МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна

Кафедра Комп'ютерні інформаційні технології

«ДО ЗАХИСТУ»

Завідувач кафедри
_____ /В. І. Шинкаренко/

«_____» _____ 20____р.

ДИПЛОМНА РОБОТА

на здобуття освітнього ступеня «магістр»

Галузь знань **12 Інформаційні технології**

Спеціальність **121 Інженерія програмного забезпечення**

Тема **Дослідження часової ефективності відображення комп'ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API**

Theme **Research of time efficiency of display of computer graphics of real time on multicore systems with use of OpenGL and Vulkan API**

Керівник дипломної роботи доц. _____ О. П. Іванов

Нормоконтролер доц. _____ О. С. Куроп'ятник

Студент групи ПЗ1921 _____ І. А. Поліщук

Student Polishchuk Illia

Дніпро – 2020

Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна

Факультет Комп'ютерних технологій і систем кафедра Комп'ютерні інформаційні технології

Спеціальність Інженерія програмного забезпечення

«ЗАТВЕРДЖУЮ»
Завідувач кафедри
_____ проф. Шинкаренко В.І.
(підпис)
«___»_____2020 р.

ЗАВДАННЯ

до дипломної роботи на здобуття ОС Магістр
(освітній ступень)

студента групи (ПЗ1921) 961-М Поліщука Іллі Андрійовича
(номер групи) (ПІБ)

1 Тема дипломної роботи: Дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API затверджена наказом по університету від «10» жовтня 2019 р. № 779ст.

2 Термін подання студентом закінченого проекту «16» грудня 2020 р.

3 Вихідні дані до дипломного проекту_____
_____
_____
_____

4 Зміст пояснювальної записки (перелік питань до розробки) проведення аналізу часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API.

5 Перелік демонстраційного матеріалу презентація дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API, результати експериментів, висновки; відео демонстрація роботи розробленого програмного інструментарію для проведення досліджень.

6. Консультанти (з назвами розділів):

Розділ	Консультант	Підпис, дата	
		завдання видав	завдання прийняв
Техніко-економічні розрахунки	доц. Гненний М.В.		
Охорона праці та безпека в надзвичайних ситуаціях	ст. викладач Музикін М.І.		

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва розділів дипломного проекту	Термін виконання розділів проекту (роботи)	Примітка
1	Вступ	1.09.2020 – 10.09.2020	
2	Огляд літератури	10.09.2020 – 15.09.2020	
3	Постановка задачі, технічне завдання	15.09.2020 – 30.09.2020	
4	Створення тестової програми	01.10.2020 – 15.10.2020	
5	Перші тестування	15.10.2020 – 22.10.2020	30%
6	Аналіз результатів	30.10.2020 – 11.11.2020	
7	Розрахунок економічних показників	11.11.2020 – 14.11.2020	
8	Охорона праці	14.11.2020 – 18.11.2020	60%
9	Оформлення пояснювальної записки	18.11.2020 – 01.12.2020	
10	Демонстраційні матеріали	5.11.2020 – 16.12.2020	100%

Дата видачі завдання «10» жовтня 2019 р.

Керівник дипломного проекту _____ Іванов О.П.
(підпис) (ПІБ)

Завдання прийняв до виконання _____ Поліщук І .А.
(підпис) (ПІБ)

РЕФЕРАТ

Об’єктом цього дослідження є відомі програмні інтерфейси OpenGL та Vulkan для роботи з комп’ютерною графікою.

Предметом дослідження є оцінка ефективності впливу збільшення кількості ядер центрального процесору на середній час відображення одного кадру комп’ютерних анімації.

Метою роботи є визначення рівня впливу кількості ядер процесору на час відображення кадрів комп’ютерних анімації з використанням OpenGL та Vulkan.

Результати та їх новизна: дослідження робить внесок у визначення практичної застосовності програмного інтерфейсу Vulkan разом із збільшенням числа процесорних ядер для зменшення часу потрібного на відображення кадрів анімації.

Пояснювальна записка складається зі вступу, 5 розділів, висновків, бібліографічного списку та 4 додатків:

Вступ – описується сутність розробки, її актуальність (3 сторінки).

у першому розділі висвітлено аналіз сучасного стану дослідження проблеми за науковими літературними джерелами також проаналізовано сучасний стану програмно-апаратного забезпечення. Складається з 15 сторінок;

у другому розділі надано обґрунтування експериментального методу дослідження. Складається з 16 сторінок;

у третьому розділі представлене проектування й розробка інструментального забезпечення для дослідження. Складається з 20 сторінок;

у четвертому розділі описано виконані дослідження. Складається з 14 сторінок;

у п‘ятому розділі розкриті питання охорона та безпеки праці. Складається з 8 сторінок;

додатки містять технічне завдання й робочий проект

Таблиць – 15, рисунків –21, бібліографія – 74 джерела.

Ключові слова: графіка реального часу, Vulkan, OpenGL, анімації, багатоядерні системи.

ЗМІСТ

ВСТУП

Передові досягнення науки і техніки в області комп’ютерної анімації, такі як імітація руху або відбиття світла загалом є дуже критичними до часу виконання завдань, які на них покладаються. Задачі, які вирішують більшість систем відображення комп’ютерної графіки є задачами так званого жорсткого реального часу (коли перевищення часу виконання поставлених завдань може призвести до невідворотних наслідків) або м’якого реального часу (коли перевищення часу вирішення небажане, але припустиме).

Візуалізація в реальному часі відноситься до швидкого зображень на комп'ютері. Це найбільш інтерактивна область комп’ютерної графіки. На екрані з'являється зображення, глядач діє або реагує, і цей відгук впливає на те, що генерується далі. Цей цикл реакції та візуалізації відбувається з досить швидкою швидкістю, щоб глядач не бачив окремі зображення, а навпаки, занурювався в динамічний процес.

Саме тому доцільно звернути увагу на можливість розподілу таких задач на декілька ядер процесору за для зменшення загального часу обробки кожного кадру анімації перед його відображенням.

Для виводу графіки на екран використовують центральний процесор та графічний процесор. Центральний процесор обчислює позицію, зміщення, форму об’єктів та передає ці дані до графічного процесору. Графічний процесор у свою чергу перетворює дані таких об’єктів у форму придатну для подальшого виведення на екран.

Актуальність роботи. Історично склалося, що центральний процесор та графічний процесор розвиваються окремо один від одного. Тому зараз, коли центральний процесор досяг успіху у паралельному виконанні задач на декількох ядрах, підходи до обробки та передачі даних на графічний процесор не є ефективними.

Один з таких підходів є використання OpenGL – програмного інтерфейсу до графічного пристрою. Цей інтерфейс розроблявся у 90-их років, коли багатоядерні процесори ще не були досить популярними. Тому методи до обробки

та передачі даних які запроваджує цей інтерфейс не дає змогу робити це паралельно, оскільки будь-яка операція з OpenGL може змінювати його глобальний стан, який не передбачає одночасні зміни і не запроваджує синхронізації щоб захиститися від стана гонитви (ситуація в якій робота чи результат операції залежить від послідовності або тривалості виконання).

Щоб виправити такі недоліки може бути використаний інтерфейс Vulkan. Цей інтерфейс не використовує глобальних об'єктів, які неможливо синхронізувати. Натомість робота з ним є складніша, оскільки об'єкти, які були сховані у OpenGL, відтепер мають бути створені та використані розробником.

Одним із таких об'єктів є командний буфер. Командний буфер це буфер, який зберігає заздалегідь заповнені команди які має виконати графічний процесор. Кожен командний буфер може використовуватись незалежно один від одного, тому кожен потік може обчислювати інформацію об'єктів та заповнювати такі буфери окремо. У кінці, ці буфери мають бути відправлені до графічного процесору, де з них буде створене зображення.

Об'єкт дослідження. Об'єктом цього дослідження є відомі програмні інтерфейси OpenGL та Vulkan для роботи з комп'ютерною графікою.

Предмет дослідження. Предметом дослідження є оцінка ефективності впливу збільшення кількості ядер центрального процесору на середній час відображення одного кадру комп'ютерних анімації.

Мета і завдання дослідження. В процесі дослідження буде з'ясовано як саме буде змінюватись ефективність відображення анімації використанням OpenGL та Vulkan. А саме:

- які можливості дає Vulkan з обробки об'єктів анімації паралельно на процесорі;
- які методи синхронізації мають бути використані за для такої паралельної обробки;
- залежність ефективності від кількості об'єктів анімації;
- залежність ефективності від кількості ядер процесору.

Наукова новизна. Набуло подальшого розвитку дослідження що до впливу використання декількох ядер центрального процесору на продуктивність відображення графіки реального часу. Інші дослідження зосереджують увагу на порівнянні різниці використання одного та усіх ядер процесору. У цьому досліджені додатково виявлено вплив поступового збільшення кількості ядер, які використовує операційна система у цілому. Також виявлено залежність цього впливу від складності тривимірного об’єкту.

Практичне значення. Практичне значення роботи викладене наступним чином. Результат проведених досліджень дозволить оцінити який вплив дає збільшення кількості ядер процесору на відображення графіки реального часу та в подальшому може бути основою для нових досліджень в даній сфері.

Апробація результатів дослідження. Процес та результати дослідницької роботи доповідались на семінарі кафедри КІТ 09.10.2020р, а також було опубліковано тези доповідей до щорічної міжнародної науково-практичній конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті».

Публікації за темою роботи. Підготовлено статтю «Дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API».

1 АНАЛІЗ ПРОБЛЕМИ МАСШТАБУВАННЯ ОБРОБКИ АНІМАЦІЇ НА БАГАТОЯДЕРНИХ ПРОЦЕСОРАХ

1.1 Історія розвитку методів програмування комп’ютерної графіки

Ранні відеокарти практично не мали пов'язаного з ними коду, що викликається, поняття "драйвери" ще не зовсім стало реальністю. Існувала концепція Video BIOS, яка була розширенням відеосервісів INT 10h BIOS, які фактично обмежувались ініціалізацією та перемиканням режимів відео [1].

Натомість усі графічні карти, принаймні в DOS, мали відображену пам’ять оперативної пам’яті, і була доступна велика документація про те, як саме встановлення різних бітів в оперативній пам’яті вплине на пікселі, що з’являлися на екрані. Не було API для малювання для виклику, якщо ви хочете, щоб щось з’явилося на екрані (будь то піксель, символ, рядок, коло, тощо), ви б написали код для переміщення байтів вправо місця в оперативній пам'яті. Були написані цілі книги про те, як правильно писати код для малювання графіки.

Відеопам'ять була відображена з сегментом A0000h. В реальному режимі середовища DOS ви можете просто сформувати покажчик на цей сегмент і обробити його як 64000-байтний масив, кожен байт відповідає одному пікселю. Встановіть для байта нове значення, піксель змінює колір на екрані. Крім того, реалізація будь-яких функцій малювання вищого рівня залежить від вас або від сторонніх бібліотек [1].

Існували деякі системи, такі як графічний інтерфейс Borland, які абстрагували примітиви графічного малювання в API з різними драйверами, які можна було викликати, щоб малювати речі на різних графічних картах. Однак, як правило, вони працювали повільніше, ніж потрібно для побудови ігор або анімацій.

Дія гри, як правило, оптимізується для певного режиму графічного відображення на певній карті. Наприклад, популярним режимом відображення був VGA 640x480 з 16 кольорами. Це буде вказано у вимогах до програмного забезпечення, і вам потрібно мати відповідне обладнання для підтримки гри. Якщо

ви придбали гру VGA, але у вас була лише карта EGA, то гра не працювала б взагалі.

У 80-х і на початку 90-х вам потрібно було зробити все на процесорі, а потім використовувати API відеокарти для показу 2D-зображення [1].

1.2 Огляд старих програмних інтерфейсів комп’ютерної графіки

На заміну старим методам програмування комп’ютерної графіки прийшли програмні інтерфейсі OpenGL та DirectX. На відміну від MS-DOS, ці інтерфейси не надають прямого доступу до пам’яті відеокарти, що негативно впливає на швидкість роботи з нею. Але натомість ці інтерфейси пропонують стандартизовані методи роботи з 2Д та 3Д графікою. Завдяки такій стандартизації, розробники відеокарт здобули можливість розробити алгоритми обробки зображення які працюють на відеочіпі, а не на процесорі. І розробники мають змогу використовувати одні і тіж виклики до цих інтерфейсів, щоб використовувати їх різні реалізації на від різних постачальників відеокарт. Це надало можливість розробляти такі програми, які будуть працювати на усіх відеочипах, до яких постачальник розробив реалізацію того чи іншого стандартизованого програмного інтерфейсу.

1.2.1 Огляд інтерфейсу OpenGL

OpenGL – це інтерфейс прикладного програмування, коротше API, який є просто бібліотекою програмного забезпечення для доступу до функцій графічного обладнання. Містить понад 500 різних команд, які використовуються для вказівки об'єктів, зображень та операції, необхідних для створення інтерактивних тривимірних комп’ютерно-графічних додатків.

OpenGL розроблений як спрощений, апаратно-незалежний інтерфейс, який можна реалізувати на багатьох різних типів графічних апаратних систем або повністю в програмному забезпеченні (якщо жодного графічного обладнання немає в системі).

Як такий, OpenGL не включає функції для виконання завдань вікна або обробки вводу користувача; натомість, потрібно буде використовувати засоби, передбачені операційною системою, де додаток буде виконано [2].

Так само OpenGL не надає жодних функцій для опису моделей тривимірних об’єктів або операції зчитування файлів зображень (наприклад, файли JPEG). Натомість, ви повинні побудувати свої тривимірні об’єкти з невеликого набору геометричних примітивів: точок, лінії та трикутників.

З тих пір, як OpenGL існує деякий час - він був вперше розроблений у Silicon Graphics Computer Systems, з версією 1.0, випущеною в липні 1994 р. - існує також безліч бібліотек програм, побудованих з використанням OpenGL для спрощення розробки додатків, наприклад написання відеоігри, створення візуалізації для наукових чи медичних цілей [2].

OpenGL реалізований як система клієнт-сервер, додаток розглядається як клієнт та реалізація OpenGL, надана виробником вашої комп’ютерної графіки обладнання, як сервер. У деяких реалізаціях OpenGL (наприклад, пов'язаних з X Window System), клієнт і сервер можуть виконуватися на різних машинах, підключених до мережі. У таких випадках клієнт видає команди OpenGL, які будуть перетворені в специфічний протокол для віконної системи, який передається на сервер через їх спільну мережу, де вони виконуються для отримання кінцевого зображення [2].

У більшості сучасних реалізацій для реалізації більшості функції OpenGL використовується апаратний графічний прискорювач встановлений на окремій платі та підключений до материнської плати комп’ютера. У будь-якому випадку це так розумно вважати клієнта своїм додатком, а сервер графічним прискорювачем (відеокартою).

1.2.2 Огляд інтерфейсу DirectX

DirectX – це колекція інтерфейсів прикладного програмування (API) корпорації Майкрософт, призначена для надання розробникам низькорівневого інтерфейсу до апаратного забезпечення ПК, на якому працюють операційні

системи на базі Windows. Кожен компонент забезпечує доступ до різних аспектів апаратного забезпечення, включаючи графіку, звук, обчислювальні технології GPU загального призначення та пристрої введення через стандартний інтерфейс.

Наявність одного стандартного API, якого повинні дотримуватися виробники обладнання, є набагато зручнішим, ніж написання шляхів коду для всіх можливих пристроїв на ринку, тим більше, що нові пристрої, випущені після доставки гри, можуть не розпізнаватися грою, тоді як використання стандартних рішень вирішують цю проблему. DirectX – це сукупність API, що використовуються в основному розробниками відеоігор для вирішення цієї потреби у стандартизації на платформах Windows та Xbox [3].

DirectX 11 є скоріше додатковим оновленням DirectX 10.1, а не основним оновленням DirectX 10, як було з DirectX 9. Microsoft ризикнула, почавши оновлення з Direct3D 10 і вимагаючи не лише нового обладнання, але і Windows Vista як мінімальну вимогу. Це було кілька років тому, і на сьогодні його використовують оскільки не тільки широко розповсюджена апаратна підтримка, але й більшість користувачів Windows зараз охоплюють Windows Vista та Windows 7. DirectX завжди враховував майбутнє, а також скільки років потрібно, щоб розробляти ігри наступного покоління, DirectX 11 буде дуже важливим для ігор на довгі роки [3].

Суперечки Direct3D проти OpenGL часто можуть здаватися релігійними, але справа в тому, що протягом багатьох років OpenGL відставав від Direct3D. Microsoft зробила велику роботу, розвиваючи Direct3D і вдосконалюючи її протягом багатьох років, але OpenGL лише відставав, не дотримуючись своїх обіцянок, оскільки кожна нова версія виходить, і раз за разом розробники страждали від тих самих проблем минулих років. Коли вперше було оголошено OpenGL 3.0, вважалося, що OpenGL нарешті повернеться в позицію, щоб конкурувати з Direct3D. На жаль, світ OpenGL пережив свою частку злетів і падінь, як у самій групі, що стояла за ним, так і через те, як API конкурував з Direct3D, а Direct3D продовжував домінувати [3].

1.3 Недоліки старих програмних інтерфейсів комп’ютерної графіки

Історично склалося, що центральний процесор та графічний процесор розвиваються окремо один від одного. Тому зараз, коли центральний процесор досяг успіху у паралельному виконанні задач на декількох ядрах, підходи до обробки та передачі даних на графічний процесор не є ефективними.

Старі інтерфейси розроблялися у 90-ті роки, коли багатоядерні процесори ще не були досить популярними. Тому методи до обробки та передачі даних які запроваджує цей інтерфейс не дає змогу робити це паралельно, оскільки будь-яка операція з ними може змінювати його глобальний стан, який не передбачає одночасні зміни і не запроваджує синхронізації щоб захиститися від стана гонитви (ситуація в якій робота чи результат операції залежить від послідовності або тривалості виконання).

Щоб виправити такі недоліки може були створені нові графічні програмні інтерфейси.

1.4 Огляд нових програмних інтерфейсів комп’ютерної графіки

Нові програмні інтерфейси для роботи з комп’ютерною графікою були створені з урахуванням того, що розробники повинні мати можливість розподіляти роботу по обробці та передачі даних на графічний процесор. Тому вони є більш складними, адже усією роботою з пам’яттю повинні керувати програмісти, які розробляють додатки з використанням цих інтерфейсів. Це зроблено для того, щоб користувачі мали можливість розподіляти роботу з пам’яттю графічного чипу по багатьом потокам, та синхронізувати доступ до цієї пам’яті максимально ефективно.

1.4.1 Огляд інтерфейсу DirectX12

DirectX 12 представляє наступну версію Direct3D - 3D-графічний API на основі DirectX. Direct3D 12 швидший та ефективніший, ніж будь-яка попередня версія. Direct3D 12 забезпечує розширені сцени, більше об’єктів, більш складні ефекти та повне використання сучасного обладнання GPU [4].

Direct3D 12 унікальний тим, що забезпечує нижчий рівень апаратної абстракції, ніж у попередніх версіях, що дозволяє значно покращити багатоядерне масштабування процесора вашого додатка. З одного боку, з Direct3D 12 ваш додаток відповідає за власне управління пам’яттю. Крім того, за допомогою Direct3D 12 ваші додатки та програми отримують вигоду від зменшення накладних витрат на графічний процесор.

Direct3D 12 надає розробникам графіки чотири основні переваги (порівняно з Direct3D 11) [4]:

- значно зменшені накладні витрати на процесор;
- значно знижене енергоспоживання;
- до (приблизно) 20% покращення ефективності графічного процесора;
- крос-платформена розробка для пристрою Windows 10 (ПК, планшет, консоль, мобільний).

Direct3D 12 призначений для використання досвідченими графічними програмістами. Це вимагає значної графічної експертизи та високого рівня тонкої настройки. Direct3D 12 розроблений для повного використання багатопоточності, обережної синхронізації процесора / графічного процесора, а також переходу та повторного використання ресурсів від однієї мети до іншої. Це методи, які вимагають значної кількості навичок програмування на рівні пам'яті [4].

Ще однією перевагою Direct3D 12 є його невеликий розмір API. Є близько 200 функцій; і приблизно одна третина з них роблять усі важкі роботи. Це означає, що ви, як розробник графіки, повинні мати можливість навчитися і освоїти повний набір API, не запам'ятовуючи занадто багато імен API.

Для проекту, який використовує всі переваги Direct3D 12, вам слід розробити спеціально налаштований двигун Direct3D 12 з нуля.

Якщо ви, як розробник графіки, розумієте використання та повторне використання ресурсів у ваших програмах, і ви можете скористатися цим, мінімізуючи завантаження та копіювання, тоді ви можете розробити та налаштувати високоефективний механізм для цих програм. Покращення

продуктивності могло б бути дуже значним, звільнивши час процесора для збільшення продуктивності [4].

Інвестиції у програмування є значними, і вам слід розглянути можливість налагодження та інструментарію проекту з самого початку. Потоки, синхронізація та інші помилки синхронізації можуть бути складними.

1.4.2 Огляд інтерфейсу Metal

Metal був оголошений на Всесвітній конференції розробників (WWDC) 2 червня 2014 року і спочатку був доступний лише на графічних процесорах A7 або новіших версіях. Apple створила нову мову для програмування графічного процесора безпосередньо за допомогою функцій шейдерів. Це мова затінення металів (MSL), заснована на специфікації C ++ 11. Через рік у WWDC [5].

API продовжував розвиватися, і WWDC 2017 представив захоплюючу нову версію API: Metal 2. Metal 2 додає підтримку віртуальної реальності (VR), доповненої реальності (AR) та прискореного машинного навчання (ML) серед багатьох нових функцій. Осінь 2017 принесла нові оновлення для Metal, включаючи блоки зображень, затінення плиток та обмін нитками, які доступні на пристроях iOS на базі чіпа A11 Bionic, який поставляється з першим графічним процесором, коли-небудь розробленим власноруч від Apple [5].

1.4.3 Огляд інтерфейсу Vulkan

Vulkan – це інтерфейс програмування для графіки та обчислювальних пристроїв. Типово пристрій Vulkan складається з процесора та ряду апаратних блоків із фіксованою функцією для прискорення використовуваних операцій в графіці та обчисленні. Процесор в пристрої, як правило, дуже широкий багатопоточний процесор і тому обчислювальна модель у Вулкані значною мірою заснована на паралельних обчисленнях. Пристрій Вулкан також має доступ до пам'яті, яка може надаватися спільно з основним процесором, на якому ваш додаток запущено. Вулкан також дає змогу контролювати виділення та звільнення цієї пам’яті [6].

Vulkan – це явний API. Тобто майже все є вашою відповідальністю. Драйвер – це програмне забезпечення, яке приймає команди та дані, що формують API, і перетворює їх у щось, що може зрозуміти обладнання. У старих API, таких як OpenGL, драйвери відстежували стан багатьох об’єктів, керували пам’яттю та синхронізацією та перевіряли наявність помилок у вашій програмі під час її роботи. Це чудово підходить для розробників, але спалює цінний час процесора, як тільки ваша програма буде налагоджена та працює належним чином. Vulkan вирішує цю проблему, передаючи майже все відстеження стану, синхронізацію та управління пам’яттю в руки розробника додатків та делегуючи перевірку правильності розширенням, які повинні бути включені. Ці розширення погано впливають на продуктивність, тому мають бути відключені у фінальних збірках.

З цих причин Вулкан і дуже багатослівний, і дещо тендітний. Вам потрібно зробити дуже багато парці, щоб Vulkan працював добре, і неправильне використання API часто може призвести до поломки зображення або навіть збою програми, де у старих API ви могли б отримати корисну помилку повідомлення. В обмін на це Vulkan забезпечує більше контролю над пристроєм, чисту модель і набагато вищу продуктивність, ніж API, які вона замінює [6].

Крім того, Vulkan був розроблений як дещо більше, ніж графічний API. Він може бути використаний для різнорідних пристроїв, таких як графічні процесори (GPU), цифрові процесори сигналів (DSP) та устаткування з фіксованою функцією. Поточна редакція Vulkan визначає категорію передачі, яка використовується для копіювання даних навколо; категорія обчислень, яка використовується для запуску шейдерів над обчислюваними робочими навантаженнями; і категорія графіки, що включає растеризацію, примітивну збірку, змішування, глибинні та трафаретні тести та інші функції, які будуть знайомі програмістам графіки [6].

1.5 Постановка задачі

У цій дипломній роботі для порівняння ефективності графічних інтерфейсів на багатоядерних системах буде застосовано старий інтерфейс

OpenGL та новий інтерфейс Vulkan оскільки вони є відкритими, та підтримують усі сучасні операційні системи, включно з операційними системами мобільних пристроїв.

На відміну від OpenGL, Інтерфейс Vulkan не використовує глобальних об’єктів які неможливо синхронізувати.

Одним із таких об’єктів є командний буфер. Командний буфер це буфер, який зберігає заздалегідь заповнені команди які має виконати графічний процесор. Кожен командний буфер може використовуватись незалежно один від одного, тому кожен потік може обчислювати інформацію об’єктів та заповнювати такі буфери окремо. У кінці, ці буфери мають бути відправлені до графічного процесору, де з них буде створене зображення.

У таблиці 1.1 наведені відмінності OpenGL та Vulkan:

Таблиця 1.1 – Відмінності OpenGL та Vulkan

OpenGL	Vulkan
Багатоплатформовий - підтримується на Windows та Linux. Для мобільних пристроїв використовується окрема підмножина специфікації OpenGL	Багатоплатформовий - підтримується на Windows, Linux і мобільних пристроїв використовуючи одну і ту ж саму специфікацією
Перевіряє вхідні данні та повертає коди помилок в обмін на зниження продуктивності.	Не перевіряє вхідні дані. Не дає опису чи гарантій що програма буде працювати і як вона буде працювати
Використовує глобальний стан, який є невеликою машиною станів. Будує напрямок своєї роботи використовуючи надані дані.	Не має глобального стану. Розробник сам має проробити усі деталі для свого додатку
Не дає доступу до керування пам’яттю.	Явний контроль над керуванням пам’яттю.

В процесі дослідження має бути з’ясовано, як саме буде змінюватись ефективність відображення анімації з використанням OpenGL та Vulkan. А саме:

- які можливості дає Vulkan з обробки об’єктів анімації паралельно на декількох ядрах процесору;
- які методи синхронізації мають бути використані за для такої паралельної обробки;
- залежність ефективності від кількості об’єктів анімації;
- залежність ефективності від кількості ядер процесору.

1.6 Огляд літератури

В даній роботі використана технічна література в якій висвітлюються основні теоретичні основи, що були застосовані при розробці відповідних компонентів системи.

Під час виконання огляду літератури розглядалися наступні питання: основні підходи і методології, що використовуються при проектуванні взаємодії додатків з програмними інтерфейсами комп’ютерної графіки, принципи, що застосовуються при розробці сучасних інтерактивних графічних програм, загальні вимоги до системи та їх різновиди, проблеми оптимізації, найбільш розповсюдженні методи, що використовуються у системах.

Існує дуже багато методів проектування відображення об’єктів комп’ютерної графіки наведені в матеріалі [7]. Ця книга ретельно зосереджується на сучасних методиках, що використовуються для створення синтетичних тривимірних зображень за частки секунд. З появою програмованих шейдерів (маленьких програм зі своєю мовою програмування, що виконуються на графічному процесорі) протягом останніх кількох років виникло і розвивалось широке коло нових алгоритмів. У цьому виданні розглядаються сучасні практичні методи візуалізації, що використовуються в іграх та інших додатках, такі як глобальне висвітлення та методи вибракування, а також пропонує багато детальної інформації про тонкі проблеми, такі як невеликі, але важливі відмінності між форматами текстур. Ця книга також не є керівництвом з програмування -

оскільки вона перевищила базовий рівень, вона передбачає, що читач досить добре знайомий із своїм середовищем програмування графіки, щоб мати змогу реалізувати описані методи, не потребуючи покрокових інструкцій. Також представляє міцну теоретичну базу та відповідну математику для галузі інтерактивної комп’ютерної графіки, все у доступному стилі.

Основні підходи та поняття для розробки графічних програм за допомогою Vulkan описані в матеріалі [6]. Усередині є огляд Vulkan на високому рівні, пам'ять та ресурси, черги та команди, бар'єри та буфери пам'яті, презентація, шейдери та конвеєри, графічні конвеєри, креслення, обробка геометрії, обробка фрагментів, синхронізація, запити та багатопрохідність. Також тут можливо знайти пояснення того, що робить кожна функція специфікації Vulkan, наприклад обмеження пристрою (таких як максимальний розмір буфера кадру, кількість байтів у константі push, тощо) та способи їх запиту. Також описано концепції синхронізації в одному з розділів. Це абсолютно важливо для правильної розробки використовуючи багатопоточність. Також надано увагу темі продуктивності та найбільш поширених помилок які можуть на неї негативно впливати.

З ціллю дослідити найпоширеніші підходи до розробки графічних додатків старим підходом був обраний ресурс [2]. Він включає у себе поняття, необхідні для ефективної розробки, такі як:

- мова шейдерів OpenGL (GLSL);
- обробка вершин, команди креслення, примітиви, фрагменти та буфери кадрів;
- керування, завантаження та арбітраж доступу до даних;
- створення більших додатків та їх розгортання на різних платформах;
- розширений рендеринг: імітація світла, художні та нефотореалістичні ефекти;
- запобігання та налагодження помилок;
- стиснення текстур.

Ресурси [9, 10] надають основи практичного використання OpenGL та Vulkan. Вони розкривають теми, такі як налагодження середи розробки, цикл рендерінгу, робота з шейдерами, буфери вершин, додавання текстур, освітлення, тощо. Також ці ресурси мають програмний код до кожної з тем, за яким можна здобути практичні навички розробки графічних додатків.

1.6.1 Огляд графічного конвеєру

Графічний конвеєр візуалізації ініціюється під час виконання операції візуалізації. Операції візуалізації вимагають наявності правильно визначеного об'єкта масиву вершин та пов'язаного об'єкта програми або об'єкта конвеєра програми, який забезпечує шейдери для програмованих етапів конвеєра [8].

Шейдери являють собою визначену користувачем програму, призначену для запуску на якомусь етапі графічного процесора.

Після запуску трубопровід працює в наступному порядку:

1. Обробка вершин:
    a. На кожну вершину, отриману з масивів вершин, діє вершинний шейдер. Кожна вершина потоку обробляється по черзі у вихідну вершину;
    b. Необов’язкові примітивні етапи тесселяції;
    c. Необов’язкова примітивна обробка геометричного шейдеру. Вихід – це послідовність примітивів.
2. Поточна обробка вершин, виходи останнього етапу коригуються або передаються в різні місця.
    a. Зворотній зв'язок про перетворення відбувається тут. Являє собою процес захоплення примітивів, згенерованих кроком (-ами) обробки вершин, запис даних з цих примітивів в об’єкти буфера;

   b. Примітивне відсікання, розділення перспективи та область перегляду перетворюються на віконний простір.
3. Розділення примітивів. Являє собою процес захоплення примітивів, згенерованих кроками обробки вершин, запис даних з цих примітивів в об’єкти буфера.
4. Перетворення сканування та примітивна інтерполяція параметрів, яка генерує ряд фрагментів.
5. Шейдер фрагментів обробляє кожен фрагмент. Кожен фрагмент генерує ряд результатів.
6. Пре-обробка за зразком, включаючи, але не обмежуючись цим:
   a. Тест на ножиці. Являє собою операцію, яка відкидає фрагменти, що потрапляють за межі певної прямокутної частини екрану.
   b. Трафаретний тест. Являє собою операцію, виконану після фрагментного шейдеру. Значення трафарету фрагмента перевіряється щодо значення в поточному буфері трафаретів; якщо тест не вдається, фрагмент вибраковується.
   c. Випробування на глибину. Вихідне значення глибини фрагмента може бути перевірено на глибину зразка, на який записується. Якщо тест не вдається, фрагмент відкидається
   d. Змішування. Приймає кольори фрагментів, що виводяться з шейдера фрагментів, і поєднує їх із кольорами в кольорових буферах, до яких ці вихідні дані відображають

Схему графічного конвеєру проілюстровано на рисунку 1.1

Рисунок 1.1 – Графічний конвеєр

Висновки до розділу 1

В першому розділі були розглянуті питання мети розробки і її доцільності, можливі способи використання і підходи до розроблюваної системи. Питання доцільності розробки розглядалося при огляді існуючих графічних інтерфейсів, та історії програмування комп’ютерної графіки. Були визначені недоліки і переваги старих та нових програмних інтерфейсів розробки комп’ютерної графіки. Було вирішено, що аналіз порівняння старих підходів, та нових багатоядерних підходів буде проводитися з використанням найбільш відкритих та поширених графічних інтерфейсів OpenGL та Vulkan.

Також було оглянуто відмінності між OpenGL та Vulkan, та оглянуто літературу, за допомогою якої стали зрозуміли такі ключові моменти, як робота графічного конвеєру, шейдери, архітектура рендеру.

2 ОБҐРУНТУВАННЯ ЕКСПЕРИМЕНТАЛЬНОГО МЕТОДУ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ АНІМАЦІЙ НА БАГАТОЯДЕРНИХ СИСТЕМАХ

Завдання розробки нових ефективних методів обробки комп’ютерної графіки не втратило своєї актуальності. На теперішній час ЕОМ постійно зазнає значне та прогресуюче зростання потужності. Зокрема збільшується кількість ядер, які можуть обчислювати данні паралельно. Відповідно до їх потужності збільшуються об’єми оброблюваної інформації, зокрема чіткість та кількість моделей комп’ютерної графіки. Чіткість досягається за допомогою збільшення кількості полігонів, тобто графічних примітивів таких як трикутник, з яких складаються тривимірні об’єкти.

Зростання кількості інформації яка може бути оброблена комп’ютером паралельно робить можливим збільшення продуктивності при обчислюванні таких моделей. Зокрема це стосується графічних моделей, які є анімованими. Анімація досягається за рахунок зміни положення у простору частини чи усього об’єкта.

Такі зміни потребують обчислення матриць, які можуть бути застосовані для зміни кута повороту, положення у просторі, чи зміщення окремих вершин одного об’єкта. Анімація скадається з постійну зміну кадрів. Ці обчислення мають бути вирахувані кожен кадр, та можуть бути розпаралелені по багатьом ядрам за для зменшення часу потрібного на відображення одного кадру. Це призводить до того, що одночасно може бути відображено більша кількість анімованих об'єктів не втрачаючи змоги розрізняти плавність у русі. Мінімальна кількість кадрів за секунду які мають бути відображені для досягнення плавності є 24 кадри, які наразі поширені серед майже усіх моделей телевізорів.

2.1 Поняття та види анімаційних моделей

3D-модель – це тривимірна фігура в просторі. Як правило, за основу беруться креслення, фотографії, та докладні описи, на основі яких фахівці створюють віртуальну модель [11].

Створення 3D-моделі об'єкта здійснюється за допомогою 3D-моделювання. На першому етапі 3D-моделювання збирається інформація: ескізи, креслення, фотографії та відео. На основі отриманої інформації 3D-дизайнер створює тривимірну модель у спеціальній комп’ютерній програмі. Після виконання моделі можна буде подивитися на неї з будь-якого кута, збільшити, зменшити, зробити необхідні налаштування. Сама модель готова до подальшого використання - друк на 3D-принтері чи будь-який інший спосіб створення прототипів [11].

Тривимірна модель являє собою безліч вершин (точок), які з'єднуються між собою утворюючи полігони.

Вершина – це точка, з координатами в тривимірній системі. X, Y, Z.

Грань, або ребро - відрізок, який з'єднує дві вершини.

Основним компонентом тривимірної графіки є багатокутник - плоский багатокутник, багато з яких утворюють тривимірну фігуру. Абсолютно будь-яка фігура буде побудована з численних простих фігур (а більшість редакторів використовують трикутники). Чим простіші фігури в комплексі, тим гладшою буде здаватися поверхня 3D-моделі [11].

Набір багатокутників несе інформацію про розмір та форму тривимірної моделі, а обрана текстура дозволяє передавати достовірну інформацію про зовнішній вигляд об’єкта та являє собою зображення на поверхні фігури [11].

Анімація 3d - автоматичне переміщення або трансформація об'єктів у просторі та часі.

Простіше кажучи, раніше потрібно було покадрово малювати пересування кожного персонажа. Тепер досить створити тривимірну модель персонажа, після чого її можна рухати в просторі без додаткових зусиль і перемалювань. Але говорити щось просто, а на ділі - пожвавлення 3d моделі персонажів досить складний процес. Щоб змусити фігурку рухатися, мало мати доступ до комп'ютера і розумним програмами. Потрібно ще й уявляти, як може пересуватися герой, які сили на нього при цьому впливають (не ті, які вищі, а, наприклад, гравітація, сила тертя і опору) [12].

Анімація моделей може бути розділена на наступні види:

2.1.1 По ключовим кадрам: з точки А в точку Б

Створення ключових кадрів це один з найбільш поширених способів створення 3д анімації персонажів. Суть методу полягає ось у чому: на шкалі часу задається кілька головних точок, в яких становище або форма об'єкта змінюється. Аніматор задає потрібні параметри моделі в зазначених кадрах, а «проміжні» стану програма розраховує автоматично [12].

Приклад: Для простоти візьмемо гумовий м`ячик, який вдаряється об землю і відскакує вгору. Щоб відобразити один такий «стрибок», процес потрібно розбити на три етапи: м'ячик у верхній точці - м'ячик на землі - м'ячик знову у верхній точці. По-хорошому слід задати більше ключових кадрів, враховувати купу дрібниць. Як то, що при падінні гумовий корпус розтягується, а при ударі сплющується [12].

2.1.2 Анімація по траєкторії

Не завжди 3d моделі персонажів це люди або тварини. Нашим героєм може бути будь-який об'єкт, наприклад, літаюча камера або НЛО (в загальному все, на що вистачить фантазії). В такому випадку миготіння лампочок і обертання по осі буде недостатньо - не цікаво. А от змусити об'єкт літати по траєкторії, та ще й «відправити» камеру стежити за переміщенням, вчасно наближаючись і віддаляючись [12].

Суть методу полягає в тому, щоб:

- задати точку старту (початок шляху об'єкта);
- позначити траєкторію (шлях, який проробляє об'єкт);
- вказати кінцеву точку (де модель повинна зупинитися).

2.1.3 Анімація в динамічному середовищі

Тривимірну модель може оточувати якась реальність, в якій обов'язково є гравітація, рух повітряних мас і інші види коливань. Все це варто враховувати, щоб анімація персонажа була досить реалістичною [12].

Строго кажучи, анімація в динамічному середовищі - швидше обчислювальна робота з глибоким зануренням в фізичні характеристики об'єктів. Але без усього цього навіть найдетальніше 3d моделювання з опрацюванням текстур не зробить персонажа жвавіше [12].

2.1.4 Motion capture: перетворення фільму в мультик

Технологія захоплення рухів - молода, але дуже популярна. Суть такого способу:

- на актора закріплюються датчики;
- поки актор рухається, камери фіксують положення датчиків;
- їх зміщення обробляє програма і створює рухомий «скелет» з набором ключових кадрів;
- отриманий пакет інформації «обтягається» оболонкою - для цього використовується 3d моделювання персонажів.

В результаті дії героя виходять реалістичними, переконливими, а аніматорам не доводиться боротися з фізикою і згадувати, де той гнеться [12].

2.2 Вибір моделей для дослідження

Вибрані моделі для дослідження були розділені на прості та складні.

У якості простих анімованих моделей була обрана модель 3д куба. Також було застосовано метод анімації по траєкторії. Моделі куба кружляють навколо своєї осі.

У якості складних анімованих моделей були обрані моделі тварин. Також ці моделі є анімовані, застосовуючи методи скелетної анімації по ключовим кадрам

2.3 Алгоритм обчислювання ефективності анімації

Для обчислення ефективності анімації була обрана величина середньої кількості кадрів які були опарацьовані та намальовані за певний період часу.

