

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування інституту, факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО РУШІЯ ДЛЯ
WINDOWS
SOFTWARE IMPLEMENTATION OF A GAME ENGINE FOR
WINDOWS

Виконав: студент(ка) 4 курсу, групи КНТ-129

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

БЕНДО Б.І.

(прізвище та ініціали)

Керівник СУББОТІН С.О.

(прізвище та ініціали)

Рецензент КОРОТКИЙ О.В.

(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет комп'ютерних наук та технологій
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2023 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

БЕНДО Богдана Ігоровича

(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація ігрового рушія для Windows
Software implementation of a game engine for Windows

керівник проєкту (роботи) СУББОТІН Сергій Олександрович, д.т.н., професор,
 (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “04” квітня 2023 року № 87

2. Строк подання студентом проєкту (роботи) 01 червня 2023 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне
завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Розробка програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

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

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СУББОТІН С.О., професор		
Нормоконтроль	КАМІНСЬКА Ж.К., асистент		

7. Дата видачі завдання « 04 » квітня 2023 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

_____ Богдан БЕНДО
(підпис) (ім'я прізвище)

Керівник проєкту (роботи)

_____ Сергій СУББОТІН
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 99 с., 17 рис., 2 табл., 2 дод., 18 джерел.

ІГРОВІ РУШІЇ, КОМП'ЮТЕРНІ ІГРИ, СУЧАСНЕ СТАНОВИЩЕ ІГРОВИХ РУШІЇВ, ШЕЙДЕР, РЕНДЕР, ГРАФІКА, ВЕКТОРИЗАЦІЯ, ПАРАЛЕЛІЗАЦІЯ, SIMD, VULKAN, WINDOWS, GLM, GLFW, МУЛЬТИПОТОЧНІСТЬ, АСИНХРОННІСТЬ.

Об'єкт дослідження – процес створення ігрового рушія.

Предмет дослідження – ігрові рушії, що використовуються для створення двомірних та тримірних комп'ютерних ігор.

Мета роботи – Виявити, дослідити та продивитись процес створення ігрового рушія. Дослідити архітектуру ігрового рушія, типи та призначення структурних компонентів. Створити свій особистий ігровий рушій.

Матеріали, методи та технічні засоби: об'єктно орієнтоване програмування, мова програмування C++, математична бібліотека GLM, бібліотека взаємодії з операційною системою GLFW, графічний інструментарій Vulkan SDK, персональний комп'ютер з процесором AMD Ryzen 5800HS під управлінням операційної системи Microsoft Windows 10.

Результати. Створено ігровий рушій для використання його з операційною системою Windows.

Висновки. Розроблено ігровий рушій для створення тримірних ігор та проведення задач типу «рендерінг» та «огляд математичної інформації у тривимірному просторі».

Галузь використання – Розробка комп'ютерних ігор.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 99 pages, 17 figures, 2 tables, 2 appendixes, 18 sources.

GAME ENGINE, COMPUTER GAMES, CURRENT STATE OF GAME ENGINES, SHADERS, RENDER, GRAPHICS, VECTORISATION, PARALLELISATION, SIMD, VULKAN, WINDOWS, GLM, GLFW, MULTITHREADING, ASYNC.

Object of study is a process of a game engine implementation.

Subject of study are game engines that's being used to create a three-dimensional and two-dimensional computer games.

The purpose of the work is to learn and analyze a process of game engine implementation. Analyze an overall game engine architecture and different game engine systems and components. Implement my own game engine.

Materials, methods, and tools: object-oriented programming, C++ programming language, mathematical library GLM, library for interaction with operational system GLFW, graphical instruments pack Vulkan SDK, personal computer with AMD Ryzen 5800HS central processing unit under control of Microsoft Windows 10 .

Results. Implemented game engine that can be used to produce computer games for operational system Windows.

Conclusions. The game engine can be used in variety of different ways: to create a computer game for operational system Windows, or to use it in tasks of render family, or to simply visualize mathematical information in three-dimensional system.

The field of use is a game development industry.

ЗМІСТ

Перелік скорочень та умовних позначок.....	7
Вступ.....	8
1 Аналіз предметної області.....	9
1.1 Аналіз завдання.....	9
1.2 Аналіз проблеми.....	10
1.3 Критерії оцінки існуючих рішень.....	11
1.4 Аналіз існуючих рішень.....	13
1.5 Підсумок.....	20
2 Розробка архітектури програми.....	21
2.1 Опис технічних відомостей.....	21
2.2 Опис архітектури програмного застосунку.....	29
2.3 Висновки за розділом.....	33
3 Розробка програми.....	35
3.1 Реалізація BAO_BWindow.....	35
3.2 Реалізація класу BAO_BApp.....	37
3.3 Реалізація класу BAO_BKeyboard.....	39
3.4 Реалізація класу BAO_BCamera.....	40
3.5 Реалізація класу BAO_BDevice.....	42
3.6 Реалізація класу BAO_BRender.....	43
3.7 Реалізація класу BAO_BPipeline.....	45
3.8 Висновок.....	46
4 Експлуатація, тестування, та експериментальне дослідження програми....	48
4.1 Опис результатів експлуатації програмного забезпечення.....	48
4.2 Аналіз проведених тестів та отриманого результату.....	48
Висновок.....	54
Перелік джерел посилання.....	55
Додаток А Текст програми.....	57
Додаток Б Слайди презентації.....	93

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	Application Programming Interface
AVX	Advanced Vector Extensions
AMD	Advanced Micro Devices
CUDA	Compue Unified Device Architecture
GLFW	Graphics Library Framework
GLM	OpenGL Mathematics
RAI	Resource Acquisition is Initialization
RT	Ray-Tracing
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
UE4	Unreal Engine 4
UE5	Unreal Engine 5

ВСТУП

У нинішній час, комп'ютерні ігри стали неймовірно комплексними програмними застосунками. Інколи стається так, що фінальний продукт не підпадає під смаки гравців навіть не зважаючи на комплексність та серйозність проекту. Однією з причин таких реакцій є функціональна обмеженість розробників ігрових застосунків, що викликана низькою якістю ігрового рушія. У нинішній час сфера розробки ігрових рушіїв розвивається семимильними кроками, але у цій сфері уся увага прикована лише до трьох великих рушіїв, аналіз та опис яких наведено у першому розділі дипломної роботи. Можна сказати, що вони є великими монополістами ринку розробки ігрових рушіїв та ігор. Їхні продукти мають як плюси, так і недоліки. Підставою для створення ігрового рушія стала скрутна ситуація з вибором між існуючими ігровими рушіями та бажання імплементувати те, що не було додано до крупних ігрових рушіїв.

Метою роботи є створення ігрового рушія для використання та створення ігрових застосунків під операційну систему Windows.

Основними завданнями роботи є: програмне створення застосунку типу «ігровий рушій» для операційної системи Windows, вивчення та аналіз проблемної області розробки ігрових рушіїв, вивчення різноманітних видів та архітектурних підходів до розробки програмних застосунків типу «ігровий рушій».

Практична цінність отриманих в роботі результатів полягає в тому, що створений ігровий рушій дозволяє використовувати низькорівневий функціонал для досягнення більш якісного рівня відображення графіки та пришвидшення проведення операцій типу «скалярний примітив».

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

У цьому розділі проводиться аналіз предметної області, опис та визначення завдання на дипломну роботу, аналіз та порівняння існуючих програмних рішень

1.1 Аналіз завдання

Основним технічним завданням є створення ігрового рушія для операційної системи Windows. Цей процес поділяється на велику кількість під-процесів, таких як: вивчення та аналіз процесу розробки ігрових рушіїв, ознайомлення з архітектурними особливостями побудови програмного забезпечення типу «ігровий рушій», ознайомлення та імплементація функцій для взаємодії користувача з програмним рушієм та системою. Останній пункт є найбільш цікавим, тому що ігровий рушій не використовується як загальноприйнятий API, чи як бібліотека набору функцій. Цей тип програмного забезпечення повинен функціонувати паралельно розробленому ігровому застосунку та виступати у ролі своєрідної програмної «прокладки» для фільтрації та імплементації запитів між високорівневим застосунком типу «комп'ютерні гра» та системою користувача.

Процес вивчення предметної області включає у себе: вивчення та аналіз предметної області розробки ігрових рушіїв та програмних засобів для рендерингу, аналіз існуючих рішень, аналіз сторонніх критеріїв, вивчення та огляд літератури за тематикою і створення ігрового рушія використовуючи новітні стандарти та технології.

З технічної точки зору, процес розробки ігрового рушія поділяється на розробку його систем, бо ігровий рушій – комплексне програмне забезпечення, яке складається з багатьох маленьких підсистем. Під час написання роботи були створені наступні елементи програмного забезпечення: графічний рушій, система отримання користувацького вводу,

система дескрипторів, програмний інструментарій, сімейство шейдерів, система рендерного «пайплайну», система «графічних ланцюгів», система-буфер та багато більш високорівневих частин програмного забезпечення. Детальний опис разом з прикладом імплементації наведено у розділі 3.

1.2 Аналіз проблеми

Предметна область поділяється на розробку ігрового рушія та на розробку гри. Ці два пункти дуже тісно пов'язані один з одним. Область включає в себе комплекс технологій та знань, що використовуються для створення комп'ютерних ігор. Це включає різні аспекти, такі як: проектування геймплею, розробка інтерфейсу користувача, створення візуальних та звукових ефектів, реалізацію штучного інтелекту та механік гри, а також оптимізацію програмного забезпечення для різних платформ. Основою розробки будь-яких ігор є ігрові рушії.

Ігровий рушій - програмне забезпечення, що використовується розробниками для створення комп'ютерних ігор. Програма цього виду надає користувачам доступ до різноманітних інструментів і функцій, необхідних для створення ігрового застосунку. Таких як: інструментарій для взаємодії з графічними компонентами системи, компоненти фізики, штучний інтелект, звукову обробку, взаємодію з гравцем та дуже багато інших компонентів, що допоможуть створити цікаву гру [1].

У нинішній час сфера розробки ігрових застосунків потребує багато інноваційних рішень від розробників ігрових рушіїв, тому що апетити гравців ростуть. Саме через це розробники ігор потребують все новіших та новіших рішень. У нинішній час, комерційна сфера розробки ігрових рушіїв належить монополістам з їхніми програмними продуктами: Unreal Engine 4, Unreal Engine 5, Unity. Порівняння зазначених раніше ігрових рушіїв наведено у підпункті 1.4. Через складність використання деяких з зазначених вище рушіїв та їхні недоліки, було прийняте рішення розробити свій

особистий ігровий рушій, що був би позбавлений недоліків програмних продуктів зазначених раніше.

Потрібно зазначити, що розробка ігрових рушіїв є складним та затратним процесом, що вимагає від команди розробників великої кількості навичок та знань з різноманітних наукових сфер. Процес розробки ігрового рушія, зазвичай, розпочинається з визначення вимог до функціональності та продуктивності фінального продукту. Після цього розробники створюють основну архітектуру рушія, включаючи систему керування ресурсами, систему відтворення звуку та графіки, систему фізики та багато іншого.

Це настільки складний та затратний процес, що деякі студії AAA рівня все частіше та частіше вирішують використовувати готові програмні рішення замість створення особистих ігрових рушіїв. Наприклад, Offworld Industries є студією розробки ігор, що була заснована у Канаді. Найбільш відомим та успішним продуктом цієї студії є комп'ютерна гра під назвою «Squad», що з англійської означає «загін» [2]. Це приближена до реальності військово-тактична гра від першої особи, що була розроблена з використанням ігрового рушія Unreal Engine 4. Розробники вирішили, що краще використовувати вже існуючі рішення на ринку для створення свого застосунку типу «гра», тому що написання нового ігрового рушія могло б бути банально не вигідним з точки зору затраченого часу та фінансів

1.3 Критерії оцінки існуючих рішень

У базовому вигляді, оцінка ігрових рушіїв може проводитися за декількома критеріями, такими як: продуктивність, функціональність, надійність, розширюваність, зручність використання та підтримка платформами, проте найбільш гарним рішенням буде розширити список критеріїв та оцінити малі деталі.

Під час оцінювання готових існуючих ігрових рушіїв, автором пояснювальної записки пропонується використовувати наступні критерії для проведення оцінки ігрових рушіїв.

Першим критерієм є мова програмування. Саме вона дозволяє користувачеві звертатися до ігрового рушія та від її швидкодії напряду залежить оптимізація майбутнього програмного продукту.

Пропонується взяти до уваги наявність системи візуального скриптингу, бо наявність системи такого рівня дає можливість користувачеві працювати з рушієм без знань мови програмування.

Наявність підтримка мультипоточного рендеру. Цей критерій відповідальний за індикацію наявності підтримки процесу мультипоточного рендеру. Наявність чи відсутність цього явища істотно відображається на рівні продуктивності ігрового рушія, коли мова стосується обробки графіки та графічних елементів.

Також пропонується взяти до уваги наявність підтримки векторизації та векторного простору, бо цей пункт відповідальний за наявність можливості використання особливої архітектурної компоновки, що складається з векторних реєстрів та особливого шляху взаємодії з ними. Цей підхід істотно підвищує швидкодію програмного забезпечення.

До критерію вище додаються набори підтримуваних векторних інструкцій. Цей критерій демонструє можливості ігрового рушія під час оптимізації гри зробленої на базі цього самого програмного забезпечення.

Підтримка інструкцій розпаралелювання операцій – критерій відповідальний за індикацію наявності підтримки інструкцій розпаралелювання операцій.

Підтримка методів вирівнювання даних у векторному просторі, що є стороннім методом оптимізації для вирішення непередбачуваних проблем, що можуть статися під час автоматичної оптимізації застосунку.

1.4 Аналіз існуючих рішень

Unreal Engine 4 - це один з найпопулярніших ігрових рушіїв, створений компанією Epic Games. Він побачив світ у 2014 році, та має відкритий вихідний код, що дає можливість розробникам з усього світу модифікувати та пристосовувати рушій до своїх особливих потреб. UE4 підтримує багато платформ: персональні комп'ютери, консолі, мобільні пристрої, VR-гарнітури, тощо [3].

Рушій має широкий набір інструментів, що дозволяють розробникам створювати ігри більш високої якості у плані графіки та фізики. Нижче наведені основні компоненти та системи цього ігрового рушія:

Unreal Editor - інтегроване середовище розробки, що дозволяє розробникам створювати ігри та інтерактивні додатки. Ця частина програми має графічний інтерфейс для користувача, та ряд інструментів, які дозволяють розробникам створювати графічні ефекти, програмувати поведінку акторів, розміщувати об'єкти у віртуальному світі, тощо [3].

Blueprint Visual Scripting - візуальна система програмування, що дозволяє розробникам створювати логіку гри без знань та навичок програмування. За допомогою цієї системи можна створювати складну логіку, що охоплює дії гравців, фізичні об'єкти, стани, тощо [3].

Material Editor - це інструмент для створення та модифікації матеріалів, що використовуються для створення текстур та графічних ефектів. За допомогою цієї системи, можна створювати матеріали, які реалістично відображають різні поверхні, такі як: камінь, дерево, метал, скло, тощо [3].

Animation Editor - система для створення анімацій персонажів та ігрових об'єктів. Animation Editor дозволяє розробникам створювати різноманітні анімації, такі як: рух, повороти, стрибок, присід, біг, тощо.

