текст] / Hubert Nguyen. – Boston. : Addison-Wesley Professional, 2007. – 1008 с. – ISBN 0321515269.

9. Eric Lengyel. Mathematics for 3D Game Programming and Computer Graphics, Third Edition [Текст] / Eric Lengyel. – Boston. : Cengage Learning PTR, 2011. – 576 с. – ISBN 1435458869.

10. Гусятин В.М. Итерационный алгоритм синтеза изображения в растровой графике реального масштаба времени [Текст] / В.М. Гусятин // Радиоэлектроника и информатика. – 1998. – №3. – С. 81–83.

11. Гусятин В.М. Метод уменьшения итераций в алгоритмах синтеза изображений реального масштаба времени [Текст] / В.М. Гусятин // Радиоэлектроника и информатика. – 2000. – №0. – С. 00–00.

12. Гусятин В.М. Векторное поле направляющих в задаче моделирования криволинейных поверхностей методом сферической интерполяции [Текст] / В.М. Гусятин, М.В. Гусятин // «Вимірювальна та обчислювальна техніка в технологічних процесах». –Хм.:ХНУ. – 2012.–№1.– С. 88-92.

13. Гусятин В. М. Интерполяция триангулированной поверхности в задачах синтеза изображений методом обратного трассирования [Текст] / В. М. Гусятин, Р. В. Сорокин // Вестник Нац. техн. ун-та «ХПИ» : сб. науч. тр. Темат. вып. : Информатика и моделирование. – Харьков : НТУ «ХПИ». – 2007. – № 39. – С. 44-48.

14. V. Gusiatin. Ray tracing synthesis of spatial curve images built by the spherical interpolation method [Текст] / V. Gusiatin, M. Gusiatin, O. Mikhal // Eastern-European Journal of Enterprise Technologies. – 2017.– Т. 3, №4.– С. 4-9.

15. V. Gusiatin. Ray tracing synthesis of images of triangulated surfaces smoothed by the spherical interpolation method [Текст] / V. Gusiatin, M. Gusiatin, O. Mikhal // Eastern-European Journal of Enterprise Technologies. – 2018.– Т. 5, №4.– С. 39-47.

16. Tamy Boubekeur. Phong Tesselation [Текст] / Tamy Boubekeur, Marc Alexa // ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2008). – 2008.– Т. 27, №5.

17. Raymarching distance fields [Электронный ресурс] / Режим доступа : www/ URL: http://iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm – 12.06.2008 г. – Загл. с экрана.

18. Modeling with distance functions [Электронный ресурс] / Режим

доступа : www/ URL: http://iquilezles.org/www/articles/distfunctions/distfunctions .htm – 12.06.2008 г. – Загл. с экрана.

19. Modeling with signed distance functions [Электронный ресурс] / Режим доступа : www/ URL: http://iquilezles.org/www/articles/sdfmodeling/sdfmodeling.htm – 24.03.2015 г. – Загл. с экрана.

20. Скотт Мейерс. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14 [Текст] / Скотт Мейерс. – К. : Диалектика, 2016. – 304 с.

21. Джеффри Рихтер. Windows via C/C++. Программирование на языке Visual C++ [Текст] / - Джеффри Рихтер, Кристоф Назар. – СПб. : Русская Редакция, 2009. – 896 с. – ISBN 978-5-388-00205-1.