Формула для обчислення середньої кількості кадрів за секунду наведена далі:

$$AFPS = \frac{N}{S}, \quad (2.1)$$

де AFPS (Average frames per second) – це середня кількість кадрів за секунду,

N – кількість кадрів які були згенеровані за період тестування

S – кількість секунд за які було проведене тестування

2.4 Аналіз програмних платформ для досліду

Були проаналізовані наступні найпоширініші операціїні системи:

2.4.1 Огляд операційної системи Windows

Windows – це сімейство комерційних операційних систем від Microsoft, які використовують графічний інтерфейс користувача.

За даними компанії Net Applications, на червень 2019 року ринкова частка Windows склала 88,33%. [13].

2.4.2 Огляд операційної системи Linux

Linux – це сімейство Unix-подібних операційних систем, заснованих на ядрі Linux, що включає набір утиліт та програм проекту GNU. Як і ядро Linux, системи на його основі, створюються та розповсюджуються відповідно до моделі розробки програмного забезпечення з вільним та відкритим кодом. [14].

Станом на середину 2010-х років системи Linux лідирують на ринках серверів (60%), є такими, що превалюють в дата-центрах підприємств і організацій (згідно Linux Foundation), займають половину ринку вбудованих систем, мають значну частку ринку нетбуків (32 % на 2009 рік). На ринку персональних комп'ютерів Linux стабільно займає 3-е місце (за різними даними, від 1 до 5%). Згідно з дослідженням Goldman Sachs, в цілому, ринкова частка Linux серед електронних пристроїв становить близько 42% [14].

2.4.3 Огляд операційної системи MacOS

За оцінками StatCounter ринкова частка macOS становить близько 18,99%. Найпопулярнішою версією macOS є Catalina (48,98% серед усіх версій macOS), слідом йдуть Mojave (21,43%), High Sierra (13,82%), Sierra (5,88%), El Capitan (4,7

%), Yosemite (3,18%). У macOS використовується ядро XNU, засноване на мікроядрі Mach і містить програмний код, розроблений компанією Apple, а також код з ОС NeXTSTEP і FreeBSD. [15].

2.5 Вибір програмної платформи для досліду

У якості операційної системи на якій проводиться дослід була обрана сучасна та найпоширініша система Microsoft Windows версії 10. Інші операційні системи такі як Linux та MacOS не були розглянуті оскільки вони використовуються здебільшого або як серверні системи, або поширені лише серед багатіїв.

2.6 Аналіз апаратних платформ для досліду

Ключовими апаратними складовими для програм відтворення комп'ютерної графіки є процесор та відеокарта.

Центральний процесор – це електронний блок або інтегральна схема, яка виконує машинні інструкції (код програм), головна частина апаратного забезпечення комп'ютера або програмованого логічного контролера. Іноді називають мікропроцесором або просто процесором [16].

Відеокарта (також звана графічною картою, графічною картою, графічним адаптером або адаптером дисплея) – це карта розширення, яка генерує подачу вихідних зображень на пристрій відображення (наприклад, монітор комп'ютера). Часто вони рекламуються як дискретні або виділені відеокарти, підкреслюючи різницю між ними та інтегрованою графікою. В основі обох - блок обробки графіки (GPU), який є основною частиною фактичних обчислень [17].

2.6.1 Процесори з архітектурою ARM

ARM – це сімейство архітектур скорочених обчислювальних наборів (RISC) для комп'ютерних процесорів, налаштованих для різних середовищ. Arm Holdings розробляє архітектуру та ліцензує її для інших компаній, які розробляють власні продукти, що реалізують одну з цих архітектур - включаючи системи на мікросхемах (SoC) та системи на модулях (SoM), що включають пам’ять, інтерфейси, радіостанції, тощо. Він також розробляє ядра, що реалізують цей

набір інструкцій, та ліцензує ці конструкції для ряду компаній, які включають ці основні конструкції у свої власні продукти [18].

Процесори, що мають архітектуру RISC, як правило, вимагають менше транзисторів, ніж ті, що мають складну архітектуру обчислювальних наборів команд (CISC) (наприклад, процесори x86, що зустрічаються в більшості персональних комп’ютерів), що покращує вартість, енергоспоживання та тепловіддачу. Ці характеристики бажані для легких, портативних пристроїв, що працюють від акумуляторів - включаючи смартфони, ноутбуки та планшетні комп’ютери та інші вбудовані системи, але певною мірою також корисні для серверів та настільних комп’ютерів. Для суперкомп’ютерів, які споживають велику кількість електроенергії, ARM також є енергоефективним рішенням [18].

2.6.2 Процесори з архітектурою x86

x86 – це сімейство архітектур набору команд, спочатку розроблене Intel на базі мікропроцесора Intel 8086 та його варіанту 8088. 8086 був представлений в 1978 році як повністю 16-бітове розширення 8-бітового мікропроцесора 8080 від Intel, з сегментацією пам'яті як рішенням для адресування більшої кількості пам'яті, ніж може бути покрито звичайною 16-бітною адресою. Термін "x86" виник, оскільки імена кількох наступників процесора Intel 8086 закінчуються на "86", включаючи процесори 80186, 80286, 80386 та 80486 [19].

Станом на 2018 рік більшість проданих персональних комп’ютерів та ноутбуків базуються на архітектурі x86, тоді як у мобільних категоріях, таких як смартфони або планшети, переважає ARM; на високому рівні x86 продовжує домінувати в обчислювальних робочих станціях та сегментах хмарних обчислень [19].

2.6.3 Процесори з архітектурою x86-64

Архітектура x86-64 (також відомий як x64, x86_64, AMD64 та Intel 64) – це 64-розрядна версія набору інструкцій x86, вперше випущена в 1999 році. Вона представила два нових режими роботи, 64-розрядний режим та режим сумісності, а також новий 4-рівневий режим пейджингового пошуку [20].

Завдяки 64-розрядному режиму та новому режиму підкачки, він підтримує значно більший обсяг віртуальної та фізичної пам’яті, ніж це було можливо у його 32-розрядних попередників, що дозволяє програмам зберігати більший обсяг даних у пам'яті. x86-64 також розширив регістри загального призначення до 64-розрядних, а також збільшив їх кількість з 8 (деякі з яких мали обмежену або фіксовану функціональність, наприклад для управління стеком) до 16 (повністю загальних), а також пропонує безліч інших удосконалень. . Операції з плаваючою комою підтримуються за допомогою обов’язкових інструкцій, подібних до SSE2, а регістри стилів x87 / MMX, як правило, не використовуються (але все ще доступні навіть у 64-розрядному режимі); натомість використовується набір з 32 векторних регістрів, по 128 біт кожен. (Кожен регістр може зберігати один або два числа з подвійною точністю або від одного до чотирьох одиничних номерів точності або різні цілі цілі формати.) У 64-розрядному режимі інструкції модифіковані для підтримки 64-розрядних операндів та 64-розрядного режиму адресації [20].

2.6.4 Відеокарти AMD Radeon

Radeon – це торгова марка графічних процесорів, оперативної пам'яті і SSD, вироблених підрозділом Radeon Technologies (колишня ATI Technologies) компанії Advanced Micro Devices (AMD). Торгова марка була створена в 2000 році компанією ATI Technologies (у 2006 році поглиненої компанією AMD). Графічні рішення цієї серії прийшли на зміну серії Rage [21].

2.6.5 Відеокарти Nvidia

Nvidia – це американська технологічна компанія, розробник графічних процесорів і систем на чіпі (SoC). Розробки компанії набули поширення в індустрії відеоігор, сфері професійної візуалізації, області високопродуктивних обчислень і автомобільної промисловості, де бортові комп'ютери Nvidia використовуються в якості основи для безпілотних автомобілів [22].

Компанія була заснована в 1993 році. На IV квартал 2018 року було найбільшим в світі виробником PC-сумісної дискретної графіки з часткою 81,2%

(статистика включає всі графічні процесори, доступні для прямої покупки кінцевими користувачами – GeForce, Quadro і прискорювачі обчислень на базі GPU Tesla). [22].

2.7 Вибір апаратної платформи для досліду

Для дослідження була обрана найпоширеніша модель процесору на архітектурі x86-64, та найпоширеніша модель відеокарти від Nvidia.

Детальні характеристики системи для дослідів наведені далі у наступних пунктах.

2.7.1 Процесор

Процесор системи для дослідів має такі характеристики:

- модель: AMD Ryzen 7 2700X;
- тип роз’єму: Socket AM4;
- кількість ядер: 8;
- тактова частота: 3700 Мгц;
- об'єм кеш пам'яті 3 рівня: 16 МБ.

2.7.2 Оперативна пам’ять

Оперативна пам’ять системи для дослідів має такі характеристики:

- модель: Kingston HyperX Predator Black (HX432C16PB3K2/16);
- тип пам’яті: DDR4;
- максимальна частота пам’яті: 3200 МГц;
- пропускна здатність: 25600 Мб/с;
- ємність одного модуля оперативної пам’яті: 8 гігабайт;
- загальна кількість встановлених модулів пам’яті: 4;
- загальна ємність оперативної пам’яті: 32 гігабайти.

2.7.3 Відеокарта

Відеокарта системи для дослідів має такі характеристики:

- модель: GIGABYTE GeForce RTX 2070 SUPER GAMING OC 3X WHITE 8G (GV-N207SGAMINGOC WHITE-8GD);

- тип пам’яті: GDDR6;
- інтерфейс: PCI Express 3.0 x16;
- обсяг пам'яті: 8 гігабайт;
- частоти роботи GPU: 1815 МГц;
- частоти роботи пам'яті: 14000 МГц.

2.7.4 Материнська плата

Материнська плата системи для дослідів має такі характеристики:

- модель: Gigabyte B450 AORUS Pro;
- тип сокету процесора: AM4;
- модель чіпсета: B450;
- тип пам’яті: DDR4;
- максимальна робоча частота пам’яті: 3200 МГц.

2.7.5 Блок живлення

Блок живлення системи для дослідів має такі характеристики:

- модель: ATX Seasonic Focus Plus Gold 850W (SSR-850FX);
- вихідна потужність: 850W;
- напруга: 110/230 В;
- частота: 50/60 Гц.

2.7.6 Пам’ять диску

Пам’ять диску системи для дослідів має такі характеристики:

- модель: Samsung 850 Pro;
- об’єм: 256 гігабайт;
- інтерфейс підключення: SATAIII;
- швидкість читання: до 550 МБ/с;
- швидкість запису: до 520 МБ/с.

2.8 Оцінка ефективності розподілу роботи по ядрам процесору

Операційна система Windows, на якій буде проводитися тестування, використовує попереджувальне планування задач.

Попереджувальне планування використовується, коли процес переходить із стану виконання в стан готовності або із стану очікування в стан готовності. Ресурси (переважно цикли процесора) виділяються процесу протягом обмеженого періоду часу, а потім забираються, і процес знову розміщується в черзі готових, якщо в цьому процесі залишається час спалаху процесора. Цей процес залишається в черзі, поки ядро не вирішить, що за пріорітетом воно має бути виконено наступним. [23].

Процеси програми для досліду, або точніше потоки процесів для обробки та відображення анімованих графічних моделей будуть використовувати ядра процесору на які їх відправить операційна система. Взагалі, для інтенсивних потоків, операційна система рідко змінює ядро на якому цей потік виконується, тому ми можемо досить впевнено замірити навантаження на ядра процесору при виконанні досліду.

Найбільш доцільна програма для виміру розподілу – це системна програма «Resource Monitor», яка поставляється з операційною системою.

«Resource Monitor» – це новий службовий компонент, що з'явився в Windows 7 і Windows Server 2008 R2, за допомогою якого можна переглядати відомості про використання апаратних ресурсів (процесора, оперативної пам'яті, фізичних дисків і мережі) і програмних ресурсів (дескрипторів файлів і модулів) в режимі реального часу. Монітор ресурсів Windows дозволяє фільтрувати результати для обраних процесів або служб, за якими можна вести моніторинг. Крім цього, завдяки монітору ресурсів можна запускати, зупиняти, припиняти і відновлювати процеси і служби, а також усувати помилки тоді, коли програма не відповідає [24].

У даній дослідницькій роботі програма «Resource Monitor» була обрана для замірювання навантаження роботи на кожне ядро у реальному часі. Скріншот програми «Resource Monitor» наведено далі.

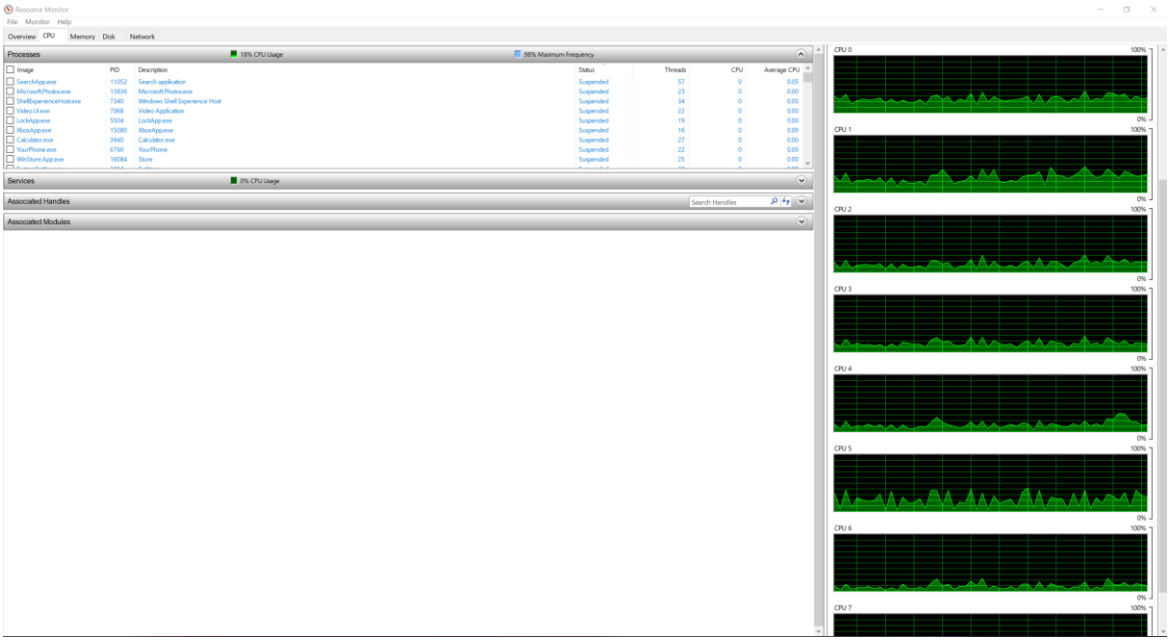

Рисунок 2.1 – Скріншот програми «Resource Monitor»

У правій частині вікна на рисунку 2.1 видно навантаження для кожного ядра. Навантаження рахується від 0 до 100 процентів, де 0 процентів це мінімальна робоча частота, 100 процентів це максимальна робоча частота. Кожну секунду данні обновлюються та показується графік з робочою частотою ядра минулої секунди.

2.9 Порівняння збільшення ефективності OpenGL та Vulkan зі збільшенням кількості ядер процесору

Щоб замірити ефективність дослідження зі збільшенням ядер процесору потрібно запустити програму та зробити досліди на системах з різною кількістю ядер. Є три способи змінити кількість ядер на системі:

1. Фізична зміна процесору процесорами з різною кількістю ядер.
2. Використання віртуальних машин та виділення такій машині різної кількості ядер.
3. Використання конфігураційного меню Windows.

Спосіб з фізичною заміною процесору є дуже дорогим, оскільки потрібно придбати декілька процесорів з різною кількістю ядер. Також, цей спосіб буде не дуже точним, оскільки у різних процесорів можуть бути різні об’єми кешу та різні частоти на яких він працює.

Спосіб з віртаульною машиною не може бути використаний у цьому дослідженні, оскільки віртуальні машини не підтримують новітні програмні інтерфейсі для роботи з комп’ютерною графікою, такі як Vulkan.

Отже було вирішено обрати спосіб з конфігуруванням кількості ядер яке буде використовувати операційна система через конфігураційне вікно операційної системи, наведене на рисунку 2.2 нижче.

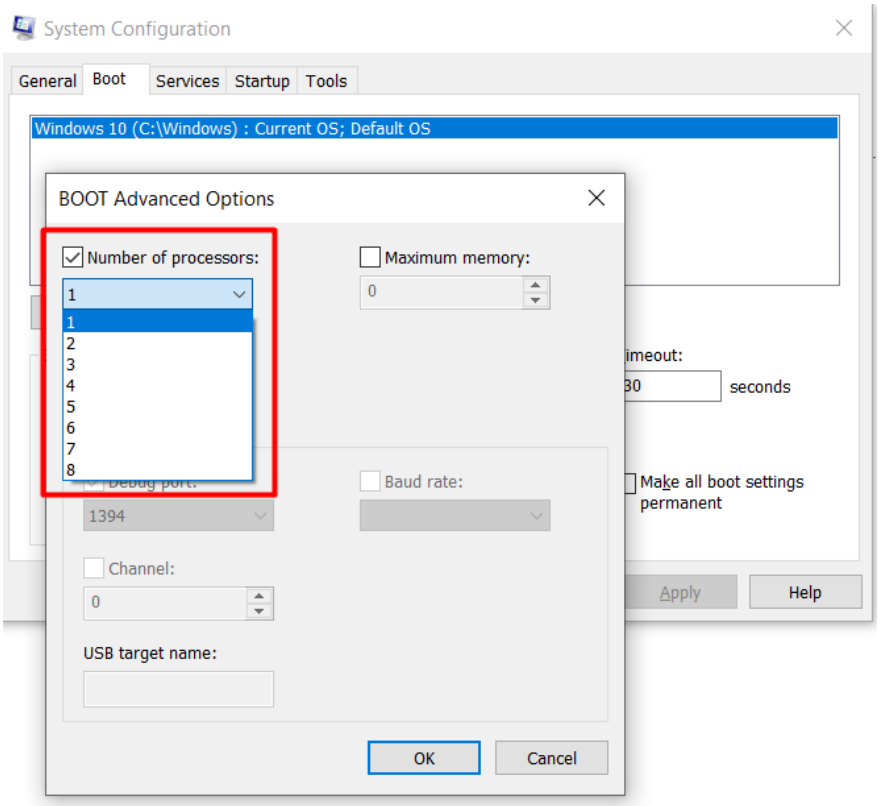

Рисунок 2.2 – Конфігураційне вікно зміни кількості ядер процесорів

Змінюючи кількість процесорів у конфігураційному вікні та перезавантажуючи комп’ютер, операційна система бачить лише задану кількість ядер процесору. Саме так й буде зроблені виміри дослідження на різних кількостях ядер.

2.10 Проектування методів обробки анімації графічних об’єктів для їх порівняння та аналізу

2.10.1 Проектування однопоточної моделі обробки графіки OpenGL

Однопоточна модель обробки та відображення графіки яка є у OpenGL є дуже простою. У загальному виді вона складається з трьох операцій:

1. Опрацювати ввід користувача
2. Відновити дані для наступного кадру
3. Відобразити кадр на екрані.

Ці операції виконуються у циклі до завершення програми. Схема однопоточного циклу рендерінгу наведена на рисунку 2.3:

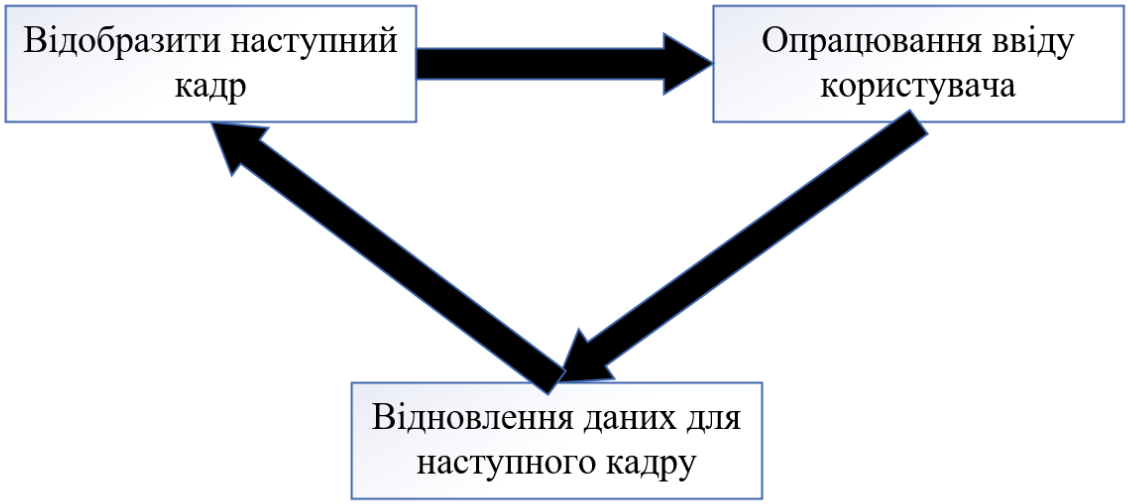

Рисунок 2.3 – Однопоточний цикл рендерінгу

2.10.2 Проектування багатопоточної моделі обробки графіки Vulkan

Багатопоточна модель обробки графіки Vulkan складається з основного потоку виконання, та допоміжних потоків. Основний потік дає завдання допоміжним потікам на обробку графічних об’єктів.

Кожен допоміжний потік відновлює буфера матриць об’єкту та команди відрисовки об’єкта.

Головний потік у свою чергу чекає завершення роботи усіх допоміжних потіків, збирає їх, та виконує відрисовку на екрані на основі даних які були створені чи змінені у допоміжних потіках для кожного графічного об’єкту анімації.

Схема багатопоточного циклу рендерінгу наведена на рисунку 2.4:

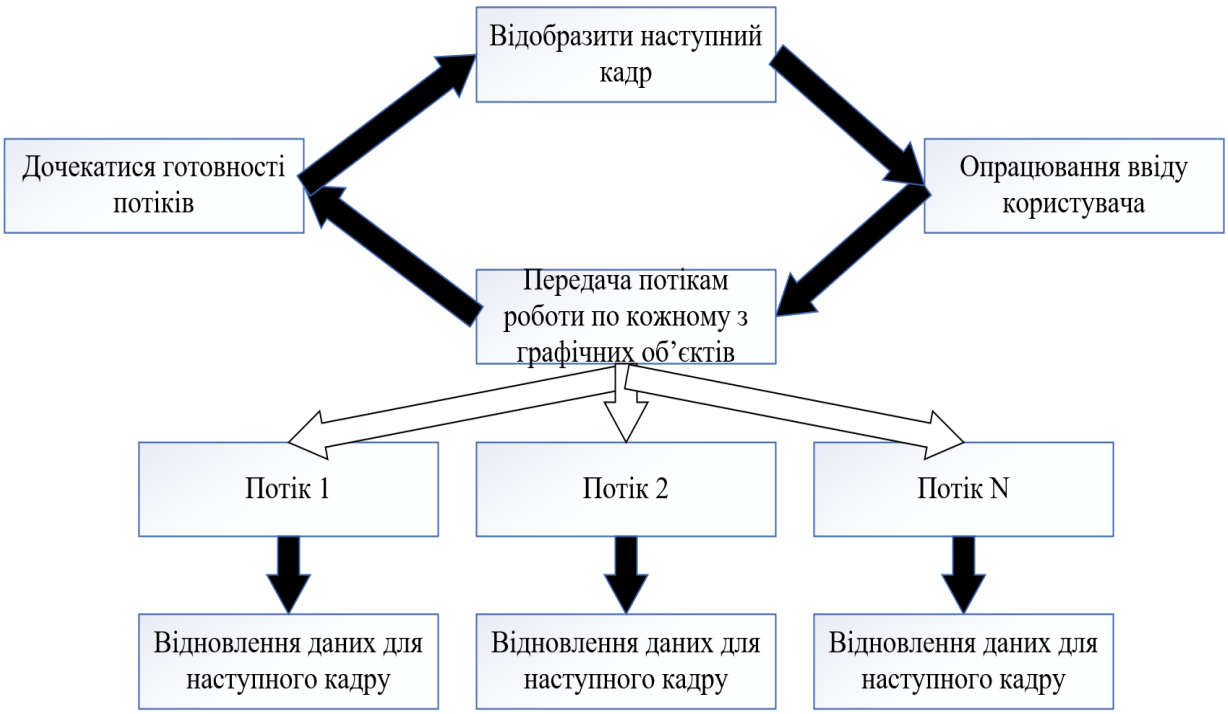

Рисунок 2.4 – Багатопоточний цикл рендерінгу

Саме так досягається перевага у використанні багатьох ядер процесору яка є у нових програмних інтерфейсів комп’ютерної грфіки.

2.11 Вибір метода виміру часу у операційній системі Windows

З моменту введення набору інструкцій x86 P5 багато розробників ігор використовували лічильник позначок часу читання, інструкцію RDTSC, для виконання синхронізації з високою роздільною здатністю. Мультимедійні таймери Windows є достатньо точними для обробки звуку та відео, але з часом кадрів, що становить дюжину мілісекунд або менше, вони не мають достатньої роздільної здатності для надання дельта-часу інформації. Багато ігор досі використовують мультимедійний таймер під час запуску для встановлення частоти процесора, і вони використовують це значення частоти для масштабування результатів від RDTSC, щоб отримати точний час. Через обмеження RDTSC, API Windows надає більш правильний спосіб доступу до цієї функціональності за допомогою процедур QueryPerformanceCounter та QueryPerformanceFrequency [25].

Тож для заміру часу буде використано рекомендований Microsoft спосіб з викликом функції програмного інтерфейсу до операційної системи Windows QueryPerformanceCounter з роздільною точністю менше ніж одна мікросекунда. Цього більш ніж досить для заміру часу виконання одного кадру. Для точності було визначено конкретні характеристики комп'ютеру на якому проводяться досліди.

Висновки до розділу 2

В другому розділі були освітлені та обґрунтовані напрямки дослідження ефективності відображення анімацій на багатоядерних системах.

Були вибрані моделі для дослідження розділені на прості та складні.

У якості простих анімованих моделей була обрана модель 3д куба з анімаційним методом по траєкторії.

У якості складних анімованих моделей були обрані моделі тварин які застосовують методи скелетної анімації по ключовим кадрам.

Для обчислення ефективності анімації була обрана величина середньої кількості кадрів які були опрацьовані та намальовані за певний період часу. Також було розглянуто формулу 2.1 за якою вираховується ця величина.

Після аналізу найпопулярніших програмних та апаратних платформ було обрано операційну систему Windows, процесор на базі архітектури x86-64 та відеокарту від Nvidia.

Далі було проаналізовано та обрано системну програми «Resource Monitor» за якою буде визначатися розподіл роботи по ядрам процесору. Також визначено, що для зміни кількості ядер буде використовуватися вікно конфігурації Windows, яке дозволяє змінювати максимальну кількість ядер які використовує операційна система.

Останні методи дослідження, які були висвітлені, це архітектура однопоточної та багатопоточної моделі відображення комп'ютерної графіки. У якості методу для вимірювання часу було обрано системну функцію QueryPerformanceCounter з роздільною точністю менше, ніж одна мікросекунда.

3 ПРОЕКТУВАННЯ Й РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ АНІМАЦІЙ НА БАГАТОЯДЕРНИХ СИСТЕМАХ

3.1 Зовнішнє проектування

3.1.1 Формалізація задачі

Формалізація задачі на рівні зовнішнього проектування представлена у вигляді діаграми варіантів використання .

Користувач представлені у вигляді актора, що взаємодіє з системою за допомогою варіантів використання. Варіанти використання надають опис можливостей, які система надає акторам.

На діаграмах можуть бути використані наступні типи відношень між варіантами використання та акторами:

- відношення асоціації – відображає зв’язок між акторами та варіантом використання. Відображається лінією зі стрілкою між акторами і варіантом використання;
- відношення включення – показує, що варіант використання включається в базову послідовність, позначається стрілкою з поміткою «include».

Користувач може виконувати наступні варіанти використання:

- задати налаштування рендерінгу;
- запустити рендерінг;
- зупинити рендерінг;
- подивитися результати рендерінгу.

Схема варіантів використання (Use-case diagram) [26] наведена на рис. 3.1

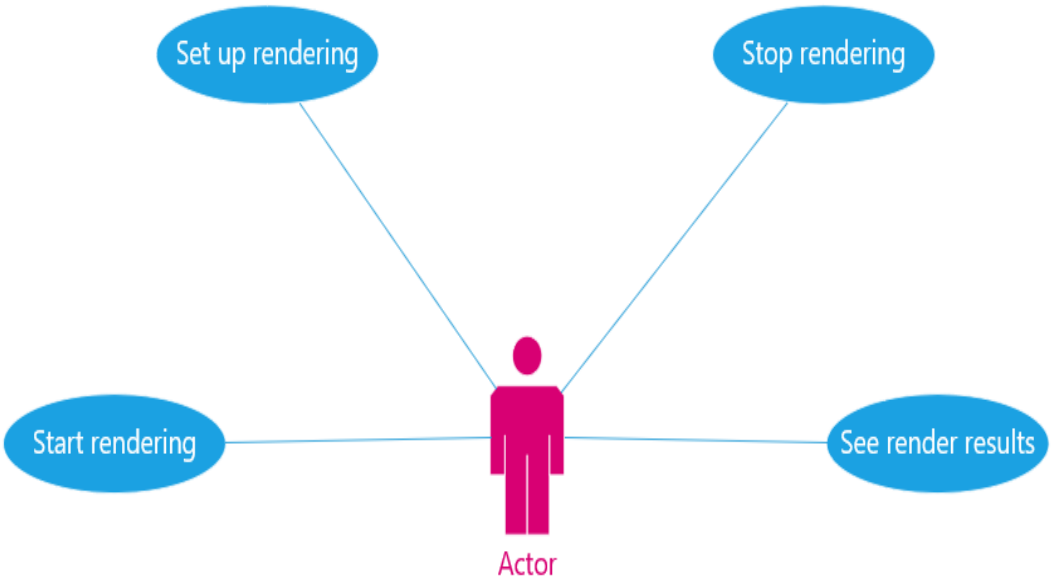

Рисунок 3.1 – Схема варіантів використання

3.1.2 Вхідні дані

Вхідними даними програми є:

- тип рендерінгу (OpenGL або Vulkan);
- складність об’єктів рендерінгу (прості або складні);
- кількість об’єктів для рендерінгу (від одного до 512).

3.1.3 Вихідні дані

Результатом роботи програми є наступні вихідні дані:

- середня кількість кадрів за секунду за час рендерінгу;
- середня кількість секунд за один кадр за час рендерінгу;
- кількість використовуваних ядер процесору.

3.1.4 Проектування динаміки системи

Для кожного варіанту використання можна побудувати діаграму послідовності (Sequence Diagram). Даний тип діаграми дозволяє розглядати динаміку взаємодії об’єктів у часі.

Для проектування розвитку подій в часі була розроблена діаграма послідовності для користувача (див. рис. 3.2), що відображає динаміку взаємодії об’єктів під час виконання. Актор є ініціатором послідовності дій.

Для відображення станів, в яких може знаходитись об’єкт або система в цілому, та умови переходу із одного стану в інший, використовується діаграма послідовності (Sequence Diagram). Діаграма використовується для характеристики поведінки елементу системи в межах її життєвого циклу у вигляді станів цих елементів та послідовності їх змін.

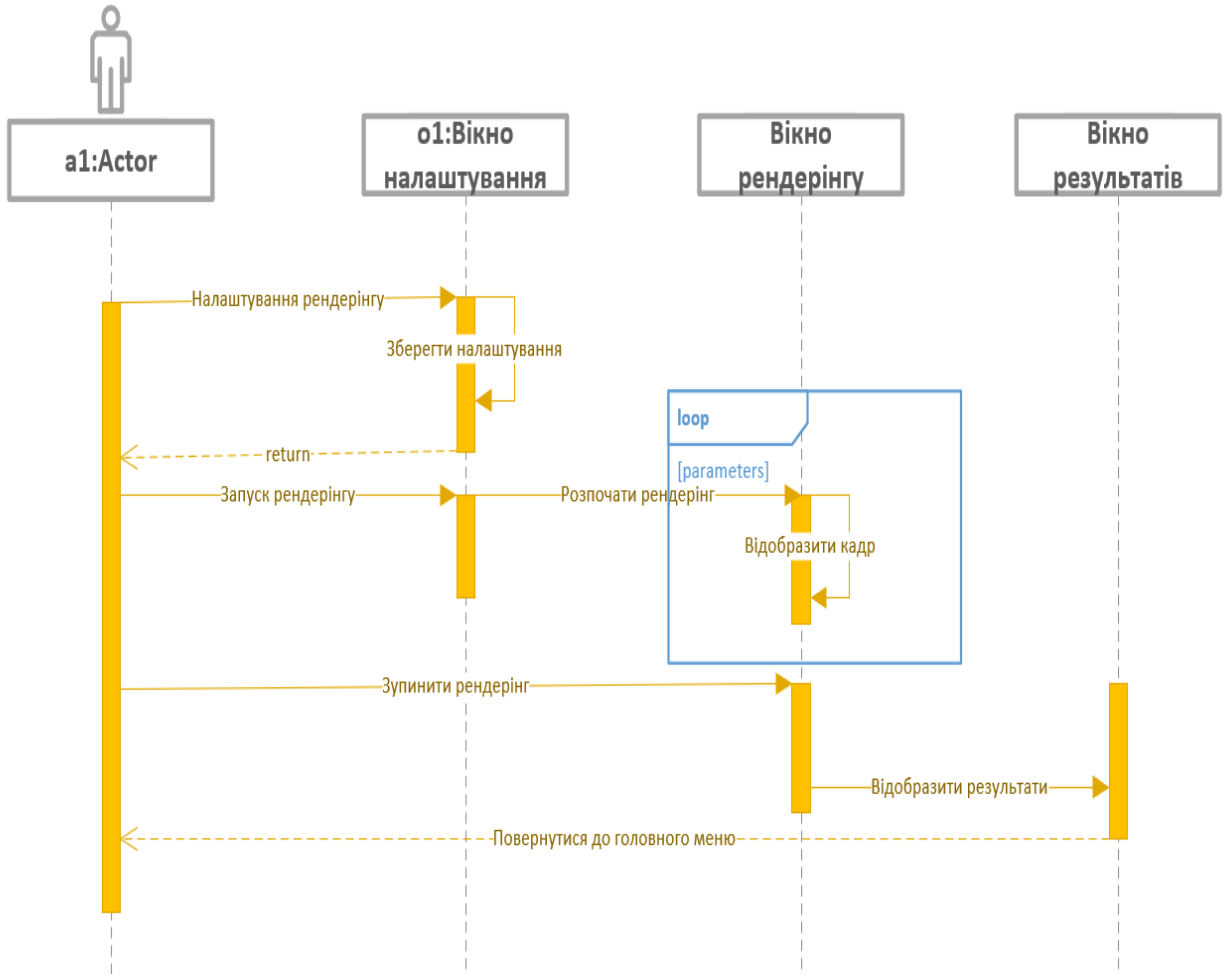

Рисунок 3.2 – Діаграма послідовності

3.1.5 Проектування інтерфейсу користувача

Проектування програмного інтерфейсу користувача грає важливу роль в етапі розробки програми, оскільки невдало спроектований інтерфейс ускладнить роботу з нею і може викликати у користувача небажання працювати із програмою.

Оскільки програмний продукт націлений на користувачів, що можуть не мати досвіду у роботі з подібними програмами, при розробці інтерфейсу необхідно дотримуватись цілого ряду правил:

- природність інтерфейсу. Не бажано змінювати звичні шляхи вирішення виробничої задачі та термінологію, що використовується у даній сфері діяльності;
- простота інтерфейсу. Інтерфейс повинен забезпечувати легкість у його вивченні та у використанні. Всі команди, повідомлення мають бути небагатослівними. Треба розміщувати та представляти елементи на екрані з урахуванням їх смислового значення та логічного взаємозв’язку;
- естетична привабливість. Проектування візуальних компонентів є найважливішою складовою;
- заголовки повинні знаходиться біля або зверху редагованих полів;
- стандартні елемента, такі як кнопки «ОК» та «Скасувати» повинні розташовуватися згідно з замовчуваннями операційної системи;
- треба пам’ятати про надсилання підтверджень користувачеві. У випадку об’ємних команд користувач повинен отримувати інформацію про відправлення йому команди;
- обробка помилок не повинна викликати складностей у користувача;
- система повинна мати можливість скасовувати розпочаті операції;
- зв’язані операції повинні бути об’єднані чи розділені;
- застосовувати правило Міллера, яке передбачає, що людина може концентруватися максимум на 5 чи 9 елементах користувацького інтерфейсу.

Принцип обліку знань користувачів передбачає наступне: інтерфейс повинен бути настільки простим для реалізації, що користувачам не потрібно багато зусиль, щоб звикнути до нього. Інтерфейс повинен використовувати зручні для користувача терміни, а керовані системою об’єкти мають бути пов’язані з робочим середовищем користувача.

Наприклад, якщо розробляється система, призначена для диспетчерів повітряного руху, контрольованими об’єктами в ній повинні бути літаки, траєкторії польоту, тощо. Основна реалізація інтерфейсу та структур даних повинна бути прихована від кінцевого користувача. Принцип узгодженості користувальницького інтерфейсу передбачає, що команди та меню системи повинні бути однакового формату, параметри повинні передаватися однаково [27].

Час на навчання користувачів з таким інтерфейсом скорочується. Також знання, отримані при роботі з одним компонентом системи можна застосувати з іншими частинами системи.

Добрим принципом є дотримання мінімальної кількості ситуації, у яких користувач буде здивований. Тобто, користувач не буде знати що саме робить система, як це інтерпретувати та як поводитись. Оскільки користувача може дратувати невизначеність системи. Природно вважати, що при однакових ситуаціях, система буде вести себе однаково, тому що у користувача під час тривалої роботи з системою формується певні звички та навички. І коли ці звички не можуть бути використані, користувачу доводиться напружуватись та розбиратись з документацією, чи навмання. Саме через це, розробники інтерфейсів повинні розробляти їх так, що би гарантувати схожість дії які потрібно виконати при вирішенні схожих проблем чи завдань

3.1.6 Створення ескізів форм

На етапі проектування до остаточного затвердження інтерфейсу користувача зручно оперувати ескізами екранів. При розробці ескізів багато уваги уділялося внутрішньопрограмній узгодженості інтерфейсу. Назви управляючих

кнопок, розташування ключових елементів форм і загальний стиль по можливості уніфікувалися або робилися схожими, з метою полегшення навчання користувачів при роботі з програмою.

Програма повинна мати простий інтерфейс і складатись з екранів та діалогів кількох типів:

– екран задання налаштувань рендерінгу (рис 3.3);
– екран рендерінгу (рис 3.4);
– екран результатів рендерінгу (рис 3.5).