Audio Editor - інструментарій для створення звукового супроводу гри. За допомогою Audio Editor можна створювати різноманітні звукові ефекти. Починаючи з музики, та закінчуючи локальними звуками.

Visual Effects - інструмент для створення візуальних ефектів, що дає можливість створювати різноманітні ефекти, такі як: ефект пострілу, ефект зникнення, ефект виникнення, тощо [3].

Перевагами Unreal Engine 4 є великий набір функцій. Рушій має багато інструментів, які дозволяють розробникам створювати різні елементи гри, включаючи: графіку, фізику, анімацію, звук, та багато іншого. Також Unreal Engine 4 має систему візуального скриптингу, що допомагає людям далеким від теми програмування працювати з компонентами рушія. Програмне забезпечення має дуже гнучкий функціонал, саме через це з його допомогою розробники створюють ігри у різних жанрах для різноманітних платформ.

Проте, Unreal Engine 4 має і проблемні сторони. Висока складність роботи з рушієм підіймає прохідну планку та відсіює новачків у сфері розробки ігор. Програмне забезпечення потребує багато системних ресурсів для нормальної роботи. Перші версії Unreal Engine 4 мали проблеми з оптимізацією, якщо точніше, то з системою візуального скриптингу: вона краще працювала на процесорах від вендора «Intel» ніж від AMD, незважаючи на суттєву перевагу останніх.

Unreal Engine 5 - це остання версія ігрового рушія Unreal Engine, який був анонсований Epic Games в травні 2020 року і випущений у травні 2021 року. Unreal Engine 5 заявлений як нове слово в гральній індустрії, яке дозволить розробникам створювати ще більш деталізовані та реалістичні ігри з меншими зусиллями. Ця версія відрізняється від попередньої більш якісним та розширеним функціоналом [3].

Ось деякі з нововведень у наступному поколінні рушія:

Nanite - технологія геометрії в реальному часі, яка дозволяє розробникам створювати детальні та складні 3D-моделі без обмежень у

розмірі. Додає можливість поліпшити якість об'єкта за допомогою нарощування кількості віртуальних полігонів.

Lumen - це система глобального освітлення в реальному часі. Вона дозволяє створювати динамічне освітлення, що реагує на кожний елемент ігрового середовища. Ця технологія забезпечує відображення натуральних тіней, реальне освітлення та ефекти, які раніше були неможливі в ігрових світах без використання RT та CUDA ядер [3].

Покращений інструментарій - розробники додали дуже багато нововведень. Blueprint має нові функції, які роблять програмування більш зручним та ефективним. Була додана підтримка автоматного форматування, здатність копіювати та вставляти блоки коду, а також більш зручна навігація.

Geometry Collection Editor, що дозволяє розробникам легко створювати та редагувати складні геометричні об'єкти.

Unreal Engine 5 дозволяє розробникам створювати більш складні та реалістичні ефекти, включаючи генерацію частинок та флюїдів в реальному часі.

Unreal Engine 5 є новим кроком у розвитку рушіїв типу «Unreal». Розробники суттєво покращили програмне забезпечення та ярко підкреслили основні переваги нового покоління. Unreal Engine 5 має передовий графічний двигун, що дає змогу користувачам створювати неймовірно реалістичну графіку не прикладаючи багато зусиль. Розробники покращили та розширили функціонал рушію, тим самим пришвидшив розробку ігор, бо нові інструменти стали набагато простішими у використанні та вивченні. Введення нових технологій «Lumen» та «Nanite» зробили бурну революцію у світі обчислення параметрів об'єктів на рівні. Використання нових низькорівневих машинних інструкцій та більш специфічного C++ коду, дало можливість розробникам рушія мінімізувати різницю у продуктивності між різними вендорами комп'ютерних компонентів. Наприклад, у Unreal Engine 4 були проблеми пов'язані з оптимізацією ігор під процесори марки AMD: центральний процесор виробництва Intel мав значну перевагу у вихідній

кількості кадрів у порівнянні з процесором виробництва AMD. У новій версії Unreal Engine ця проблема була анульована за допомогою використання більш специфічного функціоналу.

Але і Unreal Engine 5 не є доскональним та має свої недоліки, а саме проблеми зі стабільністю. Через те, що ігровий рушій є новим, у ньому міститься багато багів та проблем, які розробники обіцяють виправити у найближчому майбутньому. Незважаючи на полегшення роботи з інструментарієм рушія, Unreal Engine 5 став ще більш комплексним продуктом, тому рівень входження значно підвищився у порівнянні з попередньою версією. Через додавання нових систем та покращення рівня програмного забезпечення, вимоги до системного обладнання збільшилися: як до девайсу гравця, так і до девайсу розробника.

Unity - це популярний ігровий рушій, який використовують для створення ігор та інтерактивних додатків на різних платформах, таких як ПК, консолі, мобільні пристрої та віртуальна реальність.

Рушій розроблюється компанією Unity Technologies. Однією з особливостей Unity є система компонентів та сцен, яка дозволяє створювати складні ігрові об'єкти шляхом додавання та налаштування різноманітних компонентів ігрового світу. Це дає більшу гнучкість та додає можливість повторного використання коду, що спрощує розробку та підтримку ігрового проекту.

Unity подібний до Unreal Engine, але у них, звісно, є різноманітності. Основною мовою програмування для взаємодії з Unity є C#. Цей фактор впливає на швидкодію компонентів ігрового рушія та ігрового світу. Наприклад, Unreal Engine використовує C++, який істотно швидший та гнучкіший за C# [4].

Перевагами Unity є низький поріг входження. Цей ігровий рушій по праву вважається одним з найлегших у вивченні. Unity Technologies створили інтерактивну та повністю зрозумілу документацію. Цей параметр дає можливість швидко вивчити основний функціонал ігрового рушія та перейти

до розробки своєї особистої гри. широка підтримка різноманітних платформ та девайсів. Завдяки кросплатформеності Unity, розробники можуть створювати один код та застосовувати його на різних платформах. Це зменшує час та зусилля необхідні для розробки гри під різноманітні операційні системи та компоненти.

Основними недоліками Unity є швидкодія. У порівнянні з Unreal Engine, швидкодія рушія є значно нижчою, тому що Unity використовує C#, а не C++. Відсутність деяких функцій, що у інших рушіях вважаються базовими. Unity може виявитись менш гнучким у деяких аспектах, наприклад: у обробці фізики, обробці графіки, обробці звуку, обробці декалей, та інших малих декалей. Unity використовує старий принцип обробки графічних елементів, що суттєво знижує якість зображення та збільшує час потрібний на обробку елементу світу.

Godot Engine - це безкоштовний, відкритий ігровий рушій з власним інтегрованим редактором, що використовується для створення 2D та 3D ігор, а також інших інтерактивних додатків на різних платформах. Godot був створений у 2014 році двома розробниками з метою написання безкоштовного ігрового рушія з відкритим кодом, що був би доступний для використання всіма бажаючими.

Godot має власну скриптову мову – GDScript. За синтаксисом вона є схожою на Python. Також ця скриптова мова програмування підтримує використання C#, C++ та Visual Scripting.

Основною перевагою Godot Engine є ліцензія за якою він розповсюджується. Вона пропонує відкритий код та безкоштовне використання. Тобто, Godot Engine є повністю безкоштовним для використання у комерційних цілях ігровим рушієм. Програмне забезпечення має малий розмір, що робить його ідеальним для розробки мобільних додатків та вебігор. Також, слід зазначити, що Godot Engine містить у собі дуже багато реалізацій стандартних функцій для створення компонентів у ігровому світі.

Основними мінусами Godot Engine є обмежені можливості редагування візуальних ефектів. Рушій має обмежені можливості візуального редагування ефектів порівняно з Unreal Engine або Unity. Godot Engine має проблеми зі швидкодією – він працює повільніше на більш старих пристроях, або компонентах, порівняно з Unreal Engine або Unity. Також мінусом є мала команда розробників. Ігровий рушій створювали лише дві людини, а після написання основного коду, було вирішено створити публічний репозиторій на платформі GitHub. Таким чином, можна сказати, що ігровий рушій після релізу розроблювала величезна кількість розробників без усякої координації, чи домовленості. Також питання викликає кваліфікація деяких з розробників, бо вони не проходять відбір.

Виходячи з даних та інформації вище, створимо порівняльну таблицю для більш детального аналізу пунктів та критеріїв (табл. 1.4).

Таблиця 1.4 – Порівняння наявних рішень

	Unreal Engine 4	Unreal Engine 5	Unity	Godot Engine
Графічний рушій	Unreal Engine 4 Rendering Architecture	Unreal Engine 5 renderer	Unity Engine	Godot Engine rendering engine
Основна мова програмуванні	C++	C++	C#	C++
Наявність візуального скриптингу	Так	Так	Так	Так
Підтримка мультипоточного рендеру	Так	Так	Так, але з версії 5.0	Так, але з версії 3.2
Наявність векторизації[5]	Так	Так	Так	Частково

Продовження таблиці 1.4

1	2	3	4	5
Набори векторних інструкцій[5]	SSE, SSE-2, SSE-3, SSE-4, AVX, AVX-2	SSE, SSE-2, SSE-3, SSE-4, AVX, AVX-2, AVX-512	SSE, AVX, AVX-2	SSE-2, SSE-3
Підтримка інструкцій розпаралелювання операцій[5, 6]	Ні	Так	Ні	Так
Підтримка методів вирівнювання даних у векторному просторі[5]	Регуляризація патерну векторизації, модифікація піл-лупів, метод стріп-майнингу	Регуляризація патерну векторизації, модифікація піл-лупів, метод стріп-майнингу	Регуляризація патерну векторизації	Відсутні

З інформації зазначеної у таблиці виходить, що Unreal Engine 5 є найкращим вибором для розробки комп'ютерних та не тільки ігор. Цей рушій підтримує та реалізовує дуже багато оптимізаційних моментів, таких як: розпаралелювання інформації, векторизація, векторний простір, мультипоточний рендер, мультипоточна та асинхронна робота коду. Розробники настільки далеко зайшли у питанні оптимізації, що навіть оптимізували оптимізовані розрахунки, добавивши регуляризацію патерну векторизації, модифікацію піл-лупів та метод стріп-майнингу.

Проте, навіть цей рушій не позбавлений недоліків, таких як: проблеми зі стабільністю, багато багів, більша комплексність за Unreal Engine 4, підняття планки вимог як для гравця, так і для розробника гри.

1.5 Підсумок

Підсумовуючи, можна зазначити, що зараз індустрія розробки ігрових рушіїв дуже активна та розвивається з кожним роком, тому актуальність цієї роботи зумовлена швидким розвитком та популярністю індустрії. Багато компаній з усього світу активно працюють над створенням нових рушіїв та покращенням існуючих. З'являється все більше інструментів та технологій, які дозволяють розробникам створювати більш продуктивні ігрові рушії. Також на ринку з'являється багато нових компаній, що розробляють ігри, тим самим підвищуючи планку відповідності до існуючих програмних засобів. У нинішній час фаворитом є Unreal Engine 5, що є найбільш новітнім та досконалим рішенням у світі. Саме це програмне забезпечення задало нові стандарти розробки ігрових рушіїв у нинішній час. Проте, через різноманітні проблеми у описаних вище ігрових рушіях, автором дипломної роботи було вирішено створити свій особистий ігровий рушій. Детальний технологічний опис ігрового рушія автора дипломної роботи зроблений у наступному пункті 2 пояснювальної записки.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

Як було зазначено у попередній частині пояснювальної записки, у цьому розділі відбувається показ та опис технічних відомостей ігрового рушія та його складових: мови програмування, архітектури та інших елементів програмного забезпечення. Основною метою розділу є демонстрація технічних особливостей та обраних рішень стосовно процесу розробки програмного застосунку.

2.1 Опис технічних відомостей

Після проведеного аналізу поставленого завдання, було вирішено використовувати наступний перелік компонентів та інформаційних технологій для створення програмного застосунку «ігровий рушій»:

- мова програмування: C++;
- математичний рушій: GLM;
- бібліотека для взаємодії з операційною системою: GLFW;
- архітектурна модель ігрового рушія: компонентно-потокowa;
- сімейство операційних систем: Windows;
- графічний API: Vulkan SDK.

C++ - це строго типізована статична мова програмування загального призначення, що була розроблена у 1983 році як функціональне розширення мови C. C++ використовується для написання програмного забезпечення різних типів, включаючи операційні системи, драйвери, браузері, ігри тощо. Будучи високо продуктивною мовою програмування, C++ може безпосередньо керувати пам'яттю та виконувати низькорівневі операції, що робить її популярною для розробки програмного забезпечення, де швидкодія є найбільш пріоритетною задачею. Також мова програмування має широку підтримку бібліотек, що дозволяє швидко створювати програми різного призначення без написання своїх особистих програмних рішень. Також C++

підтримує різноманітні рівні зв'язування: динамічне та статичне зв'язування [6, 7].

C++ має широку підтримку різноманітних парадигм програмування, таких як:

Об'єктно-орієнтована парадигма (ООП) – це стиль програмування, який зосереджується на об'єктах та сутностях, що мають певні властивості та реалізують якусь поведінку. C++ має вбудовану підтримку об'єктно-орієнтованого програмування, включаючи: класи, наслідування, поліморфізм, інкапсуляцію та абстракцію [6, 7].

Процедурна парадигма - парадигма програмування, що зосереджується на написанні процедур. Це традиційний стиль програмування, який використовувався на ранніх етапах розвитку індустрії. Найбільш яскравим представником процедурних мов програмування є Асемблер [7].

Функціональна парадигма - парадигма, що зосереджується на використанні функцій в якості основного засобу побудови програм. У C++ функціональне програмування реалізується за допомогою функцій вищих порядків, лямбда-виразів, функцій-об'єктів та інших конструкцій [7].

Шаблонне програмування - це стиль програмування, у якому програміст використовує шаблони коду для створення гнучких, та повторно використовуваних рішень. Шаблонне програмування забезпечує можливість створювати загальноприйняті функції та класи, які можуть бути використані в різних контекстах [7].

C++ здатний проводити скалярні обчислення над великою кількістю даних у чотири рази швидше за C#, у вісім разів швидше за Java та у сорок разів швидше за Python. Під час використання векторного простору та векторних інструкцій додавання, C++ здатний перевершити свій скалярний результат у п'ятнадцять разів, результат C# у шістдесят шість разів, результат Java у сто тридцять три рази та час Python у шістсот шістдесят шість разів. Саме швидкість C++ стала основним фактором під час вибору мови програмування для написання ігрового рушія, бо майже усі взаємодії з

графічним рушієм потребують великої швидкості під час скалярних обчислень, а інколи навіть потребують переводу у векторний простір [5 - 8].

GLM – бібліотека заголовків для використання з мовою програмування C++. Ця бібліотека містить математичні функції та структури даних необхідні для роботи з OpenGL та іншими бібліотеками, що реалізують функції для роботи з графічним процесором девайсу. GLM надає реалізацію багатьох математичних об'єктів та операцій, таких як вектори, матриці, проекції та трансформації. Під час взаємодії з зазначеною вище бібліотекою, програміст має змогу використовувати більш зручний і зрозумілий синтаксис для виконання математичних операцій у процесі програмування графіки. Також слід зазначити, що GLM є бібліотекою з відкритим кодом, та вона є легко доступною для використання в програмах на C++. GLM не має залежностей від інших бібліотек. Багато програмістів використовують GLM разом з OpenGL для створення ігор та візуалізації графічних даних [9].