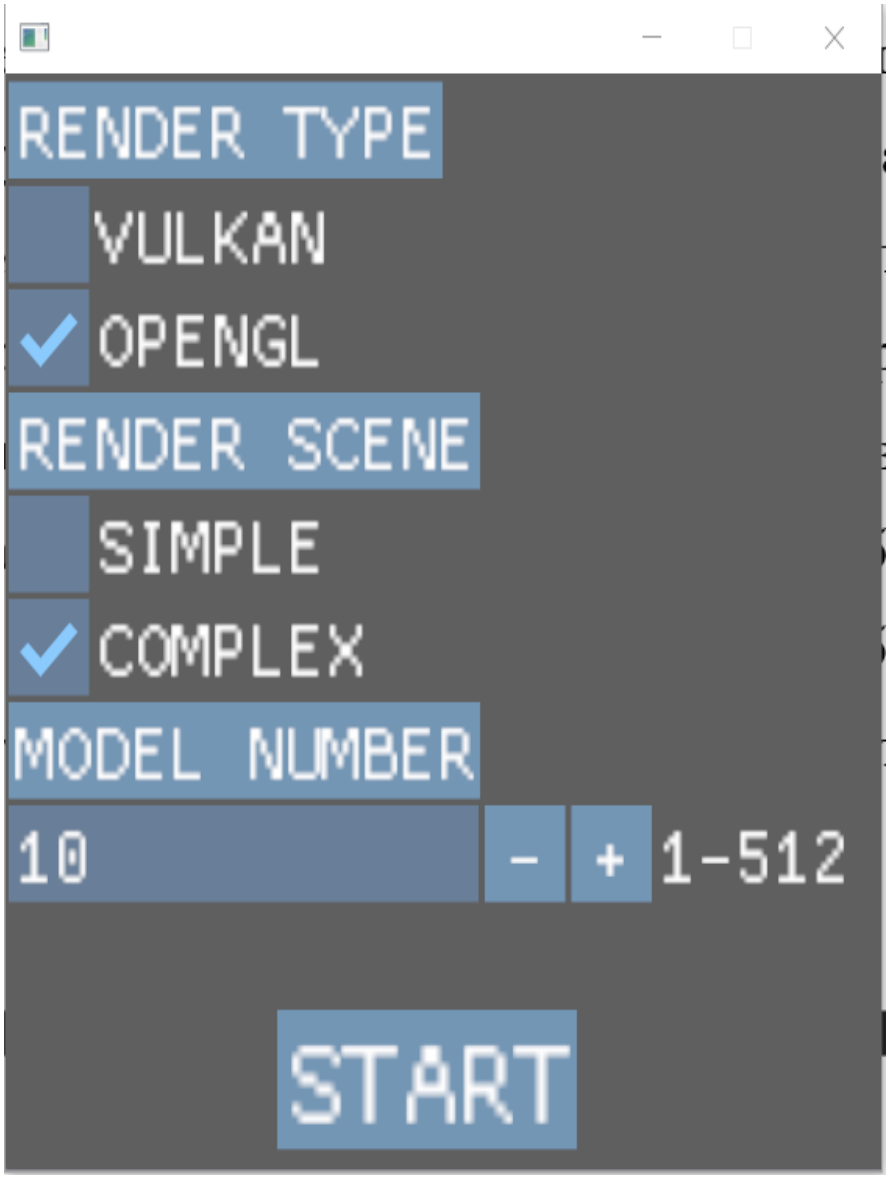

Рисунок 3.3 – Екран налаштувань

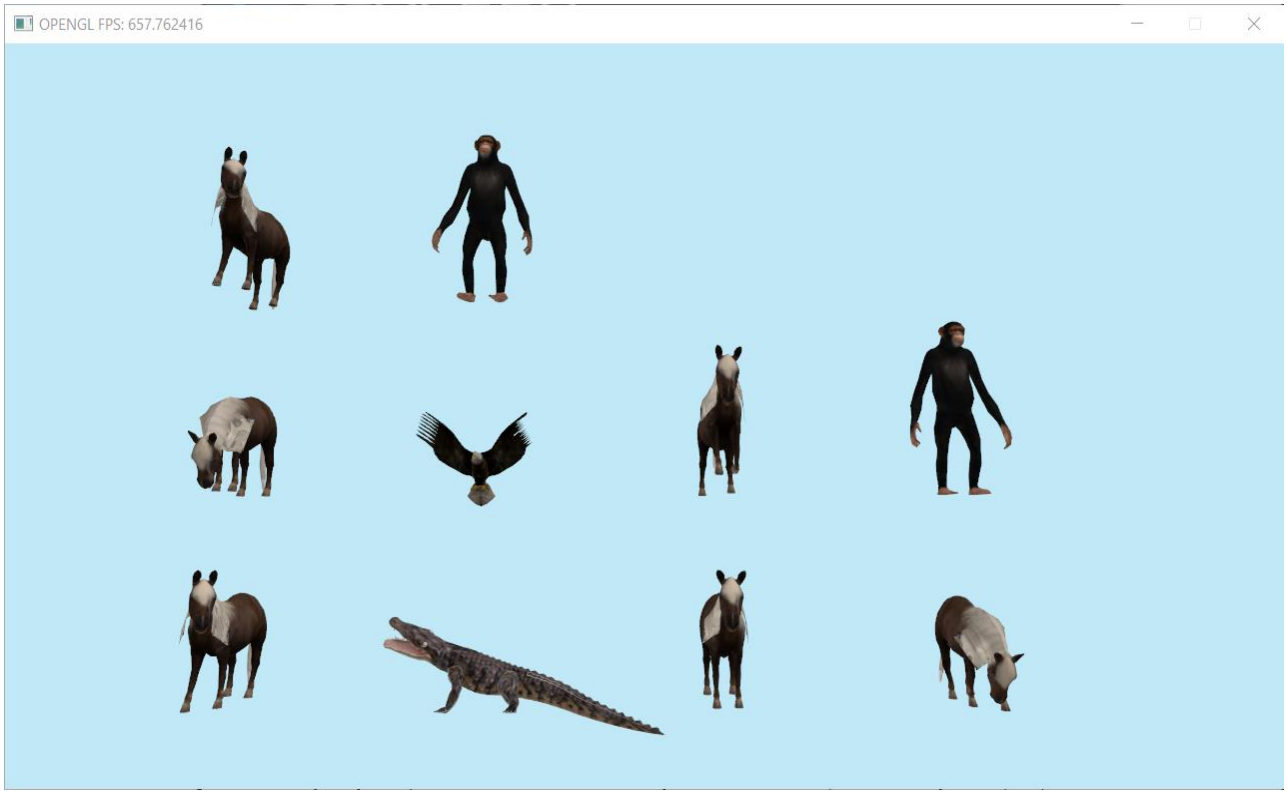

Рисунок 3.4 – Екран рендерінгу

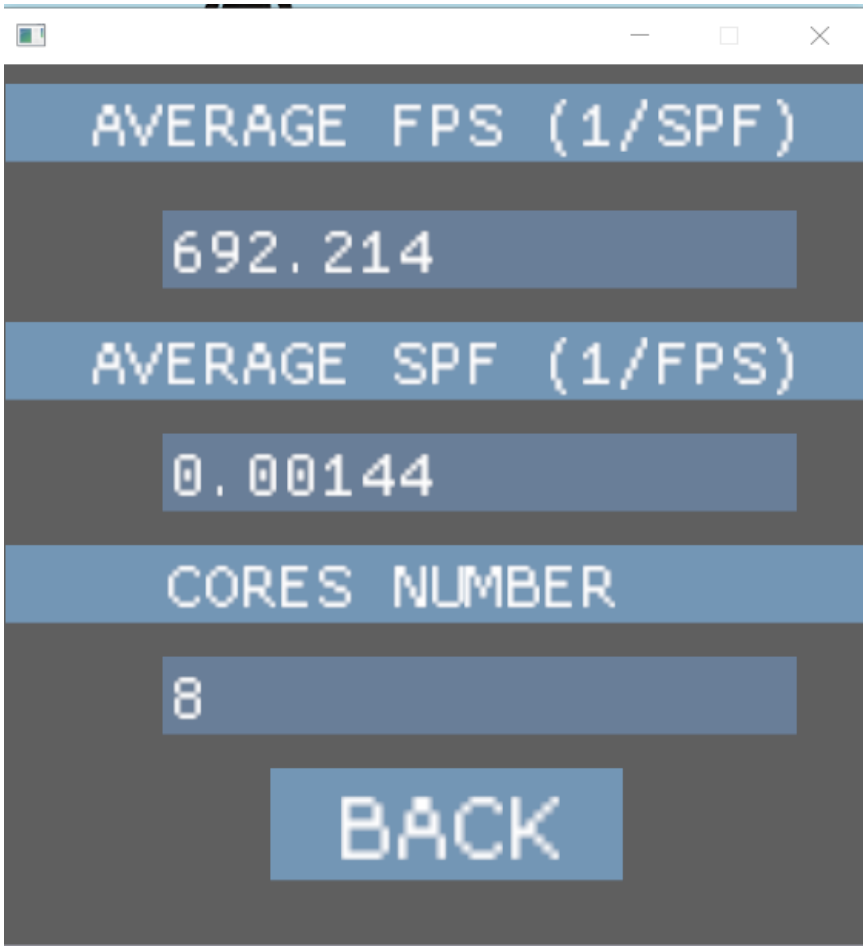

Рисунок 3.5 – Екран результатів

Розроблений інтерфейс є інтуїтивно зрозумілим для користувача, що зменшує ймовірність помилок при роботі.

Екран задання налаштувань містить в собі 2 пари радіокнопок.

Радіокнопка (Radio button), або перемикач - елемент інтерфейсу, який дозволяє користувачеві вибрати одну опцію (пункт) з визначеного набору (групи). Радиокнопки представляють собою елемент круглої (рідше - квадратної або ромбовидної) форми, а вибраний елемент виділяється найчастіше точкою всередині. Поруч з кнопкою розташовується опис обраного елемента. Радиокнопки розташовують групами по кілька штук, причому в будь-який момент обрана може бути тільки одна кнопка з групи. При ініціалізації програми будь-яку кнопку з групи, як правило, вже вибрана, але технічно можливо залишити поза обраної жодну. В такий стан групу радіокнопок привести засобами тільки самих радіокнопок неможливо [28].

Перша пара радіокнопок використовується як засіб вибору програмного інтерфейсу комп’ютерної графіки який буде використовуватися під час рендерінгу. Це може бути OpenGL або Vulkan

Друга пара радіокнопок використовується як засіб вибору складності анімованих графічних об’єктів, які будуть відображатися під час рендерінгу. Доступні два типи об’єктів. Прості являють собою прості 3д куби. Складні об’єкти являють собою анімовані за допомогою скелетної анімації моделі тварин.

Також на екрані задання налаштувань присутнє поле редагування для задання кількості відображаємих анімованих графічних об’єктів при запуску рендерінгу.

Поле редагування (Textbox, Edit field) – елемент графічного інтерфейсу користувача, що дозволяє виробляти введення і виведення текстової інформації в певній галузі інтерфейсу. Якщо записана в поле редагування інформація перевищує його розміри, необхідно використовувати вертикальну або горизонтальну смугу прокрутки, рухаючи повзунок вертикально або горизонтально відповідно за допомогою миші. У деяких інтерфейсів для зручного

відображення інформації можна змінювати розміри поля редагування. Введення даних в текстове поле відбувається за допомогою алфівітно-цифрового блоку клавіатури [28].

Мінімальна кількість відображаємих анімованих графічних об’єктів складає один. Максимальна – 512.

Після задання налаштувань користувач має змогу нажати на кнопку.

Кнопка (кнопка) – базовий елемент інтерфейсу комп'ютерних програм, принцип дії та вид аналогічної кнопки в техніці. При натисканні на кнопку відбувається програмно пов'язане з цим натисканням дія або подія. Опис пов'язаних з кнопкою дій пропонує запропонований розробником користувальницького вмісту разом із документацією до додатка. Для того, щоб ініціювати подію викликаєме кнопкою необхідно перемістити курсор на робочу область кнопки і створити однократне або двократне (в залежності від функціональної специфікації) натискання на ліву кнопку миші. Проста кнопка має два стану - "натиснути" і "віджато". Кнопка може змінювати вигляд залежно від стану та положення курсору або фокусу [28].

На формі задання налаштувань є тількі одна кнопка яка запускає рендерінг.

При натисканні на кнопку старту рендерінгу, у робочому вікні програми з’являється задані анімовані об’єкти. У рядку стану на верхній частині робочого вікна відображається поточна кількість кадрів за секунду які були відображені за останній час. Це поле відновлюється кожну секунду і надає оперативну інформацію оператору персонального коп’ютера.

Рядок стану (Status bar) – елемент графічного інтерфейсу, призначений для виведення повідомлень. Рядок стану зазвичай, має прямокутну форму і знаходиться в нижній частині робочого вікна. Рядок стану не несе ніяких інших функцій, крім виведення оперативної інформації [28].

Екран результатів рендерінгу містить три текстових поля з результатами рендерінгу такими як середнє число кадрів відображаємих за секунду, середня кількість секунд потрібних на відображення одного кадру, та кількість застосованих ядер процесору.

Поле редагування (Textbox, Edit field) - елемент графічного інтерфейсу користувача, що дозволяє виробляти введення і виведення текстової інформації в певній галузі інтерфейсу. Якщо записана в поле редагування інформація перевищує його розміри, необхідно використовувати вертикальну або горизонтальну смугу прокрутки, рухаючи повзунок вертикально або горизонтально відповідно за допомогою миші. У деяких інтерфейсів для зручного відображення інформації можна змінювати розміри поля редагування. Введення даних в текстове поле відбувається за допомогою алфівітно-цифрового блоку клавіатури [28].

Також екран результатів містить у собі кнопку при натисканні на яку користувач повертається до першого вікна налаштування рендерінгу.

3.2 Внутрішнє проектування

3.2.1 Вибір мови програмування

Для розробки програми для дослідження була обрана мова програмування C++. Ця мова програмування є об’єктно орієнтованою, має зворотну сумісність з мовою програмування C.

Оскільки програма дослідження передбачає роботу з програмними інтерфейсами комп’ютерної графіки OpenGL та Vulkan, які в свою чергу офіційно та без додаткових налаштувань можуть бути використані через програмний інтерфейс операційної системи, мова C++ ідеально підходить для того, що би використовувати програмні інтерфейси операційної системи без додатків, з можливістю застосування об’єктно орієнтованого підходу. Це пояснюється тим, що операційні системи самі працюють на мові програмування C та C++, та експортують свої інтерфейси на таких мовах.

3.2.2 Вибір системи контролю версіями

Система керування версіями (СКВ, англ. source code management, SCM) – програмний інструмент для керування версіями одиниці інформації: початкового коду програми, скрипту, веб-сторінки, вебсайту, 3D-моделі, текстового документу тощо. [29]

Система контролю повинна використовуватися для забезпечення гнучкості та надійності. Вона дозволяє зберігати попередні версії файлів, у яких були зміни. Також є можливість «відкатити» файл до попередніх версії, які були збережені. Це дозволяє передивитися останні зміни до файлу, що може допомогти виявити помилки які з’явилися після у останніх версіях продукту. [29]

Системи контролю версії є необхідним інструментом для роботи розробника програмного забезпечення. Вони дають такі переваги:

- створення різних варіантів одного файлу;
- документування змін до файлів, таких як автора зміни, рядок зміни, тощо;
- контроль доступу до читання чи запису у файли підконтрольні системі контролю версії;
- може документувати проект чи кожну версію проекту;
- дозволяє пояснювати зміни та переглядати такі пояснення.

Системи контролю версії діляться на централізовані та розподілені.

Централізовані системи версії використовують сервер який слідкує за усіми змінами. Кожен розробник має підключитися до такого серверу що би мати змогу читати чи вносити зміни до файлів проекту. Такі системи мають свої недоліки та переваги. Недоліками є те, що в разі аварійного припинення роботи серверу уся історія змін стає недоступною, та зупиняється усяка можливість роботи з проектом. Перевагою є те, що розробники зберігають значно меньше кількості локальної інформації, адже уся вона їде з сервера.

Розподілені системи у свою чергу не мають жорстко прив’язаного серверу, але не виключають його використання. Уся інформація про усі зміни у проекті зберігається у кожного розробника, який працює з проектом. Кожен розробник може змінювати та читати файли проекту навіть без доступу до серверу чи мережі Інтернет. Але це було би не зручно без серверу. Сервер потрібен для того, що би декілька розробників мали змогу завантажити на сервер, або з серверу свої чи інші зміни.

У якості системи контролю версії для цієї роботи була обрана розподілена система контролю версії Git.

Git це розподілена система керування версіями. Проект був створений Лінусом Торвальдсом, засновником ядра Linux. Git - одна з найефективніших, надійних і високопродуктивних систем управління версіями. Також є можливість застосовувати цифрові підписи до змін файлів у проекті. [30].

Ця система контролю версії є досить зручною та сучасною. Також корпорація Microsoft надає безкоштовні сервера для зберігання проектів у сервісі Github.

3.2.3 Вибір системи збірки програмного забеспечення

Система збірки є дуже важливою частиною будь якого складного проекту.

Прикладом таких систем є програма make.

У розробці програмного забезпечення Make – це інструмент автоматизації збірки, який автоматично створює виконувані програми та бібліотеки з вихідного коду, читаючи файли під назвою Makefiles, які визначають спосіб отримання цільової програми. Make виконує дії лише з файлами, які змінилися після попереднього виклику програми. Тож часу на перебудову програми стає менше [31].

У якості системи збірки для цього проекту була обрана система CMake.

Взагалі кажучі, ця система сама по собі не є системою збірки. Ліпше сказати що це є генератор інших систем збірок.

CMake описує збірку проекту у файлі CMakeLists.txt використовуючи власну мову програмування схожу на паскаль.

Перед початком роботи треба запустити програму cmake яка використовуючи описання збірки проекту у файлі CMakeLists.txt згенерує обрані файли до іншої системи збірки. Таким чином, використовуючи одні і тіж сами файли опису збірки можливо використовувати їх з різними системами збірки. Наприклад, на операційній системі Linux найчастіше генерують файли до системи збірки make. А на операційній системі Windows йде генерація файлів до системи

збірки msbuild. Також є можливість запустити Visual Studio з згенерованого проекту. Це є досить зручним тому й застосовано у цьому дипломному проекті.

3.2.4 Ієрархія та взаємодія класів системи

Виконаємо проектування системи. Для початку визначимо основні обов'язки системи:

- задання налаштувань до рендерінгу;
- рендерінг використовуючи OpenGL;
- рендерінг використовуючи Vulkan;
- рендерінг складних об'єктів;
- рендерінг простих об'єктів;
- відображення результатів рендерінгу.

Зробимо інтерфейс для рендерінгу, та реалізуємо його класами рендерінгу з OpenGL та Vulkan. Також зробимо клас сцени, яка буде ініціалізувати класи рендерінгу з об'єктами заданою кількістю та типом. Для вікна результатів та налаштування теж розробимо відповідні класи.

Схема взаємодії класів наведена на рисунку 3.6

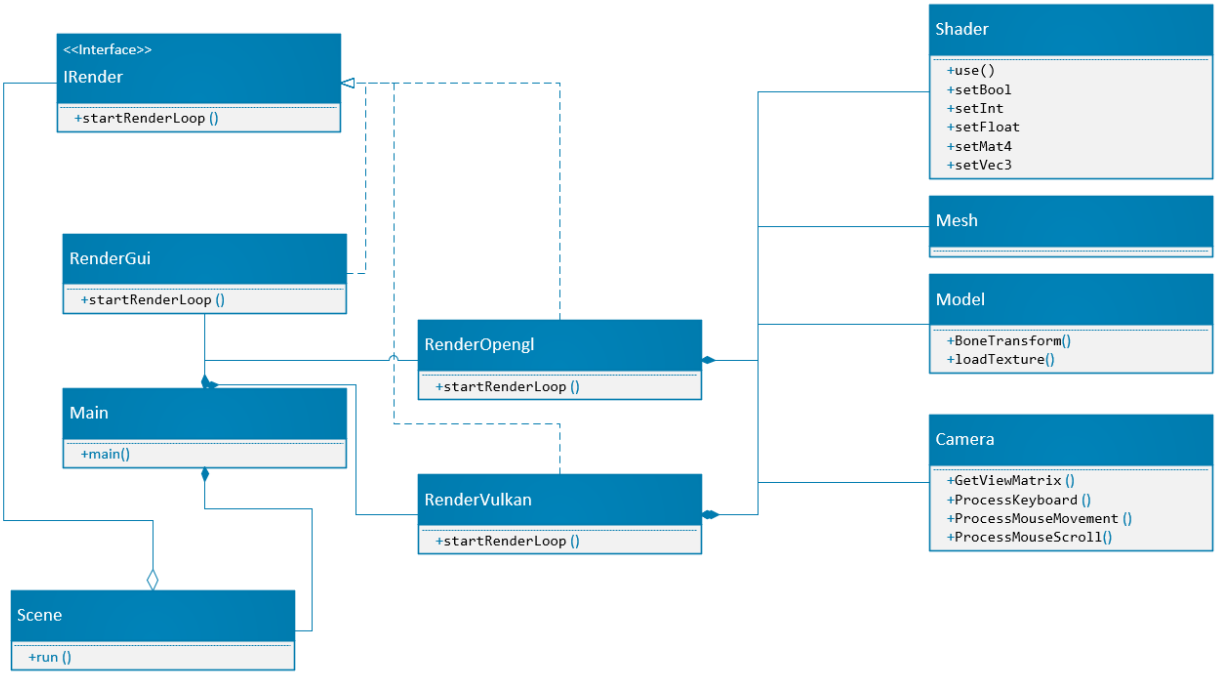

Рисунок 3.6 – Схема взаємодії класів

На рисунку 3.6 можливо побачити що програма була спроектована з використанням об’єктно орієнтованого підходу.

Було використано інтерфейс IRender. Його реалізують клас для відображення графічного інтерфейсу користувача результатів та налаштувань RenderGui. Також його реалізують класи для рендерінгу анімованих графічних моделей RenderOpengl та RenderGUI.

У свою чергу клас Scene використовує інтерфейс IRender для налаштувань рендеру та запуску не знаючи про те який саме тип рендеру це є.

Класи RenderVulkan та RenderOpengl використовують внутрішні допоміжні класи для роботи з 3д моделями, шейдерами та камерою.

Задача класу Main створити об’єкт RenderGUI, на основі вибраних налаштувань створити одну з реалізації інтерфейсу IRender, та віддати цей об’єкт у створений об’єкт класу Scene, який в свою чергу зробить підготовку 3д моделей, їх типу та кількості, та запустить рендер.

3.2.5 Використані принципи проектування

Під час проектування були використані принципи KISS та SOLID.

Принцип SOLID розшифровується таким чином:

- single responsibility - принцип єдиної відповідальності;
- open-closed - принцип відкритості / закритості;
- liskov substitution - принцип підстановки Барбари Лисков;
- interface segregation - принцип розділення інтерфейсу;
- dependency inversion - принцип інверсії залежних.

Принцип єдиної відповідальності обозначає, що кожен об'єкт повинен мати одне зобов'язання та рівень обов'язковості бути повністю інкапсульованим у класі. Усі його послуги повинні бути зроблені виключно на забезпеченні цих зобов'язань [32].

Принцип відкритості / закритості декларує, що програмні сутності (класи, модулі, функції та інше) повинні бути відкритими для розширення, але закритими

для змін. Це означає, що ці сутності можуть змінити своє введення без змін їх вихідного коду [32].

Принцип підстановки Барбари Лисков у формуліровці Роберта Мартина: «функції, які використовують базовий тип, повинні мати можливість використовувати підтипи базового типу не знайомих за цим» [32].

Принцип розділення інтерфейсу у формуліровці Роберта Мартина: «клієнти не повинні залежати від методів, які вони не використовують». Принцип розділення інтерфейсів говорить про те, що занадто «товсті» інтерфейси необхідно розподіляти на більш маленькі та специфічні, щоб клієнти маленьких інтерфейсів знали лише про методи, які необхідні їм у роботі. У тому, що при зміні методу інтерфейсу не слід змінювати клієнтів, які цей метод не використовують [32].

Принцип інверсії залежних - модулі верхніх рівнів не повинні залежати від модулів нижчих рівнів, а обидва типи модулів повинні залежати від абстракцій; самі абстракції не повинні залежати від деталей, а ось деталі повинні залежати від абстракцій [32].

KISS – це принцип проектування та програмування, при якому просторова система декларується у якості основної сукупності або цінності. Існує два варіанти розшифровки абревіатури: «будь простим, дурним» і більш коректним є «тримай коротко і просто» [33].

У проектуванні наступних принципів KISS виражається в тому, що:

Не має смислу реалізувати додаткові функції, які не будуть використовуватися зовсім або їх використання крайні маловірогідні, як правило, для більшості користувачів достатньо базової функціональності, а також умовне використання лише для зручності додатків [33];

Не варто перенавантажувати інтерфейс опціями, які не будуть потрібні більшості користувачів, значно простіше передбачити для них окремий «розширенний» інтерфейс (або зовсім відмовитися від відповідної функції) [33];

Безглуздо робити реалізацію складної бізнес-логіки, яка виконує абсолютно всі можливі варіанти впровадження систем користувача та

навколишнього середовища, - по-перших, це просто неможливо, а по-других, така фанатичність заставляє збирати «зореліт», що найчастіше ірраціонально з комерційною зору [33].

3.2.6 Використані шаблони проектування

У дипломній роботі були використані такі шаблони проектування такі як інтерфейс та стратегія.

Інтерфейс це основний шаблон проектування, що представляє собою загальний метод для структурування комп'ютерних програм для того, щоб їх було більше зрозуміти. У загальному інтерфейс – це клас, який забезпечує програму простою або більш програмно-специфічну можливість доступу до інших класів [34].

У даному випадку було використано інтерфейс IRender з яким працює клас Scene. Клас Scene нічого не знає про те, яку саме реалізацію інтерфейсу IRender на даний момент він використовує.

Стратегія це поведінковий шаблон для визначення алгоритмів. Кожен з алгоритмів є інкапсульованим. Так можно обирати алгоритм через визначення класу. Також цей шаблон Strategy дозволяє змінювати алгоритми у об’єктах, які його використовують, у будь-який час роботи програми[35].

У цій роботі можливо сказати, що стратегія це клас Scene. Передаючи різні об’єкти класів-реалізацій від класу IRender ми тим само змінюємо алгоритм яким робиться рендер.

3.2.7 Технологічна платформа

Під час реалізації системи використовувалася така технологічна платформа.

GLAD – це бібліотека для динамічної загрузки графічних програмних інтерфейсів. Вона потрібна для того що би загрузити вказівники функції OpenGL та Vulkan в час виконання програми. Це потрібно що би здобути доступ до цих програмних інтерфейсів до комп’ютерної графіки. Вони роблять це за допомогою програмного інтерфейсу до операційної системи, яка в свою чергу дає змогу

загрузити функції з файлу динамічної бібліотеки (dll). Ці функції реалізуються розробником графічних карт.

GLFW – це бібліотека для керування вікном операційної системи. Також вона може створювати начальне налаштування графічних інтерфейсів, потрібне для того що би вказати їм на яке вікно та у якому форматі має відрисовуватися комп’ютерна графіка.

Також служить для того, що би керувати та обробляти ввід з клавіатури та миші.

GLM – це бібліотека для математичних операції. Надає класи та методи для операцій над векторами та матрицями. Спеціально розроблена для того, що би бути сумісною з форматами графічних інтерфейсів комп'ютерної графіки.

ASSIMP – це бібліотека для загрузки файлів з анімованими об’єктами. Ключовою функцією є можливість завантаження різних форматів 3д моделей, які перетворюються у загальний формат визначений цією бібліотекою.

IMGUI – це бібліотека для графічного інтерфейсу. Надає змогу відрисовувати елементи керування, такі як кнопка, перемикач, та інші.

3.3 Стратегія тестування

Тестування програмного забезпечення охоплює цілий ряд видів діяльності, дуже аналогічний послідовності процесів розробки програмного забезпечення. Сюди входять постановка задачі для тесту, проектування, написання тестів, тестування тестів і, нарешті, виконання тестів і вивчення результатів тестування. Вирішальну роль грає проектування тесту. Можливий цілий спектр підходів до вироблення стратегії проектування тестів [36].

Стратегія тестування повинна відповідати на такі питання:

- що тестувати;
- якими методами.

Усі методи тестування можна поділити на дві групи: тестування за вхідними даними – «чорного ящика» [37, 38], та тестування логіки програми «білого ящика» [39]).

Тестування чорної скриньки може бути використано для методів, які мають лінійну структуру. Для того, щоб визначити, який метод тестування на чорну скриньку є більш ефективним у цій ситуації, спочатку слід розглянути кожен метод окремо.

Деякі методи з нелінійною структуру, особливо ті, що обробляють дані зі складною структурою, слід перевірити двома методами: еквівалентним методом розділу ("чорний ящик"), оскільки він дозволяє класифікувати можливі помилки, а іноді дозволяє виявляти такі помилки, які не були виявлені під час тестування методами «білого ящика», а також вхідні та вихідні дані не обмежуються специфікацією програми, а також залежать одна від одної та способу охоплення рішень.

Тестування методів класів, які мають нелінійну структуру, тобто мають умови, цикли, необхідно провести з використанням тестування методом покриття рішень («білого ящика»), у разі коли умови прості, та методом покриття умов, якщо умови складні. Для тестування білим ящиком виберемо метод покриття рішень та покриття умов. Для тестування чорним ящиком - метод припущення про помилку та метод еквівалентного розбиття.

3.3.2 Вибір стратегії налагодження програми

Розроблена система досить об’ємна і складається із взаємопов’язаних частин. Налагодження буде виконуватися за допомогою методів індукції та просування від місця виникнення помилки до місця помилки.

3.3.3 Результати налагодження програми

При налагодженні програми було виявлено деякі помилки. Наприклад, помилка при загрузці файлу текстури.

До місця виникнення ймовірної помилки ставиться точка зупину («breakpoint») зображена на рис. 3.7, на якій програма зупиниться і чекатиме покрокового виконання. На кожному наступному кроці існує можливість перегляду змісту змінних та результатів виконання функцій за допомогою

інструменту «Evaluate Expression» (див. рис. 3.8). Таким чином легко відстежити, в якому місці програми виникає помилка.

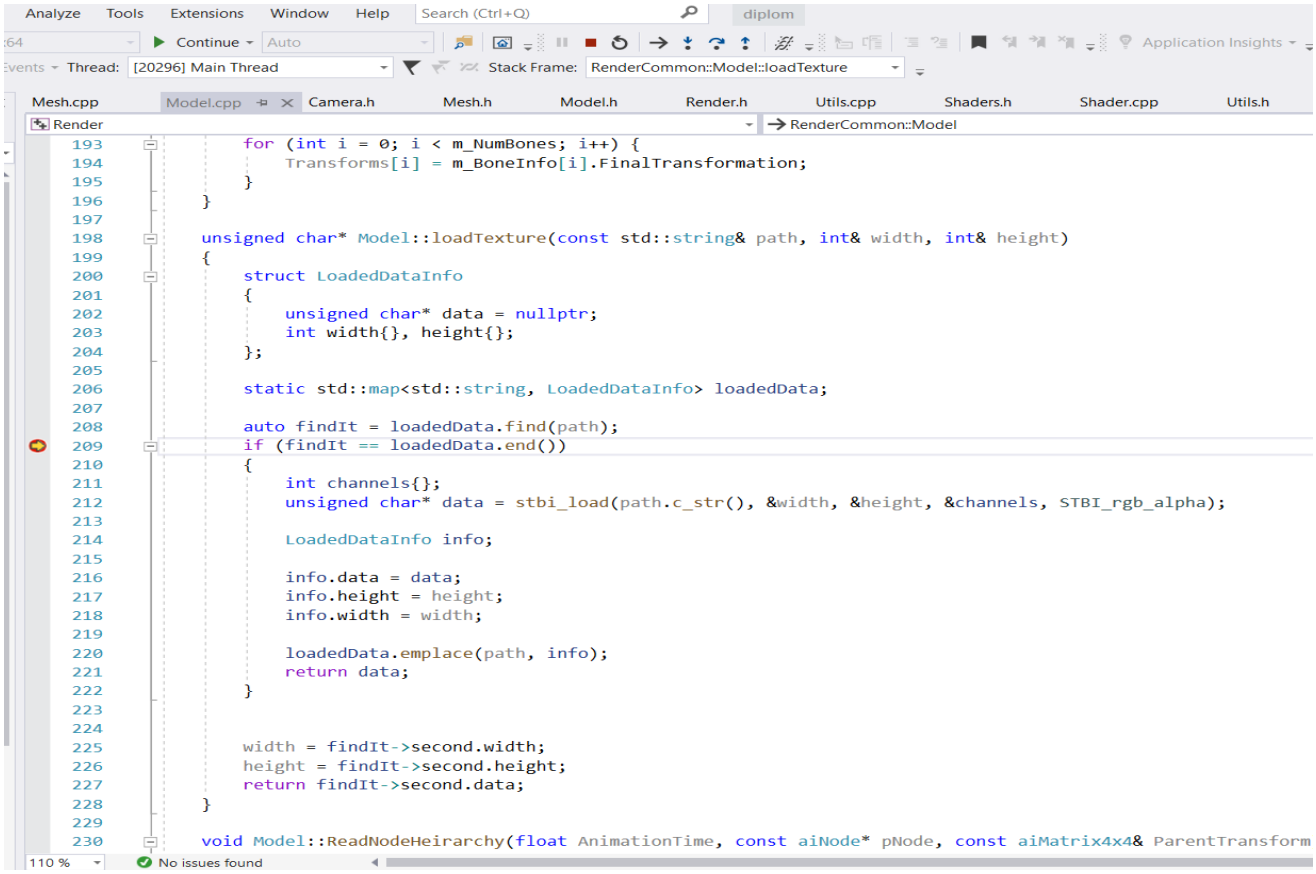

Рисунок 3.7 – Використання точки зупину програми

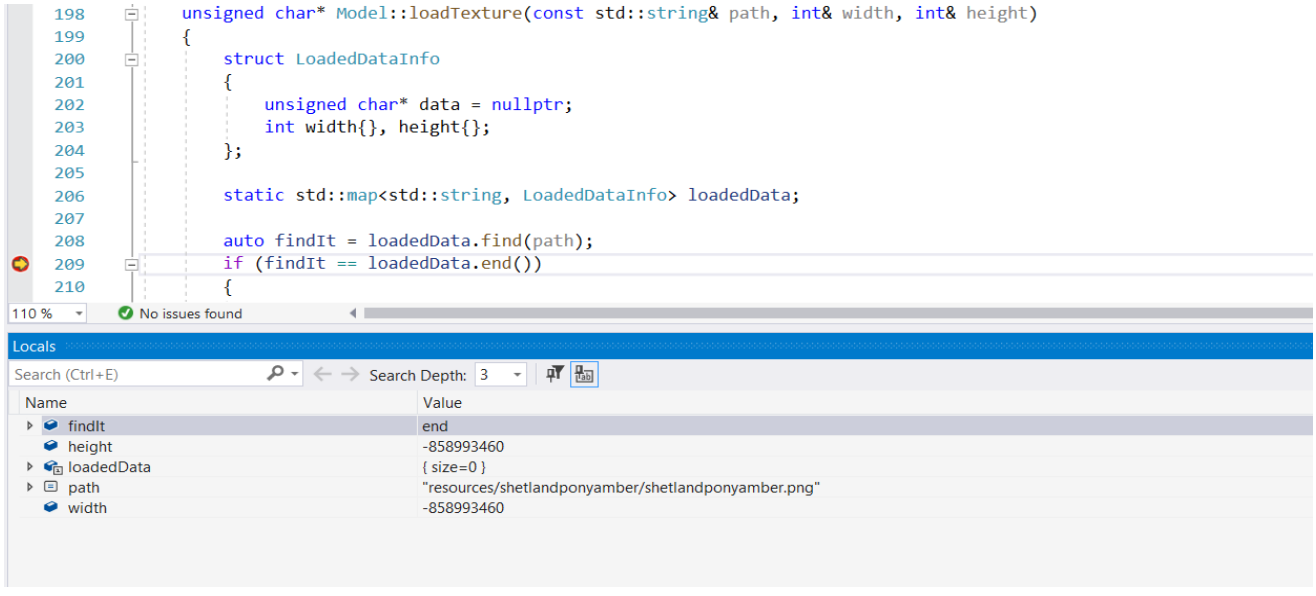

Рисунок 3.8 – Перегляд значення змінної

Інструмент «Evaluate Expression» також дозволяє викликати методи та проглядати результат, який вони повертають, що є дуже корисним для тестування. За допомогою перерахованих утиліт були налагоджені майже всі функції програми. Знайдені помилки було ліквідовано.

Висновки до розділу 3

У третьому розділі було розглянуте зовнішнє та внутрішнє проектування.

При зовнішньому проектуванні було формалізовано задачу, її вхідні та вихідні дані та схему способів використання. Також було спроектовано динаміку системи за допомогою діаграми послідовності. Спроектовано ескізи інтерфейсу користувача.

При внутрішньому проектуванні було обрано мову програмування C++. Також було обгрунтовано та обрано систему контролю версії, систему збірки програмного забезпечення. Було розроблено ієрархію класів та їх взаємодію згідно з принципами парадигми об’єктно орієнтованого програмування. Описано використані бібліотеки.

Також було описано стратегію тестування та результати налагодження програми.

Програма є придатною до використання.

4. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ АНІМАЦІЙ НА БАГАТОЯДЕРНИХ СИСТЕМАХ

4.1 Збір даних для аналізу

Збір даних проводився з використанням комп’ютеру на базі операційної системи Windows. Детальні характеристики описані у пункті 2.6.

Починаючи з одного ядра, та змінюючи кількість ядер на одне ядро, було виміряно дані для різної кількості моделей.

У даній дипломній роботі для оцінки зміни ефективності використовувалися 10, 100 та 500 анімованих графічних моделей, які одночасно відрисовувалися на екрані під час рендеру. Тести проводилися двічі. Один раз на рендері за допомогою програмного інтерфейсу до комп’ютерної графіки OpenGL. Другий раз на рендері за допомогою програмного інтерфейсу до комп’ютерної графіки Vulkan.

Також іспити подвіювалися через те, що у якості анімованих моделей було використано два типи: прості та складні. Простий тип анімованих моделей це 3д куб який обертається навколо своєї вісі. Складний тип анімованих моделей це 3д моделі тварин зі скелетною анімацією.

Для кожного типу кожної кількості моделей було зібрано інформацію, а саме:

- середня кількість кадрів за секунду;
- кількість використаної пам’яті графічного процесору;
- кількість використаної оперативної пам’яті.

Зібрані дані для десяти простих моделей наведені у таблиці 4.1. Для 100 моделей у таблиці 4.2. Для 500 моделей у таблиці 4.3.

Зібрані дані для десяти складних моделей наведені у таблиці 4.4. Для 100 моделей у таблиці 4.5. Для 500 моделей у таблиці 4.6.

Таблиця 4.1 – Зібрані дані для десяти простих моделей.

Кількість ядер	FPS OpenGL	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)	FPS Vulkan	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)
8	3524.58	79648	38844	5114.068	99272	59376
7	3442.121	79636	38840	4910.444	99268	58468
6	3501.287	79636	38832	4908.323	99268	57420
5	3506.833	79636	38944	4898.079	99224	57260
4	3512.181	79636	38872	4810.297	99268	56812
3	3341.337	79636	39280	4512.096	99268	56504
2	3033.547	79648	38392	3973.538	99268	56020
1	2135.311	79640	31484	3041.443	99272	49044

Таблиця 4.2 – Зібрані дані для ста простих моделей.

Кількість ядер	FPS OpenGL	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)	FPS Vulkan	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)
8	1025.618	79636	40148	2132.973	105448	76256
7	1038.978	79648	40196	2145.989	105404	76816
6	1012.066	79648	40204	2165.39	105400	75464
5	1044.569	79636	40156	2055.611	105448	76540
4	1059.232	79636	40176	1937.76	105404	72992
3	1011.612	79648	40048	1727.357	105448	72180
2	1013.952	79648	40756	1565.177	105440	68412
1	788.656	79632	32656	1471.663	105428	58104

Таблиця 4.3 – Зібрані дані для п’ятиста простих моделей.

Кількість ядер	FPS OpenGL	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)	FPS Vulkan	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)
8	774.727	81684	41448	1631.82	107476	84736
7	773.917	81684	41372	1622.579	107468	85160
6	773.562	81696	41332	1614.145	107468	81104
5	777.731	81684	40748	1509.396	107468	81128
4	788.579	81684	41448	1463.737	107472	78512
3	753.69	81696	41508	1353.444	107520	77472
2	758.99	81684	41336	1273.405	107520	76932
1	632.555	81680	33080	1130.3	107504	65628

Таблиця 4.4 – Зібрані дані для десяти складних моделей.

Кількість ядер	FPS OpenGL	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)	FPS Vulkan	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)
8	784.51	223360	199976	1966.821	241952	252816
7	769.067	223388	208104	1944.66	241948	252996
6	762.868	223376	203960	1968.191	241904	253304
5	732.205	223380	203904	1936.505	241896	252776
4	754.518	223368	205188	1758.827	241944	252184
3	750.211	223364	196740	1483.39	241936	247520
2	725.34	223364	195332	1137.627	241948	252452
1	551.729	223352	188768	615.572	241956	245324

Таблиця 4.5 – Зібрані дані для ста складних моделей.

Кількість ядер	FPS OpenGL	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)	FPS Vulkan	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)
8	79.59	511868	486492	386.435	530358	665228
7	86.224	511860	488072	354.583	530364	671836
6	86.977	511880	489164	313.158	530308	665024
5	82.812	511860	490944	272.014	530328	667076
4	86.203	511852	492136	233.191	530280	654472
3	83.153	511864	495544	190.856	530368	681192
2	88.262	511856	483776	133.587	530360	663656
1	76.501	511844	490944	76.183	530332	649552

Таблиця 4.6 – Зібрані дані для п’ятиста складних моделей.

Кількість ядер	FPS OpenGL	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)	FPS Vulkan	GPU пам’ять OpenGL (Кб)	Оперативна пам’ять OpenGL (Кб)
8	60.445	642636	624000	272.45	660776	847344
7	63.065	642608	625128	251.901	660724	848680
6	63.431	642620	620492	224.799	660792	835352
5	62.724	642616	619788	199.902	660696	840936
4	60.713	642616	619764	169.251	660768	851912
3	63.156	642616	622500	137.17	660764	832220
2	62.207	642632	629444	101.311	660760	831908
1	56.73	642596	610924	55.931	660760	833124

4.2 Аналіз розподілу роботи по ядрам процесору

Для того, що б відобразити розподіл роботи по ядрам процесору було обрано лише один випадок – 500 складних моделей. Цього більш ніж досить що би робити висновки про розподіл роботи по ядрам.

За для виміру розподілу роботи по ядрам була використана утиліта «Resource Monitor» описана у розділі 2.5.

На рисунку 4.1 відображено розподіл роботи по ядрам процесору використовуючи програмний інтерфейс до комп’ютерної графіки OpenGL.

На рисунку 4.2 відображено розподіл роботи по ядрам процесору використовуючи програмний інтерфейс до комп’ютерної графіки Vulkan.

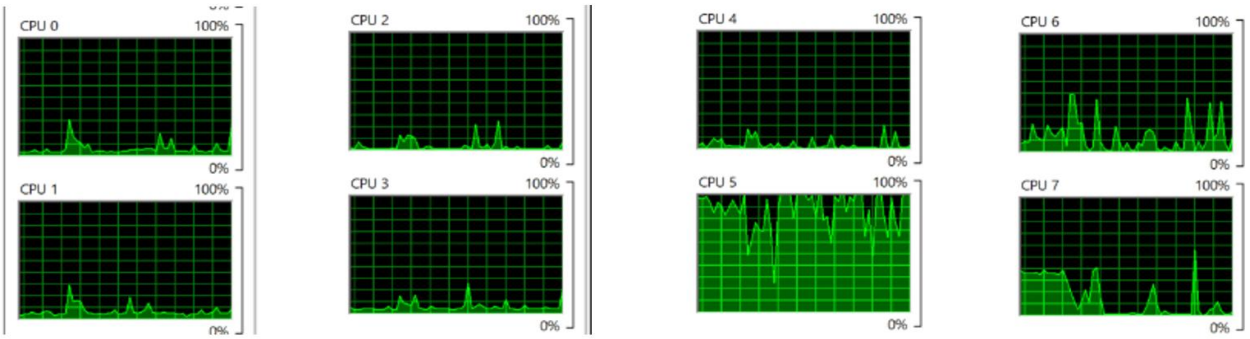

Рисунок 4.1 – Розподіл роботи по ядрам процесору з OpenGL

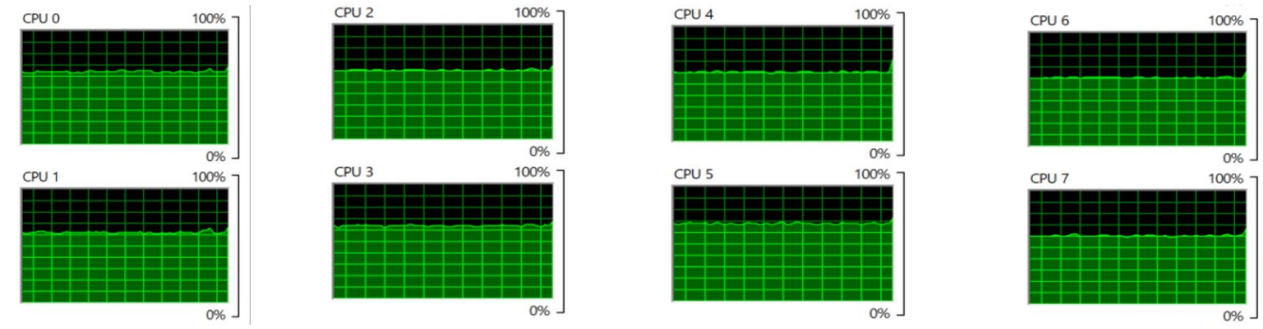

Рисунок 4.2 – Розподіл роботи по ядрам процесору з Vulkan

Як видно з рисунка 4.1, при однопоточному рендері з OpenGL лише п’яте ядро постійно навантажено майже на 100%. А усі інші ядра майже нічим не навантажені. Це не є добре, оскільки рендер сповільнюється тому що повинно

чекати на обробку інформації ядром, яке перенавантажено, замість того щоб використовувати інші ядра, як це відбувається при багатопоточному рендері на Vulkan на рисунку 4.2.

4.3 Аналіз зміни ефективності FPS зі збільшенням кількості ядер

На рисунку 4.3 та 4.4 зображено графіки зміни середньої кількості кадрів відображаємих за секунду зі зміною кількості ядер для однопоточного OpenGL та багатопоточного Vulkan типів рендерів для простих та складних моделей.

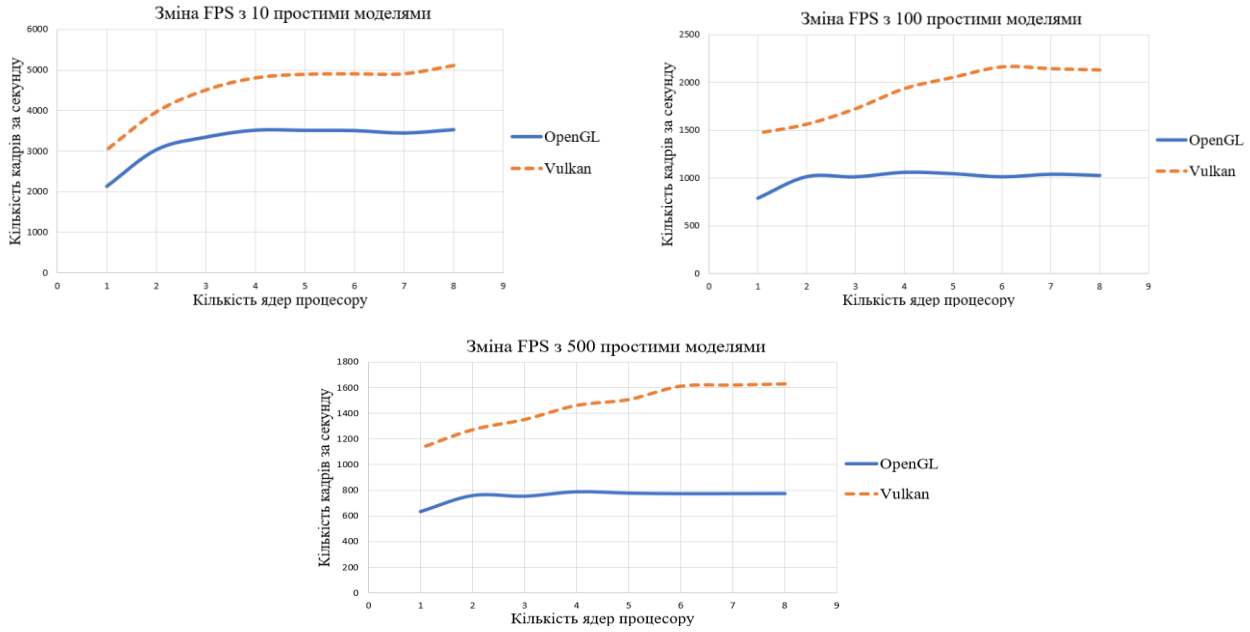

Рисунок 4.3 – Графіки зміни FPS з простими моделями

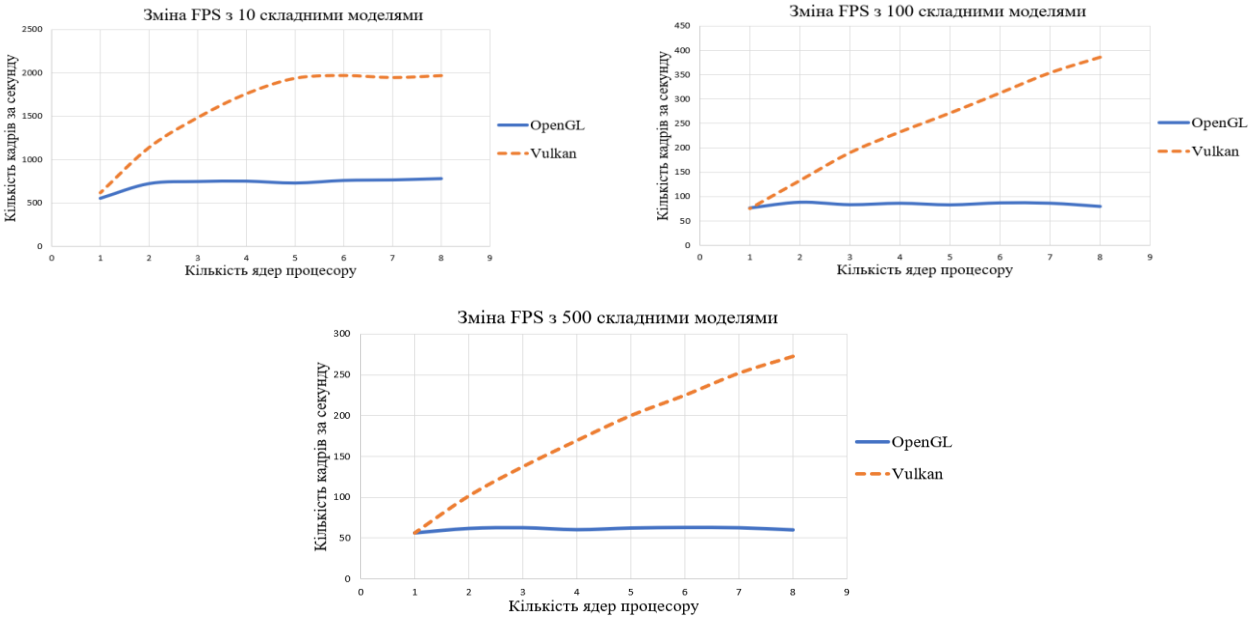

Рисунок 4.4 – Графіки зміні FPS з складними моделями

Рисунок 4.3 показує що для десяти простих моделей зріст кількості кадрів відображаємих за секунду застосовуючи багатопоточний рендер на Vulkan майже відсутній починаючи з п’яти ядер. Для ста моделей зріст зупинився на шостому ядрі, для п’ятиста на сьомому ядрі.

Це можна пояснити тим, що прости моделі, які являють собою 3д куб який обертається навколо своєї осі, навантажує процесор досить слабко, так що декілька моделей встигають обробитися лише на одному ядрі, і таким чином навіть для п’ятиста простих моделей, останні ядра не задіяні, і не впливають на ефективність.

Рисунок 4.4 показує що для десяти складних моделей ріст кількості кадрів відображаємих за секунду зупиняється на шостому ядрі, майже як і для простих моделей. Але для ста та навіть п’ятиста моделей, ріст не зупиняється, та є лінійним. Це пояснюється тим, що для скелетної анімації, яка застосовується для складних моделей, треба набагато більше процесорного часу для вирахування матриць та зміщень вершин, також кількість цих вершин значно більша ніж у простих моделей. Таким чином, для усіх ядер є робота, адже одне чи декілька ядер вже не можуть вчасно обробити велику кількість складних моделей, так що би не залишати роботи для решти ядер.

Тут можна зробити висновок, що для 3д моделей навіть значної кількості, у яких мало вершин та прості операції, велика кількість ядер є зайвою. Для малого числа складних моделей велика кількість ядер теж є зайвою, але зі збільшенням таких моделей, можна досить швидко навантажити усі ядра, так що би вони опрацьовувалися паралельно. І залучивши усі ядра процесора, ми досягнемо піку продуктивності, коли усі ядра однаково навантажені, та максимальна кількість анімованих об’єктів обробляється паралельно а не великою чергою.

Що стосується однопоточного рендеру на OpenGL, то його зріст незначно збільшується тільки на двох ядрах. Це пояснюється тим, що операційна система не перериває обробку об’єктів на одному ядрі своїми службовими процесами, а використовує вільне ядро.

Таблиця 4.7 показує різницю у кількості відображаємих кадрів за секунду у процентному відношенні між OpenGL та Vulkan різної кількості ядер для простих моделей. А таблиця 4.8 для складних моделей.

Таблиця 4.7 – Різниця FPS для простих моделей

Кількість ядер	Різниця FPS при 10 простих моделей	Різниця FPS при 100 простих моделей	Різниця FPS при 500 простих моделей
8	45%	108%	111%
7	43%	107%	110%
6	40%	114%	109%
5	40%	97%	94%
4	37%	83%	86%
3	35%	71%	80%
2	31%	54%	68%
1	42%	87%	79%

Таблиця 4.8 – Різниця FPS для складних моделей

Кількість ядер	Різниця FPS при 10 простих моделей	Різниця FPS при 100 простих моделей	Різниця FPS при 500 простих моделей
8	151%	386%	351%
7	153%	311%	299%
6	158%	260%	254%
5	164%	228%	219%
4	133%	171%	179%
3	98%	130%	117%
2	57%	51%	63%
1	12%	0%	-1%

Як видно з таблиці 4.7, для простих моделей вулкан ефективніше ніж OpenGL на у середньому на 40% для десяти моделей та на 90% для 100 та 500

моделей. При цьому зміна ефективності зі зміною кількості кадрів є незначною, приблизно 5-10%. Похибка у цифрах при цьому складає +- 25%.

Таблиця 4.8 показує що для складних моделей, одне ядро по ефективності однакове для OpenGL та Vulkan. Але далі зріст ефективності йде приблизно 50% з кожним додатковим ядром. Тож для складних моделей при одному ядрі Vulkan не є ефективним, як при простих моделей. А починаючи з двох та більше ядер Vulkan є досить ефективним, значно збільшуючи кількість кадрів видображаємих за секунду.

4.4 Аналіз зміни GPU пам’яті зі збільшенням кількості ядер

На рисунку 4.5 та 4.6 зображено графіки зміни кількості використаної пам’яті графічної карти зі зміною кількості ядер для однопоточного OpenGL та багатопоточного Vulkan типів рендерів для простих та складних моделей.

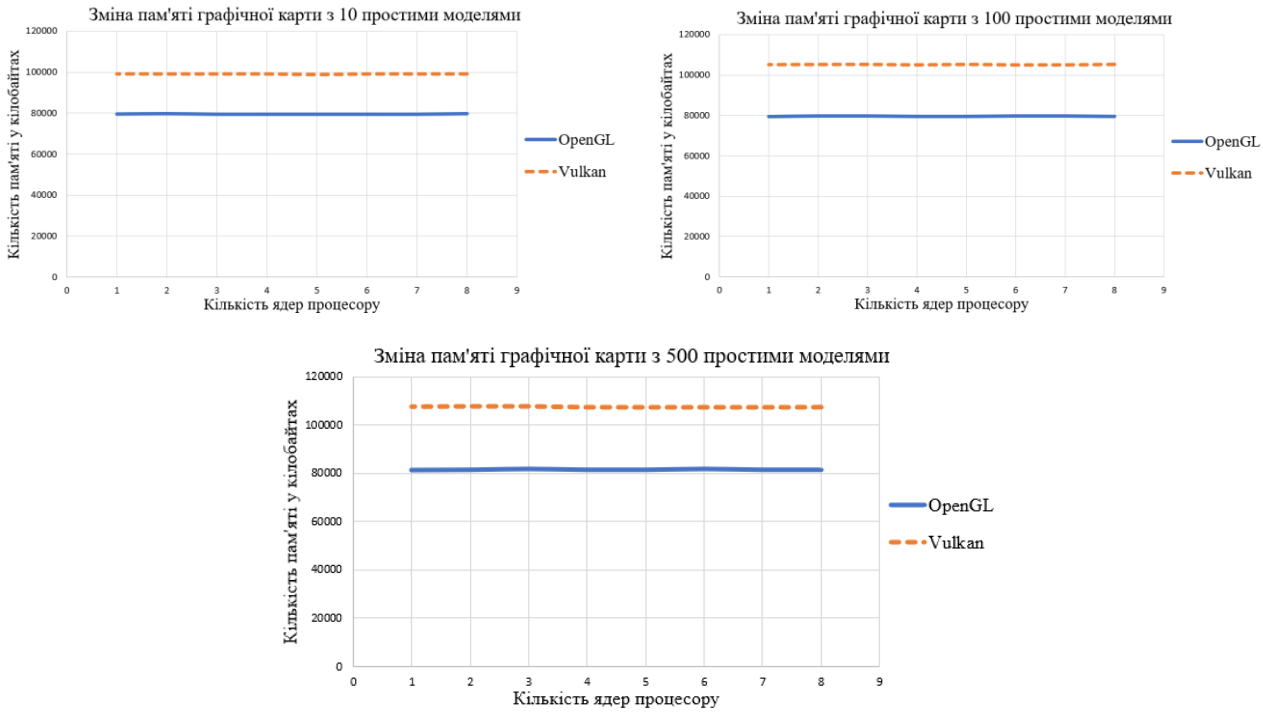

Рисунок 4.5 – Графіки зміні пам’яті графічної карти з простими моделями

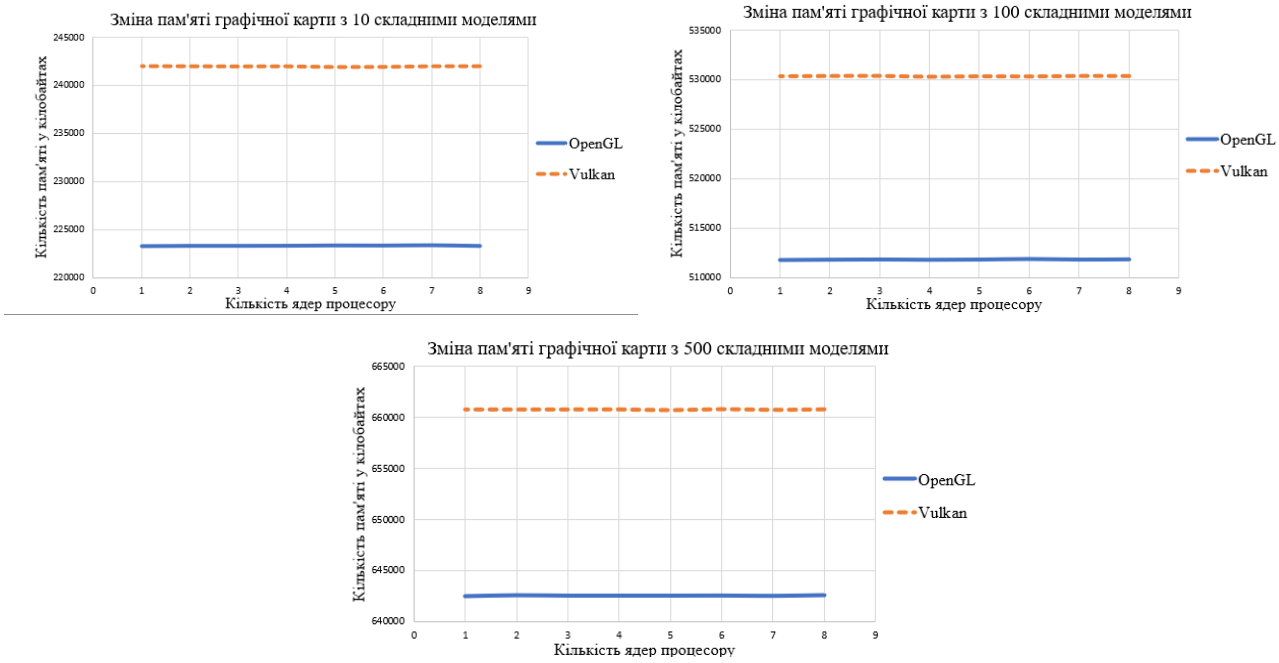

Рисунок 4.6 – Графіки зміні пам’яті графічної карти з складними моделями

Як видно з рисунків 4.5 та 4.6 пам’ять графічної карти для рендеру анімаційних моделей не збільшується зі збільшенням ядер процесору.

Таблиця 4.9 показує різницю у кількості пам’яті графічної карти у процентному відношенні між OpenGL та Vulkan для різної кількості ядер для простих моделей. А таблиця 4.10 для складних моделей.

Таблиця 4.9 – Різниця пам’яті графічної карти для простих моделей

Кількість ядер	Різниця GPU пам’яті при 10 простих моделей	Різниця GPU пам’яті при 100 простих моделей	Різниця GPU пам’яті при 500 простих моделей
8	25%	32%	32%
7	25%	32%	32%
6	25%	32%	32%
5	25%	32%	32%
4	25%	32%	32%
3	25%	32%	32%
2	25%	32%	32%
1	25%	32%	32%

Таблиця 4.10 – Різниця пам’яті графічної карти для складних моделей

Кількість ядер	Різниця GPU пам’яті при 10 складних моделей	Різниця GPU пам’яті при 100 складних моделей	Різниця GPU пам’яті при 500 складних моделей
8	8%	4%	3%
7	8%	4%	3%
6	8%	4%	3%
5	8%	4%	3%
4	8%	4%	3%
3	8%	4%	3%
2	8%	4%	3%
1	8%	4%	3%

Як видно з таблиці 4.9 Vulkan застосовує на 25% більше пам’яті для 10 простих моделей та на 32% для 100 та 500 моделей.

Для складних моделей, дані для яких наведені у таблиці 4.10, Vulkan застосовує на 8% більше пам’яті для 10 простих моделей та на 3-4% для 100 та 500 моделей.

У середньому для простих моделей графічної пам’яті Vulkan використовується на 25 мегабайт більше ніж для OpenGL. Для складних моделей на 18 мегабайт більше.

4.5 Аналіз зміни оперативної пам’яті зі збільшенням кількості ядер

На рисунку 4.7 та 4.8 зображено графіки зміни кількості використаної оперативної пам’яті зі зміною кількості ядер для однопоточного OpenGL та багатопоточного Vulkan типів рендерів для простих та складних моделей.

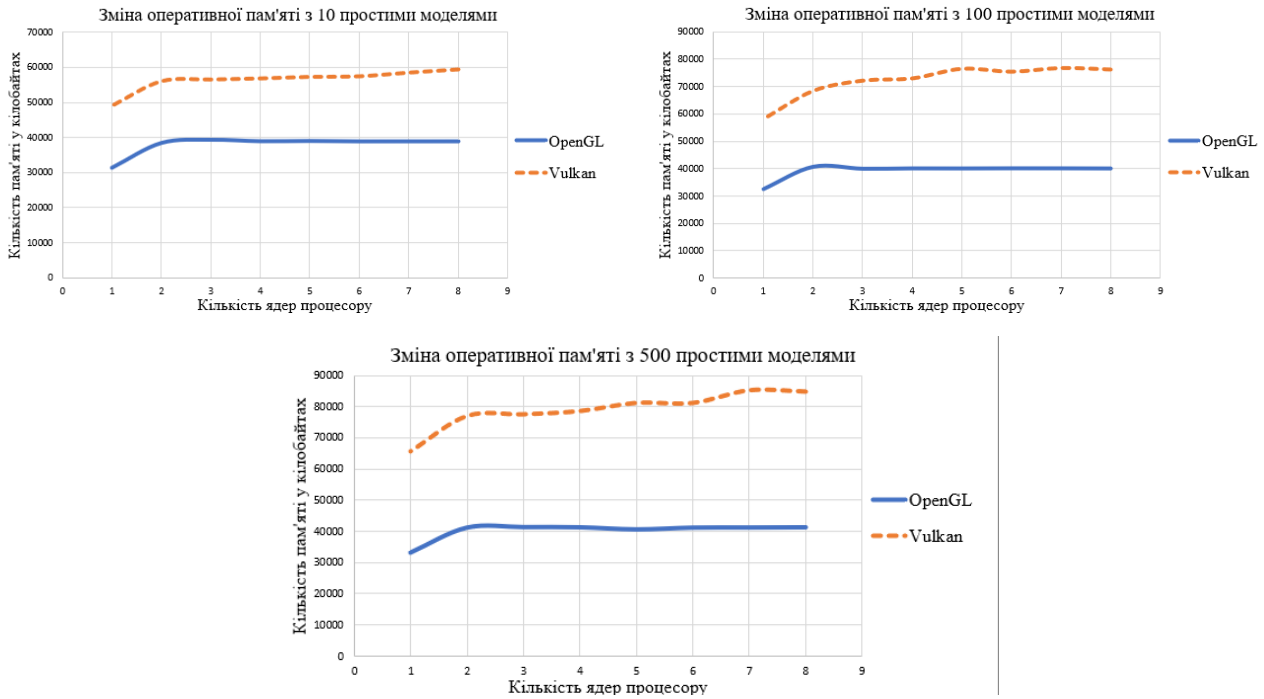

Рисунок 4.7 – Графіки зміні оперативної пам’яті з простими моделями

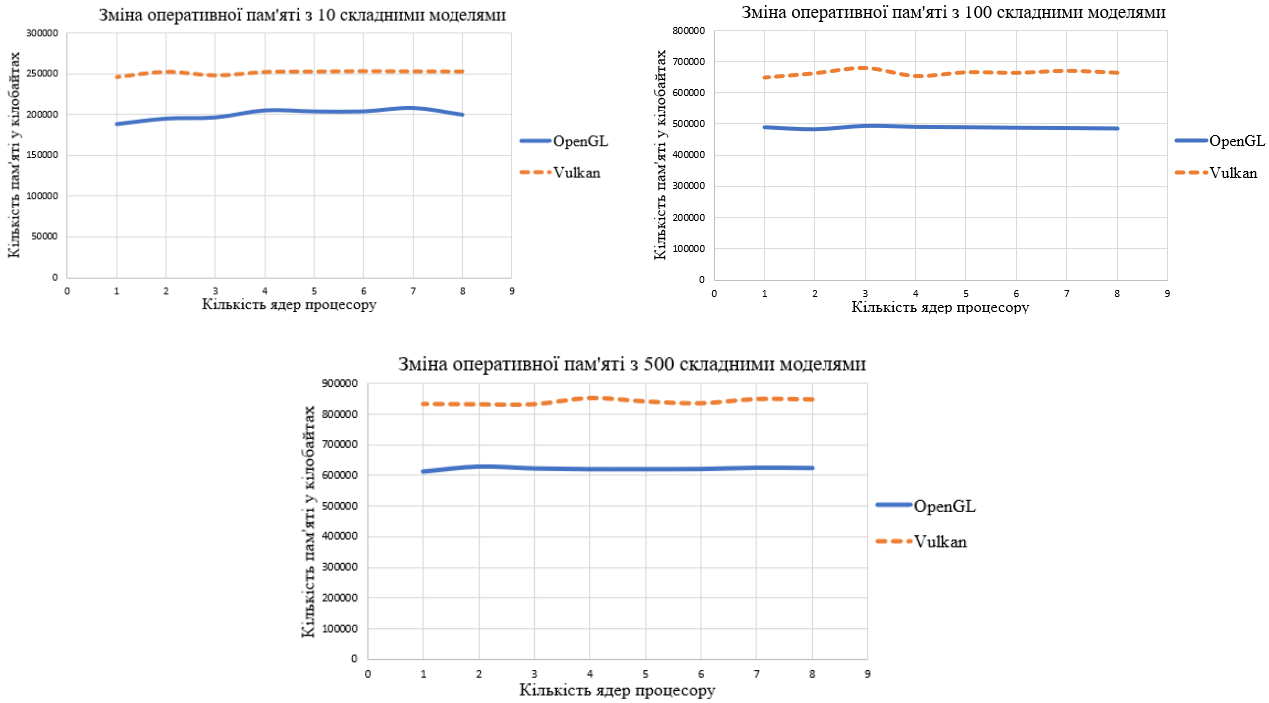

Рисунок 4.8 – Графіки зміні оперативної пам’яті з складними моделями

Як видно з рисунків 4.7 та 4.8 оперативна пам’ять для рендеру анімаційних моделей не збільшується зі збільшенням ядер процесору, так само як і пам’ять графічної карти.

Таблиця 4.11 показує різницю у кількості оперативної пам’яті у процентному відношенні між OpenGL та Vulkan для різної кількості ядер для простих моделей. А таблиця 4.12 для складних моделей.

Таблиця 4.11 – Різниця оперативної пам’яті для простих моделей

Кількість ядер	Різниця RAM пам’яті при 10 простих моделей	Різниця RAM пам’яті при 100 простих моделей	Різниця RAM пам’яті при 500 простих моделей
8	53%	90%	104%
7	51%	91%	106%
6	48%	88%	96%
5	47%	91%	99%
4	46%	82%	89%
3	44%	80%	87%
2	46%	68%	86%
1	56%	78%	98%

Таблиця 4.12 – Різниця оперативної пам’яті для складних моделей

Кількість ядер	Різниця RAM пам’яті при 10 складних моделей	Різниця RAM пам’яті при 100 складних моделей	Різниця RAM пам’яті при 500 складних моделей
8	26%	37%	36%
7	22%	38%	36%
6	24%	36%	35%
5	24%	36%	36%
4	23%	33%	37%
3	26%	37%	34%
2	29%	37%	32%
1	30%	32%	36%

Як видно з таблиці 4.11 Vulkan застосовує на 50% більше пам’яті для 10 простих моделей та на 90% для 100 та 500 моделей.

Для складних моделей, дані для яких наведені у таблиці 4.12, Vulkan застосовує на 25% більше пам’яті для 10 простих моделей та на 30-35% для 100 та 500 моделей.

У середньому для простих моделей оперативної пам’яті для Vulkan використовується на 20 мегабайт більше ніж для OpenGL для 10 моделей. На 35 мегабайт більше для 100 моделей та на 42 мегабайти більше для 500 моделей.

Для складних моделей на 50 мегабайт більше для 10 моделей, на 174 мегабайти більше для 100 моделей та на 218 мегабайт більше для 500 моделей.

Висновки до розділу 4

У четвертому розділі для кожного типу кожної кількості моделей було зібрано інформацію експериментальним шляхом.

Проаналізувавши цю інформацію можна стверджувати що ріст продуктивності зі збільшенням ядер процесору з багатопоточною моделлю рендеру на Vulkan є досить значним для використання. Але не у всіх випадках. Як що графічні моделі для обробки занадто прості, як 3д куби з декількома вершинами, велика кількість ядер не буде грати ніякої ролі, тому що декілька ядер буде досить для обробки усіх таких моделей.

Для складних моделей зі скелетною анімацією та великою кількість вершин кожне ядро додає суттєвий зріст продуктівності, оскільки навантаження на процесор досить велике, паралельна обробка таких моделей на різних ядер з кожним ядром збільшує кількість кадрів за секунду на 50%.

Також було проаналізовано та показано, що багатопоточний рендер навантажує усі ядра процесору однаково, а однопоточний рендер перевантажує лише одне ядро.

Проаналізувавши споживання пам’яті було встановлено що вона не збільшується зі збільшенням кількості ядер, тільки зі збільшенням кількості моделей.

5 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1 Вимоги безпеки при виконанні робіт на робочому місці

У даному розділі розглядаються вимоги встановлені що до безпеки при виконанні робіт на робочому місці згідно чинного законодавства, таких як санітарні норми України, накази різних міністерств чи закони прописані у конституції.

На робочому місці працівника інколи виникають шкідливі виробничі фактори, тривалий вплив яких на працюючого у визначених умовах приведе до зниження працездатності, захворювання або навіть негативного впливу на здоров'я нащадків. Наприклад: тепловий удар, опромінення, поганий рівень освітлення, шкідливі речовини в повітрі, удар електрострумом, висока напруга електричної мережі, високий рівень шуму, отруєння та інше.

Використання персонального комп’ютера під час роботи стає дедалі більшим занепокоєнням майже всіх підприємств. Поряд із ризиками безпеки, які зростають для компанії, також стоїть ризик пов’язаний з високим рівнем напруженості яке призводить до зросту відволікання уваги працівника що призводить до збільшення витрат для організацій.

Робота на комп’ютері пов'язана з наступними шкідливими факторами:

– недостатнє освітлення природним світлом;
– розумове перенапруження;
– монотонність праці;
– відволікання уваги;
– емоційне перевантаження.

Неправильна організація може сприяти розвитку остеохондрозу, погіршенню зору, напрузі кінцівок, гастриту, та погіршення самопочуття в цілому.

Для вирішення цих проблем держава розробила нормативно-правові акти, які описують правила охорони праці, техніки безпеки при роботі.

Для інженера-програміста існують наступні нормативно-правові акти:

- Закон України від 14 жовтня 1992 р. № 2694-XII “Про охорону праці”. Редакція від 16.10.2020 [60];
- Кодекс законів про працю України від 10 грудня 1971 р. № 322-VIII. Редакція від 25.10.2020 [61];
- Закон України від 23 вересня 1999 р. № 1105-XIV “Про загальнообов`язкове державне соціальне страхування”. Редакція від 25.10.2020 [62];
- Постанова Кабінету Міністрів України від 01 серпня 1992 р. № 442 “Про затвердження Порядку проведення атестації робочих місць за умовами праці”. Редакція від 28.10.2016 [63];
- Постанова Кабінету Міністрів України від 17.04.2019 №337 “Про затвердження Порядку розслідування та обліку нещасних випадків, професійних захворювань та аварій на виробництві” [64];
- НПАОП 0.00-4.12-05 Типове положення про порядок проведення навчання і перевірки знань з питань охорони праці, затверджене наказом Держнаглядохоронпраці від 26.01.2005 №15. Зареєстрованого в Мін’юсті України 15.02.2005 за №231/10511. Редакція від 14.04.2017 [65];
- НПАОП 0.700-7.15-18 Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями, затверджене наказом Міністерства соціальної політики України від 14.02.2018 № 207. Зареєстрованого в Мін’юсті України 25.04.2018 за №508/31960 [66];
- НПАОП 0.00-7.14-17 Вимоги безпеки та захисту здоров’я під час використання виробничого обладнання працівниками, затверджене наказом Міністерства соціальної політики України від 28.12.2017 № 2072. Зареєстрованого в Мін’юсті України 23.01.2018 за №97/31549 [67];
- Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин, ДСанПін 3.3.2.007-98. затверджено постановою головного санітарного лікаря України від 10 грудня 1998 року [68];

- ДСТУ 7299:2013 Дизайн і ергономіка. Робоче місце оператора. Взаємне розташування елементів робочого місця. Загальні вимоги ергономіки, затверджено та введено в дію наказом міністерства економічного розвитку і торгівлі України 14.10.2013 № 1231. Зареєстрованого в Мін'юсті України 14.02.2012 за №226/20539 [69];
- Державні будівельні норми України «Природне і штучне освітлення» ДБН В.2.5-28:2018, затверджені наказом Міністерства регіонального розвитку, будівництва та житлово-комунального господарства України 03.10.2018 № 264, введено в дію з 01.03.2019 [70];
- ДСН 3.3.6.042-99 Державні санітарні норми мікроклімату виробничих приміщень, затверджені Постановою головного санітарного лікаря України № 42 від 1 грудня 1999 року [71];
- НАПБ А.01.001-2014 Правила пожежної безпеки в Україні, затверджений наказом Міністерства внутрішніх справ України від 30.12.2014 № 1417 і зареєстрований у Міністерстві юстиції України 05.03.2015 за № 252/26697 [72] ;
- Державні санітарні норми виробничої загальної та локальної вібрації ДСН 3.3.6.039-99, початок дії від 01.12.1999 [73];
- Кодекс цивільного захисту України. Редакція від 16.10.2020 [74].

5.2 Шкідливі виробничі фактори на робочому місці

Основні шкідливі виробничі фактори на робочому місці для інженера-програміста можна поділити на виробничу та фізичну.

Виробничі шкідливі фактори:

- підвищена або знижена температура повітря робочої зони, а також поверхонь обладнання, матеріалів;
- підвищена запиленість та загазованість повітря робочої зони;
- підвищена напруженість електричного і магнітного полів;
- недостатня освіченість робочих зон;

- невідповідність ергономічних показників робочого місця діючим вимогам;
- підвищена або низька температура повітря робочої зони;
- підвищена температура поверхонь ПК;
- підвищена яскравість;
- ризик виникнення пожеж;
- підвищена контрастність;
- підвищений рівень статичної електрики.

Фізичні шкідливі фактори:

- нервово-емоційне напруження;
- інтелектуальне навантаження (сприйняття сигналів і їх оцінка);
- сенсорна навантаження (тривалість зосередженого спостереження, щільність сигналів, спостереження за екранами відеотерміналів);
- емоційне навантаження (ступінь відповідальності за результат своєї діяльності);
- монотонність навантажень (тривалість виконання повторюваних операцій);
- напруження зору.

Щоб знизити шкідливий вплив при роботі інженером-програмістом, далі будуть розглянуті рекомендації щодо організації робочого місця для зменшення шкідливого впливу при роботі з ЕОМ. Також перевіримо, чи відповідає робоче місце, де розроблялась програма, параметрам мікроклімату, освітленості, шуму та вібрації.

5.2.1 Характеристика робочого місця

Робоче місце, де була розроблена програма, має наступні характеристики:

- кількість працюючих: 1 людина;
- довжина приміщення: 5 м;
- ширина приміщення: 3 м;
- висота приміщення: 2.8 м.

За підрахунками отримаємо наступні показники:

- загальна площа дорівнює площі на одне робоче місце: 15м2;
- об’єм приміщення дорівнює об’єму на одне робоче місце: 62.4 м3.

Відповідно до вимог з матеріалу [68] – обсяг приміщення має бути не менше 20 м3 на людину; - площа приміщення має бути не менша ніж 6 м2 на людину. Виявлено що приміщення відповідає стандартам.

Приміщення відноситься до класу звичайних приміщень, тому що у цьому приміщенні відсутні фактори, через які може бути створена підвищена небезпека.

5.2.2 Освітлення

Штучне освітлення не робить шкідливого впливу на стан працівника за умови належного проектування. У цьому випадку це покращує умови зорової роботи, зменшує втому, підвищує продуктивність праці, благотворно впливає на виробниче середовище, надаючи позитивний психологічний вплив на працівника, підвищує безпеку праці та зменшує травматизм.

Якщо освітлення на робочому місці було розроблено неправильно, або воно недостатнє, людина під час роботи може страждати від перенапруження очей, погіршення уваги, початку передчасної втоми. Надмірно яскраве освітлення викликає сліпоту, роздратування і сльозотечу в очах. Неправильний напрямок світла на робочому місці може створити різкі тіні, відблиски, дезорієнтувати працівника. Всі ці причини можуть призвести до нещасних випадків або професійних захворювань, тому важливо правильно розрахувати світло.

Освітленість приміщень комп’ютерних центрів нормована ДБН В.2.5-28-2018 [70]. Під час роботи було використано природне одностороннє бічне та штучне освітлення. Виходячи з того, що категорія візуальних робіт на дисплеї комп’ютера належить до III категорії, при звичайних умовах освітленість робочого місця повинна становити від 200 до 400 люкс. Фактична освітленість на робочому місці становить 194,8 лк. Згідно з ДБН В.2.5-28-2018 [70] в приміщенні використовується природне і штучне освітлення. Відповідно до виконаних підрахунків, існуючих джерел світла достатньо для роботи з дисплеєм.