GLM має широкий набір інструментів. Особливо можна виділити функції для роботи з векторами. Бібліотека містить дуже багато функцій для обробки операцій над векторами, наприклад: додавання, віднімання, множення, ділення тощо. Крім того, вона містить функції для визначення довжин векторів, функціонал для нормалізації вектора, визначення скалярного добутку двох векторів, та інше. Також GLM має функції для роботи з матрицями, вона має функції для проведення базисних математичних операцій над матрицями, таких як: додавання, віднімання та множення матриць. Крім того, GLM містить функціонал та структуру даних для створення матриць трансформації та функціонал для роботи з кутами - функції для конвертації кутів між різними системами вимірювання, а також інструментарій для проведення обчислення тригонометричних функцій. Слід зазначити, що GLM також реалізовує і функціонал для роботи з проекціями. Бібліотека містить інструментарій для створення матриць проекції у форматі перспективної та ортогоальної матриць. GLM має широкий набір функцій для роботи з кольорами. Бібліотека має функціонал для конвертації кольорів між

різними форматами, наприклад RGB та HSV. Також бібліотека містить функції для створення та зміни кольорів, функції математичної обробки кольорів, тощо [9].

Вище зазначені функції дозволяють перетворювати інформацію відповідно до потреб програми, та отримувати значення, що необхідні для обчислювання графічних об'єктів. Вектори можуть бути використані для представлення координат точок у тривимірному та двовимірному просторі, матриці для опису трансформацій об'єктів, а функції тригонометрії для обчислення позиції точок на екрані. Слід зазначити, що GLM дозволяє виконувати математичні операції з високою точністю та швидкістю, що є дуже важливим для ефективної роботи з графікою, пікселями, вертикалями та іншими графічними елементами програмного забезпечення. Бібліотека також дозволяє використовувати оптимізований код для різних платформ, що дозволяє розробникам легко переносити свої програми на різні операційні системи та апаратні платформи.

Узагальнюючи, GLM є потужним інструментом для виконання математичних операцій. Бібліотека дозволяє розробникам зосередитися на реалізації функціональності програми, не витрачаючи багато часу на вирішення математичних завдань та оптимізацію роботи з графікою.

GLFW - бібліотека, яка дозволяє розробникам легко створювати віконні додатки та інтерфейси для роботи з OpenGL на різних платформах. Бібліотека була створена у 2002 році, та причиною для створення було масштабне незадоволення розробників тогочасними варіантами бібліотек для роботи з графікою. GLFW була створена як проста та легка бібліотека для роботи з OpenGL. Основною метою, на момент створення, було бажання допомогти розробникам створювати віконні додатки та інтерфейси з максимальною легкістю для різних платформ та операційних систем [9].

Команда розробників з Університету Кембріджу, включаючи Кемерона Спейтса, головного розробника бібліотеки на протязі декількох років, створила GLFW. Ця бібліотека є простим та легким у використанні

інтерфейсом створення вікон та контекстів для роботи з OpenGL. У нинішній час бібліотека постійно розвивається та має активну спільноту розробників. Програмне забезпечення підтримує різноманітні функції, такі як: створення вікон та контекстів для OpenGL, обробка користувацького вводу, обробка подій, обробка помилок та звітів про стан системи, тощо. GLFW також підтримує сучасні функції OpenGL, включаючи мультисемплінг, вбудовані шейдери та текстури високої роздільності для Retina-дисплеїв, а також OpenGL ES для розробки мобільних додатків. Крім того, бібліотека має декілька додаткових функцій. Наприклад, підтримка виводу вікон на різних моніторах, підтримка створення курсорів та їх відображення на екрані, можливість установки обмежень на розмір вікна та його мінімальну та максимальну ширину та висоту [9].

Програмний засіб надає програмісту підтримку багатьох платформ, що дозволяє створювати додатки для використання на різних платформах без потреби у створенні нового коду. GLFW є вільним для розповсюдження програмним забезпеченням з відкритим кодом, що дозволяє використовувати його для комерційних проектів безкоштовно. Однією з істотних переваг бібліотеки є те, що вона є легкою у використанні та налаштуванні. Її API було розроблене з орієнтацією на простоту та легкість використання. GLFW також має низку прикладів, які допомагають початківцям швидко зрозуміти суть програмного забезпечення та швидко навчитися працювати з бібліотекою. Крім того, GLFW підтримує різні програмні застосунки для роботи з графікою, такі як: OpenGL, OpenGL ES та Vulkan [9].

Загалом, GLFW є потужною та ефективною бібліотекою для створення вікон та контекстів, що є особливо корисною для розробників, які хочуть створювати додатки з підтримкою різноманітних платформ та взаємодію з високоякісною обробкою графіки та вводу.

Vulkan SDK - це набір інструментів для розробки програм з використанням графічної бібліотеки Vulkan. Він був створений компанією Khronos Group в 2016 році та призначений для створення

високопродуктивних 3D-додатків та ігор для різних платформ та операційних систем. Vulkan SDK працює на багатьох архітектурах та операційних системах, включаючи Windows, Linux, MacOS та Android. Цей набір інструментів підтримує багато різних функцій. Крім того, він містить різноманітні інструменти для аналізу та оптимізації продуктивності додатків, що розробляються на його основі. Vulkan SDK навіть має функціонал для проведення неявної динамічної оптимізації під час ран-тайму застосунку, що дуже позитивно сказується на швидкодії програми, тому що він реалізовує не тільки методи скалярної оптимізації, а ще й різноманітні методи векторної та паралельної оптимізації, що дає змогу виконувати більше роботи за менший проміжок часу чи за меншу кількість циклів центрального процесора [10].

Vulkan SDK підтримує різноманітні архітектури процесорів, включаючи: Intel, AMD, NVIDIA, Qualcomm та ARM. Програмне забезпечення складається з декількох компонентів. Основними з них є: бібліотеки, драйвери, заголовочні файли та документація. Бібліотеки містять реалізацію функціоналу Vulkan API, а драйвери забезпечують взаємодію між програмою та апаратним забезпеченням девайсу користувача [10].

Vulkan заснований та працює на базі об'єктної моделі, де кожен об'єкт має різні аспекти та імплементує різний функціонал для взаємодії з графічним компонентом девайсу. Під час розробки програмного забезпечення з використанням Vulkan, програмістам потрібно створювати об'єкти та налаштовувати їх для досягнення зрозумілого результату. Всі об'єкти у системі є незалежними один від одного, тому користувачі можуть керувати кожним елементом окремо, та управляти ресурсами ефективніше ніж у тому самому OpenGL [10].

Однією з основних відмінностей між Vulkan та OpenGL є те, що Vulkan є низькорівневим API, тоді як OpenGL є високорівневим. Це означає, що розробка на Vulkan є більш складною, але вона забезпечує більш високу продуктивність ніж розробка на OpenGL. Крім того, Vulkan є орієнтованим на потоки API, тоді як OpenGL не підтримує мультипотоківість. Однак,

завдяки більш прямому доступу до апаратних ресурсів, Vulkan має більший рівень доступу до апаратних ресурсів, ніж OpenGL [10, 11].

Для оцінювання та порівняння двох програмних засобів, пропонується використати наступні критерії створені автором дипломної роботи: орієнтованість на потоки, тип API, показник швидкодії, рівень доступу до апаратних ресурсів, рівень використання пам'яті пристрою, сумісність з операційними системами, переносимість без портування, наявність підтримки мобільних пристроїв, підтримка використання мультипоточного та мультіядерного програмування, кількість доступних функцій, підтримка старих операційних систем, підтримка асинхронної обробки інформації, підтримка геометричних шейдерів, підтримувані мови програмування для взаємодії з програмним застосунком, підтримка мультиплатформенної обробки графічної інформації, графічні можливості, ресурсозалежність, наявність візуалізатора профілювання, наявність інспектора шейдерів, тип підтримуваної мови програмування шейдерів та підтримувані системи контролю версій [11]. Нижче наведена таблиця порівняння між двома технологіями (табл. 2.1)

Таблиця 2.1 – Порівняння Vulkan та OpenGL

	Vulkan	OpenGL
Орієнтованість на потоки	Так	Ні
Тип API	Низькорівневний	Використовуваний
Показник швидкодії	Високий	Залежить від імплементації
Рівень доступу до апаратних ресурсів	Прямий	Обмежений
Сумісність з операційними системами	Мультиплатформений	Мультиплатформений

Продовження таблиці 2.1

1	2	3
Переносимість без портування	Так	Так
Підтримка мобільних пристроїв	Так	Так
Підтримка використання мультитотоку	Так	Ні
Кількість доступних функцій	Велика	Менша за Vulkan API
Підтримка старих операційних систем	Ні	Так
Підтримка асинхронної обробки інформації	Так	Ні
Підтримка геометричних шейдерів	Так	Ні
Підтримувані мови програмування	C, C++, Java, Rust, Python, C#.	C++, C, Python
Підтримка мультиплатформенної обробки графічної інформації	Так	Ні
Графічні можливості	Можливість працювати з шейдерами та налаштуваннями графіки на низькому рівні	Високорівневі можливості, відсутні налаштування на низькому рівні

Продовження таблиці 2.1

1	2	3
Ресурсозалежність	Вимагає від розробника власноручного управління ресурсами	Автоматичний розподіл та управління
Інспектор шейдерів	Vulkan Shader Editor, FXC	Shader Builder, glslangValidator
Мова шейдерів	GLSL, SPIR-V	GLSL
Контроль версій	Git, SVN	Git, SVN, CVS

Виходячи з інформації зазначеною у таблиці, можна зробити висновок, що Vulkan API є дуже тонким та потужним інструментом взаємодії з комп'ютерною графікою, що має підтримку низькорівневих функцій та може працювати у багатьох режимах та з великою кількістю ядер девайсу.

2.2 Опис архітектури програмного застосунку

Основним архітектурним шаблоном було обрано компонентно-потоківу архітектуру [12].

Вона є однією з найсучасніших, де рушій складається з різних компонентів, кожен з яких відповідає за певний аспект роботи рушія. Кожен компонент може мати свою власну архітектуру та бути розробленим окремо від інших компонентів. Компонентна архітектура забезпечує високий рівень розширюваності та підтримки. Наприклад, якщо розробники хочуть додати нову функціональність, вони можуть розробити новий компонент та просто додати його до існуючого рушія. Це дає змогу зберігати основний функціонал рушія незмінним, тим самим зменшуючи ризик розробки нової функціональності. Крім того, компонентна архітектура забезпечує певний рівень стандартизації, оскільки кожен компонент має свій власний інтерфейс та специфікації. У компонентній архітектурі кожен компонент може бути

розроблений з використанням будь-якої архітектури, яка найкраще підходить для відповідної функціональності. Наприклад, для компоненту, який відповідає за графіку, може бути використана архітектура типу «А», а для компоненту, який відповідає за звук, може бути використана архітектура типу «Б» [12].

Зазвичай архітектура ігрового рушія виглядає наступним чином (рисунок 2.1) та поділяється на об'єкти типу «менеджер» та «рушій».

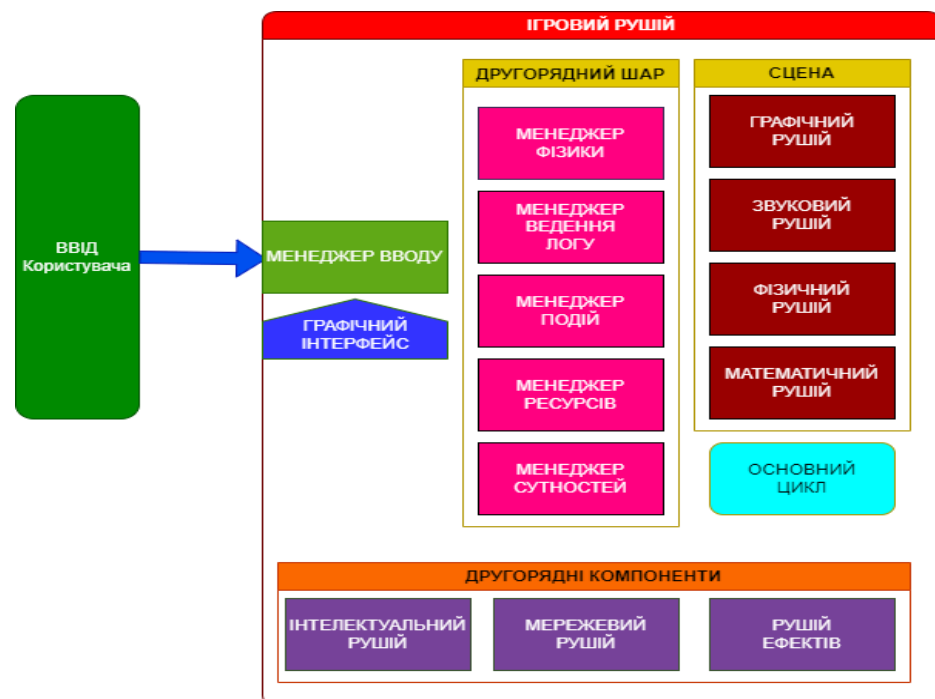


Рисунок 2.1. – Схема стандартної узагальненої компоновки [17]

Архітектура складається з чотирьох шарів: сцена, другорядний шар, другорядні компоненти та шар основного циклу ігрового рушія. Під час розробки ігрового рушія було вирішено відмовитись від подібного компонування та використати компонентно-потоківий підхід.

На рисунку 2.2 зображено систему виводу графічної інформації користувачу при використанні компонентно-потоківий системи

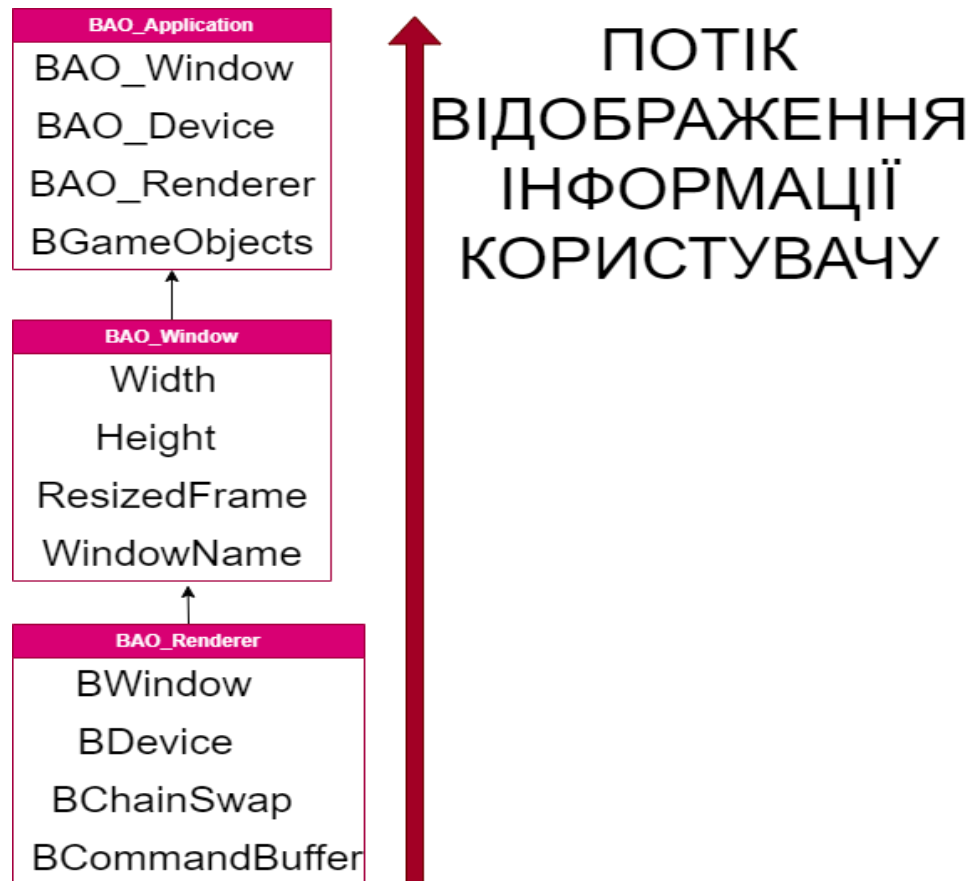


Рисунок 2.2 – Система виводу графічної інформації користувача [17]

На цій схемі видно компоненти системи: BAO_Renderer, BAO_Window, BAO_Application у ієрархічному порядку від найближчого до програмного функціоналу до найближчого до користувацького функціоналу. Таким чином програма поділяється на шари не тільки по призначенню, але і за рівнем глибини.

BAO_Renderer є представником найнижчого шару, що забезпечує обробку графічної інформації з використанням графічного процесору девайсу та пов'язаний трансферним потоком з BAO_Window, який у свою чергу є представником «вищого шару», тобто частини ігрового рушія, що не працює з програмним функціоналом напряду, а тільки відображає інформацію. З точки зору архітектури, система має наступний вигляд (рисунок 2.3).

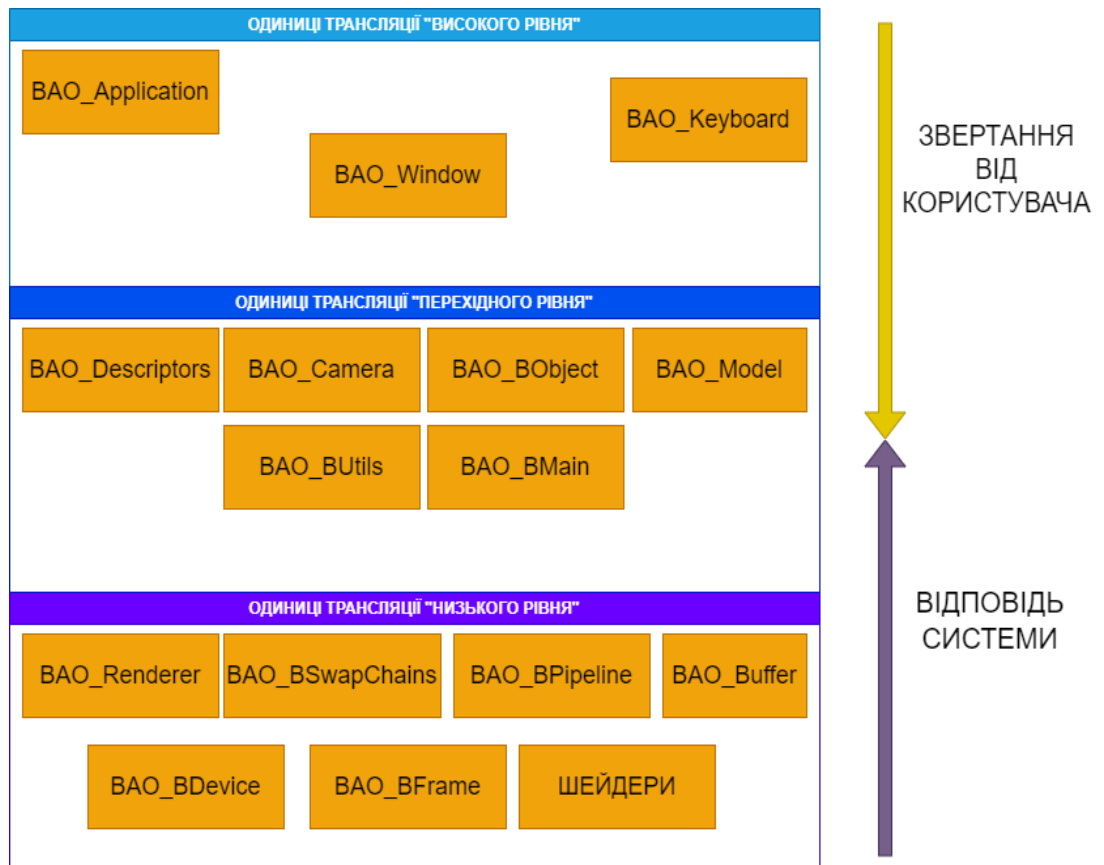


Рисунок 2.3 – Розподілення на шари [17]

Одиниці трансляції високого рівня – функціонал, що забезпечує взаємодію з користувачем за допомогою різноманітних графічних інструментів, наприклад: графічний елемент «кнопка», графічний елемент «текстове поле», тощо. Є стартовою точкою при зверненні користувача до системи [12, 13].

Одиниці трансляції перехідного рівня – функціонал, що не має безпосереднього доступу до функціоналу низького рівня та у той же час не виступає безпосереднім зв'язуючим елементом між користувачем та функціоналом системи [12, 13].

Одиниці трансляції низького рівня – безпосередній функціонал та «серце» програмного застосунку. На цьому рівні відбувається звертання до компонентів девайсу, до операційної системи та до різноманітних утворюючих компонентів [13].

Усі звертання користувача йдуть від найвищого рівня до найнижчого, а відповідь системи йде від найнижчого рівня. Тобто, у системи з такою компоновкою не може бути звертань до одиниць трансляції перехідного рівня. Потрібно зазначити, що звертання користувача зазвичай «покладено на потік» тобто воно оброблюється незалежно від стану системи. Наприклад, користувач бажає одночасно завантажувати якийсь векторний примітив та переглядати свої об'єкти на сцені. Компонентно-потоківий підхід розділяє цю процедуру на два незалежних, але захищених потоки.

«Потік А» створюється та запускається під час звернення до компоненту завантажування векторного примітиву.

«Потік Б» відображає сцену з об'єктами.

Таким чином у компонентно-потоківому архітектурному стилі реалізовано використання мультитядерної розробки, що допомагає значно пришвидшувати швидкість роботи та взаємодії компонентів ігрового рушія.

2.3 Висновки за розділом

Виходячи з отриманої та зазначеної вище інформації, робиться висновок, що ігровий рушій буде мати приближене до середнього значення між абстракцією та залежністю. Через використання компонентно-потоківого архітектурного підходу, у рушія буде можливість працювати над багатьма завданнями не втрачаючи рівень швидкості.

Готовий математичний рушій та бібліотека для взаємодії з операційною системою, є найкращим вибором для пришвидшення процесу створення ігрового рушія. GLM має широкий функціонал, що знайде застосування під час роботи з векторною графікою та іншими елементами графічного рушія. GLFW дає можливість взаємодіяти з операційною системою не використовуючи API операційної системи, бо кожна функція цієї бібліотеки є своєрідною обгорткою для виклику функції системи.

Vulkan SDK є найліпшим вибором поміж усіх інших графічних API, бо завдяки витонченій компоновці високорівневих та низькорівневих функцій розробники змогли досягти великої швидкості роботи застосунку та високого рівня якості вихідної інформації.

У наступному розділі будуть наведені: приклади програмної реалізації компонентів, аналіз деяких частин системи програмного забезпечення та результати розробки частин програми.

3 РОЗРОБКА ПРОГРАМИ

Як було зазначено у попередній частині пояснювальної записки, у цьому розділі відбувається опис, аналіз та наводяться приклади реалізації деяких з частин системи.

3.1 Реалізація BAO_BWindow

Операційна система Window побудована на терміні «віконна система».

Віконна система - це складова частина операційної системи, яка керує відображенням вікон та взаємодіє з користувачем. Вона надає засоби для створення, керування та управління вікнами, а також обробки подій, пов'язаних з вікнами.

Для взаємодії з функціоналом операційної системи, а саме для створення вікон, була обрана бібліотека GLFW. Це кросплатформова бібліотека програмування для створення віконних інтерфейсів, обробки подій вікна та управління контекстами OpenGL.

Вікно в GLFW представляє собою прямокутну область на екрані, де можна відображати графічні об'єкти, або взаємодіяти з користувачем. Вікно має свої особисті розміри, унікальний заголовок, та може мати різні системні налаштування. GLFW забезпечує можливість створювати контексти для взаємодії з OpenGL, які, у свою чергу, пов'язані з віконною системою. Контекст визначає область пам'яті, в якій відбувається малювання графіки і обробка графічних команд, а GLFW дозволяє створювати та керувати контекстом у вікнах [1]. Бібліотека надає можливість обробки різних подій пов'язаних з вікнами, такі як: переміщення, зміна розміру, закриття, натискання клавіші миші тощо. Розробник може зареєструвати обробники подій і реагувати на них залежно від потреб свого додатку. Також GLFW

надає набір функцій для базового керування вікном, наприклад: закриття, переміщення, зміна розміру, приховування та відображення, тощо [9].

Клас вікна у реаліях GLFW є звичайною обгорткою для виклику низькорівневого функціоналу операційної системи Windows. Робиться це з використанням Windows API [14].

Вхідними даними для конструктору BAO_BWindow є дві змінні типу шістнадцятибітне ціле число та масив символів, який представлений у формі вряпера «рядок». У першій змінній міститься число, що відповідає за ширину вікна, у другій змінній міститься інформація про висоту вікна. У свою чергу, рядок містить ім'я для майбутнього вікна [1].

Всього клас має вісім функцій: чотири функції типу «отримати», що повертають інформацію про різні компоненти системи, та дві функції колбеку та дві функції відповідальні за створення та ініціалізацію вікна.

Клас вікна повинен бути унікальним, тому під час розробки було вирішено видалити конструктор копіювання та оператор привласнення, щоб запобігти неочікуваних багів та проблем з роботою програмного забезпечення [14]. На рисунку 3.1 показано звичайне пусте вікно, що створюється за допомогою базового функціоналу. Воно не має у собі ніякого функціоналу, окрім примітивних можливостей налаштування стану: закрити, згорнути, розгорнути. На рисунку 3.2 зображено структуру класу BAO_BWindow.

Bendo B.I. Diplom

— □ ×

Рисунок 3.1 – Базове вікно.

BAO_BWindow
bool ShouldClose() VkExtent2D getExtent() bool resetResize() GLFWWindow* getWindow() bool wasResized()
void resizeBFrameBBuffer() void initBWindow() uint16_t height uint16_t width bool blsBFrameBBufferResized std::string bWindowName GLFWWindow* BWindow

Рисунок 3.2 – Структура класу BAO_BWindow [18]

Цей клас належить до компонентів, що знаходяться на найвищому рівні у всій ієрархії шарів застосування.

З інформації зазначеної вище витікає, що клас вікна є одним з найпростіших по складності, але одним із найголовніших елементів програмного застосування, тому що саме клас вікна дає змогу користувачеві візуально побачити реакцію застосування на його дії.

3.2 Реалізація класу BAO_VApp

Клас BAO_VApp є об'єктно-орієнтованою конструкцією [15], що використовується для створення і керування основним потоком виконання програми. Це високорівневий об'єкт, що містить у собі усю необхідну для запуску додатка функціональну частину. Ця структурна одиниця відповідає за керування життєвим циклом ігрового додатка. Вона містить методи для ініціалізації рушія, створення вікна, обробки подій, оновлення гри, малювання графіки, тощо. Це дозволяє контролювати порядок виконання різних етапів гри і забезпечувати правильну роботу додатка. До структурної одиниці можуть опціонально включати логіку для завантаження та управління ресурсами гри: текстурами, звуками, моделями, картами, тощо. Клас може виконувати завантаження ресурсів перед початком гри і звільняти

їх після закінчення. Це допомагає ефективно управляти ресурсами і запобігти витокам пам'яті. Проте, найкращим рішенням у цьому випадку було б створити зовнішню одиницю трансляції, що буде працювати у двох режимах: як система виділення пам'яті, та як система-прибиральник сміття [16]. Клас включає у собі логіку для керування станом гри. Він має змінні, що представляють поточний стан гри: рівень, стан персонажа, стан ворогів. Опціонально, він може також мати методи для зміни стану гри, для переходу між різними режимами або етапами гри.

BAO_VApp не має безпосередніх вхідних даних, але він забезпечує взаємодію користувача з різноманітними системами. Всього клас має дві функції: почати роботу, та завантажити існуючі ігрові об'єкти. Клас «Застосунок» тримає у собі копію вікна, копію класу для зчитування взаємодії користувача з елементами маніпуляції персональним комп'ютером, та містить копію на об'єкт рендеру. Також застосунок містить структуру даних типу «хеш-таблиця» для збереження ігрових об'єктів, та розумний покажчик типу «унікальний» на простір дескрипторів. На рисунку 3.3 наведено структуру класу BAO_VApp.

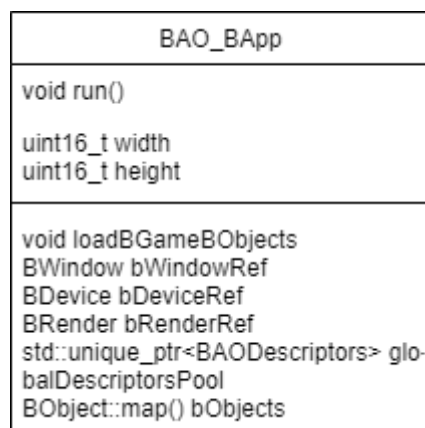


Рисунок 3.3 – Структура класу BAO_VApp [18]

Цей клас належить до компонентів, що знаходяться на найвищому рівні у всій ієрархії шарів застосунку.

З інформації зазначеної вище, витікає, що клас застосунку є фундаментом для запуску рішення типу «комп'ютерна програма» на девайсі користувача, тому що саме за допомогою цього класу користувач здатний взаємодіяти із застосунком та, інколи, навіть контролювати перебіг подій у логіці програмного застосунку.

3.3 Реалізація класу BAO_VKeyboard

BAO_VKeyboard вводу має функції для зчитування введення користувача з різних пристроїв, таких як: клавіатура, миша, геймпади, тощо. У світі ігрових рушіїв важливу роль мають так звані події [1]. Клас вводу має функції для обробки введених подій та зумовлений сценарій реагування на них. Наприклад, він перевіряє, чи була натиснута певна клавіша і у випадку якщо вона була натиснута, то він викликає відповідні функції, або методи гри. Це дозволяє відслідковувати дії користувача та виконувати відповідні дії у грі. Клас вводу слугує для керування рухом персонажа, взаємодії з об'єктами, виконання різних дій гравця в грі, тощо. Клас може включати методи для переміщення персонажа, взаємодії з предметами або виконання спеціальних дій [1].

Вхідними даними для класу є швидкість руху, показчик на вікно, змінну дельта-часу для своєчасного оновлення буферів вікна, та посилення на ігровий об'єкт. Також клас приймає ввід користувача та будує свою реакцію спираючись на порівняння вводу зі структурою, що містить різноманітні коди клавіш операційної системи, що були взяті базуючись на інформації від персонального комп'ютера користувача. На рисунку 3.4 наведено схему класу BAO_VKeyboard.

BAO_BKeyboard
<pre>struct BKeyMap { } void bMoveInPlaneXZ</pre>
<pre>BKeyMap bKeys float moveSpeed float lookSpeed</pre>

Рисунок 3.4 – Структура класу BAO_BKeyboard [18]

З архітектурної точки зору, цей клас належить до одиниць трансляції високого рівня.