5.2.3 Мікроклімат

Для постійних робочих місць операторів ПК встановлюються оптимальні параметри мікроклімату, якщо виконати їх неможливо, використовуйте допустимі параметри. В таблиці 5.1 наведені оптимальні параметри мікроклімату в приміщеннях, де виконуються роботи операторського типу.

Таблиця 5.1 – Параметри мікроклімату для приміщень з ПК

Період року	Параметри мікроклімату	Величина
Холодний	Температура повітря в приміщенні	22 – 24° С
	Відносна вологість	40-60%
	Швидкість руху повітря	До 0.1 м/с
Теплий	Температура повітря в приміщенні	23 - 25° С
	Відносна вологість	40-60%
	Швидкість руху повітря	До 0.1 – 0.2 м/с

Температура і вологість повітря в приміщенні, виміряні за допомогою приладів, відповідають зазначеним у таблиці для теплого періоду року. З метою нормалізації параметрів мікроклімату слід використовувати кондиціонування повітря в приміщеннях або забезпечувати вентиляційні системи свіжим повітрям. Норми подачі свіжого повітря наведені у табл. 5.2

Таблиця 5.2 – Норми подачі свіжого повітря в приміщення з ПК

Характеристика приміщення	Об'ємна витрата свіжого повітря, що подається в приміщення, $м^3$ на одну людину в годину
Об'єм до 20$м^3$ на людину	не менше 30
20 - 40 $м^3$ на людину	не менше 20

Єдиний ПК, що знаходиться в приміщенні, є джерелом тепла, крім підтримки оптимальних параметрів мікроклімату в приміщенні в холодну пору року, використовуються нагріті поверхні опалювальної системи.

5.2.4 Шум та вібрація

Шум і вібрація негативно впливають на умови праці, шкідливо впливаючи на організм людини. Люди, які працюють в умовах тривалого шуму, відчувають дратівливість, головні болі, запаморочення, втрату пам’яті, підвищену стомлюваність, зниження апетиту, біль у вухах тощо. Також під впливом шуму концентрація уваги знижується, порушуються фізіологічні функції, з’являється втома внаслідок збільшення витрат енергії та психічного напруження погіршується мовна комутація. Все це - причини зниження ефективності та продуктивності праці. У приміщенні, де розроблювалась програма, причинною шуму і вібрації являються комп’ютер, джерело безперебійного живлення з вентилятором та кондиціонер. При їхній роботі рівень вібрації не вище 33 дБ, рівень шуму не повинен перевищувати 50 дБА, що є нормою для даного виду діяльності відповідно до ДСН 3.3.6.039-99 [73].

В якості звукопоглинальних засобів слід використовувати негорючі або негорючі спеціальні перфоровані плити, панелі, мінеральну вату.

5.3 Дії працівників в надзвичайних ситуаціях

В аварійних ситуаціях програміст зобов’язаний:

- у разі раптового збою живлення послідовно вимкніть периферію, процесор та відключіть прилад від мережі;
- у разі появи ознак горіння (дим, запах горіння) вимкніть обладнання, знайдіть джерело займання та вживайте заходів до його усунення, повідомити про це керівництву або системному адміністратору;
- у разі пожежі негайно повідомити про це охорону, вжити необхідних заходів для евакуації людей відповідно до плану евакуації підприємства та розпочати гасіння первинними засобами пожежогасіння.

Головною особливістю дій малих підприємств при виникненні надзвичайних ситуацій є перш за все захист персоналу та відвідувачів..

Стаття 130 Кодексу цивільного захисту України [74] передбачає, що на підприємствах, штат яких не перевищує 50 чоловік, розробляються та затверджуються інструкції щодо дій у випадку надзвичайної ситуації.

Розроблена інструкція не повинна суперечити положенням та вимогам Кодексу цивільного захисту України [74].

Інструкція повинна бути розробленою та підписаною посадовою особою підприємства з питань цивільного захисту. Потім вона затверджується керівником підприємства. Вже після цього кожен працівник повинен ознайомитись і підписати цю інструкцією.

Крім інструкції повинен бути розроблений план евакуації при пожежі або загрозі вибуху.

Висновки до розділу 5

У ході аналізу нормативно правових норм для охорони праці були знайдені та проаналізовані нормативно правові акти. Були надані шкідливі виробничі фактори можуть негативно впливати на робітників, та навіть їх нащадків.

Були проаналізовані норми освітлення, мікроклімату та шуму приміщення. Виявлено що норми де розроблялася програма до диплому є допустимими.

Також було надано рекомендації дії працівників в надзвичайних ситуаціях, таких як збій живлення, ознаки горіння та інші.

ВИСНОВКИ

Результатом цієї дипломної роботи є визначення ефективності відображення комп’ютерної графіки реального часу, а саме визначення впливу використання можливостей багатоядерних систем під час обробки та генерації кадрів анімованих тривимірних об’єктів.

Головною особливістю є порівняння та використання графічних програмних інтерфейсів OpenGL та Vulkan. Графічний програмний інтерфейс Vulkan прийшов на заміну інтерфейсу OpenGL, який не може повноцінно розкрити потенціал багатоядерних систем через те, що був розроблений у часи коли багатоядерні системи не були досить поширеними. У свою чергу інтерфейс Vulkan має усі на то необхідні засоби для того, що би максимально ефективно розподілити роботу по обробці та зміни змінних при генеруванні кадрів до постійно змінюваних тривимірних анімованих моделей.

Результатом дослідження є наступні висновки:

- як що тривимірний об’єкт занадто простий, то для них буде досить невеликої кількості ядер, оскільки зі зростанням ядер продуктивність не змінюється;
- мала кількість об’єктів, яка не перевищує чи перевищує на декілька значень кількість ядер буде задіювати лише частину усіх ядер, оскільки одне ядро буде встигати виконати обробку декількох моделей, перед тим як інше ядро матиме змогу «здобути» об’єкт до обробки;
- велика кількість складних (100-500) тривимірних об’єктів, такі як об’єкти зі скелетною анімацією та великою кількістю вершин, буде максимально ефективно використовувати усі ядра процесору. З додаванням кожного ядра зріст продуктивності буде лінійним;
- розподіл роботи по ядрам процесору з OpenGL використовуючи одне ядро не є ефективним, оскільки тільки одне ядро навантажено на 100%. При багатоядерному рендері з Vulkan усі ядра навантажені на близько 70% (+-20%).

БІБЛІОГРАФІЧНИЙ СПИСОК

1. «MS-DOS графічне програмування» [Ел. ресурс]. Available: https://retrocomputing.stackexchange.com/questions/11219/how-did-old-ms-dos-games-utilize-various-graphic-cards. [Дата звернення: 15.11.2020].
2. Kessenich J. OpenGL Programming Guide Ninth Edition J. Kessenich, G. Sellers, D. Shreiner – Boston: Addison–Wesley. – 976 с.
3. Sherrod A. Beginning DirectX R 11 Game Programming A. Sherrod, W. Jones – Boston: Cengage Learning PTR, 2011. – 384 с.
4. «DirectX 12» [Ел. ресурс]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d12/what-is-directx-12-. [Дата звернення: 15.11.2020].
5. Begbie C. Metal by Tutorials Second Edition C. Begbie, M. Horga – Boston: Razeware LLC, 2019. – 735 с.
6. Seller G. Vulkan Programming Guide G. Seller – Boston: Addison-Wesley Professional, 2016. – 478 с.
7. Tomas A. Real-Time Rendering 3rd edition A. Tomas, E. Haines, N. Hoffman – Massachusetts: A K Peters, Ltd, 2008. – 1045 с.
8. «Графічний конвеєр» [Ел ресурс]. Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview. [Дата звернення: 17.11.2020].
9. «Вивчити OpenGL» [Ел ресурс]. Available: https://learnopengl.com/. [Дата звернення: 18.11.2020].
10. «Навчання Vulkan» [Ел ресурс]. Available: https://vulkan-tutorial.com/. [Дата звернення: 19.11.2020].
11. «Види 3д моделей» [Ел ресурс]. Available: https://koloro.ua/blog/3d-tekhnologii/3d-model-vidy-urovni-slozhnosti-sostavnye-chasti.html. [Дата звернення: 19.11.2020].
12. «Види 3д анімації» [Ел ресурс]. Available: https://klona.ua/blog/3d-animaciya/kak-ozhivit-geroya-3d-animatsiya-personazhei. [Дата звернення: 20.11.2020].

13. «ОС Windows» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Windows. [Дата звернення: 20.11.2020].
14. «ОС Linux» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Linux. [Дата звернення: 20.11.2020].
15. «ОС MacOS» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/MacOS. [Дата звернення: 20.11.2020].
16. «Центральний процесор» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Центральный_процессор. [Дата звернення: 21.11.2020].
17. «Відеокарта» [Ел ресурс].Available: https://en.wikipedia.org/wiki/Video_card. [Дата звернення: 21.11.2020].
18. «ARM Архітектура» [Ел ресурс]. Available: https://en.wikipedia.org/wiki/ARM_architecture. [Дата звернення: 21.11.2020].
19. «X86 Архітектура» [Ел ресурс]. Available: https://en.wikipedia.org/wiki/X86. [Дата звернення: 22.11.2020].
20. «X86_64 Архітектура» [Ел ресурс]. Available: https://en.wikipedia.org/wiki/X86-64. [Дата звернення: 22.11.2020].
21. «Відеокарти Radeon» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Radeon. [Дата звернення: 23.11.2020].
22. «Відеокарти Nvidia» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Nvidia. [Дата звернення: 24.11.2020].
23. «Архітектури комп'ютеру» [Ел ресурс]. Available: https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/. [Дата звернення: 25.11.2020].
24. «Монітор ресурсів» [Ел ресурс]. Available: http://www.oszone.net/10487#01. [Дата звернення: 26.11.2020].
25. «Багатоядерні процесори» [Ел ресурс]. Available: https://docs.microsoft.com/en-us/windows/win32/dxtecharts/game-timing-and-multicore-processors. [Дата звернення: 27.11.2020].

26. «What is Use Case Diagram?» [Ел. ресурс]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/. [Дата звернення: 28.11.2020].
27. «Інженерія програмного забезпечення» [Ел ресурс]. Available: https://er.nau.edu.ua/bitstream/NAU/25927/1/ВЕРСТКА%201-388.pdf. [Дата звернення: 28.11.2020].
28. «Основні елементи інтерфейсу користувача» [Ел ресурс]. Available: https://sites.google.com/site/srd18docs/graficeskij-interfejs/osnovnye-elementy-polzovatelskogo-interfejsa. [Дата звернення: 28.11.2020].
29. «Система керування версіями» [Ел ресурс]. Available: https://uk.wikipedia.org/wiki/Система_керування_версіями. [Дата звернення: 28.11.2020].
30. «Система GIT» [Ел ресурс]. Available: https://uk.wikipedia.org/wiki/Git. [Дата звернення: 29.11.2020].
31. «Система Make» [Ел ресурс]. Available: https://uk.wikipedia.org/wiki/Make. [Дата звернення: 29.11.2020].
32. «Принцип SOLID» [Ел ресурс]. Available: https://web-creator.ru/articles/solid. [Дата звернення: 29.11.2020].
33. «Принцип KISS» [Ел ресурс]. Available: https://web-creator.ru/articles/kiss. [Дата звернення: 29.11.2020].
34. «Шаблон проектування Інтерфейс» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Интерфейс_(шаблон_проектирования). [Дата звернення: 30.11.2020].
35. «Шаблон проектування Стратегія» [Ел ресурс]. Available: https://ru.wikipedia.org/wiki/Стратегия_(шаблон_проектирования). [Дата звернення: 30.11.2020].
36. «Стратегії тестування» [Ел. ресурс]. Available: http://dit.isuct.ru/Publish_RUP/core.base_rup/guidances/concepts/test_strategy_9981F03E.html. [Дата звернення: 30.11.2020].

37. «Тестування по стратегії чорного ящика» [Ел. ресурс]. Available: https://ru.wikipedia.org/wiki/Тестирование_по_стратегии_чёрного_ящика. [Дата звернення: 1.12.2020].
38. Эшби Р. , Чёрный ящик // Введение в кибернетику, 1959.
39. «Тестування білим ящиком» [Ел. ресурс]. Available: https://ru.wikipedia.org/wiki/Тестирование_белого_ящика. [Дата звернення: 1.12.2019].
40. «Історія обчислювальних технологій» [Ел. ресурс]. Available: https://web.archive.org/web/20070405172339/http://accad.osu.edu/~waynec/history/lesson1.html. [Дата звернення: 1.12.2020].
41. «Виникнення технології комп’ютерної графіки» [Ел. ресурс]. Available: https://web.archive.org/web/20070405181508/http://accad.osu.edu/~waynec/history/lesson2.html. [Дата звернення: 2.12.2020].
42. «Розвиток комп'ютерної графіки» [Ел. ресурс]. Available: https://web.archive.org/web/20070405172607/http://accad.osu.edu/~waynec/history/lesson3.html. [Дата звернення: 2.12.2020].
43. «Основні дослідження комп'ютерної графіки» [Ел. ресурс]. Available: https://web.archive.org/web/20060906015152/http://accad.osu.edu/~waynec/history/lesson4.html. [Дата звернення: 2.12.2020].
44. «Університецькі дослідження комп'ютерної графіки» [Ел. ресурс]. Available:https://web.archive.org/web/20070501041000/http://accad.osu.edu/~waynec/history/lesson5.html. [Дата звернення: 2.12.2020].
45. «Комп'ютерна графіка для реклами» [Ел. ресурс]. Available: https://web.archive.org/web/20070405172514/http://accad.osu.edu/~waynec/history/lesson6.html. [Дата звернення: 3.12.2020].
46. «Графічні стандарти» [Ел. ресурс]. Available: https://web.archive.org/web/20070402165202/http://accad.osu.edu/~waynec/history/lesson7.html. [Дата звернення: 3.12.2020].

47. «Комерційні анімаційні компанії» [Ел. ресурс]. Available: https://web.archive.org/web/20070506064841/http://accad.osu.edu/~waynec/history/lesson8.html. [Дата звернення: 3.12.2020].
48. «Комп'ютерні художники» [Ел. ресурс]. Available: https://web.archive.org/web/20070403055941/http://accad.osu.edu/~waynec/history/lesson9.html. [Дата звернення: 3.12.2020].
49. «Програмне забезпечення CAD» [Ел. ресурс]. Available: https://web.archive.org/web/20070402202233/http://accad.osu.edu/~waynec/history/lesson10.html. [Дата звернення: 3.12.2020].
50. «Виробничі компанії» [Ел. ресурс]. Available: https://web.archive.org/web/20070313205609/http://accad.osu.edu/~waynec/history/lesson11.html. [Дата звернення: 3.12.2020].
51. «Аналогові підходи та композитування» [Ел. ресурс]. Available: https://web.archive.org/web/20070328205521/http://accad.osu.edu/~waynec/history/lesson12.html. [Дата звернення: 3.12.2020].
52. «Моделювання польоту» [Ел. ресурс]. Available: https://web.archive.org/web/20061203065540/http://accad.osu.edu/~waynec/history/lesson13.html. [Дата звернення: 3.12.2020].
53. «CG у фільмах» [Ел. ресурс]. Available: https://web.archive.org/web/20070517144718/http://accad.osu.edu/~waynec/history/lesson14.html. [Дата звернення: 3.12.2020].
54. «Відображення обладнання та робочої станції» [Ел. ресурс]. Available: https://web.archive.org/web/20070513132711/http://accad.osu.edu/~waynec/history/lesson15.html. [Дата звернення: 3.12.2020].
55. «Графічний інтерфейс і персональний комп'ютер» [Ел. ресурс]. Available: https://web.archive.org/web/20070531212526/http://accad.osu.edu/~waynec/history/lesson16.html. [Дата звернення: 3.12.2020].
56. «Віртуальна реальність та штучне середовище» [Ел. ресурс]. Available: https://web.archive.org/web/20070427081552/http://accad.osu.edu/~waynec/history/lesson17.html. [Дата звернення: 3.12.2020].

57. «Наукова візуалізація» [Ел. ресурс]. Available: https://web.archive.org/web/20070402201801/http://accad.osu.edu/~waynec/history/lesson18.html. [Дата звернення: 3.12.2020].
58. «Візуальний реалізм та складність зображення» [Ел. ресурс]. Available: https://web.archive.org/web/20070604164225/http://accad.osu.edu/~waynec/history/lesson19.html. [Дата звернення: 3.12.2020].
59. «Комп'ютерна графіка іконки» [Ел. ресурс]. Available: https://web.archive.org/web/20070428033134/http://accad.osu.edu/~waynec/history/lesson20.html. [Дата звернення: 3.12.2020].
60. Закон України від 14 жовтня 1992 р. № 2694-XII "Про охорону праці". Редакція від 16.10.2020
61. Кодекс законів про працю України від 10 грудня 1971 р. № 322-VIII. Редакція від 25.10.2020
62. Закон України від 23 вересня 1999 р. № 1105-XIV "Про загальнообов`язкове державне соціальне страхування". Редакція від 25.10.2020
63. Постанова Кабінету Міністрів України від 01 серпня 1992 р. № 442 "Про затвердження Порядку проведення атестації робочих місць за умовами праці". Редакція від 28.10.2016
64. Постанова Кабінету Міністрів України від 17.04.2019 №337 "Про затвердження Порядку розслідування та обліку нещасних випадків, професійних захворювань та аварій на виробництві"
65. НПАОП 0.00-4.12-05 Типове положення про порядок проведення навчання і перевірки знань з питань охорони праці, затверджене наказом Держнаглядохоронпраці від 26.01.2005 №15. Зареєстрованого в Мін'юсті України 15.02.2005 за №231/10511. Редакція від 14.04.2017
66. НПАОП 0.700-7.15-18 Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями, затверджене наказом Міністерства соціальної політики України від 14.02.2018 № 207. Зареєстрованого в Мін'юсті України 25.04.2018 за №508/31960

67. НПАОП 0.00-7.14-17 Вимоги безпеки та захисту здоров’я під час використання виробничого обладнання працівниками, затверджене наказом Міністерства соціальної політики України від 28.12.2017 № 2072. Зареєстрованого в Мін’юсті України 23.01.2018 за №97/31549
68. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин, ДСанПін 3.3.2.007-98. затверджено постановою головного санітарного лікаря України від 10 грудня 1998 року
69. ДСТУ 7299:2013 Дизайн і ергономіка. Робоче місце оператора. Взаємне розташування елементів робочого місця. Загальні вимоги ергономіки, затверджено та введено в дію наказом міністерства економічного розвитку і торгівлі України 14.10.2013 № 1231. Зареєстрованого в Мін’юсті України 14.02.2012 за №226/20539
70. Державні будівельні норми України «Природне і штучне освітлення» ДБН В.2.5-28:2018, затверджені наказом Міністерства регіонального розвитку, будівництва та житлово-комунального господарства України 03.10.2018 № 264, введено в дію з 01.03.2019
71. ДСН 3.3.6.042-99 Державні санітарні норми мікроклімату виробничих приміщень, затверджені Постановою головного санітарного лікаря України № 42 від 1 грудня 1999 року
72. НАПБ А.01.001-2014 Правила пожежної безпеки в Україні, затверджений наказом Міністерства внутрішніх справ України від 30.12.2014 № 1417 і зареєстрований у Міністерстві юстиції України 05.03.2015 за № 252/26697
73. Державні санітарні норми виробничої загальної та локальної вібрації ДСН 3.3.6.039-99, початок дії від 01.12.1999
74. Кодекс цивільного захисту України. Редакція від 16.10.2020

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Перший проректор Дніпровського
національного університету
залізничного транспорту
імені академіка В. Лазаряна
Б.Є. Боднар

СИСТЕМА ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП`ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Технічне завдання
ЛИСТ ЗАТВЕРДЖЕННЯ
1116130.01189-01-ЛЗ

Завідувач кафедри КІТ
проф. В.І. Шинкаренко
_____

Керівник розробки
доц. О.П. Іванов
_____

Виконавець
студент групи ПЗ1921
І.А. Поліщук
_____

Нормоконтролер
доц. О.С. Куроп’ятник
_____

2020

ЗАТВЕРДЖЕНО
1116130.01189-01

СИСТЕМА ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП`ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

1116130.01189-01

Аркушів 26

2020

АНОТАЦІЯ

Документ 1116130.01189-01 «Система дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням Opengl та Vulkan API. Технічне завдання» входить до складу програмної документації до дипломного проекту.

У даному документі представлене призначення та область застосування програмного продукту, основні вимоги, стадії та строки виконання проекту, техніко-економічні показники, що пред’являються до програмного продукту.

ЗМІСТ

ВСТУП

Передові досягнення науки і техніки в області комп’ютерної анімації, такі як імітація руху або відбиття світла загалом є дуже критичними до часу виконання завдань, які на них покладаються. Задачі, які вирішують більшість систем відображення комп’ютерної графіки є задачами так званого жорсткого реального часу (коли перевищення часу виконання поставлених завдань може призвести до невідворотних наслідків) або м’якого реального часу (коли перевищення часу вирішення небажане, але припустиме).

Візуалізація в реальному часі відноситься до швидкого зображень на комп'ютері. Це найбільш інтерактивна область комп’ютерної графіки. На екрані з'являється зображення, глядач діє або реагує, і цей відгук впливає на те, що генерується далі. Цей цикл реакції та візуалізації відбувається з досить швидкою швидкістю, щоб глядач не бачив окремі зображення, а навпаки, занурювався в динамічний процес.

Саме тому ми звертаємо нашу увагу на можливість розподілу таких задач на декілька ядер процесору за для зменшення загального часу обробки кожного кадру анімації перед його відображенням.

Для виводу графіки на екран використовують центральний процесор та графічний процесор. Центральний процесор обчислює позицію, зміщення, форму об’єктів та передає ці дані до графічного процесору. Графічний процесор у свою чергу перетворює дані таких об’єктів у форму придатну для подальшого виведення на екран.

Історично склалося, що процесор та графічний процесор розвиваються окремо один від одного. Тому зараз, коли центральний процесор досяг успіху у паралельному виконанні задач на декількох ядрах, підходи до обробки та передачі даних на графічний процесор не є ефективними.

Один з таких підходів є використання OpenGL – програмного інтерфейсу до графічного пристрою. Цей інтерфейс розроблявся у 90-их років, коли багатоядерні процесори ще не були досить популярними. Тому методи до обробки

та передачі даних які запроваджує цей інтерфейс не дає змогу робити це паралельно, оскільки будь-яка операція з OpenGL може змінювати його глобальний стан, який не передбачає одночасні зміни і не запроваджує синхронізації щоб захиститися від стана гонитви (ситуація в якій робота чи результат операції залежить від послідовності або тривалості виконання).

Щоб виправити такі недоліки може бути використаний інтерфейс Vulkan. Цей інтерфейс не використовує глобальних об’єктів які неможливо синхронізувати. На томість робота з ним є складніша, оскільки об’єкти які були сховані у OpenGL відтепер мають бути створені та використані розробником.

Одним із таких об’єктів є командний буфер. Командний буфер це буфер, який зберігає заздалегідь заповнені команди які має виконати графічний процесор. Кожен командний буфер може використовуватись незалежно один від одного, тому кожен потік може обчислювати інформацію об’єктів та заповнювати такі буфери окремо. У кінці, ці буфери мають бути відправлені до графічного процесору, де з них буде створене зображення.

1 ПІДСТАВА ДО РОЗРОБКИ

Підставою для розробки даного програмного продукту є наказ №779 ст. «Про призначення керівників та затвердження тем магістерських робіт» від 10.10.2019 р., затверджений ректором Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна.

Тема: «Дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API».

Керівник дипломного проекту – доц. Іванов О.П.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення

За допомогою даного продукту користувач матиме змогу оцінювати ефективність відображення комп’ютерної графіки з використанням графічних програмних інтерфейсів OpenGL та Vulkan. Також він матиме змогу оцінювати ефективність використовуючи різну кількість тривимірних анімованих моделей. Ефективність подається у кількості кадрів які відображає програма за секунду. Так він матиме змогу збирати ці дані на різних системах з різною кількістю ядер процесору, та робити висновки про те, як змінюється ця ефективність, виконуючи тестування з різною кількістю об’єктів та різною кількістю ядер процесору.

2.2 Експлуатаційне призначення

Програмний продукт призначений для забезпечення швидкого збору інформації що до ефективності того чи іншого програмного інтерфейсу на різних системах використовуючи готові тестові дані, такі як анімовані тривимірні об’єкти.

Також дасть можливість зрозуміти який програмний інтерфейс є ефективнішим на конкретній системі.

3 ВИМОГИ ДО ПРОГРАМИ

3.1 Вимоги до функціональних характеристик

Програма повинна:

- надавати можливість відображати підготовлені тривимірні моделі;
- надавати можливість обирати кількість моделей, які будуть відображатися;
- надавати можливість обирати програмний інтерфейс до графічної карти, який буде використовуватися при обробці кадрів;
- робити рендер анімованих тривимірних моделей з заданими налаштуваннями;
- надавати результати циклу відображення моделей у вигляді кількості кадрів за секунду, кількості секунд за кадр та кількості використаних ядер процесору на системі.

Вхідні данні:

- кількість тривимірних об’єктів що буде відображатися на екрані;
- вид програмного інтерфейсу до графічної карти який буде застосовано (OpenGL чи Vulkan);
- вид об’єктів які будуть відображатися. (Прості тривимірні куби чи складні анімовані моделі тварин з скелетною анімацією).

Вихідні данні:

- кількість кадрів за секунду;
- кількість секунд за кадр;
- кількість використаних ядер процесору на системі.

3.2 Вимоги до надійності

Одним із критеріїв правильного функціонування програмного продукту є забезпечення надійності роботи програмного продукту. Вимоги до надійності програмного продукту повинні відповідати наступним вимогам:

- кількість відмов системи не повинна перевищувати однієї відмови на 2000 запусків системи (під відмовою слід вважати непрацездатність системи після її запуску, тобто необхідність запустити систему повторно).

3.3 Умови експлуатації

Для належної роботи програмного забезпечення повинні виконуватися наступні вимоги:

1. ЕОМ повинні відповідати вимогам чинних в Україні стандартів, нормативних актів з охорони праці [1];
2. програмний комплекс повинен використовуватись в приміщеннях, призначених для роботи ЕОМ з наступними кліматичними умовами [2]: температура – 21-25 ^{o}C, відносна вологість повітря 40-60%;
3. комп’ютери повинні відповідати вимогам чинних в Україні стандартів, нормативів з охорони праці.

3.4 Вимоги до складу та параметрів технічних засобів

Для коректного функціонування програмного продукту вимагається наявність ЕОМ, що задовольняє наступним мінімальним вимогам:

- ОС Windows 10;
- ОЗП не менш ніж 4096 Мб;
- інтегрована або дискретна графічна картка з підтримкою Vulkan.

3.5 Вимоги до інформаційної та програмної сумісності

Для функціонування програмного продукту необхідна ОС Windows 10 та остання версія драйверів до графічної карти.

3.6 Вимоги до маркування та упаковки

Програма може зберігатися на змінних носіях (flash-памя’ть). Упаковка продукту повинна забезпечувати захист від механічних пошкоджень. Упаковка повинна мати маркування:

«Система дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням Opengl та Vulkan API».

Розробник: Поліщук І. А.

Версія 1.0

ДНУЗТ, КІТ

2020

3.7 Вимоги до транспортування та зберігання

Транспортування може виконуватись будь-яким способом, що виключає механічний і електромагнітний вплив на носії інформації. Місце збереження повинне відповідати умовам збереження носія, на якому знаходиться програмний комплекс.

Термін збереження обумовлений збереженням інформації на носії. Рекомендується проводити профілактичні роботи з перевірки якості носіїв кожні шість місяців.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу програмної документації повинні входити:

- технічне завдання;
- робочий проект:
  - специфікація;
  - текст програми;
  - опис програми;
  - керівництво користувача. Керівництво по роботі з системою відображення комп`ютерної графіки реального часу на багатоядерних системах.

Вся документація до програми повинна задовольняти вимогам державного стандарту до оформлення програмних документів ГОСТ 19.101-77 «Єдина система програмної документації. Види програм та програмних документів» [3].

5 ВИЗНАЧЕННЯ ВИТРАТ НА ПРОЕКТУВАННЯ ПРОГРАМИ

Капітальні вкладення включають всі витрати, пов’язані з реалізацією проекту. До них відносяться витрати на придбання технологій і обладнання, доставку на підприємство, монтаж і налагодження, на розробку проекту по заміні технологій і обладнання. Якщо у підприємства не вистачає власних коштів на цю заміну, воно може взяти кредит в банку. В цьому випадку капітальні витрати називаються інвестиціями.

Нова техніка, технологія, засоби автоматизації, що розробляються і впроваджуються у виробництво, повинні приносити певний корисний результат – ефект. Ефект може проявлятися у поліпшенні умов праці працюючих (соціальний), в зниженні шкідливого впливу виробництва на навколишнє середовище (екологічний), у підвищенні безпеки держави (оборонний), та, врешті, в економії витрат підприємства на виробництво продукції та збільшенні його прибутку (економічний).

Згідно моделі COCOMO, розмір проекту S вимірюється в рядках коду LOC (KLOC), а трудовитрати в людино-місяцях [4] [5].

$$E = a \cdot S^b \cdot EAF, \tag{5.1}$$

де E – витрати праці на проект (в людино-місяцях);

S^b – розмір коду (в KLOC);

EAF – фактор уточнення витрат (effort adjustment factor)

EAF використовується для адаптації вашої оцінки на основі умов середовища розробки. Існує 15 різних драйверів витрат, які можна використовувати для розрахунку вашого EAF. Вони згруповані в 4 різні категорії; атрибути товару, атрибути комп’ютера, персональні атрибути та атрибути проекту (див. таблицю 5.1). Кожен драйвер витрат оцінюється за шкалою «Дуже низький» до «Надвисокий» залежно від того, як цей драйвер витрат вплине на ваш розвиток. Ці рейтинги базуються на статистичному аналізі історичних даних, зібраних із 83 минулих проектів [6].

$$EAF = \text{RELY} * DATA * CPLX * TIME * STOR * VIRT * TURN * ACAP * \quad (5.2)$$
$$* AEXP * PCAP * VEXP * LEXP * MODP * TOOL * SCED$$

Таблиця 5.1 – Список 15 драйверів витрат та їх рейтинги для COCOMO

Категорія	Драйвер витрат	Дуже низька	Низька	Нормальна	Висока	Дуже висока	Надвисока
Атрибути продукту	RELY Необхідна надійність програмного забезпечення	0.75	**0.88**	1.00	1.15	1.40	-
	DATA Розмір бази даних	-	0.94	**1.00**	1.08	1.16	-
	CPLX Складність продукту	**0.70**	0.85	1.00	1.15	1.30	1.65
Атрибути комп'ютеру	TIME Обмеження часу виконання	-	-	**1.00**	1.11	1.30	1.66
	STOR Основне обмеження зберігання	-	-	**1.00**	1.06	1.21	1.56
	VIRT Стабільність віртуальної машини	-	**0.87**	1.00	1.15	1.30	-
	TURN Час роботи комп'ютера	-	**0.87**	1.00	1.07	1.15	-
Атрибути персоналу	ACAP Досвід аналітика	1.46	1.19	1.00	**0.96**	0.71	-
	AEXP Досвід застосування	1.29	1.13	**1.00**	0.91	0.82	-
	PCAP Досвід програміста	1.42	1.17	**1.00**	0.86	0.70	-
	VEXP Досвід роботи з віртуальною машиною	1.21	1.10	1.00	**0.90**	-	-
	LEXP Мовний досвід	1.14	1.07	1.00	**0.95**	-	-
Атрибути проекту	MODP Сучасні практики програмування	1.24	1.10	1.00	0.91	**0.82**	-
	TOOL Використання програмних засобів	1.24	1.10	1.00	0.91	**0.83**	-
	SCED Необхідний графік розробки	1.23	1.08	**1.00**	1.04	1.10	-

Розрахунок коефіцієнту EAF з обраними значеннями з таблиці 5.1:

$$EAF = 0.88 * 1.00 * 0.70 * 1.00 * 1.00 * 0.87 * 0.87 * 0.96 * 1.00 * 1.00 * 0.9 *$$
$$* 0.95 * 0.82 * 0.83 * 1.00 = 0.28$$

Для простих систем використаємо значення коефіцієнтів з ресурсу [10]:

$$a = 2{,}4; \mathrm{b} = 1{,}05. \quad (5.3)$$

Розмір програмного коду програмного засобу складають:

$$S = 4740; \ KS = 4{,}74, \quad (5.4)$$

$$E = 2{,}4 \cdot 4{,}74^{\,1{,}05} \cdot 0.28 = 3.4. \quad (5.5)$$

Отже, згідно моделі COCOMO, орієнтовні трудовитрати на проект складуть приблизно 3 людино-місяців, що складає 60 робочих днів або 12 робочих

тижнів, 4 робочі тижні на місяць, 5 робочих днів на тиждень, 8 робочих годин на тиждень, одним виконавцем:

$$N_{\text{чол}} = 1;\ N_{\text{міс}} = 3; N_{\text{тиж}} = 12; N_{\text{днів}} = 60; Z_{\text{тиж}} = 4; Z_{\text{днів}} = 5; Z_{\text{год}} = 8. \qquad (5.6)$$

Основними статтями витрат прийняті:

1. Основна заробітна плата.
2. Відрахування на соціальні потреби.
3. Накладні витрати.
4. Витрати на персональний комп'ютер і ліцензійні базові програмні засоби.

Основна заробітна плата (ОЗП) оцінює працю інженера-програміста зі створення програмного продукту і визначається виходячи з кількості розробників, часу виконання розробки, а також заробітної плати.

Таблиця 5.2 – Середньомісячна заробітна плата працівників у сфері інформації та телекомунікаційних технологій за період з січня 2020р. до вересня 2020р. за даними довідкової інформаційної Державної служби статистики України [7].

Рік	Місяць	Середня заробітна плата, грн
2020 Кількість місяців: 9	Січень	23597,00
	Лютий	24864,00
	Березень	33141,00
	Квітень	24109,00
	Травень	24304,00
	Червень	24681,00
	Липень	25922,00
	Серпень	26059,00
	Вересень	26329,00

Відповідно до даних наведених в табл. 5.2. можна розрахувати суму зарплат інженера-програміста за 9 місяців:

$$S_{зарпл} = \sum_{i=1}^{10} \text{Середня заробітна плата за і–й місяць.} \quad (5.7)$$

$$S_{зарпл} = 23597{,}00+24864{,}00 + 33141{,}00 + 24109{,}00 + 24304{,}00+ 24681{,}00 + +25922{,}00 + 26059{,}00 + 26329{,}00 = 233006{,}00. \quad (5.8)$$

Середньомісячну заробітну плату інженера-програміста за останні 9 місяців:

$$S_{міс} = \frac{S_{зарпл}}{N_{stat}}, \quad (5.9)$$

$$S_{міс} = \frac{233006{,}00}{9} = 25889{,}55 \text{ грн/міс.} \quad (5.10)$$

А також погодинну середню заробітну плату:

$$N_{год} = \frac{S_{міс}}{Z_{тиж} * Z_{днів} * Z_{год}}, \quad (5.11)$$

$$N_{год} = \frac{25889.55}{4*5*8} = 161.8 \text{ грн/год.} \quad (5.12)$$

Описаний в проекті програмний продукт буде розроблений одним програмістом в період з 01.09.20 до 01.12.20, що складає 60 днів або 12 робочих тижнів. Витрати робочого часу прийняті за 40 годин у тиждень. Погодинна ставка кваліфікованого інженера–програміста складає 161.8 грн/год.

Таким чином, витрачено робочого часу:

$$t_{розробки} = N_{чол} \cdot N_{тиж} \cdot N_{год}, \quad (5.13)$$

де $N_{чол}$ – кількість виконавців, *чол*;

$N_{тиж}$ – тривалість розробки;

$N_{год}$ – витрати робочого часу, год;

$$t_{розробки} = 1 \cdot 12 \cdot 40 = 480 \text{ чол/год.} \quad (5.14)$$

ОЗП визначається за формулою:

$$ОЗП = t_{розробки} \cdot N \cdot K_{KB}, \quad (5.15)$$

де N – погодинна ставка;

K_{KB} – коефіцієнт кваліфікації програміста, приймається

$$K_{KB} = 0{,}75 \text{ грн/год.} \quad (5.16)$$

ОЗП складає:

$$\text{ОЗП} = 480 \cdot 161.8 \cdot 0{,}75 = 58248 \text{ грн.} \quad (5.17)$$

Відповідно до чинного Законодавства України нарахування на заробітну плату у вигляді Єдиного внеску на загальнообов'язкове державне соціальне страхування (ЄСВ) становить 22% від окладу працівника [8]. Таким чином ФОП з нарахування становить:

$$C_{\text{соц}} = \frac{\text{ОЗП} \cdot 22\%}{100\%}, \quad (5.18)$$

$$C_{\text{соц}} = \frac{58248 \cdot 22\%}{100\%} = 12814.56 \text{ грн.} \quad (5.19)$$

Отримані результати за (15) та (19) підсумовуються.

$$\text{ОПВ} = C_{\text{соц}} + \text{ОЗП} = 71062.56 \frac{\text{грн}}{\text{год}}. \quad (5.20)$$

Вони визначають основні прямі витрати. Накладні витрати враховують загально господарчі витрати по забезпеченню проведення роботи: витрати на опалення, електроенергію, амортизація будівель, зарплату адміністративного персоналу та інше. Вони становлять 30-40 % від суми прямих витрат на оплату праці:

$$\text{С}_{\text{накл}} = \frac{\text{ОЗП} \cdot 35\%}{100\%}, \quad (5.21)$$

$$\text{С}_{\text{накл}} = \frac{58248 \cdot 35\%}{100\%} = 20386.8 \text{ грн.} \quad (5.22)$$

На протязі усього терміну використання нової техніки підприємство щорічно витрачає певні кошти, пов’язані з її експлуатацією.

Накладні витрати на проект визначаються терміном розробки програмної системи в залежності від вартості комп’ютеру та інших складових і включають в себе:

– вартість витратних матеріалів;

– витрати на ремонт;

– заробітна плата ремонтника;
– оренда приміщення;
– додаткові витрати – прибирання приміщення, охорона, оренда, комунальні послуги;
– амортизаційні витрати на персональний комп'ютер і програмне забезпечення;

Розрахунок витрат на електроенергію

Витрати на електроенергію ($С_{ел}$) визначаються за формулою:

$$С_{ел}=Р\cdot В\cdot Т_{розр}, \quad (5.23)$$

де Р – середня потужність комп’ютера та допоміжних споживачів електричної енергії, приймаємо за 0,32 *кВт/год;* [9].

В – вартість 1 *кВт* за даними з сайту Міністерства Фінансів складає 1,7 грн [10].

$Т_{розр}$ – час роботи з ЕВМ, прийнято рівним робочому часу, розраховується за формулою:

$$С_{ел} = 0{,}32\cdot 1{,}7\cdot 480=261.12 \text{грн.} \quad (5.24)$$

Витрати на витратні матеріали ($С_{вм}$) протягом всього терміну експлуатації приблизно 10% від вартості комп’ютеру. Вартість робочої станції приймається 45 000 грн., термін експлуатації – 5 років. За робочу станцію приймається ноутбук HP Spectre x360 [11] Отже, можна визначити ці витрати за період створення програмного засобу:

$$С_{вм} = В_{ком}\cdot \frac{N_Д}{N_{експ}\cdot 365}\cdot \frac{10\%}{100\%}, \quad (5.25)$$

де $B_{КОМ}$ – вартість персонального комп’ютеру;

$N_Д$ – кількість днів розробки програмного продукту;

$N_{експ}$ – термін експлуатації персонального комп’ютеру, 5 років

Витрати на витратні матеріали визначаються так:

$$C_{вм} = 45000\cdot \frac{60}{5\cdot 365}\cdot \frac{10}{100} = 147.94 \text{ грн.} \quad (5.26)$$

Заробітна плата ремонтника ($C_{рем}$) визначена наступним чином: на ремонт 50 комп’ютерів потрібен один інженер-системотехнік. Його

середньомісячна заробітна плата за даними work.ua приймається 18000 грн [9]. Тоді в перерахунку на один комп'ютер його заробітна плата складає:

$$C_{\text{рем}} = \frac{C_{\text{рем}}'}{N_{КОМ}},$$

$$C_{\text{рем}} = \frac{18000}{50} * 2 = 720, \tag{5.27}$$

де $C_{\text{рем}}'$ – середньомісячна заробітна плата;

За статистикою витрати на комплектуючи вироби ($С_{КОМ}$) для ремонту персонального комп'ютера складає 10% від його вартості за термін його експлуатації, тобто рівні витратам на витратні матеріали:

$$C_{КОМ} = C_{ВМ} = 147.94 \text{ грн.} \tag{5.28}$$

Амортизаційні відрахування на персональний комп'ютер (АПК) визначені з положення, що амортизаційний період в даний час дорівнює терміну морального старіння обчислювальної техніки і складає 5 років. Отже, за 5 років амортизаційні відрахування на персональний комп'ютер дорівнюють вартості комп'ютера. За період проектування амортизаційні відрахування складуть:

$$\text{АКП} = \text{В}_{КОМ} \cdot \frac{N_{\text{Д}}}{N_{\text{експ}} \cdot 365}, \tag{5.29}$$

$$\text{АКП} = 45000 \cdot \frac{2}{5 \cdot 12} = 1500. \tag{5.30}$$

Амортизаційні відрахування на програмне забезпечення (АПЗ) залежать від його циклу заміни. Якщо прийняти термін морального старіння для Windows 5 років та Visual Studio за 1 рік то амортизаційні відрахування на програмне забезпечення дорівнюють його вартості.

Для функціонування персонального комп'ютера використовувалася операційна система Windows 10, для написання програмного забезпечення - програмне середовище Visual Studio 2019.

Розрахунок амортизаційних витрат на програмне забезпечення (АВП) приведений в табл. 5.3.

Таблиця 5.3 – Програмне забезпечення, що використовується в проекті

Найменування програмного забезпечення	Кіль-ість, *шт*	Вартість програмного забезпечення, *грн*	Джерело придбання	Амортизаційні витрати, *грн*
Windows 10 Pro	1	6633 [13]	Microsoft.com	221
Visual Studio	1	14608 [14]	visualstudio. Com	2434
Всього:	2	21241		2655

$$АКП_w = 6633 \cdot \frac{2}{5 \cdot 12} = 221{,}1 \text{ грн}, \quad (5.31)$$

$$АКП_w = 14608 \cdot \frac{2}{12} = 2434{,}66 \text{ грн}. \quad (5.32)$$

В результаті отримали суму амортизаційних витрат на програмне забезпечення (АПО) дорівнює 2655,76 грн.

$$АПО = 2655{,}76 \text{ грн}. \quad (5.33)$$

Комунальні витрати, та оплата послуг сторонніх організацій (у тому числі оренда та прибирання приміщень, охорона) є індивідуальними в залежності від кожного проекту, та визначаються окремо. Тому приймемо суму витрат на комунальні послуги ($C_{дод}$) у розмірі 50 % від загального обсягу накладних витрат (5.10), що становить 12944.77 грн.

$$C_{дод} = \frac{S_{міс}}{2} = 12944{,}77 \text{ грн/міс}. \quad (5.34)$$

Оренду приміщень приймемо рівною 13816 гривень на місяць ($C_{ор}$). Вартість оренди взята з статистичних даних платформи Krysha.ua [15].

$$C_{ор} = 13816 \text{ грн/міс}. \quad (5.35)$$

Сумарні експлуатаційні витрати на один персональний комп’ютер складають:

$$C_{\text{експ}} = C_{\text{ел}} + C_{\text{ВМ}} + C_{\text{рем}} + \text{АКП} + \text{АПО} + C_{\text{ор}} + C_{\text{дод}}; \quad (5.36)$$

$$C_{\text{експ}} = 261.12 + 147.94 + 720 + 1500 + 2655{,}76 + 13816 + \\ + 12944{,}77 = 32348.95. \quad (5.37)$$

Результати розрахунків зведено у табл. 5.4.

Таблиця 5.4 – Експлуатаційні витрати на ПК і ПЗ.

Найменування витрат	Витрати, грн
Витрати на електроенергію $C_{\text{ел}}$	261
Вартість витратних матеріалів $C_{\text{ВМ}}$	147
Витрати на ремонт $C_{\text{рем}}$	720
Амортизація персонального комп'ютера АКП	1500
Амортизація програмного забезпечення АПО	2655
Оренда приміщення $C_{\text{ор}}$	13816
Додаткові витрати $C_{\text{дод}}$	12944
Всього $C_{\text{експ}}$	32045

Таким чином, витрати на створення програмного продукту складають:

$$C_{\text{розробки}} = \text{ОЗП} + C_{\text{соц}} + C_{\text{накл}} + C_{\text{експ}}, \quad (5.38)$$

$$C_{\text{розробки}} = 58248 + 12814.56 + 20386.8 + 32045.59 = \\ = 123494.95\text{грн}. \quad (5.39)$$

Розрахунок витрат зведено у табл. 5.5.

Таблиця 5.5 – Кошторис витрат на розробку програмного засобу

Найменування витрат	Витрати, грн
Основна заробітна плата ОЗП	58248
Відрахування на соціальні потреби $C_{\text{соц}}$	12814
Накладні витрати $C_{\text{накл}}$	20386
Експлуатаційні витрати $C_{\text{експ}}$	32045
Всього $C_{\text{розробки}}$	123494

За отриманими значеннями техніко-економічних показників проекту складено кошторис витрат на розроблення системи аналізу часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням OpenGL та Vulkan API.

6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії та етапи розробки програми представлені у табл. 6.1.

Таблиця 6.1 – Стадії та етапи розробки

Стадії розробки	Етапи розробки	Терміни виконання
1.Технічне завдання (ТЗ)	Постановка задачі	21.10.2020 – 22.10.2020
	Огляд літератури та аналіз аналогів	22.10.2020 – 25.10.2020
	Розробка структур вхідних і вихідних даних	26.10.2020 – 31.10.2020
	Визначення вимог до програми. Вибір та обґрунтування мови програмування	1.11.2020 – 4.11.2020
	Узгодження та затвердження ТЗ	5.11.2020 – 12.11.2020
2. Робочий проект	Розробка та програмування логіки програми	13.11.2020 – 20.11.2020
	Розробка і реалізація інтерфейсу користувача	21.11.2020 – 24.11.2020
	Відлагодження програми	25.11.2020 – 26.11.2020
	Розробка, узгодження та затвердження програмної документації	27.11.2020 – 30.11.2020
3. Впровадження	Підготовка і передача програми та програмної документації замовнику	1.12.2020 – 21.12.2020

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

Контроль здійснюється за допомогою виконання набору тестів з метою знаходження помилок в програмі та його специфікації. Контроль виконання роботи забезпечується керівником розробки.

Прийом програми здійснюється уповноваженою комісією.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. "ДСанПіН 3.3.2-007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин [Текст] / Постанова Головного державного санітарного лікаря України від 10 грудня 1998 р. № 7 – К.,1998.".
2. "ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень [Текст]/ Постанова Головного Державного санітарного лікаря України від 01.12.1999 № 42 - К., 1999.".
3. "ГОСТ 19.101-77. Виды програм и программных документов [Текст]/ Постановление Государственного комитета стандартов Совета Министров СССР от 20 мая 1977 г. – М., 1977.".
4. "Методики оценки трудозатрат по разработке программного обеспечения информационных систем," [Ел. ресурс]. Available: http://repository.enu.kz/bitstream/handle/data/12881/metodika-trudozatrat.pdf . [Дата звернення: 10.12.2020].
5. "Методики оценки трудозатрат," [Ел. ресурс]. Available: http://www.hups.mil.gov.ua/periodic-app/article/11953/soi_2014_8_33.pdf. [Дата звернення: 10.12.2020].
6. " Оцінка вартості програмного забезпечення," [Ел. ресурс]. Available: https://www.computing.dcu.ie/~renaat/ca421/report.html. [Дата звернення: 10.12.2020].
7. "Головне управління статистики у м. Києві," [Ел. ресурс]. Available: http://www.kiev.ukrstat.gov.ua/p.php3?c=1139&lang=1. [Дата звернення: 11.2020].
8. "Єдиний соціальний внесок," [Ел. ресурс]. Available: https://index.minfin.com.ua/ua/labour/social/. [Дата звернення: 11.2020].

9. "Скільки електрики споживає комп’ютер," [Ел. ресурс]. Available: https://realadmin.ru/perefiriya/skolko-watt-potreblyaet-pc.html. [Дата звернення: 11.2020].
10. "Тарифи на електроенергію," [Ел. ресурс]. Available: https://pret.com.ua/tariff?hard_tag_meta=for_company. [Дата звернення: 11.2020].
11. "Сайт роздрібної торгівлі Rozetka," [Ел. ресурс]. Available: https://rozetka.com.ua/hp_1s7g8ea/p236474341/. [Дата звернення: 11.2020].
12. "Середня заробітня плата адміністратора," [Ел. ресурс]. Available: https://www.work.ua/ru/salary-kyiv-системный+администратор/. [Дата звернення: 11.2020].
13. "Сайт роздрібної торгівлі Rozetka," [Ел. ресурс]. Available: https://soft.rozetka.com.ua/microsoft_fqc_09131/p3936301/. [Дата звернення: 11.2020].
14. "Сайт роздрібної торгівлі Exe.ua," [Ел. ресурс]. Available: https://soft.rozetka.com.ua/microsoft_fqc_09131/p3936301/. [Дата звернення: 11.2020].
15. "Статистика цін на аренду житла," [Ел. ресурс]. Available: https://dnepropetrovsk.krysha.ua/tseny/adtype-arenda. [Дата звернення: 11.2020].

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Перший проректор Дніпровського
національного університету
залізничного транспорту
імені академіка В. Лазаряна
Б.Є. Боднар

СИСТЕМА ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП`ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Робочий проект
ЛИСТ ЗАТВЕРДЖЕННЯ
1116130.01189-01-ЛЗ

Завідувач кафедри КІТ
проф. В.І. Шинкаренко
_____

Керівник розробки
доц. О.П. Іванов
_____

Виконавець
студент групи ПЗ1921
І.А. Поліщук
_____

Нормоконтролер
доц. О.С. Куроп’ятник
_____

2020

ЗАТВЕРДЖЕНО
1116130.01189-01 13 01

СИСТЕМА ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП`ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Опис програми

1116130.01189-01 13 01

Аркушів 16

2020

ЗМІСТ

1 ЗАГАЛЬНІ ВІДОМОСТІ

Програмний продукт «Система дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням Opengl та Vulkan API» призначений для перевірки та збору даних що до ефективності відображення комп’ютерної графіки. Ефективність оцінюється у кількості кадрів які відображаються за секунду.

Існує багато підходів та технології що до розробки програм для відображення тривимірної графіки. Але вони не беруть до уваги сучасний розвиток процесорів, які у теперешній мають багато ядер які можуть обробляти команди до процесору паралельно.

Основною задачею даного програмного продукту є тестування ефективності відображення графіки використовуючи однопоточний метод рендеру з OpenGL та багатопоточний метод рендеру з Vulkan, для подальшого порівняння та аналізу.

Програма написана за допомогою мови програмування C++.

Застосування такого програмного продукту корисно для того, що би збирати дані що до ефективності того чи іншого підходу до відображення тривимірної графіки на різних системах.

2 ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

За допомогою даного продукту користувач має змогу тестувати та оцінювати ефективність відображення тривимірної графіки. Програмний продукт забезпечує різні варіанти тестування, такі як різні програмні інтерфейси до комп’ютерної графіки OpenGL чи Vulkan, різні види тривимірних моделей для тестування (прості куби, чи анімовані моделі тварин), та різну кількість таких об’єктів які будуть одночасно відображатися на екрані. Також програма надає можливості що до опрацювання та виводу результату таких циклів відображення у вигляді середньої кількості кадрів які були відображені за секунду.

3 ОПИС ЛОГІЧНОЇ СТРУКТУРИ

3.1 Алгоритм програми

Порядок алгоритму роботи програми є наступним:

- надання можливості налаштувати тип графічного програмного інтерфейсу, кількість відображаємих об’єктів, та їх складність;
- розпочати рендер та відобразити його на екрані;
- надати результати рендеру, такі як середня кількість кадрів за секунду, середня кількість секунд за один кадр та кількість використаних ядер.

Детальний алгоритм програми наведено на діаграмі послідовності на рис. 3.1:

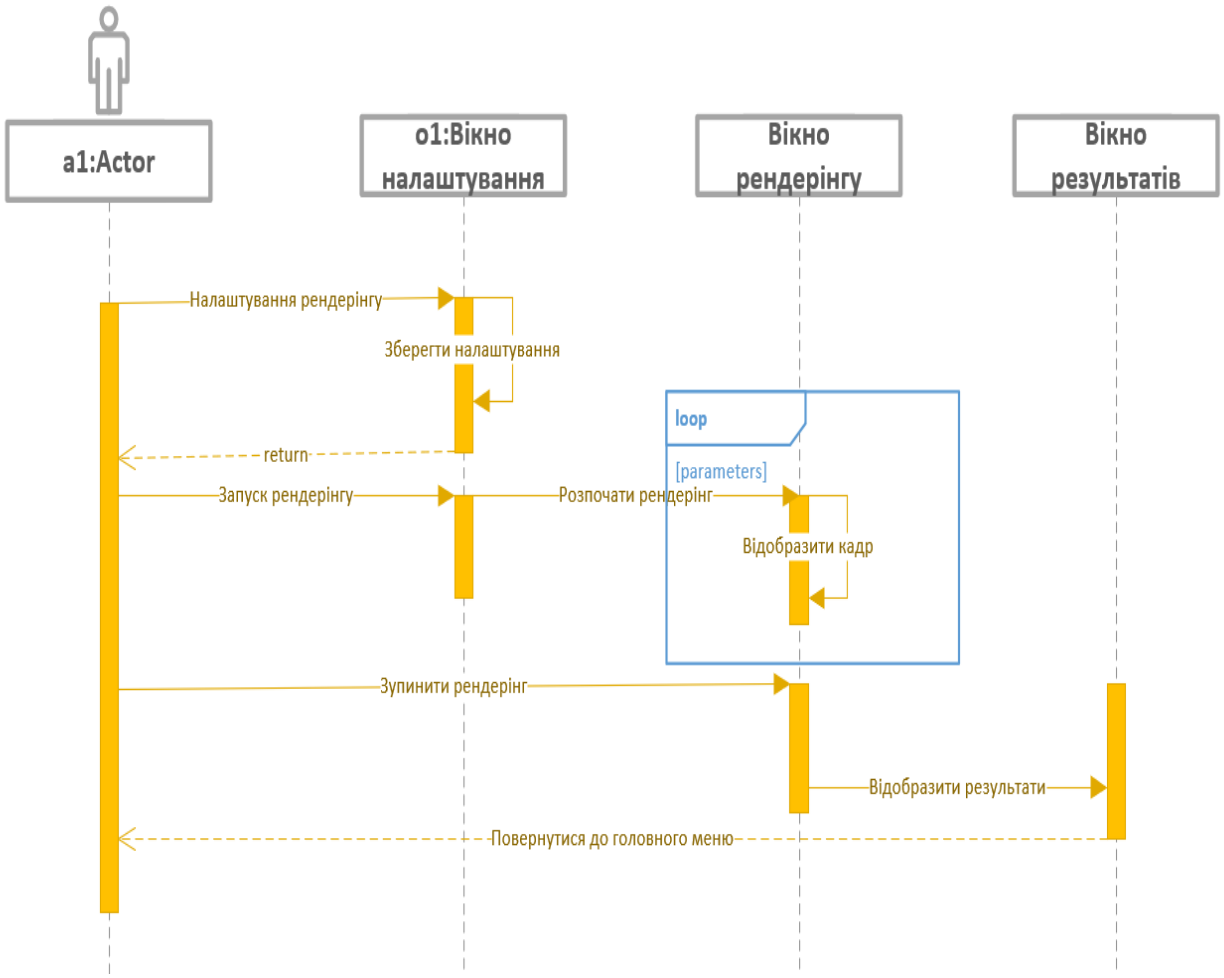

Рисунок 3.1 – Діаграма послідовності

3.2 Використані методи

Оцінка ефективності відображення комп’ютерної графіки буде відбуватися за допомогою середньої кількості кадрів які відображаються за секунду під час відображення. Розрахунок середньої кількості кадрів наведено у формулі 3.1.

$$AFPS = \frac{N}{S}, \tag{3.1}$$

де AFPS (Average frames per second) – це середня кількість кадрів за секунду,

N – кількість кадрів які були згенеровані за період тестування

S – кількість секунд за які було проведене тестування.

3.3 Структура програми

Структура програми наведена на рис. 3.2:

Рисунок 3.2 – Структура програми

Схема взаємодії складових частин програми наведена на діаграмі класів на рис. 3.3

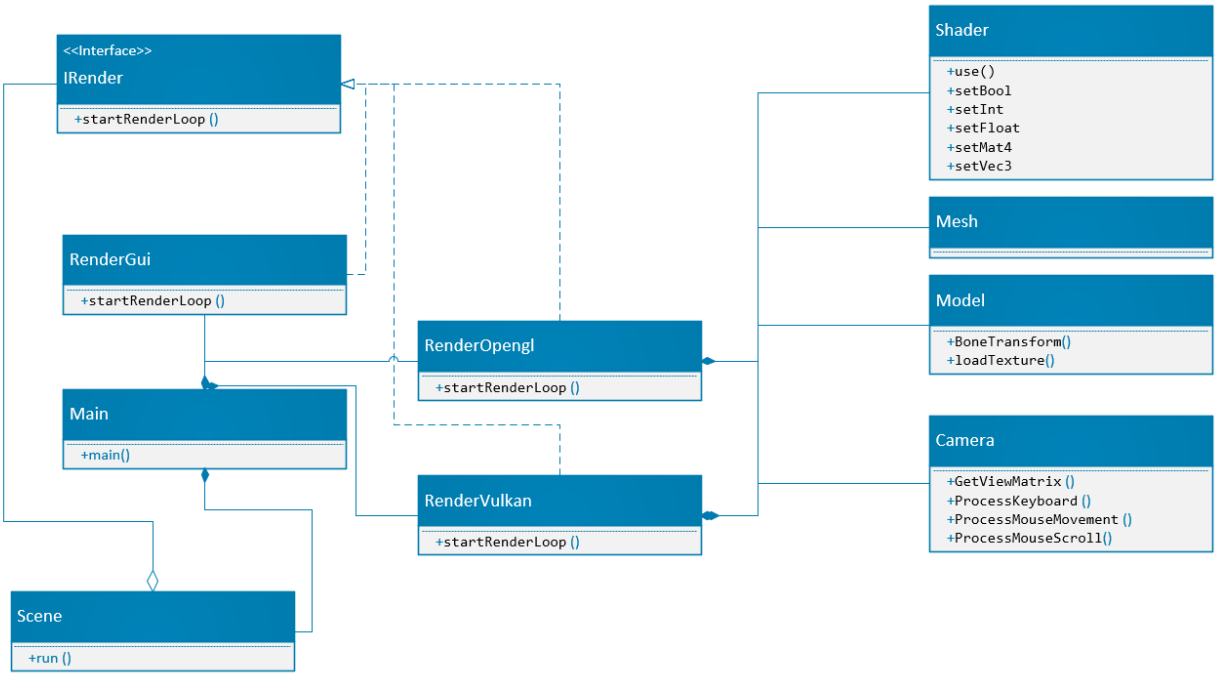

Рисунок 3.3 – Діаграма класів

На рисунку 3.3 можливо побачити що програма була спроектована з використанням об’єктно орієнтованого підходу.

Було використано інтерфейс IRender. Його реалізують клас для відображення графічного інтерфейсу користувача результатів та налаштувань RenderGui. Також його реалізують класи для рендерінгу анімованих графічних моделей RenderOpengl та RenderGUI.

У свою чергу клас Scene використовує інтерфейс IRender для налаштувань рендеру та запуску не знаючи про те який саме тип рендеру це є.

Класи RenderVulkan та RenderOpengl використовують внутрішні допоміжні класи для роботи з 3д моделями, шейдерами та камерою.

Задача класу Main створити об’єкт RenderGUI, на основі вибраних налаштувань створити одну з реалізації інтерфейсу IRender, та віддати цей об’єкт у створений об’єкт класу Scene, який в свою чергу зробить підготовку 3д моделей, їх типу та кількості, та запустить рендер.

4 ВИКОРИСТАНІ ТЕХНІЧНІ ЗАСОБИ

Для коректної роботи програма повинна виконуватися на Windows-сумісних комп'ютерах що мають наступні мінімальні характеристики:

- двох ядерний процесор з тактовою частотою не менше 2,1 ГГц;
- графічний відеочип Nvidia Geforce GTX 1070;
- 8 Гб DDR3 доступної оперативної пам’яті;
- жорсткий диск на 256 Гб;
- пристрій виводу зображення з роздільною здатністю 1080x1920 пікселів.

Вимоги до інформаційної і програмної сумісності наступні:

- операційна система Windows 10 версії 20H2 і пізніше;
- драйвер до графічної карти з підтримкою програмних інтерфейсів до комп’ютерної графіки OpenGL та Vulkan.

5 ВИКЛИК І ЗАВАНТАЖЕННЯ

Дистрибутивний носій з даним програмним продуктом містить у собі технічну документацію до цього продукту, код програми, та власне саму програму у виді виконуваного файлу. Також носії містить папку з файлами анімованих тривимірних моделей, які будуть відображатися на екрані під час роботи.

Перед початком роботи рекомендовано скопіювати виконувану програму та папку з файлами анімованих тривимірних об’єктів на внутрішній диск комп’ютеру. При цьому програма та папка повинні бути розташовані у одній директорії.

Для завантаження програми необхідно два рази натиснути лівою клавішею міші по іконці виконуваного файлу скопійованого з дистрибутивного носія.

6 ВХІДНІ ДАНІ

Вхідні дані повинні передаватися у першому вікні налагоджування. Вони задають налаштування наступного рендеру тривимірних об’єктів.

Вхідні дані вікна налагоджування:

Вхідними даними програми є:

- тип рендерінгу (OpenGL або Vulkan);
- складність об’єктів рендерінгу (прості або складні);
- кількість об’єктів для рендерінгу (від одного до 512).

7 ВИХІДНІ ДАНІ

Вихідні дані надаються після рендеру. Також вихідними даними є сам рендер тривимірних об’єктів, які відображаються на екрані.

Вихідні дані:

- відображення тривимірних об’єктів на екрані з заданими налаштуваннями;
- середня кількість кадрів за секунду за час рендерінгу;
- середня кількість секунд за один кадр за час рендерінгу;
- кількість використовуваних ядер процесору.

8 ОПИС ПРИЗНАЧЕНОГО ДЛЯ КОРИСТУВАЧА ІНТЕРФЕЙСУ

8.1 Опис станів програми

Стани у яких може знаходитись програма описані в табл. 8.1.

Таблиця 8.1 – Стани програми

Номер стану	Стан	Опис стану
1	Налаштування рендеру	Програма відображає вікно налаштування рендеру де можуть бути задані вхідні дані, описані у розділі 6, та чекає доки користувач не розпочне сам рендер.
2	Рендер	Програма відображає тривимірні об’єкти на екрані та їх анімації з заданими налаштуваннями, доки користувач не натисне клавішу «ESC».
3	Результати рендуру	Програма відображає вікно з результатам рендеру, які наведені у пункті 7.

8.2 Опис переходів між станами програми

Схема переходів програми представлена на рис. 8.1

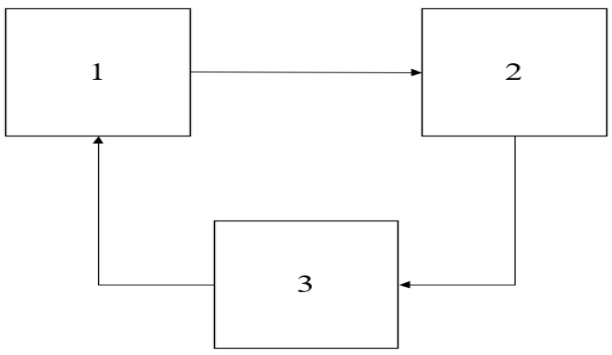

Рисунок 8.1 – Схема переходів між станами програми

8.3 Опис керування діалогом

Діалог роботи з програмою можна описати наступним чином:

- користувач налаштовує програму у вікні налаштувань;
- користувач запускає рендер згідно обраних налаштувань;
- користувач припиняє рендер та переходить на вікно результатів;
- у вікні результатів користувач копіює результати проведеного рендеру для подальшого аналізу, та переходить у вікно налаштувань.

8.4 Формування екранів

Програма має наступні екрани:

- екран налаштувань (рис. 8.2);
- екран рендеру (рис 8.3);
- екран результатів рендеру (рис 8.4).

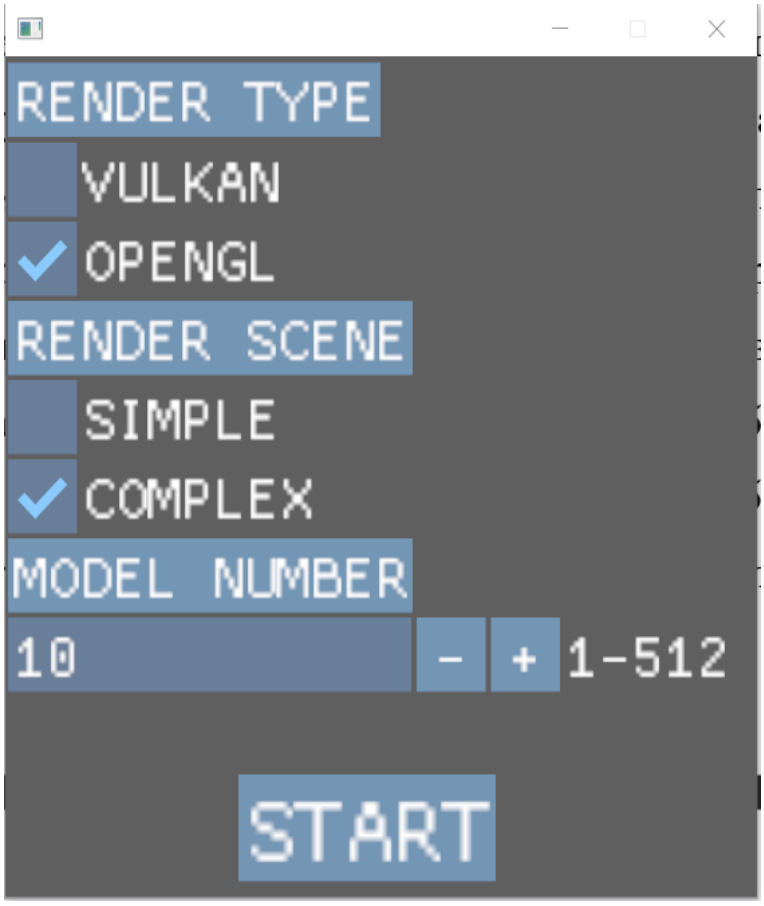

Рисунок 8.2 – Вікно налаштувань

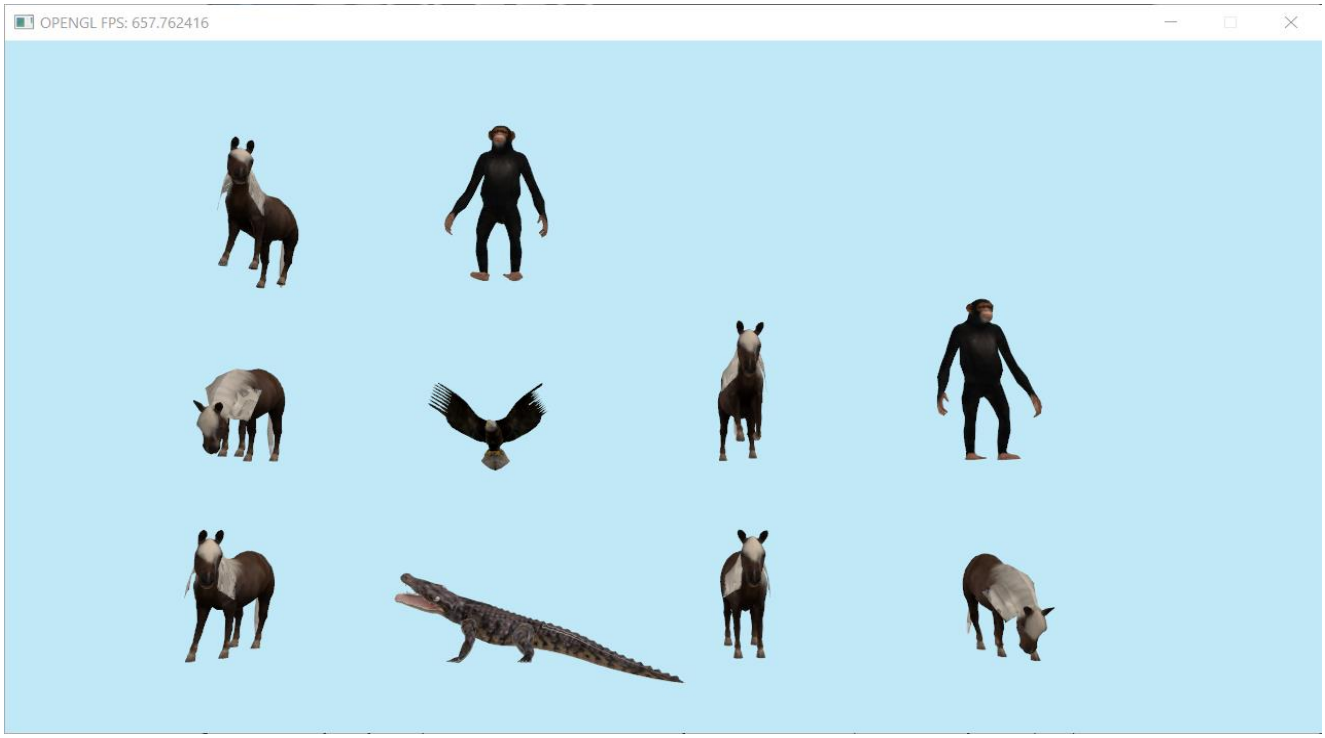

Рисунок 8.3 – Вікно рендеру

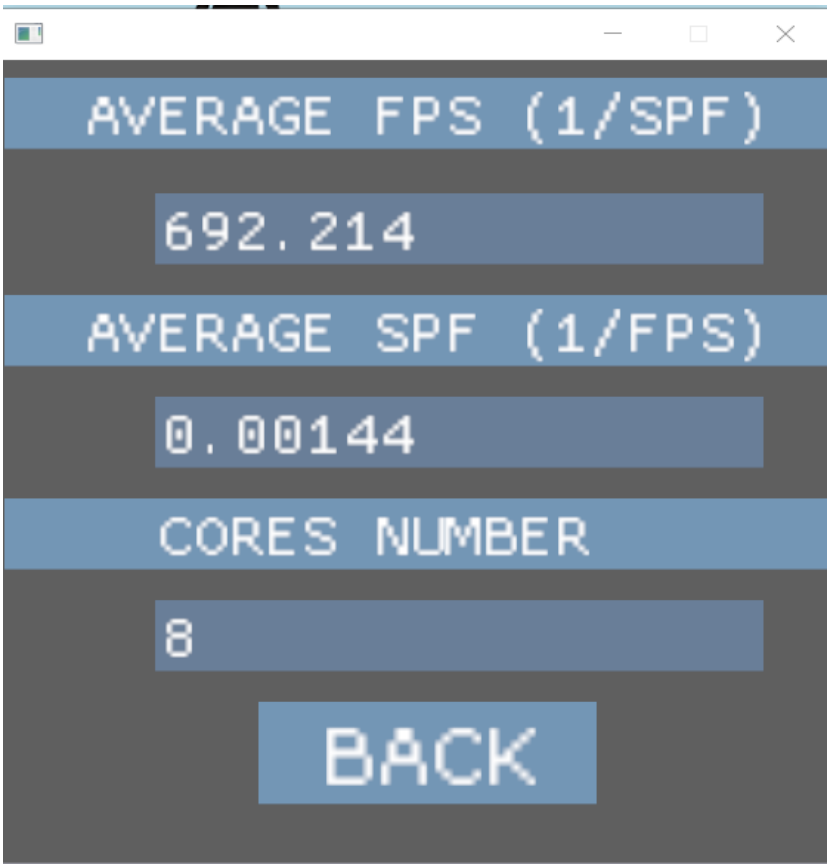

Рисунок 8.4 – Вікно результатів

9 ПОРЯДОК РОБОТИ З ПРОГРАМОЮ

1. Розгортання програми та введення налаштувань здійснюється головним чи молодшим адміністратором.
2. Налаштування подальших рендерів здійснюється головним інженером.
3. Рендер та збір результатів здійснюється інженером-технологом, по буднях з 10:00 до 18:00.
4. Відправка результатів рендеру по буднях з 9:00 до 12:00.
5. Розрахунок наступного плану дня (інженер-технолог) до 16:00.
6. Узгодження плану (начальник цеху) до 17:30.

10 ПОВІДОМЛЕННЯ

Для даної повідомлення надаються тільки у виникненні збоїв.

Можуть бути надані такі повідомлення:

- драйвер графічної карти не підтримує інтерфейс OpenGL;
- драйвер графічної карти не підтримує інтерфейс Vulkan;
- відсутні файли моделей тривимірних об’єктів.

Причина і способи вирішення помилок наведені в табл. 10.1.

Таблиця 10.1 – Повідомлення програми

Повідомлення	Пояснення повідомлення	Спосіб вирішення
Opengl not supported	Драйвер графічної карти не підтримує інтерфейс OpenGL. Потрібно обновити.	Встановити останню версію драйверів до графічної карти з сайту виробника. Переконатися що вони підтримують програмні інтерфейси OpenGL та Vulkan
Vulkan not supported	Драйвер графічної карти не підтримує інтерфейс Vulkan. Потрібно обновити.	Встановити останню версію драйверів до графічної карти з сайту виробника. Переконатися що вони підтримують програмні інтерфейси OpenGL та Vulkan
Can’t find model files	Файли з тривимірними моделями не знайдені. Перевстановіть програму.	Видалити файли програми та перекопіювати їх з дистрибутивного носія

ЗАТВЕРДЖЕНО
1116130.01189-01 І3 01-ЛЗ

СИСТЕМА ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП`ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Керівництво користувача. Керівництво по роботі з системою відображення комп`ютерної графіки реального часу на багатоядерних системах

1116130.01189-01 І3 01

Аркушів 9

2020

ЗМІСТ

ВСТУП

Програмний продукт «Система дослідження часової ефективності відображення комп`ютерної графіки реального часу на багатоядерних системах з використанням Opengl та Vulkan API» призначений для перевірки та збору даних що до ефективності відображення комп’ютерної графіки. Ефективність оцінюється у кількості кадрів які відображаються за секунду.

Основною задачею даного програмного продукту є тестування ефективності відображення графіки використовуючи однопоточний метод рендеру з OpenGL та багатопоточний метод рендеру з Vulkan, для подальшого порівняння та аналізу.

Застосування такого програмного комплексу корисне в областях для яких буде створюватися програмне забезпечення в області комп’ютерної графіки. Це прискорить процес аналізу ефективності відображення комп’ютерної графіки на різних системах з різною кількістю ядер процесора.

Для роботи з даним програмним продуктом користувач повинен мати як мінімум освітньо-кваліфікаційний рівень підготовки бакалавра або спеціаліста.

Перед початком роботи необхідно ознайомитись з даним керівництвом користувача.

1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

1.1 Функціонал програми

Вхідні дані повинні передаватися у першому вікні налагоджування. Вони задають налаштування наступного рендеру тривимірних об’єктів.

Вхідні дані вікна налагоджування:

Вхідними даними програми є:

– тип рендерінгу (OpenGL або Vulkan);
– складність об’єктів рендерінгу (прості або складні);
– кількість об’єктів для рендерінгу (від одного до 512).

Вихідні дані надаються після рендеру. Також вихідними даними є сам рендер тривимірних об’єктів, які відображаються на екрані.

Вихідні дані вікна результатів:

– середня кількість кадрів за секунду за час рендерінгу;
– середня кількість секунд за один кадр за час рендерінгу;
– кількість використовуваних ядер процесору.

1.2 Вимоги до складу і параметрів технічних засобів

Для коректної роботи програма повинна виконуватися на Windows-сумісних комп'ютерах що мають наступні мінімальні характеристики:

- двох ядерний процесор з тактовою частотою не менше 2,1 ГГц;
- графічний відеочип Nvidia Geforce GTX 1070;
- 8 ГбDDR3 доступної оперативної пам’яті;
- пристрій виводу зображення з роздільною здатністю 1080x1920 пікселів.

1.3 Вимоги до інформаційної і програмної сумісності

Вимоги до інформаційної і програмної сумісності наступні:

- операційна система Windows 10 версії 20H2 і пізніше;
- драйвер до графічної карти з підтримкою програмних інтерфейсів до комп’ютерної графіки OpenGL та Vulkan.

2 ПІДГОТОВКА ДО РОБОТИ

2.1 Склад і зміст дистрибутивного носія даних

Дистрибутивний носій з даним програмним продуктом містить у собі технічну документацію до цього продукту, код програми, та власне саму програму у виді виконуваного файлу. Також носії містить папку з файлами анімованих тривимірних моделей, які будуть відображатися на екрані під час роботи.

2.2 Порядок завантаження даних і програм

Перед початком роботи рекомендовано скопіювати виконувану програму та папку з файлами анімованих тривимірних об’єктів на внутрішній диск комп’ютеру. При цьому програма та папка повинні бути розташовані у одній директорії.

Для завантаження програми необхідно два рази натиснути лівою клавішею міші по іконці виконуваного файлу скопійованого з дистрибутивного носія.

2.3 Порядок перевірки працездатності

Для перевірки працездатності даного програмного продукту необхідно виконати наступні дії:

- виконати дії по копіюванню програми вказані у пункті 2.2;
- запустити програму;
- розпочати рендер обравши OpenGL у якості графічного програмного інтерфейсу;
- перезапустити програму та розпочати рендер обравши Vulkan у якості графічного програмного інтерфейсу.

Як що при обох рендерах з OpenGL та Vulkan на екрані з’явиться анімовані моделі, можна стверджувати що на системі де проводилася перевірка програма працює коректно.

3 ОПИС ОПЕРАЦІЇ

3.1 Налаштування рендеру

Перед початком налаштувань рендеру, користувач повинен виконати дії вказані у пункті 2 – «Підготовка до роботи», а саме необхідно скопіювати програму з дистрибутивного носія, та запустити її.

Після запуску програми користувач потрапляє на вікно налаштування параметрів до подальшого рендеру тривимірних моделей. На рис. 3.1 представлений вигляд вікна налашутвань.