З інформації вище виходить, що клас для обробки вводу користувача є одним з фундаментальних постулатів програмного забезпечення типу «ігровий рушій», та він допомагає розробникам гри управляти взаємодією гравця з програмним продуктом для забезпечення більшого рівня зручності і точності керування. Цей пункт відкриває можливості для реалізації різноманітних взаємодій та функціональності гри.

3.4 Реалізація класу BAO_BCamera

Клас камери відображає як буде видно гру з певної позиції та кута огляду, це дозволяє контролювати перегляд сцени з різних точок [1]. Клас BAO_BCamera має властивості, що визначають її позицію і орієнтацію. Це дозволяє розміщувати камеру у віртуальному просторі та спрямовувати її на потрібні об'єкти, або ділянки сцени. Камера включає параметри, що визначають тип проекції: перспективна, чи ортогональна проекція, та функціонал для налаштування проекційної матриці. Слід зазначити, що клас може опціонально включати методи для керування оглядом сцени, наприклад, для зміни позиції та орієнтації камери - це дозволяє моделювати рухи камери, панорамування, нахил, збільшення або зменшення масштабу, тощо. Проте, від цієї ідеї було вирішено відмовитись через те, що найкращим рішенням у такій ситуації буде створення компонента «пружина», чи «палка», яка з'єднує компонент камери з персонажем. Цей клас містить

методи для розрахунку матриці трансформації, а саме для розрахунку матриці трансформувannya та матриці проєкції. Ці математичні одиниці використовуються для перетворення координат об'єктів у систему координат камери з наступним перетворенням їх на 2D простір екрану. На рисунку 3.5 наведено схему класу BAO_VCamera

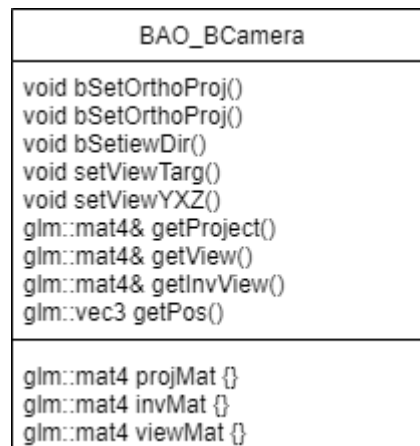


Рисунок 3.5 – Структура класу BAO_VCamera [18]

Вхідними даними для класу є: структури даних типу «тривимірний вектор», що містять у собі інформації про положення камери, положення гравця, різноманітні напрямки руху, та числа з плаваючою крапкою, що передають інформацію про: ліву та праву сторони, про верхній та нижній кордони, та число дистанції.

З архітектурної точки зору, цей клас належить до компонентів перехідного рівня, що не здатні приймати безпосередній користувацький ввід, проте можуть отримувати інформацію від інших компонентів та функцій системи. Для більш детального ознайомлення з візуальним поданням поняття «компонент перехідного рівня».

З інформації зазначеної вище випливає, що клас камери є важливою складовою ігрового рушія, оскільки саме він дозволяє реалістично відтворити перспективу та огляд у майбутній грі. Він забезпечує точність керування та створює враження іммерсивності для гравця.

3.5 Реалізація класу BAO_BDevice

Клас BAO_BDevice – структурна одиниця, що збирає інформацію про систему та комп'ютер користувача. Вона потрібна для отримання та управління характеристиками та можливостями пристрою, на якому запущена гра та ігровий рушій [1]. Клас дозволяє отримати інформацію про апаратні можливості. Наприклад, про тип і модель графічної картки, кількість доступної оперативної пам'яті, версії графічного API, кількість ядер процесору, можливість використання векторного простору, та інші характеристики пристрою. Це дозволяє адаптувати програмний продукт типу «гра» до можливостей конкретного пристрою, та оптимізувати її продуктивність. Клас пристрою надає можливість отримати налаштування графіки, які підтримуються пристроєм. Кожний пристрій користувача поділяється на компоненти. Наприклад, на екран, графічну картку, центральний процесор, оперативну пам'ять, тощо. BAO_BDevice допомагає отримати доступ до значення роздільної здатності екрану, отримати інформацію про підтримку шейдерів, функцій згладжування, тощо. Ця інформація дозволяє розробникам гри автоматично встановлювати оптимальні налаштування графіки. Також клас надає інформацію про наявність та тип підключених пристроїв користувацького вводу.

Вхідними даними для класу є інформація про підтримку ширів дебагу, інформація про сім'ю графічного пристрою, інформація про підтримку зміни графічних ланцюгів. На рисунку 3.6 наведено схему класу BAO_BDevice.

BAO_BDevice
void bSetOrthoProj() void bSetOrthoProj() void bSetiewDir() void setViewTarg() void setViewYXZ() glm::mat4& getProject() glm::mat4& getView() glm::mat4& getInvView() glm::vec3 getPos()
glm::mat4 projMat {} glm::mat4 invMat {} glm::mat4 viewMat {}

Рисунок 3.6 - Структура класа BAO_BDevice [18]

З архітектурної точки зору, цей клас належить до так званих одиниць трансляції низького рівня, тому що він безпосередньо взаємодіє з компонентами девайсу.

Загалом, клас пристрою надає розробнику гри доступ до інформації про пристрій, на якому запущений програмний продукт. Це дозволяє розробнику адаптувати гру до можливостей пристрою, динамічно налаштовувати графіку та ввід відповідно до характеристик компонентів девайсу, та забезпечувати оптимальну продуктивність та швидкодію застосунку.

3.6 Реалізація класу BAO_BRender

Клас BAO_BRender є компонентом, який відповідає за візуалізацію графіки та відображення графічних об'єктів на екрані. Він забезпечує відтворення графічної сцени та взаємодію з графічним апаратним обладнанням. Клас рендеру виконує рендеринг графічних об'єктів, таких як: персонажі, об'єкти світу, текстури, освітлення, тощо [1, 11]. Він перетворює 3D-моделі та інші графічні ресурси у візуальні зображення, що на наступному етапі відображаються на екрані користувача через взаємодію з класом вікна. Клас рендеру на автоматичному рівні забезпечує процес візуалізації під-процесом динамічної оптимізації. Функціонал графічного API

використовує різні техніки, такі як: вибір оптимального рівня деталізації, використання техніки розбиття на сцени, оптимізація шейдерів, та інші методи, щоб забезпечити швидку та ефективну візуалізацію на автоматичному рівні. Клас рендеру відповідає за управління графічними буферами та ресурсами. Наприклад, програмна структура працює з буферами вершин, буферами текстур, буферами кадрів.. Клас керує завантаженням, розподілом, та використанням ресурсів для забезпечення ефективного використання пам'яті.

Вхідними даними для класу рендеру слугують посилання на клас вікна, посилання на клас девайсу, унікальний розумний показчик на функціонал зміни графічних ланцюгів, структуру даних типу «вектор», що має у собі команди для буфера. Також клас має індекс нинішнього зображення, індекс нинішнього кадру, та булеве значення для перевірки на початок кадру [11].

BAO_BRender має 12 створених функцій, три з яких є функціями типу «отримати» та створені з використанням статичного зв'язування. Структурна одиниця здатна виконувати різноманітну роботу над кадрами, графічними ланцюгами, буферами, тощо. Структуру класу BAO_BRender наведено на рисунку 3.7.

BAO_BRender
VkRenderPass getBSwapBChainRenderPass() float getAspectRatio() bool blsBFrameInProgress() VkCommandBuffer getCurrentBBuffer() int getBFrameBIndex() VkCommandBuffer beginBFrame() void endBFrame() void beginBSwapBChainBRenderPass() void endBSwapBChainBRenderPass()
void createBCommandBBuffers() void freeBCommandBBuffers() void recreateBSwapBChain() BWindow& bWindow BDevice& bDevice std::unique_ptr<BSwapBChain> bSwapBChain std::vector<VkCommandBuffer> commandBBuffers uint32_t currentBImageBIndex uint32_t currentBFrameBIndex bool blsBFrameStarted()

Рисунок 3.7 – Структура класу BAO_BRender [18]

З архітектурної точки зору, цей клас належить до так званих одиниць трансляції низького рівня, тому що він безпосередньо взаємодіє з компонентами девайсу та низькорівневим графічним Vulkan SDK.

З інформації зазначеної вище, випливає, що структурна одиниця рендеру є одним з фундаментальних постулатів архітектури ігрового рушія, оскільки він відповідає за створення візуального досвіду для гравця, та забезпечує графічну реалістичність, автоматичну оптимізацію продуктивності та швидкодії раз із управлінням графічними ресурсами девайсу користувача.

3.7 Реалізація класу BAO_VPipeline

Клас BAO_VPipeline є компонентом, що відповідає за організацію та керування послідовними етапами обробки графічних даних під час рендерингу [1]. Він виконує важливу роль у визначенні та контролі процесу візуалізації графіки і графічних ефектів у грі. Основна мета пайплайну - оптимізація рендерингу та досягнення високої продуктивності [11]. Ця структурна одиниця має у собі різноманітні етапи обробки та роботи з шейдерами. Наприклад, клас може оброблювати вершинні шейдери. На цьому етапі обробки виконуються операції з окремими вершинами моделі. Ведеться перетворення координат, нормалей та текстурних координат. Пайплайн може оброблювати геометричні шейдери, проводити генерацію додаткових вершин, проводити повторне обчислення нормалей, відкидати та дублювати вершини, тощо. Структуру класу BAO_VPipeline наведено на рисунку 3.8.

BAO_BPipeline
<pre>struct bPipelngConfig { } void bind() void defaultBPipelineConfig() void enableAlphaBlending()</pre>
<pre>std::vector<char> readFile() void createBGraphicsBPipeline() void createBShaderModule()</pre>

Рисунок 3.8 – Структура класа BAO_BPipeline [18]

Вхідними даними для класу є: структура-налаштування пайп-лайну, шлях до фрагментарного шейдеру, шлях до вертикального шейдеру, показчик на модуль шейдеру.

З архітектурної точки зору, цей клас належить до так званих одиниць трансляції низького рівня, тому що він взаємодіє з компонентами пристрою користувача, та проводить обробку інших низькорівневих елементів – шейдерів.

3.8 Висновок

З інформації та описаних пунктів вище, витікає висновок, що програмне забезпечення розроблене автором дипломної роботи дійсно підпадає під архітектурний стандарт компонентно-потоків системи, та у той же час забезпечує об'єктно-орієнтований підхід. На даному етапі, у програмному забезпеченні є ряд як недоліків, так і переваг. Наприклад, з архітектурної точки зору, деякі елементи потребували векторизації. Найкращим рішенням було б векторизувати функції пов'язані з роботою з великою кількістю даних: вертикалі, вектори, трикутники, тощо. Проте, потрібно зазначити, що векторизація коду застосунку – дуже довгий та складний процес, тому з точки зору підходу «час-якість», трата такої великої кількості годин на оптимізацію декількох функцій – далеко не найкраща ідея.

Також потрібно привернути увагу до використання розумних показників у класі рендеру, застосунку, тощо. Ці класи використовують

унікальний показчик, тобто, структура типу «показчик» може мати лише одне посилання на об'єкт. Його не можна копіювати, чи передавати іншим показчикам. Таким чином забезпечується безпека поводження з комп'ютерною пам'яттю, тому що розумний показчик імплементує комп'ютерну ідіому з короткою назвою «RAII». Українською мовою воно розшифровується як «отримання ресурсу – є ініціалізацією» тобто, це означає, що отримання процесу повинно бути невід'ємно пов'язане зі знищенням того ж самого процесу. Саме з використанням цієї ідіоми були створені розумні показчики. Використання «унікального показчика» дає програмісту можливість не слідкувати за його станом та не видаляти його руками з використанням оператора delete - структура даних все зробить сама.

У наступному розділі будуть наведені тести програмного забезпечення та аналіз результатів його роботи

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

У розділі 3 було зазначено, що у наступному розділі буде проведено та наведено результати різноманітних тестів розробленої системи. Ця частина звіту відповідає за відображення результатів роботи над створенням програмного застосунку типу «ігровий рушій».

4.1 Опис результатів експлуатації програмного забезпечення

Програма реалізує базовий функціонал, а саме: створення шейдерів, компіляція шейдерів, створення вікна користувача, створення програмного застосунку, взаємодія з користувачем за допомогою клавіатури, відображення інформації як у двовимірному просторі, так і у тривимірному, ведення логу роботи програми. Також програма здатна опціонально працювати з одним видом світла, та завантажувати тривимірні меші, що складаються виключно з набору вертикалей та точок типу «тривимірний вектор». Функція завантажування текстур була реалізована на початковому етапі створення програмного забезпечення, але у майбутньому було вирішено відмовитись від неї через велику кількість проблем, що були створені під час її роботи. Більш детальну інформацію про результати графічної роботи наведено у пункті 4.2 у вигляді доказів працездатності програмного забезпечення.

4.2 Аналіз проведених тестів та отриманого результату

Під час першого тестового запуску застосунку «програмний рушій», сталася критична помилка з шейдерами – вони не могли скомпілюватися через помилку на рівні `debugLayers`. Система лову критичних помилок відпрацювала штатно, та викликала `debugBreak` для зупинки роботи програмного засобу, а також вивела інформацію від Vulkan SDK. На рисунку

4.2 зазначений список помилок отриманий від програмного засобу під час критичної помилки викликаній неможливістю скомпілювати шейдери.

```
D:\VSProjects\BAO_Engine\BAO_Engine>D:\VulkanSDK\Bin\glslc.exe BAO_BTestShader.vert -o BAO_BTestShader.vert.spv
BAO_BTestShader.vert:6: error: 'dataToPush' : block instance name redefinition
BAO_BTestShader.vert:13: error: 'dataToPush' : cannot be used (maybe an instance name is needed)
BAO_BTestShader.vert:13: error: 'dataToPush' : undeclared identifier
BAO_BTestShader.vert:13: error: 'pushOffset' : vector swizzle too long
BAO_BTestShader.vert:13: error: 'pushOffset' : unknown swizzle selection
BAO_BTestShader.vert:13: error: 'pushOffset' : unknown swizzle selection
BAO_BTestShader.vert:13: error: 'pushOffset' : vector swizzle selection out of range
7 errors generated.

D:\VSProjects\BAO_Engine\BAO_Engine>D:\VulkanSDK\Bin\glslc.exe BAO_BTestShader.frag -o BAO_BTestShader.frag.spv
BAO_BTestShader.frag:5: error: 'dataToPush' : block instance name redefinition
BAO_BTestShader.frag:13: error: 'dataToPush' : cannot be used (maybe an instance name is needed)
BAO_BTestShader.frag:13: error: 'dataToPush' : undeclared identifier
BAO_BTestShader.frag:13: error: 'fillColor' : vector swizzle too long
BAO_BTestShader.frag:13: error: 'fillColor' : unknown swizzle selection
BAO_BTestShader.frag:13: error: 'fillColor' : unknown swizzle selection
BAO_BTestShader.frag:13: error: 'fillColor' : unknown swizzle selection
BAO_BTestShader.frag:13: error: 'fillColor' : unknown swizzle selection
BAO_BTestShader.frag:13: error: 'fillColor' : unknown swizzle selection
BAO_BTestShader.frag:13: error: 'constructor' : not enough data provided for construction
BAO_BTestShader.frag:13: error: 'assign' : cannot convert from 'const float' to 'layout( location=0) out highp 4-component vector of float'
10 errors generated.

D:\VSProjects\BAO_Engine\BAO_Engine>pause
```

Рисунок 4.2 – Список помилок викликаний під час першого запуску

Таким чином, можна побачити, що система лову помилок працює штатно та повністю виконує свою зазначену роботу. На рисунку 4.3 наведено приклад результату штатної роботи частини програмного забезпечення під назвою «компілятор шейдерів».

```
D:\VSProjects\BAO_Engine\BAO_Engine>D:\VulkanSDK\Bin\glslc.exe BAO_BTestShader.vert -o BAO_BTestShader.vert.spv
D:\VSProjects\BAO_Engine\BAO_Engine>D:\VulkanSDK\Bin\glslc.exe BAO_BTestShader.frag -o BAO_BTestShader.frag.spv
D:\VSProjects\BAO_Engine\BAO_Engine>pause
```

Рисунок 4.3. – Штатний вивід компілятора шейдерів

Після стадії компіляції шейдерів починається їх асемблування, растерізація, поєднання кольорів, та вивід на екран. У світі графіки, рендеру та ігрових рушіїв, існує процедура під назвою «the born of the first polygon» [16]. Це процес виводу трикутника на екран, тому що усі полігони є трикутниками та складаються з трьох точок у просторі. Частіше за все, полігон репрезентують як вектор на три елементи типу float. Полігон є найменшою графічною структурною одиницею, що подібна атому у

звичайному світі. На рисунку 4.4 зображено процедуру під назвою «the born of the first polygon», тобто відображення першого статичного полігону, до якого прив'язаний тестовий шейдер, що задає йому жовтий колір.

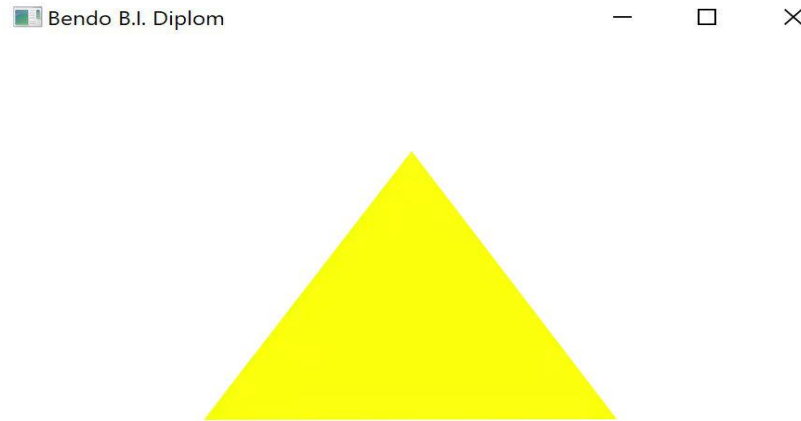


Рисунок 4.4. – Вивід першого полігону

На наступному етапі було вирішено протестувати зміну структури шейдера та її рекомпіляцію під час виконання програми. На рисунку 4.5 можна побачити зміну кольора шейдера, це явище зумовлене процесом модифікації структури файлу шейдера та процесом компіляції файлу з новою інформацією.

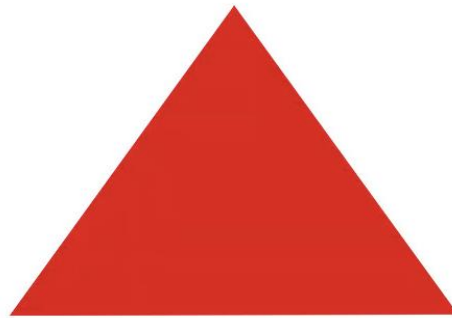


Рисунок 4.5 – Змінений колір полігону на червоний з жовтого

На наступному етапі тестування програмного забезпечення було вирішено викликати функцію інтерполяції частин шейдера. Таким чином, графічний процесор був змушений оброблювати не один колір для всього полігону, а одразу три: RGB(Red-Green-Blue) кольори були накладені на полігон за годинниковою стрілкою. У деяких частинах полігону, це явище викликало створення нових кольорів, наприклад, жовтого під час поєднання червоного та зеленого. Результат цього етапу відображений на рисунку 4.6.



Рисунок 4.6 – Інтерпольований полігон

На наступному етапі було проведено тестування рендеру у тримірному просторі. Було створено примітив типу «куб» з забарвленими різними кольорами сторонами. Вивід зазначений на рисунку 4.7. Візуально можна побачити деякі напливи сторін та кольорів на різні грані через використання однорідних координат. Проте, вивід стверджує, що програмне забезпечення здатне використовувати як двомірний, так і тримірний рендер для відображення інформації користувачеві.

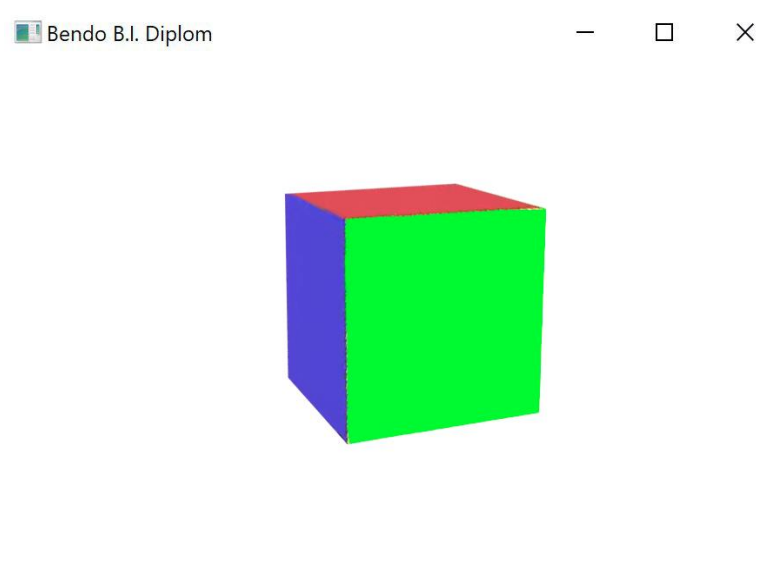


Рисунок 4.7 – Тримірний куб

Виходячи з інформації зазначеної вище, можна зробити висновок, що програма реалізує базові функції, такі як: створення шейдерів, компіляція шейдерів, створення вікна користувача, створення програмного застосунку, створення системи взаємодії з користувачем, відображення інформації як у двовимірному просторі, так і у тривимірному. Також програма здатна працювати з одним видом світла, та завантажувати тривимірні меші, що складаються виключно з набору вертикалей та точок типу «тривимірний вектор».

ВИСНОВОК

У першому розділі пояснювальної записки було проаналізовано поставлене завдання, предметну область та існуючі аналоги у сфері розробки ігрових рушіїв. У висновку вказано, що найкращим, але не ідеальним існуючим рішенням є Unreal Engine 5.

У другому розділі пояснювальної записки було описано процес розробки архітектури програмного забезпечення типу «ігровий рушій», де були зазначено, що програмний застосунок побудований з використанням мови програмування C++ та компонентно-поточної архітектури.

У третьому розділі пояснювальної записки наводився опис імплементованих компонентів системи з використанням UML діаграм для репрезентації структури класу у зрозумілому людині форматі. Було розказано про структуру та функціонал наступних класів: BAO_BWindow, клас BAO_BApp, клас BAO_BDevice, клас BAO_BRender, клас BAO_BCamera, клас BAO_BKeyboard та клас BAO_BPipeline.

У четвертому розділі пояснювальної записки було наведено інформацію про використання та тести програмного забезпечення. Було наведено отриманий після тестів графічний результат.

Резюмуючи, із урахуванням інформації вище, впливає висновок, що під час виконання поставленого завдання, було успішно розроблено програмний застосунок типу «ігровий рушій», що здатний приймати користувацький ввід та візуально відображати графічну інформацію пов'язану з об'єктами рендеру. Ігровий рушій справно працює під управлінням операційної системи Windows. Проте потрібно зазначити, що створений ігровий рушій має свої особисті як переваги, так і недоліки. До основних недоліків можна віднести катастрофічно малий функціонал у порівнянні з існуючими рішеннями, та відсутність широкого спектру оптимізаційних прийомів які присутні у Unreal Engine 5, що є найкращим ігровим рушієм виходячи з висновку першої частини дипломної роботи

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Gregory, J. Game Engine Architecture / J. Gregory. – Natick: A K Peters/CRC Press, 2018. – 1240 p.
2. Offworld Industries Website Title [Electronic Resource]. – Access mode: <https://www.offworldindustries.com/>.
3. Cookson, A. Unreal Engine 4 Game Development in 24 Hours, Sams Teach Yourself / A. Cookson, T. Johnson, R. DowlingSoka, C. Crumpler. - Indianapolis: Sams Publishing, 2016 – 496 p.
4. Manning, J. Unity Game Development Cookbook / J. Manning, T. Nugent, P. Buttfield-Addison – California: O'Reilly Media, 2019 – 406p.
5. Kusswurm, D. Modern x86 Assembly Language Programming: 32-bit, 64-bit, SSE, AVX / D. Kusswurm. – New York: Apress, 2014 – 700 p.
6. Llopis, N. C++ For Game Programmers / N. Llopis. – Boston : Charles River Media, 2003 – 412 p.
7. Williams, A. C++ Concurrency in Action / A. Williams. – New York: Manning, 2012 – 585 p
8. McCool, M. Structured Parallel Programming / M. McCool, J. Reinders, A. Robison. – Burlington: Morgan Kaufmann, 2012 – 432 p
9. Sung, K. Build Your Own 2D Game Engine and Create Great Web Games. 2nd Ed / K. Sung, J. Pavleas. – New York: Apress, 2022 – 741 p.
10. Lapinski, P. Vulkan Cookbook: Work through recipes to unlock the full potential of the next generation graphics API / P. Lapinski. – Birmingham: Packt, 2017 – 700 p.
11. Haines, E. Real-Time Rendering, Fourth Edition / E. Hainess, T. Akenine-Moller, N. Hoffman. - Natick: A K Peters/CRC Press, 2018. - 1178 p.
12. McShaffry, M. Game Coding Complete, Fourth Edition / M. McShaffry, D. Graham. – Boston: Cengage Learning PTR, 2012 – 960 p.

13. Busby, J. Mastering Unreal Technology: Introduction to Level Design with Unreal Engine 3 / J. Busby, Z. Parrish, J. Willson. – Indianapolis: Sams Publishing, 2009 – 889 p.
14. Hunt, A. The Pragmatic Programmer: From Journeyman to Master / A. Hunt, D. Thomas. – Illinois: Addison-Wesley Professional, 1999 – 352 p.
15. Martin R. Clean Code: A Handbook of Agile Software Craftsmanship / R. Martin, D. Wampler. – New York: Pearson, 2008 – 464 p.
16. Feathers M. Working Effectively with Legacy Code / M. Feathers. – Hoboken: Prentice Hall, 2004 – 464 p.
17. C4 Notation [Electronic Resource]. – Access mode: <https://c4model.com>.
18. UML Notation [Electronic Resource]. – Access mode: https://uk.wikipedia.org/wiki/Unified_Modeling_Language.

ДОДАТОК А
Текст програми

A.1 файл BAO_BApplication.h

```

#ifndef BAOAPPLICATIONFILE_H
#define BAOAPPLICATIONFILE_H

#include <string>
#include "BAO_BWindow.h"
#include "BAO_BPipeline.h"
#include "BAO_BSwapChain.h"
#include "BAO_BDevice.h"
#include "BAO_BModel.h"
#include <memory>
#include <array>
#include <stdexcept>

namespace bao
{
    class BApplication
    {
    public:

        int16_t width = 900;
        int16_t height = 600;

    public:

        inline BApplication()
        {
            LoadModels();
            RecreateChainSwapper();
            CreatePipeLayout();
            CreateCommandBuffer();
        }

        BApplication(const BApplication& other) =
delete;
        BApplication& operator=(const BApplication&
other) = delete;
        void run();

        ~BApplication();

    private:

```

```

void LoadModels();
void RecreateChainSwapper();
void CommandBufferOnRecording(int32_t imgIndex);
void CreatePipeLayout();
void DrawFrame();
void CreatePipe();
void CreateCommandBuffer();
void ClearBufferOfCommands();

bao::BWindow bWindowHandler_
{
    width, height, "Bendo B.I. Diplom"
};

bao::BDevice LocalDeviceInstance {
bWindowHandler_ };
    std::unique_ptr<bao::BSwapChain> ChainSwapper;
    std::vector<VkCommandBuffer> commandBuffer;
    std::unique_ptr<BModel> bModel;
    std::unique_ptr<BPipeline> bPipelineHandler_;
    VkPipelineLayout pipeLayout;
};
}

#endif

```

A.2 – файл BAO_BApplication.cpp

```

#include "BAO_BApplication.h"

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>

using namespace bao;

struct PushBuffer
{
    glm::vec2 pushOffset;
    alignas(16) glm::vec3 fillColor;
};

```

```

bao::BApplication::~BApplication()
{
    vkDestroyPipelineLayout(LocalDeviceInstance.GetDevice()
e(), pipeLayout, nullptr);
}

void bao::BApplication::run()
{
    while (!bWindowHandler_.bShouldClose())
    {
        glfwPollEvents();
        DrawFrame();
    }

    vkDeviceWaitIdle(LocalDeviceInstance.GetDevice());
}

void BApplication::LoadModels()
{
    std::vector<BModel::Vertex> vVertices{
        {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
        {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
        {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
    };

    bModel =
std::make_unique<BModel>(LocalDeviceInstance,
vVertices);
}

void bao::BApplication::CreatePipeLayout()
{
    VkPushConstantRange constantRangePush{};
    constantRangePush.stageFlags =
VK_SHADER_STAGE_VERTEX_BIT |
VK_SHADER_STAGE_FRAGMENT_BIT;
    constantRangePush.offset = 0;
    constantRangePush.size = sizeof(PushBuffer);

    VkPipelineLayoutCreateInfo pipeLayoutInfo{};
    pipeLayoutInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipeLayoutInfo.setLayoutCount = 0;

```

```

        pipelineLayoutInfo.pSetLayouts = nullptr;
        pipelineLayoutInfo.pushConstantRangeCount = 1;
        pipelineLayoutInfo.pPushConstantRanges =
&constantRangePush;
        pipelineLayoutInfo.pNext = nullptr;
        pipelineLayoutInfo.flags = 0;

        if
(vkCreatePipelineLayout(LocalDeviceInstance.device(),
&pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS)
        {
            BAOFATAL(EReportCodes::PIPE_FAILURE, "Faield to
create a Pipeline", true);
        }
    }

void bao::BApplication::CreatePipe()
{
    BPipeLineCfg bPCfg{};
    BPipeline::configureDefaultPipe(bPCfg);
    bPCfg.RenderPass = ChainSwapper->getRenderPass();
    bPCfg.PipelineLayout = pipelineLayout;
    bPipelineHandler_ =
std::make_unique<BPipeline>(LocalDeviceInstance,
        "BAO_BTestShader.vert.spv",
        "BAO_BTestShader.frag.spv", bPCfg);
}

void bao::BApplication::CreateCommandBuffer()
{
    commandBuffer.resize(ChainSwapper->imageCount());
    VkCommandBufferAllocateInfo AllocationInformation{};
    AllocationInformation.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    AllocationInformation.level =
VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    AllocationInformation.cPool =
LocalDeviceInstance.GetCommandPool();
    AllocationInformation.commandBufferCount =
static_cast<uint32_t>(commandBuffer.size());

    if
(vkAllocateCommandBuffers(LocalDeviceInstance.GetDevice()
, &AllocationInformation, commandBuffer.data())