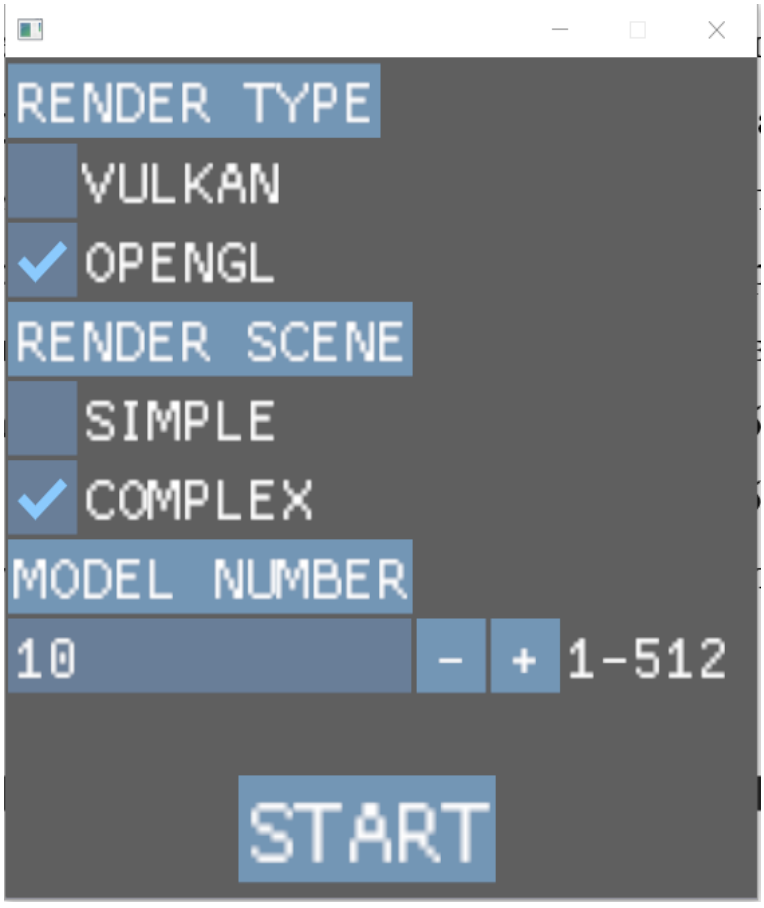

Рисунок 3.1 – Вікно налаштувань

- Далі користувачу потрібно обрати налаштування до подальшого рендеру наступним чином:
- обрати тип програмного інтерфейсу, який буде використовуватися під час рендеру. Це може бути OpenGL чи Vulkan;

- обрати складність тривимірних об’єктів які будуть відображатися під час рендеру. Обравши складність «SIMPLE» користувач буде бачити під час рендеру прості тривимірні фігури кубу, які кружляють навколо своєї вісі. А обравши складність «COMPLEX» користувач буде бачити під час рендеру складні моделі тварин зі скелетною анімацією;
- обрати кількість тривимірних об’єктів, які одночасно будуть відображатися не екрані під час рендеру. Кількість об’єктів може бути від 1 до 512.

Заключною дією користувача є запуск рендеру. Для цього необхідно натиснути клавішу «START».

3.2 Процес рендеру

Перед початком рендеру, необхідно виконати дії з налаштувань вказані у пункті 3.1, а саме необхідно налаштувати параметри рендеру, та розпочати його натиснувши на кнопку «START».

Далі користувач зможе бачити наступне вікно, де буде відображатися тривимірні моделі згідно попередніх налаштувань. (Вид вікна може змінюватись при різних попередніх налаштуваннях).

На рис. 3.1 представлений вигляд вікна рендеру:

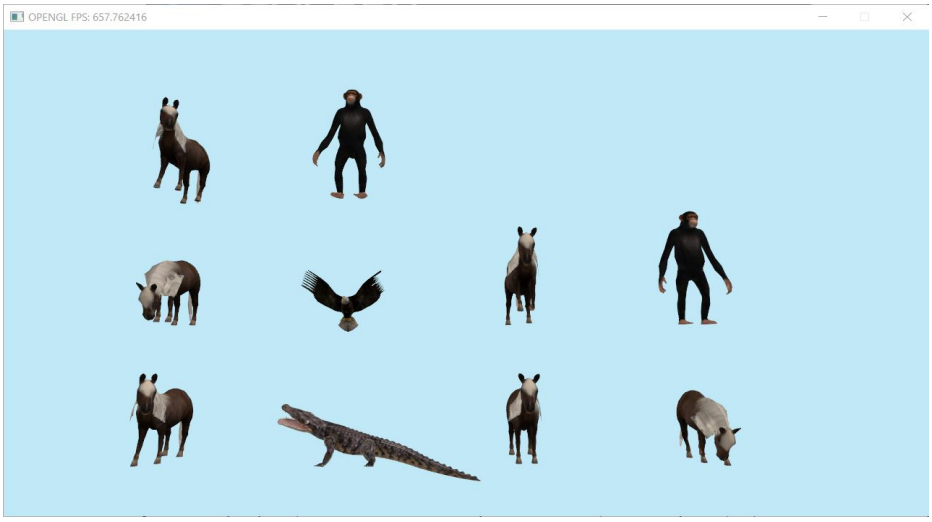

Рисунок 3.2 – Вікно рендеру

У лівому верхньому куті програми, користувач має змогу відстежувати середню кількість кадрів яка відображається за одну секунду для даної сцени рендеру.

Для завершення роботи рендеру, користувачу необхідно натиснути клавішу «ESC» на клавіатурі.

3.3 Перегляд результатів

Перед переглядом результатів, користувачу необхідно виконати дії описані у пунктах 3.1 та 3.2. А саме запустити програму, налаштувати її та розпочати рендер.

Для переходу у вікно перегляду результатів користувач повинен під час рендеру натиснути клавішу «ESC».

Приклад вікна результатів наведено на рис. 3.3:

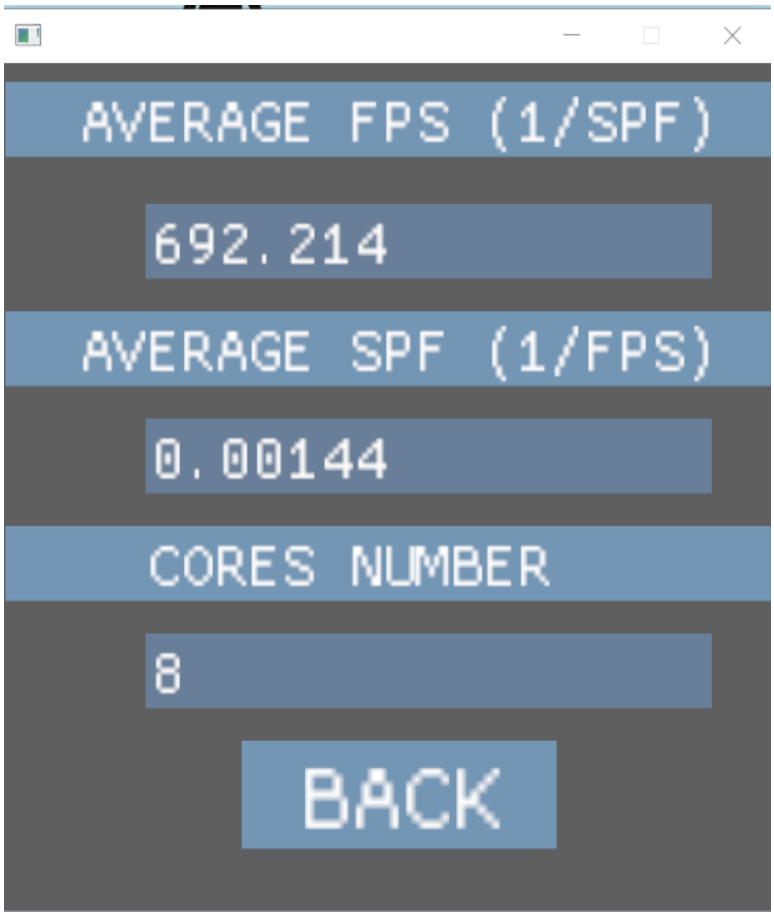

Рисунок 3.3 – Вікно результатів

Для закінчення перегляду результатів та повернення до вікна налаштувань, користувач повинен натиснути кнопку «BACK».

4. АВАРІЙНІ СИТУАЦІЇ

У таблиці 4.1 наведені відомі аварійні ситуації, повідомлення програми та їх спосіб вирішення:

Таблиця 4.1 – Відомі аварійні ситуації

Повідомлення	Пояснення повідомлення	Спосіб вирішення
Opengl not supported	Драйвер графічної карти не підтримує інтерфейс OpenGL. Потрібно обновити.	Встановити останню версію драйверів до графічної карти з сайту виробника. Переконатися що вони підтримують програмні інтерфейси OpenGL та Vulkan
Vulkan not supported	Драйвер графічної карти не підтримує інтерфейс Vulkan. Потрібно обновити.	Встановити останню версію драйверів до графічної карти з сайту виробника. Переконатися що вони підтримують програмні інтерфейси OpenGL та Vulkan
Can’t find model files	Файли з тривимірними моделями не знайдені. Перевстановіть програму.	Видалити файли програми та перекопіювати їх з дистрибутивного носія

У інших аварійних ситуаціях рекомендується виконати дії по перекопіюванню та встановленню драйверів з таблиці 4.1, та переконатися що система де виконується програма відповідає вимогам наведеним у пунктах 1.2 та 1.3.

ЗАТВЕРДЖЕНО

1116130.01189-01 12 01-ЛЗ

СИСТЕМА ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП`ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Текст програми

1116130.01189-01 12 01

Аркушів 18

2020

ЗМІСТ

1 СТРУКТУРА ПРОГРАМИ

Структура програми наведена на рисунку 1.1

Рисунок 1.1 – Структура програми

2 ТЕКСТ ПРОГРАМИ

Mesh.h

```
#pragma once
#include <string>
#include <vector>
#include <array>
#include <map>
#include <glm/glm.hpp>
#include <assimp/scene.h>

namespace RenderCommon
{
		struct Vertex {
	inline static constexpr size_t c_maxBonePerVertexCount = 8;

				glm::vec3 vertPos{};
				glm::vec3 VertNormal{};
				glm::vec2 VertText{};
	std::uint32_t BoneIDs[c_maxBonePerVertexCount] = { 0 };
	float Weights[c_maxBonePerVertexCount] = { 0 };
		};

   struct Texture {
      enum class Type {
         diffuse,
         specular
      } type;

      std::string path;
      std::string glslName;

      static std::string toString(Type type);
   };

   class Mesh {
   public:
      Mesh(std::vector<Vertex> vertices, std::vector<uint32_t> indices,
std::vector<Texture> textures);
      ~Mesh();
   public:
      std::vector<Vertex>   m_vertices;
      std::vector<uint32_t> m_indices;
      std::vector<Texture>  m_textures;
   };
}
```

Mesh.cpp

```
#include "Mesh.h"

namespace RenderCommon
{
		Mesh::~Mesh()
		{

		}


		Mesh::Mesh(std::vector<Vertex>		vertices,
std::vector<uint32_t> indices, std::vector<Texture> textures) :
			m_vertices{  std::move(vertices)  },  m_indices{
std::move(indices) }, m_textures{ std::move(textures) }
		{

		}

		std::string Texture::toString(Type type)
		{
			if (type == Type::diffuse)
				return "texture_diffuse";
			else if (type == Type::specular)
				return "texture_specular";
		}
}
```

Model.h

```
#pragma once

#include "Mesh.h"
#include <assimp/scene.h>
#include <assimp/Importer.hpp>
#include <map>
#include <filesystem>

namespace RenderCommon
{
   inline static glm::mat4 Assimp2Glm(const aiMatrix4x4& from)
   {
      return glm::mat4(
         from.a1, from.b1, from.c1, from.d1,
         from.a2, from.b2, from.c2, from.d2,
         from.a3, from.b3, from.c3, from.d3,
         from.a4, from.b4, from.c4, from.d4
      );
   }

   struct BoneInfo
   {
      aiMatrix4x4 BoneOffset;
      aiMatrix4x4 FinalTransformation;
   };

   class Model
   {
   public:
      using Textures = std::vector<std::pair<std::filesystem::path,
Texture::Type>>;
      Model(std::filesystem::path path,
std::vector<std::pair<std::filesystem::path, Texture::Type>> textures,
int animationNumber);

      void BoneTransform(float TimeInSeconds,
std::vector<aiMatrix4x4>& Transforms);

      static unsigned char* loadTexture(const std::string& path, int&
width, int& height);

      std::vector<Mesh> meshes;
   private:
      Assimp::Importer* m_import;

      std::map<std::string, std::uint32_t> m_BoneMapping; // maps a
bone name to its index
      std::uint32_t m_NumBones = 0;
      std::vector<BoneInfo> m_BoneInfo;
      aiMatrix4x4 m_GlobalInverseTransform;


   private:
      std::vector<std::pair<std::filesystem::path,
Texture::Type>>
m_textures;
      int m_animationNumber = 0;
   };
}
```

RenderGUI.h

```
#pragma once

#include "Render.h"
#include <memory>
```

```
class RenderGUI
{
public:
	RenderGUI();
	~RenderGUI();

	RenderGuiData startRenderLoop(RenderGuiData result);
private:
	class Impl;
	std::unique_ptr<Impl> m_impl;
};
```

RenderGUI.cpp

```
#include "RenderGUI.h"

#include "glad/glad.h"
#include "GLFW/glfw3.h"

#include <string>
#include <iostream>
#include <thread>

#include "imgui.h"
#include "imgui_impl_glfw.h"
#include "imgui_impl_opengl3.h"

using namespace std::literals;


class RenderGUI::Impl
{
public:
	Impl()
	{
	}

	static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
	{
		auto _this = reinterpret_cast<RenderGUI::Impl*>(glfwGetWindowUserPointer(window));
		glViewport(0, 0, width, height);
	}

	~Impl()
	{
	}

	void init()
	{
		initWindow();

		glEnable(GL_MULTISAMPLE);
		glEnable(GL_FRAMEBUFFER_SRGB);
	}

	void initWindow()
	{
		if (!glfwInit())
			throw std::runtime_error{ "glfwInit has failed" };

		glfwWindowHint(GLFW_CLIENT_API, GLFW_OPENGL_API);

		glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);

		glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
		glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
		glfwWindowHint(GLFW_SAMPLES, 8);
		glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
		m_window = glfwCreateWindow(561, 585, " ", NULL, NULL);
		if (!m_window)
			throw std::runtime_error{ "glfwCreateWindow has failed" };

		glfwSetWindowCenter(m_window);

		glfwSetInputMode(m_window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);

		glfwMakeContextCurrent(m_window);
		glfwSwapInterval(0);

		glfwSetWindowUserPointer(m_window, this);

		if (!gladLoadGLLoader(reinterpret_cast<GLADloadproc>(glfwGetProcAddress)))
			throw std::runtime_error{ "gladLoadGLLoader has failed" };

		glfwSetFramebufferSizeCallback(m_window, [](GLFWwindow* window, int x, int y) {
			auto _this = reinterpret_cast<RenderGUI::Impl*>(glfwGetWindowUserPointer(window));
			std::cout << "X: " << x << " Y: " << y << std::endl;
				glViewport(0, 0, x, y);
				// render();
		});
	}

	RenderGuiData startRenderLoop(RenderGuiData result)
	{
		init();

		glEnable(GL_DEPTH_TEST);
		glEnable(GL_CULL_FACE);
		glClearColor(1.f, 1.f, 1.f, 1.0f);

		IMGUI_CHECKVERSION();
		ImGui::CreateContext();
		ImGuiIO& io = ImGui::GetIO(); (void)io;

		ImGui::StyleColorsDark();

		// Setup Platform/Renderer backends
		ImGui_ImplGlfw_InitForOpenGL(m_window, true);
		ImGui_ImplOpenGL3_Init("#version 130");


	   if (result.averageFps > 0)
		{
			renderResultMenu(result);
		}

		result = renderMainMenu(result);

		ImGui_ImplOpenGL3_Shutdown();
		ImGui_ImplGlfw_Shutdown();
		ImGui::DestroyContext();

		glfwDestroyWindow(m_window);

		return result;
	}

	RenderGuiData renderMainMenu(RenderGuiData result)
	{
```

```
		bool cbSimple = result.simpleScene;
		bool cbComplex = !result.simpleScene;

		bool cbVulkan = result.renderType == RenderGuiData::RenderType::Vulkan;
		bool cbOpengl = result.renderType == RenderGuiData::RenderType::OpenGL;

		bool cbHigh = result.sceneLoad == RenderGuiData::SceneLoad::High;
		bool cbMedium = result.sceneLoad == RenderGuiData::SceneLoad::Med;
		bool cbLow = result.sceneLoad == RenderGuiData::SceneLoad::Low;


		int modelNumber = result.modelNumber;
		while (true)
		{
			if (glfwGetKey(m_window, GLFW_KEY_ESCAPE) == GLFW_PRESS || glfwWindowShouldClose(m_window))
			{
				result.renderType = RenderGuiData::RenderType::Exit;
				break;
			}

			glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


			// Start the Dear ImGui frame
			ImGui_ImplOpenGL3_NewFrame();
			ImGui_ImplGlfw_NewFrame();
			ImGui::NewFrame();

			ImGui::Begin(" ");

			ImGui::SetWindowFontScale(3.5);
			ImGui::Button("RENDER TYPE");

			if (ImGui::Checkbox("VULKAN", &cbVulkan))
			{
				cbVulkan = true;
				cbOpengl = false;
			}

			if (ImGui::Checkbox("OPENGL", &cbOpengl))
			{
				cbVulkan = false;
				cbOpengl = true;
			}

			ImGui::Button("RENDER SCENE");

			if (ImGui::Checkbox("SIMPLE", &cbSimple))
			{
				cbSimple = true;
				cbComplex = false;
			}

			if (ImGui::Checkbox("COMPLEX", &cbComplex))
			{
				cbSimple = false;
				cbComplex = true;
			}

			ImGui::Button("MODEL NUMBER");

			if (ImGui::InputInt("1-512", &modelNumber, 10, 512))
			{
				cbHigh = false;
				cbMedium = false;
				cbLow = false;
			}

			if (modelNumber < 1)
				modelNumber = 1;
			else if (modelNumber > 512)
				modelNumber = 512;


			ImVec2 windowSize = ImGui::GetIO().DisplaySize;

			ImGui::BeginChild(1, { 50, 50 });
			ImGui::EndChild();

			ImGui::SetCursorPosX(180);
			ImGui::BeginChild(2);
			ImGui::SetWindowFontScale(1.5);
			bool startPressed = ImGui::Button("START");

			ImGui::EndChild();
			ImGui::End();


			ImGui::Render();

		ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());


			glfwSwapBuffers(m_window);
			glfwPollEvents();

			if (startPressed)
				break;
		}

		if (result.renderType != RenderGuiData::RenderType::Exit)
		{
			if (cbOpengl)
				result.renderType = RenderGuiData::RenderType::OpenGL;
			else if (cbVulkan)
				result.renderType = RenderGuiData::RenderType::Vulkan;
		}

		if (cbHigh)
			result.sceneLoad = RenderGuiData::SceneLoad::High;
		else if (cbMedium)
			result.sceneLoad = RenderGuiData::SceneLoad::Med;
		else if (cbLow)
			result.sceneLoad = RenderGuiData::SceneLoad::Low;

		result.modelNumber = modelNumber;

		result.simpleScene = cbSimple;

		return result;
	}

	void renderResultMenu(RenderGuiData result)
	{
```

```
		while (true)
		{
			if (glfwGetKey(m_window, GLFW_KEY_ESCAPE) == GLFW_PRESS || glfwWindowShouldClose(m_window))
			{
				result.renderType = RenderGuiData::RenderType::Exit;
				break;
			}

			glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


			// Start the Dear ImGui frame
			ImGui_ImplOpenGL3_NewFrame();
			ImGui_ImplGlfw_NewFrame();
			ImGui::NewFrame();

			ImGui::Begin(" ");

			ImGui::SetWindowFontScale(3.5);

			ImGui::BeginChild(22, { 50, 5 });
			ImGui::EndChild();

			ImGui::Button("  AVERAGE FPS (1/SPF)  ");

			ImGui::BeginChild(1, { 50, 25 });
			ImGui::EndChild();

			double res = result.averageFps;
			ImGui::SetCursorPosX(110);
			ImGui::InputDouble("  ", &res, 0.0, 0.0, "%.3f");

			ImGui::BeginChild(1543, { 50, 15 });
			ImGui::EndChild();


			ImGui::Button("  AVERAGE SPF (1/FPS)  ");

			ImGui::BeginChild(63241, { 50, 15 });
			ImGui::EndChild();

			double res2 = 1/result.averageFps;
			ImGui::SetCursorPosX(110);
			ImGui::InputDouble(" ", &res2, 0.0, 0.0, "%.5f");

			ImGui::BeginChild(63212341, { 50, 15 });
			ImGui::EndChild();

			ImGui::Button("  CORES NUMBER ");

			ImGui::BeginChild(6324321, { 50, 15 });
			ImGui::EndChild();

			int res3 = std::thread::hardware_concurrency();
			ImGui::SetCursorPosX(110);
			ImGui::InputInt("  ", &res3, 0, 0);

			ImGui::BeginChild(9876543, { 50, 15 });
			ImGui::EndChild();

			ImGui::SetCursorPosX(180);
			ImGui::BeginChild(2);
			ImGui::SetWindowFontScale(1.5);
			bool startPressed = ImGui::Button(" BACK ");

			ImGui::EndChild();
			ImGui::End();
			ImGui::Render();

		ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());


			glfwSwapBuffers(m_window);
			glfwPollEvents();

			if (startPressed)
				break;
		}
	}
private:
	GLFWwindow* m_window;
};

RenderGUI::RenderGUI()
	: m_impl{ std::make_unique< RenderGUI::Impl>() }
{

}

RenderGUI::~RenderGUI() = default;

RenderGuiData RenderGUI::startRenderLoop(RenderGuiData result)
{	return m_impl->startRenderLoop(result);
}
```

RenderOpengl.h

```
#pragma once

#include "Render.h"
#include <memory>


class RenderOpengl : public IRender
{
public:
	RenderOpengl();
	~RenderOpengl();

	double startRenderLoop(std::vector<ModelInfo> modelInfos) override;
private:
	class Impl;
	std::unique_ptr<Impl> m_impl;
};
```

RenderOpengl.cpp

```
#include "RenderOpengl.h"

#include "glad/glad.h"
#include "GLFW/glfw3.h"
#include "Shader.h"
#include "Shaders.h"
#include "Camera.h"

#include <string>
#include <iostream>

#include "imgui.h"
#include "imgui_impl_glfw.h"
#include "imgui_impl_opengl3.h"
```

```
#include "Model.h"

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include <glm/gtc/type_ptr.hpp>
#include <Utils.h>

using namespace std::literals;


class RenderOpengl::Impl
{
public:
	struct MeshRenderData
	{
		unsigned int VAO = 0, VBO = 0, EBO = 0;
	};

	struct OpenglModel
	{
		std::unique_ptr<RenderCommon::Model>
model;
		std::vector<MeshRenderData>
meshRenderData;

		std::map<RenderCommon::Texture::Type,	int>
textures;

		glm::vec3 position{};
		glm::vec3 scale{};

		ModelInfo info{};
	};
public:
	Camera	camera{	glm::vec3(8.207467,	2.819616,
18.021290) };

	std::vector<OpenglModel> m_models;

	int m_modelsMeshCount = 0;

	std::map<std::string, int> m_textureCache;
public:
	Impl()
	{
	}

	static	void	framebuffer_size_callback(GLFWwindow*
window, int width, int height)
	{
		auto	_this	=
reinterpret_cast<RenderOpengl::Impl*>(glfwGetWindowUserPointer(
window));
		glViewport(0, 0, width, height);
		//_this->render();
	}

	~Impl()
	{
	}

	void init(std::vector<ModelInfo> modelInfos)
	{
		initWindow();

		glEnable(GL_MULTISAMPLE);
		glEnable(GL_FRAMEBUFFER_SRGB);

		auto	models	=
prepareModels(std::move(modelInfos));
		loadModels(std::move(models));
	}

	void initWindow()
	{
		if (!glfwInit())
			throw	std::runtime_error{	"glfwInit
has failed" };

		glfwWindowHint(GLFW_CLIENT_API,
GLFW_OPENGL_API);

		glfwWindowHint(GLFW_CONTEXT_VERSION_MAJO
R, 4);

		glfwWindowHint(GLFW_CONTEXT_VERSION_MINO
R, 6);
		glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
		glfwWindowHint(GLFW_SAMPLES, 8);
		m_window	=	glfwCreateWindow(1280,	720,
"OpenGL", NULL, NULL);
		if (!m_window)
			throw	std::runtime_error{
"glfwCreateWindow has failed" };

		glfwMakeContextCurrent(m_window);


		glfwSwapInterval(0);

		glfwSetWindowUserPointer(m_window, this);

		if
(!gladLoadGLLoader(reinterpret_cast<GLADloadproc>(glfwGetProc
Address)))
			throw	std::runtime_error{
"gladLoadGLLoader has failed" };

		glfwSetWindowCenter(m_window);

		glfwSetFramebufferSizeCallback(m_window,
[](GLFWwindow* window, int x, int y) {
			auto	_this	=
reinterpret_cast<RenderOpengl::Impl*>(glfwGetWindowUserPointer(
window));
				glViewport(0, 0, x, y);
				// render();
			});

		glfwSetCursorPosCallback(m_window,
[](GLFWwindow* window, double xpos, double ypos) {
			auto	_this	=
reinterpret_cast<RenderOpengl::Impl*>(glfwGetWindowUserPointer(
window));
			static float lastX = 400, lastY = 300;
			static bool firstMouse = true;
			if (firstMouse)
			{
				lastX = xpos;
				lastY = ypos;
				firstMouse = false;
			}

			float xoffset = xpos - lastX;
			float yoffset = lastY - ypos;
			lastX = xpos;
			lastY = ypos;

			if	(glfwGetKey(window,
GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)

		glfwSetInputMode(window,	GLFW_CURSOR,
GLFW_CURSOR_DISABLED);
			else
```

```
	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);

				if (glfwGetKey(window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)
					_this->camera.ProcessMouseMovement(xoffset, yoffset);
			});
	}

	std::vector<OpenglModel> prepareModels(std::vector<ModelInfo> modelInfos)
	{
		std::vector<OpenglModel> models;

		for (auto& modelInfo : modelInfos)
		{
			OpenglModel model1;

			model1.info = modelInfo;

			model1.model = std::make_unique<RenderCommon::Model>(modelInfo.modelPath,

	RenderCommon::Model::Textures{

	{modelInfo.texturePath, RenderCommon::Texture::Type::diffuse },
				},

	modelInfo.animationNumber
				);

			model1.position = { modelInfo.posX, modelInfo.posY, modelInfo.posZ };
			model1.scale = { modelInfo.scaleX, modelInfo.scaleY, modelInfo.scaleZ };


	models.emplace_back(std::move(model1));
		}

		for (OpenglModel& model : models)
		{
			for (size_t i = 0; i < model.model->meshes.size(); ++i)
			{
				++m_modelsMeshCount;
			}
		}

		return models;
	}

	void loadModels(std::vector<OpenglModel> models)
	{
		for (OpenglModel& model : models)
		{
			for (size_t i = 0; i < model.model->meshes.size(); ++i)
			{

	model.meshRenderData.push_back(createMeshRenderData(model.model->meshes[i]));

				for (auto& texture : model.model->meshes[i].m_textures)
				{
					auto findIt = model.textures.find(texture.type);
					if (model.textures.end() == findIt)

	model.textures.emplace(texture.type, createTextureImage(texture.path));
				}
			}
		}

		m_models = std::move(models);
	}

	MeshRenderData createMeshRenderData(RenderCommon::Mesh& mesh)
	{
		MeshRenderData meshRenderData;
		glGenVertexArrays(1, &meshRenderData.VAO);
		glGenBuffers(1, &meshRenderData.VBO);
		glGenBuffers(1, &meshRenderData.EBO);

		glBindVertexArray(meshRenderData.VAO);
		glBindBuffer(GL_ARRAY_BUFFER, meshRenderData.VBO);

		glBufferData(GL_ARRAY_BUFFER, mesh.m_vertices.size() * sizeof(RenderCommon::Vertex), &mesh.m_vertices[0], GL_STATIC_DRAW);


	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, meshRenderData.EBO);

	glBufferData(GL_ELEMENT_ARRAY_BUFFER, mesh.m_indices.size() * sizeof(unsigned int),
				&mesh.m_indices[0], GL_STATIC_DRAW);

		// vertex positions
		glEnableVertexAttribArray(0);
		glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(RenderCommon::Vertex), (void*)0);
		// vertex normals
		glEnableVertexAttribArray(1);
		glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(RenderCommon::Vertex), (void*)offsetof(RenderCommon::Vertex, Normal));
		// vertex texture coords
		glEnableVertexAttribArray(2);
		glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(RenderCommon::Vertex), (void*)offsetof(RenderCommon::Vertex, TexCoords));

		glEnableVertexAttribArray(3);
		glVertexAttribIPointer(3, 4, GL_INT, sizeof(RenderCommon::Vertex), (void*)offsetof(RenderCommon::Vertex, BoneIDs));

		glEnableVertexAttribArray(4);
		glVertexAttribIPointer(4, 4, GL_INT, sizeof(RenderCommon::Vertex), (void*)(offsetof(RenderCommon::Vertex, BoneIDs) + 4 * sizeof(std::uint32_t)));

		glEnableVertexAttribArray(5);
		glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(RenderCommon::Vertex), (void*)offsetof(RenderCommon::Vertex, Weights));

		glEnableVertexAttribArray(6);
		glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(RenderCommon::Vertex), (void*)(offsetof(RenderCommon::Vertex, Weights) + 4 * sizeof(float)));

		glBindVertexArray(meshRenderData.VAO);
```

```
                glDrawElements(GL_TRIANGLES, mesh.m_indices.size(), GL_UNSIGNED_INT, 0);

                glBindVertexArray(0);

                return meshRenderData;
        }

        int createTextureImage(const std::string& path)
        {
                auto findIt = m_textureCache.find(path);

                if(findIt != m_textureCache.end())
                {
                        return findIt->second;
                }

                unsigned int textureID;
                glGenTextures(1, &textureID);

                int width, height, nrComponents;

                unsigned char* data = RenderCommon::Model::loadTexture(path.c_str(), width, height);

                if (!data)
                        throw std::runtime_error{ "Texture failed to load at path: "s + path };

                glBindTexture(GL_TEXTURE_2D, textureID);


                glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
                glGenerateMipmap(GL_TEXTURE_2D);

                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

                m_textureCache.emplace(path, textureID);

                return textureID;
        }

        double startRenderLoop(std::vector<ModelInfo> modelInfos)
        {
                init(std::move(modelInfos));

                glEnable(GL_DEPTH_TEST);
                glEnable(GL_CULL_FACE);
                glClearColor(135 / 255.f, 206 / 255.f, 235 / 255.f, 1.0f);

                Shader ourShaderSimple(s_shader_v_simple, s_shader_f_simple);
                Shader ourShader(s_shader_v, s_shader_f);

                Shader* currentShader = &ourShader;

                ourShader.use();
                ourShader.setInt("material.texture_diffuse1", 0);
                //ourShader.setInt("material.texture_specular1", 1);

                auto startSeconds = glfwGetTime();
                std::uint64_t frameCount = 0;
    while (!glfwWindowShouldClose(m_window))
    {
                if (glfwGetKey(m_window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
                        break;

                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                showFPS();
                processInput();

                auto currentTime = glfwGetTime();

                glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)1280 / (float)720, 0.1f, 100.0f);
                glm::mat4 view = camera.GetViewMatrix();

                for (auto& model : m_models)
                {
                        if (model.info.simpleModel)
                                currentShader = &ourShaderSimple;
                        else
                                currentShader = &ourShader;

                        currentShader->use();

                        glm::mat4 modelMat = glm::mat4(1.0f);
                        modelMat = glm::translate(modelMat, model.position);
                        modelMat = glm::scale(modelMat, model.scale);

                        if (model.info.simpleModel)
                                modelMat = glm::rotate(modelMat, (float)currentTime, glm::vec3(0.5f, 1.0f, 0.0f));

                        currentShader->setMat4("model", modelMat);
                        currentShader->setMat4("PVM", projection * view * modelMat);

                        std::vector<aiMatrix4x4> Transforms;
                        model.model->BoneTransform(currentTime, Transforms);


                        if (!model.info.simpleModel)
                                for (int i = 0; i < Transforms.size(); i++) {
                                        currentShader->setMat4("gBones["s + std::to_string(i) + "]", RenderCommon::Assimp2Glm(Transforms[i]));
                                }

                        auto findIt = model.textures.find(RenderCommon::Texture::Type::diffuse);
                        if (findIt == model.textures.end())
                                throw std::runtime_error{ "Can't find diffuse texture" };


                glActiveTexture(GL_TEXTURE0);

                glBindTexture(GL_TEXTURE_2D, findIt->second);
```

```
                                        findIt = model.textures.find(RenderCommon::Texture::Type::specular);
                                        if (findIt != model.textures.end())
                                        {

            glActiveTexture(GL_TEXTURE0 + 1);

            glBindTexture(GL_TEXTURE_2D, findIt->second);
                                        }
                                        else
                                        {

            glActiveTexture(GL_TEXTURE0 + 1);

            glBindTexture(GL_TEXTURE_2D, model.textures.find(RenderCommon::Texture::Type::diffuse)->second);
                                        }

                                        for (int i = 0; i < model.model->meshes.size(); ++i)
                                        {

            glBindVertexArray(model.meshRenderData[i].VAO);

            glDrawElements(GL_TRIANGLES, model.model->meshes[i].m_indices.size(), GL_UNSIGNED_INT, 0);

            glBindVertexArray(0);
                                        }
                                }

                                glfwSwapBuffers(m_window);
                                frameCount++;

                                auto endSeconds = glfwGetTime();
                                auto renderSeconds = endSeconds - startSeconds;

                                if (renderSeconds > 60)
                                        break;

        glfwPollEvents();
    }

                auto endSeconds = glfwGetTime();
                auto renderSeconds = endSeconds - startSeconds;
                auto averageFps = frameCount / renderSeconds;

                glfwDestroyWindow(m_window);

                return averageFps;
        }

        void showFPS()
        {
                double currentTime = glfwGetTime();
                static double lastTime = 0;
                static double nbFrames = 0;
                double delta = currentTime - lastTime;

                nbFrames++;

                if (delta >= 1.0) { // If last cout was more than 1 sec ago
                        double fps = double(nbFrames) / delta;

                        glfwSetWindowTitle(m_window, (std::string("OPENGL FPS: ") + std::to_string(fps)).c_str());
                        lastTime = currentTime;

                        nbFrames = 0;
                }
        }

        void processInput()
        {
                static float deltaTime = 0.0f;  // Time between current frame and last frame
                static float lastFrame = 0.0f; // Time of last frame

                float currentFrame = glfwGetTime();
                deltaTime = currentFrame - lastFrame;
                lastFrame = currentFrame;

                if (glfwGetKey(m_window, GLFW_KEY_W) == GLFW_PRESS)

            camera.ProcessKeyboard(Camera_Movement::FORWARD, deltaTime);
                if (glfwGetKey(m_window, GLFW_KEY_S) == GLFW_PRESS)

            camera.ProcessKeyboard(Camera_Movement::BACKWARD, deltaTime);
                if (glfwGetKey(m_window, GLFW_KEY_A) == GLFW_PRESS)

            camera.ProcessKeyboard(Camera_Movement::LEFT, deltaTime);
                if (glfwGetKey(m_window, GLFW_KEY_D) == GLFW_PRESS)

            camera.ProcessKeyboard(Camera_Movement::RIGHT, deltaTime);


                if (glfwGetKey(m_window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)
                        glfwSetInputMode(m_window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
                else
                        glfwSetInputMode(m_window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        }
private:
        GLFWwindow* m_window;
};

RenderOpengl::RenderOpengl()
        : m_impl{ std::make_unique< RenderOpengl::Impl>() }
{

}

RenderOpengl::~RenderOpengl() = default;

double RenderOpengl::startRenderLoop(std::vector<ModelInfo> modelInfos)
{
        return m_impl->startRenderLoop(std::move(modelInfos));
}
```

Shader.h

```
#pragma once

#include <string>
#include <filesystem>
#include "glad/glad.h"
#include "glm/glm.hpp"

class Shader final
{
public:
   Shader(std::string vertexSource, std::string fragmentSource);
   ~Shader();
```

```
    void use();
    void setBool(const std::string& name, bool value) const;
    void setInt(const std::string& name, int value) const;
    void setFloat(const std::string& name, float value) const;
    void setMat4(const std::string& name, const glm::mat4& mat);
    void setVec3(const std::string& name, const glm::vec3& vec);
private:
    GLuint compileShader(const char* source, GLenum shaderType);
    GLuint linkShader(GLuint vertex, GLuint fragment);
    void reportError(GLuint Id, std::string message);

            GLint getUniformLocation(const std::string& name) const;
private:
    unsigned int m_programID = 0;
public:
    unsigned int& Program = m_programID;
};
```

Shader.cpp

```
#include "Shader.h"
#include "glad/glad.h"
#include <cassert>
#include "glm/gtc/type_ptr.hpp"

using namespace std::literals;

Shader::Shader(std::string vertexSource, std::string fragmentSource)
{
            GLuint vertex = compileShader(vertexSource.c_str(),
GL_VERTEX_SHADER);
            GLuint fragment = compileShader(fragmentSource.c_str(),
GL_FRAGMENT_SHADER);

            m_programID = linkShader(vertex, fragment);

            glDeleteShader(vertex);
            glDeleteShader(fragment);
}

Shader::~Shader()
{
            if (m_programID)
                        glDeleteProgram(m_programID);

            m_programID = 0;
}

void Shader::use()
{
            assert(m_programID);
            glUseProgram(m_programID);
}

void Shader::setBool(const std::string& name, bool value) const
{
            assert(m_programID);
            try
            {
                        glUniform1i(getUniformLocation(name.c_str()),
(int)value);
            }
            catch (...) {}
}

void Shader::setInt(const std::string& name, int value) const
{
            assert(m_programID);
            try
            {
            glUniform1i(getUniformLocation(name.c_str()), value);
            }
            catch (...) {}
}

void Shader::setFloat(const std::string& name, float value) const
{
            assert(m_programID);
            try
            {
            glUniform1f(getUniformLocation(name.c_str()), value);
            }
            catch (...) {}
}

void Shader::setMat4(const std::string& name, const glm::mat4& mat)
{
            try
            {
            glUniformMatrix4fv(getUniformLocation(name.c_str()), 1,
GL_FALSE, glm::value_ptr(mat));
            }
            catch (...) {}
}

void Shader::setVec3(const std::string& name, const glm::vec3& vec)
{
            try
            {
            glUniform3fv(getUniformLocation(name.c_str()), 1,
glm::value_ptr(vec));
            }
            catch (...) {}
}

GLuint Shader::compileShader(const char* source, GLenum
shaderType)
{
            GLuint shader = 0;

            shader = glCreateShader(shaderType);
            glShaderSource(shader, 1, &source, NULL);
            glCompileShader(shader);

            GLint success = 0;
            glGetShaderiv(shader, GL_COMPILE_STATUS,
&success);

            if(!success)
                        reportError(shader, "shader compilation failed");

            return shader;
}

GLuint Shader::linkShader(GLuint vertex, GLuint fragment)
{
            GLuint programID = glCreateProgram();
            glAttachShader(programID, vertex);
            glAttachShader(programID, fragment);
            glLinkProgram(programID);

            GLint success = 0;
            glGetProgramiv(programID, GL_LINK_STATUS,
&success);

            if (!success)
                        reportError(programID, "shader linking failed");

            return programID;
}

void Shader::reportError(GLuint Id, std::string message)
{
            GLchar infoLog[512] = { 0 };
            glGetShaderInfoLog(Id, sizeof(infoLog), NULL, infoLog);
            throw std::runtime_error{ "ERROR: " + message + ":" +
infoLog };
}

GLint Shader::getUniformLocation(const std::string & name) const
```

```
{
	auto res = glGetUniformLocation(m_programID, name.c_str());
	if (res < 0)
		throw std::runtime_error{ "glGetUniformLocation for "s + name + " not found" };

	return res;
}
```

RenderVulkan.h

```
#pragma once

#include "Render.h"
#include <memory>

class RenderVulkan : public IRender
{
public:
	RenderVulkan();
	~RenderVulkan();

	double startRenderLoop(std::vector<ModelInfo> modelInfos) override;
private:
	class Impl;
	std::unique_ptr<Impl> m_impl;
};
```