```

```

        != VK_SUCCESS)
    {
        EReportCodes::ALLOC_FAILURE, "Failed to allocate
command buffer", true
    }
}

void bao::BApplication::RecreateChainSwapper()
{
    auto localExtentValue = bWindowHandler_.GetExtent();

    while (localExtentValue.height == 0 ||
localExtentValue.width == 0)
    {
        localExtentValue = bWindowHandler_.GetExtent();
        glfwWaitEvents();
    }
    vkDeviceWaitIdle(LocalDeviceInstance.GetDevice());

    ChainSwapper = nullptr;

    if (ChainSwapper == nullptr)
    {
        ChainSwapper =
std::make_unique<BSwapChain>(LocalDeviceInstance,
localExtentValue);
    }
    else
    {
        ChainSwapper =
std::make_unique<BSwapChain>(LocalDeviceInstance,
localExtentValue, std::move(ChainSwapper));
        if (ChainSwapper->imageCount() !=
commandBuffer.size())
        {
            ClearBufferOfCommands();
            CreateCommandBuffer();
        }
    }
    CreatePipe();
}

void BApplication::ClearBufferOfCommands()
{

```

```

        vkFreeCommandBuffers(LocalDeviceInstance.GetDevice()
, LocalDeviceInstance.GetCommandPool(),

static_cast<uint32_t>(commandBuffer.size()),
commandBuffer.data());
        commandBuffer.clear();
    }

void bao::BApplication::CommandBufferOnRecording(int32_t
imgIndex)
{
    static int32_t Frame = NULL;
    Frame = (Frame++) % 100;

    VkCommandBufferBeginInfo beginInformation{};
    beginInformation.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

    if (vkBeginCommandBuffer(commandBuffer[imgIndex],
&beginInformation) != VK_SUCCESS)
    {
        EReportCodes::REC_FAILURE, "Faield to record",
true
    }

    VkRenderPassBeginInfo renderPassInformation{};
    renderPassInformation.sType =
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
    renderPassInformation.renderPass = ChainSwapper-
>getRenderPass();
    renderPassInformation.framebuffer = ChainSwapper-
>getFramebuffer(imgIndex);
    renderPassInformation.renderArea.offset = { 0, 0 };
    renderPassInformation.renderArea.extent =
ChainSwapper->getSwapChainExtent();

    std::array<VkClearColor, 2> ValuesToClear{};
    ValuesToClear[0].color = { 0.1f, 0.1f, 0.1f };
    ValuesToClear[1].depthStencil = { 1.0f, 0 };
    renderPassInformation.clearValueCount =
static_cast<uint32_t>(ValuesToClear.size());
    renderPassInformation.pClearValues =
ValuesToClear.data();

```

```

    vkCmdBeginRenderPass(commandBuffer[imgIndex],
&renderPassInformation, VK_SUBPASS_CONTENTS_INLINE);

    VkViewport viewP{};
    viewP.x = 0.0f;
    viewP.y = 0.0f;
    viewP.width = static_cast<float>(ChainSwapper-
>getSwapChainExtent().width);
    viewP.height = static_cast<float>(ChainSwapper-
>getSwapChainExtent().height);
    viewP.minDepth = 0.0f;
    viewP.maxDepth = 1.0f;
    VkRect2D scissor{ {0, 0}, ChainSwapper-
>getSwapChainExtent()};
    vkCmdSetViewport(commandBuffer[imgIndex], 0, 1,
&viewP);
    vkCmdSetScissor(commandBuffer[imgIndex], 0, 1,
&scissor);

    bPipelineHandler->bind(commandBuffer[imgIndex]);
    bModel->bind(commandBuffer[imgIndex]);

    for (int32_t iterator = 0; iterator < 4; iterator++)
    {
        PushBuffer dataToPush{};
        dataToPush.pushOffset = { -0.5f + Frame * 0.02f,
-0.4f + iterator * 0.25f };
        dataToPush.fillColor = { 0.0f, 0.0f, 0.2f + 0.2f
* iterator };

        vkCmdPushConstants(commandBuffer[imgIndex],
pipeLayout, VK_SHADER_STAGE_VERTEX_BIT |
VK_SHADER_STAGE_FRAGMENT_BIT, 0,
                                sizeof(PushBuffer),
&dataToPush);
        bModel->startDrawing(commandBuffer[imgIndex]);
    }

    vkCmdEndRenderPass(commandBuffer[imgIndex]);

    if (vkEndCommandBuffer(commandBuffer[imgIndex]) !=
VK_SUCCESS)
    {

```

```

        EReportCodes::CBUFFER_FAILURE, "Faield to obtain
a command buffer", true
    }
}

void bao::BApplication::DrawFrame()
{
    uint32_t IndexOfImage;
    auto Resulted = ChainSwapper-
>acquireNextImage(&IndexOfImage);

    if (Resulted == VK_ERROR_OUT_OF_DATE_KHR)
    {
        RecreateChainSwapper();
        return;
    }

    if (Resulted != VK_SUCCESS && Resulted !=
VK_SUBOPTIMAL_KHR)
    {
        EReportCodes::NCIMG_FAILURE, "Faield to obtain
next chain image", true
    }

    CommandBufferOnRecording(IndexOfImage);
    Resulted = ChainSwapper-
>submitCommandBuffers(&commandBuffer[IndexOfImage],
&IndexOfImage);

    if (Resulted == VK_ERROR_OUT_OF_DATE_KHR ||
        Resulted == VK_SUBOPTIMAL_KHR ||
        bWindowHandler_.isFrameResized())
    {
        bWindowHandler_.resetFlagOfResize();
        RecreateChainSwapper();
        return;
    }

    if (Resulted != VK_SUCCESS)
    {
        EReportCodes::CCIMG_FAILURE, "Faield to obtain a
current image chain swapper", true
    }
}

```

```
}
```

A.3 файл BAO_BWindow.h

```
#ifndef BAOBWINDOWFILE_H
#define BAOBWINDOWFILE_H

#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <string>

namespace bao
{
    class BWindow
    {
    public:

        inline BWindow(const int16_t& width, const
int16_t& height, const std::string& name) : width(width),
        height(height),
        bWindowName(name)
        {
            init();
        }

        inline ~BWindow()
        {
            glfwDestroyWindow(bWindow);
            glfwTerminate();
        }

        inline void createSurfaceForBWindow(VkInstance
vkInst, VkSurfaceKHR* vkSurf)
        {
            if (glfwCreateWindowSurface(vkInst, bWindow,
nullptr, vkSurf) != VK_SUCCESS)
            {
                BAOFATAL(EReportCodes::VBUFFER_FAILURE,
"Faield to create a window surface", true);
            }
        }

        inline VkExtent2D GetExtent() {
```

```

        return { static_cast<uint32_t>(width),
static_cast<uint32_t>(height) };
    }

    inline bool bShouldClose()
    {
        return glfwWindowShouldClose(bWindow);
    }

    BWindow(const BWindow& other) = delete;
    BWindow& operator=(const BWindow& other) =
delete;

    inline bool isFrameResized()
    {
        return bIsFrameResized;
    }

    inline void resetFlagOfResize()
    {
        if (bIsFrameResized == true)
        {
            bIsFrameResized = false;
            return;
        }
        bIsFrameResized = true;
    }

private:

    inline void framebufferResize(GLFWwindow* w,
int32_t wi, int32_t h)
    {
        auto bWindowLocal =
reinterpret_cast<BWindow*>(glfwGetWindowUserPointer(w));
        bWindowLocal->bIsFrameResized = true;
        bWindowLocal->width = wi;
        bWindowLocal->height = h;
    }

    inline void init()
    {
        glfwInit();
        glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
    }

```

```

        glfwWindowHint(GLFW_CLIENT_API,
GLFW_NO_API);

        bWindow = glfwCreateWindow(width, height,
bWindowName.c_str(), nullptr, nullptr);
        glfwSetWindowUserPointer(bWindow, this);
    }

    bool bIsFrameResized = false;
    int32_t width;
    int32_t height;

    std::string bWindowName;

    GLFWwindow* bWindow;
};
}

#endif

```

A.4 файл BAO_BDevice.h

```

#ifndef BAOBDEVICEFILE_H
#define BAOBDEVICEFILE_H

#include "BAO_BWindow.h"
#include "BAO_Logger.h"

#include <string>
#include <vector>
#include <vulkan/vulkan.h>

namespace bao
{

    struct SupportedChainSwappersInformation
    {
        std::vector<VkPresentModeKHR> VCurrentModes;
        VkSurfaceCapabilitiesKHR KHRCapabilities;
        std::vector<VkSurfaceFormatKHR> vKHRFormats;
    };

    struct QueuedIndecies
    {

```

```

uint32_t GraphicsProcessingUnitFamily;
uint32_t CurrentFamily;
bool doesCurrentFamilyOwnsAValue = false;
bool doesGPUFamilyOwnsAValue = false;

inline bool isComplete()
{
    return doesGPUFamilyOwnsAValue &&
doesCurrentFamilyOwnsAValue;
}
};

class BDevice
{
public:
#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool areValidationLayersEnabled = true;
#endif

    BDevice(BWindow &window);
    ~BDevice();

    inline VkQueue GetCurrentQueue()
    {
        return pPresentQueue;
    }

    inline VkCommandPool GetCommandPool()
    {
        return cPool;
    }

    inline VkQueue GetGQueue()
    {
        return pGQueue;
    }

    inline VkDevice GetDevice()
    {
        return pDevice;
    }
}

```

```

        inline VkSurfaceKHR GetKHRSurface()
        {
            return pSurface;
        }

        void BufferCopyFun(VkBuffer srcBuffer, VkBuffer
dstBuffer, VkDeviceSize size);
        void BufferCopyToImageFun(VkBuffer buffer,
VkImage image, uint32_t width, uint32_t height, uint32_t
layerCount);
        SupportedChainSwappersInformation
GetSupportedChainSwappers() { return
queryChainSwappersAcceptance(phDevice); }
        uint32_t LookForMemType(uint32_t typeFilter,
VkMemoryPropertyFlags PhysDeviceProperties);
        void CreateBBuffer(VkDeviceSize size,
VkBufferUsageFlags usage, VkMemoryPropertyFlags
PhysDeviceProperties, VkBuffer &buffer,
                                VkDeviceMemory &bufferMemory);
        VkCommandBuffer StartExecutingSTimeCmds();
        void StopExecutingSTimeCmds(VkCommandBuffer
commandBuffer);
        QueuedIndecies LookForPhysDeviceFamilies() {
return LookForQueuedFamilies(phDevice); }
        VkFormat LookForSupportedVKFormat(const
std::vector<VkFormat>& candidates, VkImageTiling tiling,
VkFormatFeatureFlags features);

        void AssemblyImageWithInformationFun(const
VkImageCreateInfo &imageInfo, VkMemoryPropertyFlags
PhysDeviceProperties, VkImage &image,
                                VkDeviceMemory
&imageMemory);

        VkPhysicalDeviceProperties PhysDeviceProperties;

private:
        void AssemblyInst();
        void AssemblyLogicDevice();
        void AssemblyPoolOfCommands();
        void ChoosePhysDevice();
        void AssemblyDBManager();
        void AssemblySurfaceInformation();

```

```

        bool DoesSupportVKValidationLayers();
        bool IsCurrentDeviceAcceptable(VkPhysicalDevice
GetDevice);
        std::vector<const char*> GetExtentions();
        QueuedIndecies
LookForQueuedFamilies(VkPhysicalDevice GetDevice);
        bool
LookForDivesExtentionsAcceptance(VkPhysicalDevice
GetDevice);
        SupportedChainSwappersInformation
queryChainSwappersAcceptance(VkPhysicalDevice GetDevice);
        void
RearrangeDBManagerCInformation(VkDebugUtilsMessengerCreat
eInfoEXT& createInfo);
        void DoesGflwRequiresInstanceExtension();

        VkInstance inst;
        VkDebugUtilsMessengerEXT bMessageDebug;
        VkPhysicalDevice phDevice = VK_NULL_HANDLE;
        BWindow &window;
        VkCommandPool cPool;

        VkDevice pDevice {};
        VkSurfaceKHR pSurface;
        VkQueue pGQueue;
        VkQueue pPresentQueue;

        const std::vector<const char*>
nValidationLayersHandler =
{"VK_LAYER_KHRONOS_validation"};
        const std::vector<const char*> deviceExtensions =
{VK_KHR_SWAPCHAIN_EXTENSION_NAME};
        };

}

#endif

```

A.5 файл BAO_BDevice.cpp

```

#include "BAO_BDevice.h"

#include <cstring>

```

```

#include <iostream>
#include <set>
#include <unordered_set>

using namespace bao;

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

    return VK_FALSE;
}

VkResult  CreateDebugUtilsMessengerEXT(VkInstance  inst,
const          VkDebugUtilsMessengerCreateInfoEXT
*pInformationRepresentation,
                                                    const
VkAllocationCallbacks          *pCallbackForAllocation,
VkDebugUtilsMessengerEXT *pVkDebugMessage)
{
    auto          tmpLocal          =
(PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr
(inst, "vkCreateDebugUtilsMessengerEXT");

    if (tmpLocal != nullptr)
    {
        return tmpLocal(inst, pInformationRepresentation,
pCallbackForAllocation, pVkDebugMessage);
    }
    else
    {
        return VK_ERROR_EXTENSION_NOT_PRESENT;
    }
}

```

```

}

void DestroyDebugUtilsMessengerEXT(VkInstance inst,
VkDebugUtilsMessengerEXT messageForDebugging, const
VkAllocationCallbacks *VulkanAllocCB)
{
    auto tmpLocal =
(PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(
inst, "vkDestroyDebugUtilsMessengerEXT");

    if (tmpLocal != nullptr)
    {
        tmpLocal(inst, messageForDebugging,
VulkanAllocCB);
    }
}

BDevice::BDevice(BWindow &window) : window{window}
{
    AssemblyInst();
    AssemblyDBManager();
    AssemblySurfaceInformation();
    ChoosePhysDevice();
    AssemblyLogicDevice();
    AssemblyPoolOfCommands();
}

BDevice::~~BDevice()
{
    vkDestroyCommandPool(pDevice, cPool, nullptr);
    vkDestroyDevice(pDevice, nullptr);

    if (areValidationLayersEnabled)
    {
        DestroyDebugUtilsMessengerEXT(inst,
bMessageDebug, nullptr);
    }

    vkDestroySurfaceKHR(inst, pSurface, nullptr);
}