RenderVulkan.cpp

```
#include "RenderVulkan.h"

#include "glad/vulkan.h"
#include "GLFW/glfw3.h"

//#define GLM_FORCE_RADIANS
//#define GLM_FORCE_DEFAULT_ALIGNED_GENTYPES

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/string_cast.hpp>

#include <stb_image.h>

#include "ShadersGen/Shaders.h"

#include "Utils.h"
#include "Camera.h"
#include "Model.h"

#include <iostream>
#include <vector>
#include <set>
#include <array>
#include <algorithm>
#include <optional>
#include <cassert>
#include <memory>
#include <functional>
#include <cstring>
#include <chrono>
#include <filesystem>
#include <thread>

namespace fs = std::filesystem;
using namespace std::literals;

#ifdef NDEBUG
constexpr bool enableValidationLayers = false;
#else
constexpr bool enableValidationLayers = true;
#endif

constexpr uint32_t g_WIDTH = 1280;
constexpr uint32_t g_HEIGHT = 720;

const std::array<const char*, 1> g_validationLayers = {
	"VK_LAYER_KHRONOS_validation",
};

const std::array<const char*, 1> g_deviceExtensions = {
	VK_KHR_SWAPCHAIN_EXTENSION_NAME
};

static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
	VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
	VkDebugUtilsMessageTypeFlagsEXT messageType,
	const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
	void* pUserData)
{
	std::cerr << "VULKAN: " << pCallbackData->pMessage << std::endl << std::endl;

	return VK_FALSE;
}

static GLADapiproc glad_vulkan_callback(const char* name, void* user)
{
	return glfwGetInstanceProcAddress((VkInstance)user, name);
}

static void errorCcallback(int error, const char* description)
{
	std::cerr << "glfwError #" << error << " ; desc: " << description << std::endl;
}


class RenderVulkan::Impl
{
public:
	static VkVertexInputBindingDescription getBindingDescription() {
		VkVertexInputBindingDescription bindingDescription{};

		bindingDescription.binding = 0;
		bindingDescription.stride = sizeof(RenderCommon::Vertex);
		bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

		return bindingDescription;
	}

	static std::array<VkVertexInputAttributeDescription, 7> getAttributeDescriptions()
	{
		std::array<VkVertexInputAttributeDescription, 7> attributeDescriptions{};

		attributeDescriptions[0].binding = 0;
		attributeDescriptions[0].location = 0;
		attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
		attributeDescriptions[0].offset = offsetof(RenderCommon::Vertex, Position);

		attributeDescriptions[1].binding = 0;
		attributeDescriptions[1].location = 1;
		attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
		attributeDescriptions[1].offset = offsetof(RenderCommon::Vertex, Normal);
```

```
		attributeDescriptions[2].binding = 0;
		attributeDescriptions[2].location = 2;
		attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
		attributeDescriptions[2].offset = offsetof(RenderCommon::Vertex, TexCoords);

		attributeDescriptions[3].binding = 0;
		attributeDescriptions[3].location = 3;
		attributeDescriptions[3].format = VK_FORMAT_R32G32B32A32_UINT;
		attributeDescriptions[3].offset = offsetof(RenderCommon::Vertex, BoneIDs);

		attributeDescriptions[4].binding = 0;
		attributeDescriptions[4].location = 4;
		attributeDescriptions[4].format = VK_FORMAT_R32G32B32A32_UINT;
		attributeDescriptions[4].offset = offsetof(RenderCommon::Vertex, BoneIDs) + sizeof(RenderCommon::Vertex::BoneIDs) / 2;

		attributeDescriptions[5].binding = 0;
		attributeDescriptions[5].location = 5;
		attributeDescriptions[5].format = VK_FORMAT_R32G32B32A32_SFLOAT;
		attributeDescriptions[5].offset = offsetof(RenderCommon::Vertex, Weights);

		attributeDescriptions[6].binding = 0;
		attributeDescriptions[6].location = 6;
		attributeDescriptions[6].format = VK_FORMAT_R32G32B32A32_SFLOAT;
		attributeDescriptions[6].offset = offsetof(RenderCommon::Vertex, Weights) + sizeof(RenderCommon::Vertex::Weights) / 2;

		return attributeDescriptions;
	}

	struct UniformBufferObject {
		constexpr static inline size_t MaxBoneTransforms = 100;
		alignas(16) glm::mat4 BoneTransform[MaxBoneTransforms];
		alignas(16) glm::vec3 viewPos;
	};

	struct PushConstantBufferObject {
		alignas(16) glm::mat4 PVM;
		alignas(16) glm::mat4 model;
	};
public:
	std::function<void(VkShaderModule)> m_shaderModuleDeleter =
		[this](VkShaderModule smodule) {
		vkDestroyShaderModule(m_device, smodule, nullptr);
	};

	std::function<void(VkBuffer)> m_bufferDeleter = [this](VkBuffer buffer) {
		if (buffer)
			vkDestroyBuffer(m_device, buffer, nullptr);
	};

	std::function<void(VkDeviceMemory)> m_deviceMemoryDeleter = [this](VkDeviceMemory deviceMemory) {
		if (deviceMemory)
			vkFreeMemory(m_device, deviceMemory, nullptr);
	};

	std::function<void(VkImage)> m_imageDeleter = [this](VkImage image) {
		if (image)
			vkDestroyImage(m_device, image, nullptr);
	};

	std::function<void(VkImageView)> m_imageViewDeleter = [this](VkImageView imageView) {
		if (imageView)
			vkDestroyImageView(m_device, imageView, nullptr);
	};

	std::function<void(VkSampler)> m_samplerDeleter = [this](VkSampler sampler) {
		if (sampler)
			vkDestroySampler(m_device, sampler, nullptr);
	};

	std::function<void(VkCommandPool)> m_commandPoolDeleter = [this](VkCommandPool commandPool) {
		if (commandPool)
			vkDestroyCommandPool(m_device, commandPool, nullptr);
	};

	using CommandBufferDeleter = std::function<void(VkCommandBuffer)>;

	using unique_ptr_shared_module = std::unique_ptr<std::remove_pointer_t<VkShaderModule>, decltype(m_shaderModuleDeleter)>;
	using unique_ptr_buffer = std::unique_ptr<std::remove_pointer_t<VkBuffer>, decltype(m_bufferDeleter)>;
	using unique_ptr_device_memory = std::unique_ptr<std::remove_pointer_t<VkDeviceMemory>, decltype(m_deviceMemoryDeleter)>;
	using unique_ptr_image = std::unique_ptr<std::remove_pointer_t<VkImage>, decltype(m_imageDeleter)>;
	using unique_ptr_image_view = std::unique_ptr<std::remove_pointer_t<VkImageView>, decltype(m_imageViewDeleter)>;
	using unique_ptr_sampler = std::unique_ptr<std::remove_pointer_t<VkSampler>, decltype(m_samplerDeleter)>;
	using unique_ptr_command_pool = std::unique_ptr<std::remove_pointer_t<VkCommandPool>, decltype(m_commandPoolDeleter)>;
	using unique_ptr_command_buffer = std::unique_ptr<std::remove_pointer_t<VkCommandBuffer>, CommandBufferDeleter>;
public:
	struct QueueFamilyIndices {
		std::optional<uint32_t> graphicsFamily;
		std::optional<uint32_t> presentFamily;

		bool isComplete() {
			return graphicsFamily.has_value() && presentFamily.has_value();
		}
	};

	struct SwapChainSupportDetails {
		VkSurfaceCapabilitiesKHR capabilities{};
		std::vector<VkSurfaceFormatKHR> formats;
		std::vector<VkPresentModeKHR> presentModes;
	};

	struct MeshVertexBuffer
	{
		MeshVertexBuffer(RenderVulkan::Impl* _this)
:
```

```
                m_vertexBuffer{ nullptr, _this->m_bufferDeleter },
                m_vertexBufferMemory{ nullptr, _this->m_deviceMemoryDeleter }
            {}

            unique_ptr_buffer m_vertexBuffer;
            unique_ptr_device_memory m_vertexBufferMemory;
        };

        struct MeshTextureImage
        {
            MeshTextureImage(RenderVulkan::Impl* _this) :
                textureImage{ nullptr, _this->m_imageDeleter },
                textureImageMemory{ nullptr, _this->m_deviceMemoryDeleter },
                textureImageView{ nullptr, _this->m_imageViewDeleter },
                textureSampler{ nullptr, _this->m_samplerDeleter }
            {}

            uint32_t mipLevels = 0;
            unique_ptr_image textureImage;
            unique_ptr_device_memory textureImageMemory;
            unique_ptr_image_view textureImageView;
            unique_ptr_sampler textureSampler;
        };

        struct MeshUniformBuffer
        {
            MeshUniformBuffer(RenderVulkan::Impl* _this) :
                uniformBuffer{ nullptr, _this->m_bufferDeleter },
                uniformBuffersMemory{ nullptr, _this->m_deviceMemoryDeleter }
            {}

            unique_ptr_buffer uniformBuffer;
            unique_ptr_device_memory uniformBuffersMemory;
            void* uniformBufferMemoryMapping;
        };

        struct VulkanModel
        {
            std::unique_ptr<RenderCommon::Model> model;

            std::map<RenderCommon::Texture::Type, MeshTextureImage*> meshTextureImages;
            std::vector<MeshVertexBuffer> meshVertexBuffers;
            std::vector<MeshUniformBuffer> meshUniformBuffers;
            std::vector<VkDescriptorSet> meshDescriptorSet;

            glm::vec3 position{};
            glm::vec3 scale{};

            std::vector<PushConstantBufferObject> pushConstant{};
            std::vector<UniformBufferObject> uniformBuffer{};

            ModelInfo info{};
        };

        struct ThreadData {
            ThreadData(RenderVulkan::Impl* self) :
                m_this{ self },
                commandPool{ nullptr, self->m_commandPoolDeleter }
            {
            }

            ThreadData(ThreadData&&) = default;

            ~ThreadData()
            {
                commandBuffers.clear();
            }

            RenderVulkan::Impl* m_this;
            unique_ptr_command_pool commandPool;

        std::vector<std::vector<unique_ptr_command_buffer>> commandBuffers;
            int usedCommandBuffers = 0;
        };
public:
        GLFWwindow* m_window = nullptr;

        VkInstance m_instance = VK_NULL_HANDLE;
        VkDebugUtilsMessengerEXT m_debugMessenger = VK_NULL_HANDLE;

        VkSurfaceKHR m_surface = VK_NULL_HANDLE;

        VkPhysicalDevice m_physicalDevice = VK_NULL_HANDLE;

        VkDevice m_device = VK_NULL_HANDLE;
        VkQueue m_graphicsQueue = VK_NULL_HANDLE;
        VkQueue m_presentQueue = VK_NULL_HANDLE;

        VkSwapchainKHR m_swapChain = VK_NULL_HANDLE;

        std::vector<VkImage> m_swapChainImages;
        VkFormat m_swapChainImageFormat;
        VkExtent2D m_swapChainExtent;

        std::vector<VkImageView> m_swapChainImageViews;

        VkRenderPass m_renderPass = VK_NULL_HANDLE;

        VkDescriptorSetLayout m_descriptorSetLayout = VK_NULL_HANDLE;
        VkDescriptorPool m_descriptorPool = VK_NULL_HANDLE;

        VkPipelineLayout m_pipelineLayout = VK_NULL_HANDLE;

        VkPipeline m_graphicsPipeline = VK_NULL_HANDLE;
        VkPipeline m_graphicsPipelineSimple = VK_NULL_HANDLE;

        std::vector<VkFramebuffer> m_swapChainFramebuffers;

        VkCommandPool m_commandPool = VK_NULL_HANDLE;

        std::vector<VkCommandBuffer> m_commandBuffers;

        std::vector<VkSemaphore> m_imageAvailableSemaphores;
        std::vector<VkSemaphore> m_renderFinishedSemaphores;

        std::vector<VkFence> m_inFlightFences;
        std::vector<VkFence> m_imagesInFlight;
```

```
	VkImage m_depthImage = VK_NULL_HANDLE;
	VkDeviceMemory m_depthImageMemory = VK_NULL_HANDLE;
	VkImageView m_depthImageView = VK_NULL_HANDLE;

	VkSampleCountFlagBits m_msaaSamples = VK_SAMPLE_COUNT_1_BIT;

	VkImage colorImage = VK_NULL_HANDLE;
	VkDeviceMemory colorImageMemory = VK_NULL_HANDLE;
	VkImageView colorImageView = VK_NULL_HANDLE;
public:
	bool m_framebufferResized = false;
	int m_framebufferWidth = g_WIDTH;
	int m_framebufferHeight = g_HEIGHT;

	Camera camera{ glm::vec3(8.207467, 2.819616, 18.021290) };

	double m_deltaTime{};
	double m_currentTime{};
	double m_lastTime{};
```

Scene.h

```
#include "Render.h"

class Scene
{
public:
	Scene(IRender& render, RenderGuiData guiData);

	double run();
private:
	IRender& m_render;

	std::vector<ModelInfo> m_modelInfos;
};
```

Scene.cpp

```
#include "Scene.h"
#include <array>
#include "glm/glm/glm.hpp"

namespace
{

	enum ModelType
	{
		Chimp, // 1
		Bird, // 3
		Crocodile, // 1
		Pony, // 8
	};

	const std::array<ModelInfo, 4>& getModelInfos()
	{
		static const std::array<ModelInfo, 4> modelInfos = [] {
			ModelInfo modelChimp;

			modelChimp.modelPath = "resources/chimp/chimp.FBX";
			modelChimp.texturePath = "resources/chimp/chimp_diffuse.jpg";
			modelChimp.maxAnimationNumber = 1;


			ModelInfo modelBird;

			modelBird.modelPath = "resources/bird/Bird.FBX";
			modelBird.texturePath = "resources/bird/Fogel_Mat_Diffuse_Color.png";
			modelBird.maxAnimationNumber = 3;

			modelBird.scaleX = 0.0025f;
			modelBird.scaleY = 0.0025f;
			modelBird.scaleZ = 0.0025f;

			ModelInfo modelCrocodile;

			modelCrocodile.modelPath = "resources/crocodile/Crocodile.fbx";
			modelCrocodile.texturePath = "resources/crocodile/Crocodile.jpg";

			modelCrocodile.scaleX = 0.015f;
			modelCrocodile.scaleY = 0.015f;
			modelCrocodile.scaleZ = 0.015f;


		modelCrocodile.maxAnimationNumber = 1;


			ModelInfo modelPony;

			modelPony.modelPath = "resources/shetlandponyamber/ShetlandPonyAmberM.fbx";
			modelPony.texturePath = "resources/shetlandponyamber/shetlandponyamber.png";
			modelPony.maxAnimationNumber = 8;

			modelPony.scaleX = 0.015f;
			modelPony.scaleY = 0.015f;
			modelPony.scaleZ = 0.015f;

			return std::array<ModelInfo, 4>{ modelChimp, modelBird, modelCrocodile, modelPony };
		}();

		return modelInfos;
	}

	ModelInfo getModelInfo(ModelType type)
	{
		return getModelInfos()[type];
	}

	ModelInfo getSimpleCubeModelInfo()
	{
		ModelInfo modelInfo;

		modelInfo.modelPath = "resources/cube.obj";
		modelInfo.texturePath = "resources/container.jpg";
		modelInfo.simpleModel = true;

		return modelInfo;
	}
}

Scene::Scene(IRender& render, RenderGuiData guiData) :
	m_render{ render }
{
	std::vector<int> currModelNum;

	int modelNumber = guiData.modelNumber;
	while (modelNumber)
	{
		if (modelNumber >= 12)
		{
			currModelNum.push_back(12);
```

```
                modelNumber -= 12;
            }
            else
            {

        currModelNum.push_back(modelNumber);
                break;
            }
        }

        for (int j = 0; j < 12; ++j)
        {
            for (int i = 0;  j < currModelNum.size() && i < currModelNum[j]; ++i)
            {
                ModelInfo modelInfo{};
                if (guiData.simpleScene)
                {
                    modelInfo = getSimpleCubeModelInfo();
                    modelInfo.simpleModel = true;
                }
                else
                {
                    modelInfo = getModelInfo(ModelType(fixedRundom[(i + 1) * (j + 1)]));
                    modelInfo.simpleModel = false;
                }

                int yOffset = i / 4;

                modelInfo.posY += yOffset * 4;
                modelInfo.posX += (i - (yOffset * 4)) * (i % 2 == 0 ? 5 : 5);
                modelInfo.posZ -= j * 6;

                if(modelInfo.maxAnimationNumber)

        modelInfo.animationNumber = ((i + 1) * (j + 1)) % modelInfo.maxAnimationNumber;


        m_modelInfos.push_back(modelInfo);
            }
        }
}


double Scene::run()
{
        return m_render.startRenderLoop(m_modelInfos);
}
}
```

Main.cpp

```
#include "RenderOpengl.h"
#include "RenderVulkan.h"
#include "RenderGUI.h"
#include "Scene.h"

#include <iostream>
#include <string>
#include <filesystem>

using namespace std::literals;
namespace fs = std::filesystem;

#ifdef __linux__

#include <libgen.h>
#include <unistd.h>
#include <linux/limits.h>

namespace
{
    void SetupCurrentDirectory()
    {
        std::string modulePath;
        modulePath.resize(PATH_MAX);

        ssize_t count = readlink("/proc/self/exe", &modulePath[0], PATH_MAX);
        if(count < 0)
            throw std::runtime_error("readlink failed"s + std::to_string(errno));

        char* path = dirname(&modulePath[0]);
                                    chdir(path);
    }
}
#endif

#ifdef _WIN32

#include "Windows.h"

namespace
{
        void SetupCurrentDirectory()
        {
                size_t length = MAX_PATH;
                std::wstring modulePath;

                while (true)
                {
                        modulePath.resize(length);
                        DWORD size = GetModuleFileName(NULL, &modulePath[0], MAX_PATH);

                        DWORD err = GetLastError();

                        if (err == ERROR_SUCCESS)
                                break;

                        if (err == ERROR_INSUFFICIENT_BUFFER)
                                length *= 2;
                        else
                                throw std::runtime_error{ "GetModuleFileName error: "s + std::to_string(err) };
                }

                if (0 == SetCurrentDirectory(fs::path{ modulePath }.parent_path().c_str()))
                        throw std::runtime_error{ "SetCurrentDirectory error: "s + std::to_string(GetLastError()) };
        }
}
#endif

#include <thread>

#ifdef _WIN32
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow)
#else
int main()
#endif
try {
        SetupCurrentDirectory();

        RenderGuiData guiData{};
        while (true)
        {
                RenderGUI gui;
                guiData = gui.startRenderLoop(guiData);
```

```
                if (guiData.renderType == RenderGuiData::RenderType::OpenGL)
                {
                        RenderOpengl openglRender;
                        Scene scene{ openglRender, guiData };
                        guiData.averageFps = scene.run();
                };
                        guiData.averageFps = scene.run();
                }
                else
                {
                        break;
                }
        }
}
catch (const std::exception& ex)
{
        auto wtf = ex.what();
        std::cerr << "Main exception: " << ex.what() << std::endl;
        std::cin.get();
}
catch (...)
{
        std::cerr << "\n\nUNSPESIFIED EXCEPTION!!!\n\n";
        std::cin.get();
}
```

```
                }
                else if (guiData.renderType == RenderGuiData::RenderType::Vulkan)
                {
                        RenderVulkan vulkanRender;
                        Scene scene{ vulkanRender, guiData
```

ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП'ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТОЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Автор – Поліщук І.А., студент групи ПЗ1921
Науковий керівник – к.т.н., доц. Іванов О.П.
Дніпровський національний університет імені академіка В. Лазаряна
Україна

Передові досягнення науки і техніки в області комп'ютерної анімації, такі як імітація руху або відбиття світла загалом є дуже критичними до часу виконання завдань, які на них покладаються. Задачі, які вирішують більшість систем відображення комп'ютерної графіки є задачами так званого жорсткого реального часу (коли перевищення часу виконання поставлених завдань може призвести до невідворотних наслідків) або м'якого реального часу (коли перевищення часу вирішення небажане, але припустиме).

Саме тому ми звертаємо нашу увагу на можливість розподілу таких задач на декілька ядер процесору за для зменшення загального часу обробки кожного кадру анімації перед його відображенням.

Для виводу графіки на екран використовують центральний процесор та графічний процесор. Центральний процесор обчислює позицію, зміщення, форму об'єктів та передає ці дані до графічного процесору. Графічний процесор у свою чергу перетворює дані таких об'єктів у форму придатну для подальшого виведення на екран.

Історично склалося, що центральний процесор та графічний процесор розвиваються окремо один від одного. Тому зараз, коли центральний процесор досяг успіху у паралельному виконанні задач на декількох ядрах, одноядерні підходи до обробки та передачі даних на графічний процесор не є ефективними.

Один з таких підходів є використання OpenGL – програмного інтерфейсу до графічного пристрою. Цей інтерфейс розроблявся у 90-их років, коли багатоядерні процесори ще не були досить популярними. Тому методи до обробки та передачі даних які запроваджує цей інтерфейс не дає змогу робити це паралельно, оскільки будь-яка операція з OpenGL може змінювати його глобальний стан, який не передбачає одночасні зміни і не запроваджує синхронізації щоб захиститися від стана гонитви (ситуація в якій робота чи результат операції залежить від послідовності або тривалості виконання).

Щоб виправити такі недоліки може бути використаний інтерфейс Vulkan. Цей інтерфейс не використовує глобальних об'єктів які неможливо синхронізувати. Натомість робота з ним є складніша, оскільки об'єкти які були сховані у OpenGL відтепер мають бути створені та використані розробником.

Одним із таких об'єктів є командний буфер. Командний буфер це буфер, який зберігає заздалегідь заповнені команди які має виконати графічний процесор. Кожен командний буфер може використовуватись незалежно один від одного, тому кожен потік може обчислювати інформацію об'єктів та заповнювати такі буфери окремо. У кінці, ці буфери мають бути відправлені до графічного процесору, де з них буде створене зображення.

Інші дослідження зосереджують увагу на порівнянні різниці використання одного та усіх ядер процесору. У цьому досліджені буде додатково з'ясовано, як саме буде змінюватись ефективність відображення анімації з використанням OpenGL та Vulkan зі збільшенням кількості ядер процесору, які можливості дає Vulkan з обробки об'єктів анімації паралельно на процесорі, які методи синхронізації мають бути використані за для такої паралельної обробки та залежність ефективності від кількості та складності анімованих об'єктів.

ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ВІДОБРАЖЕННЯ КОМП'ЮТЕРНОЇ ГРАФІКИ РЕАЛЬНОГО ЧАСУ НА БАГАТО-ЯДЕРНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ OPENGL ТА VULKAN API

Автор: Поліщук І. А.

Науковий керівник – к.т.н., доц. Іванов О.П

Дніпровський національний університет імені академіка В. Лазаряна

ВСТУП

Передові досягнення науки і техніки в області комп'ютерної анімації, такі як імітація руху або відбиття світла загалом є дуже критичними до часу виконання завдань, які на них покладаються.

Саме тому ми звертаємо нашу увагу на можливість розподілу таких задач на декілька ядер процесору за для зменшення загального часу обробки кожного кадру анімації перед його відображенням.

Для виводу графіки на екран використовують центральний процесор та графічний процесор. Центральний процесор обчислює позицію, зміщення, форму об'єктів та передає ці дані до графічного процесору. Графічний процесор у свою чергу перетворює дані таких об'єктів у форму придатну для подальшого виведення на екран.

Історично склалося, що центральний процесор та графічний процесор розвиваються окремо один від одного. Тому зараз, коли центральний процесор досяг успіху у паралельному виконанні задач на декількох ядрах, підходи до обробки та передачі даних на графічний процесор не є ефективними.

Один з таких підходів є використання OpenGL – програмного інтерфейсу до графічного пристрою. Цей інтерфейс розроблявся у 90-их років, коли багатоядерні процесори ще не були досить популярними. Тому методи до обробки та передачі даних які запроваджує цей інтерфейс не дає змогу робити це паралельно, оскільки будь-яка операція з OpenGL може змінювати його глобальний стан, який не передбачає одночасні зміни і не запроваджує синхронізації щоб захиститися від стана гонитви (ситуація в якій робота чи результат операції залежить від послідовності або тривалості виконання).

Щоб виправити такі недоліки може бути використаний інтерфейс Vulkan. Цей інтерфейс не використовує глобальних об'єктів які неможливо синхронізувати. На томість робота з ним є складніша, оскільки об'єкти які були сховані у OpenGL відтепер мають бути створені та використані розробником.

ПОСТАНОВКА ЗАДАЧІ

Розробити систему для дослідження та дослідити як саме буде змінюватись ефективність відображення анімації з використанням OpenGL та Vulkan, а саме залежність ефективності від кількості об'єктів анімації та залежність ефективності від кількості ядер процесору.

МЕТОДИКА ДОСЛІДЖЕННЯ

Вибрані моделі для дослідження були розділені на прості та складні.

У якості простих анімованих моделей була обрана модель 3д куба. Також було застосовано метод анімації по траєкторії. Моделі куба кружляють навколо своєї осі.

У якості складних анімованих моделей були обрані моделі тварин анімовані за допомогою скелетної анімації.

Для обчислення ефективності анімації була обрана величина середньої кількості кадрів які були опарацьовані та намальовані за певний період часу.

Формула для обчислення середньої кількості кадрів за секунду наведена далі:

$$AFPS = \frac{N}{S}, \qquad (1)$$

де AFPS (Average frames per second) – це середня кількість кадрів за секунду; N – кількість кадрів які були згенеровані за період тестування; S – кількість секунд за які було проведене тестування.

Щоб замірити ефективність дослідження зі збільшенням ядер процесору потрібно запустити програму та зробити досліди на системах з різною кількістю ядер. Було вирішено обрати спосіб з конфігуруванням кількості ядер яке буде використовувати операційна система через конфігураційне вікно операційної системи, наведене на рисунку 1 нижче.

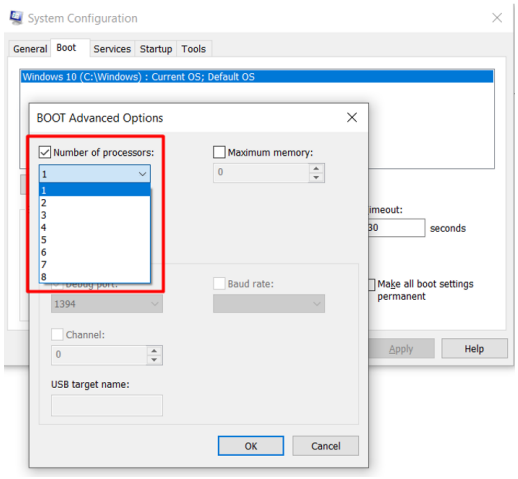

Рисунок 1 – Вікно конфігурації

Змінюючи кількість процесорів у конфігураційному вікні та перезавантажуючи комп'ютер, операційна система бачить лише задану кількість ядер процесору. Саме так й буде зроблені виміри дослідження на різних кількостях ядер.

Однопоточна модель обробки та відображення графіки яка є у OpenGL є дуже простою. У загальному виді вона складається з трьох операцій:

4. Опрацювати ввід користувача
5. Відновити дані для наступного кадру
6. Відобразити кадр на екрані.

Ці операції виконуються у циклі до завершення програми. Схема однопоточного циклу рендерінгу наведена на рисунку 2:

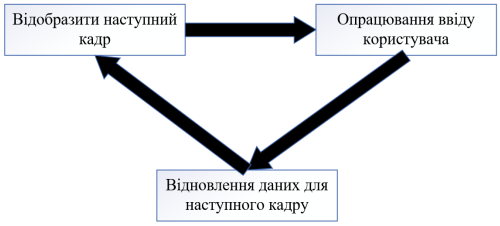

Рисунок 2 – Однопоточний рендер

Багатопоточна модель обробки графіки Vulkan складається з основного потоку виконання, та допоміжних потоків. Основний потік дає завдання допоміжним потікам на обробку графічних об’єктів.

Кожен допоміжний потік відновлює буфера матриць об’єкту та команди відрисовки об’єкта.

Головний потік у свою чергу чекає завершення роботи усіх допоміжних потіків, збирає їх, та виконує відрисовку на екрані на основі даних які були створені чі змінені у допоміжних потіках для кожного графічного об’єкту анімації.

Схема багатопоточного циклу рендерінгу наведена на рисунку 3:

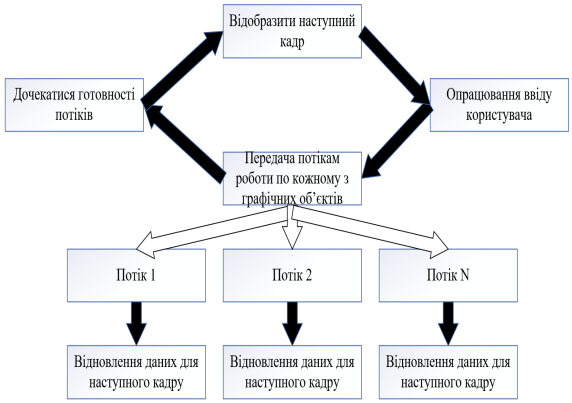

Рисунок 3 – Багатопоточний рендер

Для заміру часу буде використано рекомендований Microsoft спосіб з викликом функції програмного інтерфейсу до операційної системи Windows QueryPerformanceCounter з роздільною точністю менше ніж одна мікросекунда. Цього більш ніж досить для заміру часу виконання одного кадру.

ПРОГРАМНЕ ЗАБЕЗБЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ

Для цього дослідження було спроектоване та розроблене програмне забезпечення для рендеру анімованих об’єктів.

У головному меню можливо настроювати кількість об’єктів, тип рендеру та тип об’єкта. Головне меню інструментального засобу показане на рисунку 4:

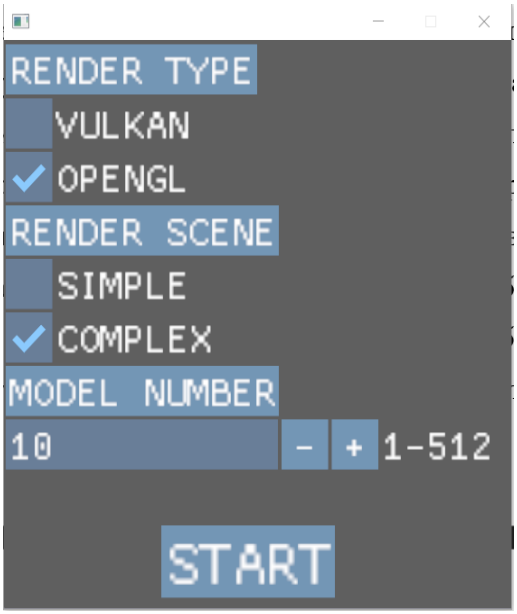

Рисунок 4 – Головне меню ПЗ

Процес рендеру після налаштування розробленого ПЗ показано на рисунку 5 та 6:

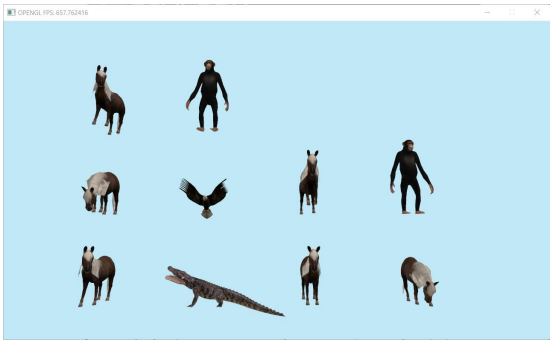

Рисунок 5 – Рендер складних моделей

Рисунок 6 – Рендер простих моделей

Після закінчення рендеру буде показано вікно з вимірами необхідними для дослідження. Вікно з результатами наведено на рисунку 7:

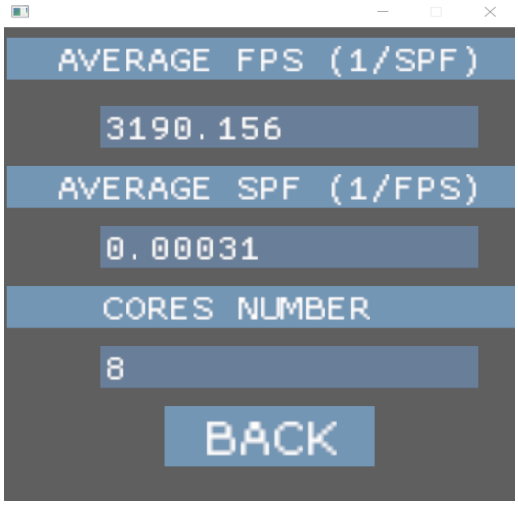

Рисунок 7 – Вікно результатів

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

У даній роботі для оцінки зміни ефективності використовувалися 10, 100 та 500 анімованих графічних моделей, які одночасно відрисовувалися на екрані під час рендеру. Тести проводилися двічі. Один раз на рендері за допомогою програмного інтерфейсу до комп'ютерної графіки OpenGL. Другий раз на рендері за допомогою програмного інтерфейсу до комп'ютерної графіки Vulkan.

Також іспити подвіювалися через те, що у якості анімованих моделей було використано два типи: прості та складні.

Для кожного типу кожної кількості моделей було зібрано інформацію, а саме:

- Середня кількість кадрів за секунду.
- Кількість використаної пам'яті графічного процесору.
- Кількість використаної оперативної пам'яті

На рисунку 7 та 8 зображено графіки зміни середньої кількості кадрів відображаємих за секунду зі зміною кількості ядер для однопоточного OpenGL та багатопоточного Vulkan типів рендерів для простих та складних моделей:

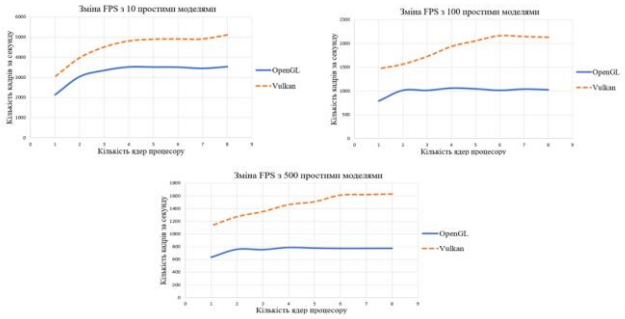

Рисунок 7 – Зміна FPS з простими моделями

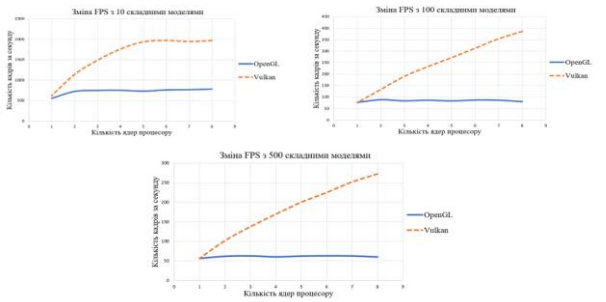

Рисунок 8 – Зміна FPS з складними моделями

Рисунок 7 показує що для десяти простих моделей зріст кількості кадрів відображаємих за секунду застосовуючи багатопоточний рендер на Vulkan майже відсутній починаючи з п’яти ядер. Для ста моделей зріст зупинився на шостому ядрі, для п’ятиста на сьомому ядрі.

Це можна пояснити тим, що прости моделі, які являють собою 3д куб який обертається навколо своєї осі, навантажує процесор досить слабко, так що декілька моделей встигають обробитися лише на одному ядрі, і таким чином навіть для п’ятиста простих моделей, останні ядра не задіяні, і не впливають на ефективність.

Рисунок 8 показує що для десяти складних моделей ріст кількості кадрів відображаємих за секунду зупиняється на шостому ядрі, майже як і для простих моделей. Але для ста та навіть п’ятиста моделей, ріст не зупиняється, та є лінійним. Це пояснюється тим, що для скелетної анімації, яка застосовується для складних моделей, треба набагато більше процесорного часу для вирахування матриць та зміщень вершин, також кількість цих вершин значно більша ніж у простих моделей. Таким чином, для усіх ядер є робота, адже одне чи декілька ядер вже не можуть вчасно обробити велику кількість складних моделей, так що би не залишати роботи для решти ядер.

Що стосується однопоточного рендеру на OpenGL, то його зріст незначно збільшується тільки на двох ядрах. Це пояснюється тим, що операційна система не перериває обробку об’єктів на одному ядрі своїми службовими процесами, а використовує вільне ядро.

На рисунку 9 відображено розподіл роботи по ядрам процесору використовуючи програмний інтерфейс до комп’ютерної графіки OpenGL.

На рисунку 10 відображено розподіл роботи по ядрам процесору використовуючи програмний інтерфейс до комп’ютерної графіки Vulkan.

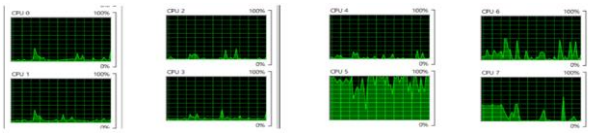

Рисунок 9 – Розподіл роботи по ядрам з OpenGL

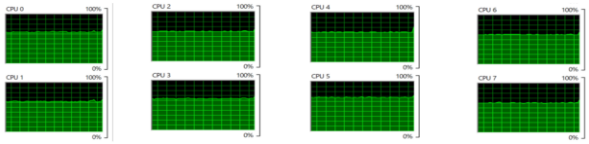

Рисунок 10 – Розподіл роботи по ядрам з Vulkan

Як видно з рисунка 9, при однопоточному рендері з OpenGL лише п’яте ядро постійно навантажено майже на 100%. А усі інші ядра майже нічим не навантажені. Це не є добре, оскільки рендер сповільнюється тому що повинно чекати на обробку

інформації ядром, яке перенавантажено, замість того щоб використовувати інші ядра, як це відбувається при багатопоточному рендері на Vulkan на рисунку 10.

На рисунку 11 та 12 зображено графіки зміни кількості використаної пам’яті графічної карти зі зміною кількості ядер для однопоточного OpenGL та багатопоточного Vulkan типів рендерів для простих та складних моделей.

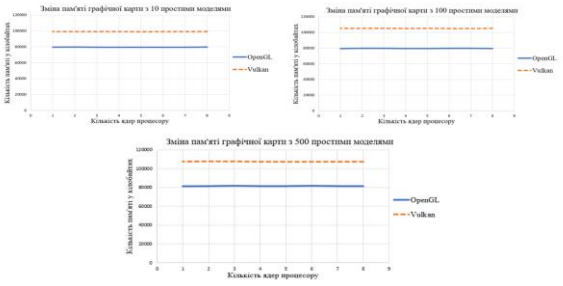

Рисунок 11 – Зміна GPU пам’яті з простими моделями

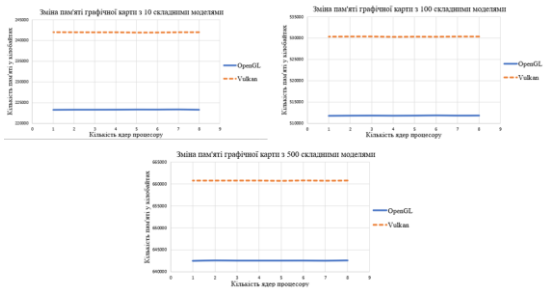

Рисунок 12 – Зміна GPU пам’яті з складними моделями

Як видно з рисунків 11 та 12 пам’ять графічної карти для рендеру анімаційних моделей не збільшується зі збільшенням ядер процесору, теж саме було встановлено і для оперативної пам’яті.

ВИСНОВКИ

Для 3д моделей навіть значної кількості, у яких мало вершин та прості операції, велика кількість ядер є зайвою. Для малого числа складних моделей велика кількість ядер теж є зайвою, але зі збільшенням таких моделей, можна досить швидко навантажити усі ядра, так що би вони опрацьовувалися паралельно. І залучивши усі ядра процесора, ми досягнемо піку продуктивності, коли усі ядра однаково навантажені, та максимальна кількість анімованих об’єктів обробляється паралельно а не великою чергою. Оскільки навантаження на процесор досить велике, паралельна обробка таких моделей на різних ядер з кожним ядром збільшує кількість кадрів за секунду на 50%.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Kessenich J. OpenGL Programming Guide Ninth Edition J. Kessenich, G. Sellers, D. Shreiner – Boston: Addison–Wesley, ‘. – 976 с.
2. Seller G. Vulkan Programming Guide G. Seller – Boston: Addison-Wesley Professional, 2016. – 478 с.