```

```

        vkDestroyInstance(inst, nullptr);
    }

void BDevice::AssemblyInst()
{
    if (areValidationLayersEnabled &&
        !DoesSupportVKValidationLayers())
    {
        throw std::runtime_error("validation layers
requested, but not available!");
    }

    VkApplicationInfo BApplicationInformationLocal = {};
    BApplicationInformationLocal.sType =
VK_STRUCTURE_TYPE_APPLICATION_INFO;
    BApplicationInformationLocal.pApplicationName =
"Bendo B.I. Diplom";
    BApplicationInformationLocal.applicationVersion =
VK_MAKE_VERSION(1, 0, 0);
    BApplicationInformationLocal.pEngineName =
"BAO_Engine";
    BApplicationInformationLocal.engineVersion =
VK_MAKE_VERSION(1, 0, 0);
    BApplicationInformationLocal.apiVersion =
VK_API_VERSION_1_0;

    VkInstanceCreateInfo VulkanInformationStructure = {};
    VulkanInformationStructure.sType =
VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    VulkanInformationStructure.pApplicationInfo =
&BApplicationInformationLocal;

    auto localRequiredExtensions = GetExtentions();
    VulkanInformationStructure.enabledExtensionCount =
static_cast<uint32_t>(localRequiredExtensions.size());
    VulkanInformationStructure.ppEnabledExtensionNames =
localRequiredExtensions.data();

```

```

        VkDebugUtilsMessengerCreateInfoEXT
VKDebugInformationStructure;
        if (areValidationLayersEnabled)
        {
            VulkanInformationStructure.enabledLayerCount    =
static_cast<uint32_t>(nValidationLayersHandler.size());
            VulkanInformationStructure.ppEnabledLayerNames    =
nValidationLayersHandler.data();

RearrangeDBManagerCInformation(VKDebugInformationStructur
e);
            VulkanInformationStructure.pNext                =
(VkDebugUtilsMessengerCreateInfoEXT
*)&VKDebugInformationStructure;
        }
        else
        {
            VulkanInformationStructure.enabledLayerCount = 0;
            VulkanInformationStructure.pNext = nullptr;
        }

        if (vkCreateInstance(&VulkanInformationStructure,
nullptr, &inst) != VK_SUCCESS) {
            BAOFATAL(EReportCodes::VINST_FAILURE, "Faield to
create a VInformationStructureInstance", true);
        }

        DoesGflwRequiresInstanceExtension();
    }

void BDevice::ChoosePhysDevice()
{
    uint32_t NumberOfDevices = 0;
    vkEnumeratePhysicalDevices(inst,      &NumberOfDevices,
nullptr);
    if (NumberOfDevices == 0)
    {

```

```

        BAOFATAL(EReportCodes::GPU_FAILURE, "Faield to
find GPU with Vulkan support", true);
    }
    std::cout << "Device count: " << NumberOfDevices <<
std::endl;
    std::vector<VkPhysicalDevice>
vDevices(NumberOfDevices);
    vkEnumeratePhysicalDevices(inst, &NumberOfDevices,
vDevices.data());

    for (const auto &GetDevice : vDevices)
    {
        if (IsCurrentDeviceAcceptable(GetDevice))
        {
            phDevice = GetDevice;
            break;
        }
    }

    if (phDevice == VK_NULL_HANDLE)
    {
        BAOFATAL(EReportCodes::GPU_FAILURE, "Faield to
find GPU", true);
    }

    vkGetPhysicalDeviceProperties(phDevice,
&PhysDeviceProperties);
    std::cout << "physical device: " <<
PhysDeviceProperties.deviceName << std::endl;
}

void bao::BDevice::AssemblyLogicDevice()
{
    bao::QueuedIndecies indices =
LookForQueuedFamilies(phDevice);

    std::vector<VkDeviceQueueCreateInfo>
CreatedQueueInformation;

```

```

        std::set<uint32_t>          uniqueQueueFamilies          =
{indices.GraphicsProcessingUnitFamily,
indices.CurrentFamily};

        float queuePriority = 1.0f;
        for (uint32_t queueFamily : uniqueQueueFamilies)
        {
            VkDeviceQueueCreateInfo DeviceInformationQueue =
{
};
            DeviceInformationQueue.sType =
VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
            DeviceInformationQueue.queueCount = 1;
            DeviceInformationQueue.queueFamilyIndex =
queueFamily;
            DeviceInformationQueue.pQueuePriorities =
&queuePriority;

CreatedQueueInformation.push_back(DeviceInformationQueue)
;
        }

        VkPhysicalDeviceFeatures deviceFeatures = {};
        deviceFeatures.samplerAnisotropy = VK_TRUE;

        VkDeviceCreateInfo VKLocalDeviceInformation = {};
        VKLocalDeviceInformation.sType =
VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

        VKLocalDeviceInformation.queueCreateInfoCount =
static_cast<uint32_t>(CreatedQueueInformation.size());
        VKLocalDeviceInformation.pQueueCreateInfos =
CreatedQueueInformation.data();

        VKLocalDeviceInformation.pEnabledFeatures =
&deviceFeatures;
        VKLocalDeviceInformation.enabledExtensionCount =
static_cast<uint32_t>(deviceExtensions.size());

```

```

        VKLocalDeviceInformation.ppEnabledExtensionNames =
deviceExtensions.data();

        if (areValidationLayersEnabled)
        {
            VKLocalDeviceInformation.enabledLayerCount =
static_cast<uint32_t>(nValidationLayersHandler.size());
            VKLocalDeviceInformation.ppEnabledLayerNames =
nValidationLayersHandler.data();
        }
        else
        {
            VKLocalDeviceInformation.enabledLayerCount = 0;
        }

        if (vkCreateDevice(phDevice,
&VKLocalDeviceInformation, nullptr, &pDevice) !=
VK_SUCCESS) {
            BAOFATAL(EReportCodes::LDP_FAILURE, "Faield to
create logical device", true);
        }

        vkGetDeviceQueue(pDevice,
indices.GraphicsProcessingUnitFamily, 0, &pGQueue);
        vkGetDeviceQueue(pDevice, indices.CurrentFamily, 0,
&pPresentQueue);
    }

void BDevice::AssemblyPoolOfCommands()
{
    QueuedIndecies bQueueIndiciesOfFamily =
LookForPhysDeviceFamilies();

    VkCommandPoolCreateInfo VKPoolOfCommandsInfo = {};
    VKPoolOfCommandsInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    VKPoolOfCommandsInfo.flags =
VK_COMMAND_POOL_CREATE_TRANSIENT_BIT |
VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;

```

```

VKPoolOfCommandsInfo.queueFamilyIndex          =
bQueueIndiciesOfFamily.GraphicsProcessingUnitFamily;

    if (vkCreateCommandPool(pDevice,
&VKPoolOfCommandsInfo, nullptr, &cPool) != VK_SUCCESS)
    {
        BAOFATAL(EReportCodes::VCP_FAILURE, "Faield to
create command pool", true);
    }
}

void BDevice::AssemblySurfaceInformation()
{
    window.createSurfaceForBWindow(inst, &pSurface);
}

bool BDevice::IsCurrentDeviceAcceptable(VkPhysicalDevice
GetDevice)
{
    QueuedIndecies      LocalQueueIndiciesOfFamily      =
LookForQueuedFamilies(GetDevice);

    bool                extensionsSupported              =
LookForDivesExtentionsAcceptance(GetDevice);

    bool swapChainAdequate = false;
    if (extensionsSupported) {
        SupportedChainSwappersInformation
swapChainSupport          =
queryChainSwappersAcceptance(GetDevice);
        swapChainAdequate  =
!swapChainSupport.vKHRFormats.empty()    &&
!swapChainSupport.VCurrentModes.empty();
    }

    VkPhysicalDeviceFeatures
ListOfSupportedDeviceFeatures;
    vkGetPhysicalDeviceFeatures(GetDevice,
&ListOfSupportedDeviceFeatures);

```

```

        return LocalQueueIndiciesOfFamily.isComplete()    &&
        extensionsSupported && swapChainAdequate &&

```

```

ListOfSupportedDeviceFeatures.samplerAnisotropy;
}

```

```

void
BDevice::RearrangeDBManagerCInformation(VkDebugUtilsMessengerCreateInfoEXT &createInfo)
{
    createInfo = {};
    createInfo.sType =
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
    createInfo.messageType =
VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |

VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |

VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
    createInfo.messageSeverity =
VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |

VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
    createInfo.pUserData = nullptr;
    createInfo.pfnUserCallback = debugCallback;
}

```

```

void BDevice::AssemblyDBManager()
{
    if (!areValidationLayersEnabled)
    {
        return;
    }

```

```

        VkDebugUtilsMessengerCreateInfoEXT
        LocalDebugUtilsCreateInformation;

```

```
RearrangeDBManagerCInformation(LocalDebugUtilsCreateInformation);
```

```
    if (CreateDebugUtilsMessengerEXT(inst,
&LocalDebugUtilsCreateInformation, nullptr,
&bMessageDebug) != VK_SUCCESS)
    {
        BAOFATAL(EReportCodes::DEBUGMES_FAILURE, "Failed
to setp debug messenger", true);
    }
}
```

```
bool BDevice::DoesSupportVKValidationLayers()
{
```

```
    uint32_t layerCount;
    vkEnumerateInstanceLayerProperties(&layerCount,
nullptr);
```

```
    std::vector<VkLayerProperties>
availableLayers(layerCount);
    vkEnumerateInstanceLayerProperties(&layerCount,
availableLayers.data());
```

```
    for (const char *NameForLayer :
nValidationLayersHandler)
    {
```

```
        bool exactLayer = false;
```

```
        for (const auto &LayerProps : availableLayers)
        {
```

```
            if (strcmp(NameForLayer,
LayerProps.layerName) == 0)
            {
```

```
                exactLayer = true;
                break;
```

```
            }
        }
```

```

        if (!exactLayer)
        {
            return false;
        }
    }
    return true;
}

std::vector<const char*> BDevice::GetExtentions()
{
    uint32_t glfwAmountOfExtensions = 0;
    const char **glfwExtensions;
    glfwExtensions =
glfwGetRequiredInstanceExtensions(&glfwAmountOfExtensions
);

    std::vector<const char*> vExtentions(glfwExtensions,
glfwExtensions + glfwAmountOfExtensions);

    if (areValidationLayersEnabled)
    {

vExtentions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
    }

    return vExtentions;
}

void BDevice::DoesGflwRequiresInstanceExtension()
{
    uint32_t localExtensionsAmount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr,
&localExtensionsAmount, nullptr);
    std::vector<VkExtensionProperties>
vLocalExtensions(localExtensionsAmount);
    vkEnumerateInstanceExtensionProperties(nullptr,
&localExtensionsAmount, vLocalExtensions.data());

    std::cout << "obtained extensions:" << std::endl;

```

```

        std::unordered_set<std::string> sExtensionsToPrint;
        for (const auto &extension : vLocalExtensions)
        {
            std::cout << "\t" << extension.extensionName <<
std::endl;

sExtensionsToPrint.insert(extension.extensionName);
        }

        std::cout << "needed extensions:" << std::endl;
        auto requiredExtensions = GetExtensions();
        for (const auto &required : requiredExtensions)
        {
            std::cout << "\t" << required << std::endl;
            if (sExtensionsToPrint.find(required) ==
sExtensionsToPrint.end())
            {
                BAOFATAL(EReportCodes::GLFW_FAILURE, "Missing
GLFW", true);
            }
        }
    }

bool
BDevice::LookForDivesExtentionsAcceptance(VkPhysicalDevic
e GetDevice) {
    uint32_t localExtensionsAmount;
    vkEnumerateDeviceExtensionProperties(GetDevice,
nullptr, &localExtensionsAmount, nullptr);

    std::vector<VkExtensionProperties>
vExtensionsToPrint(localExtensionsAmount);
    vkEnumerateDeviceExtensionProperties(GetDevice,
nullptr, &localExtensionsAmount,
vExtensionsToPrint.data());

```

```

        std::set<std::string>
sRequiredExtensions(deviceExtensions.begin(),
deviceExtensions.end());

        for (const auto &extension : vExtentionsToPrint)
        {

sRequiredExtensions.erase(extension.extensionName);
        }

        return sRequiredExtensions.empty();
}

QueuedIndecies
BDevice::LookForQueuedFamilies(VkPhysicalDevice
GetDevice)
{
    QueuedIndecies LocalQueueFamilyIndicies;

    uint32_t queueFamilyCount = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(GetDevice,
&queueFamilyCount, nullptr);

    std::vector<VkQueueFamilyProperties>
vFamilyOfQueues(queueFamilyCount);
    vkGetPhysicalDeviceQueueFamilyProperties(GetDevice,
&queueFamilyCount, vFamilyOfQueues.data());

    int iterator = 0;
    for (const auto &queueFamily : vFamilyOfQueues)
    {
        if (queueFamily.queueCount > 0 &&
queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT)
        {

LocalQueueFamilyIndicies.GraphicsProcessingUnitFamily =
iterator;

LocalQueueFamilyIndicies.doesGPUFamilyOwnsAValue = true;

```

```

    }

    VkBool32 bIsSupportedPresently = false;
    vkGetPhysicalDeviceSurfaceSupportKHR(GetDevice,
    iterator, pSurface, &bIsSupportedPresently);

    if (queueFamily.queueCount > 0 &&
    bIsSupportedPresently)
    {
        LocalQueueFamilyIndicies.CurrentFamily =
    iterator;

    LocalQueueFamilyIndicies.doesCurrentFamilyOwnsAValue =
    true;
    }

    if (LocalQueueFamilyIndicies.isComplete())
    {
        break;
    }

    iterator++;
}

return LocalQueueFamilyIndicies;
}

SupportedChainSwappersInformation
BDevice::queryChainSwappersAcceptance(VkPhysicalDevice
GetDevice)
{
    SupportedChainSwappersInformation
    ChainSwapperSupportDetails;
    vkGetPhysicalDeviceSurfaceCapabilitiesKHR(GetDevice,
    pSurface, &ChainSwapperSupportDetails.KHRCapabilities);

    uint32_t FormatCounter;
    vkGetPhysicalDeviceSurfaceFormatsKHR(GetDevice,
    pSurface, &FormatCounter, nullptr);

```

```

        if (FormatCounter != 0)
        {

ChainSwapperSupportDetails.vKHRFormats.resize(FormatCounter);

            vkGetPhysicalDeviceSurfaceFormatsKHR(GetDevice,
pSurface,                                     &FormatCounter,
ChainSwapperSupportDetails.vKHRFormats.data());
        }

            uint32_t PresentisedModeAmount;
            vkGetPhysicalDeviceSurfacePresentModesKHR(GetDevice,
pSurface, &PresentisedModeAmount, nullptr);

            if (PresentisedModeAmount != 0)
            {

ChainSwapperSupportDetails.VCurrentModes.resize(PresentisedModeAmount);

vkGetPhysicalDeviceSurfacePresentModesKHR(GetDevice,
pSurface,                                     &PresentisedModeAmount,
ChainSwapperSupportDetails.VCurrentModes.data());
            }

            return ChainSwapperSupportDetails;
        }

VkFormat BDevice::LookForSupportedVKFormat(const
std::vector<VkFormat> &vCandidates,    VkImageTiling
VkImgTiling, VkFormatFeatureFlags VkFeatureFlags)
{
    for (VkFormat format : vCandidates)
    {
        VkFormatProperties LocalProperties;
        vkGetPhysicalDeviceFormatProperties(phDevice,
format, &LocalProperties);
    }
}

```

```

        if (VkImgTiling == VK_IMAGE_TILING_LINEAR &&
(LocalProperties.linearTilingFeatures & VkFeatureFlags)
== VkFeatureFlags)
        {
            return format;
        }
        else if (VkImgTiling == VK_IMAGE_TILING_OPTIMAL
&& (LocalProperties.optimalTilingFeatures &
VkFeatureFlags) == VkFeatureFlags)
        {
            return format;
        }
    }

```

```

        BAOFATAL(EReportCodes::FORMAT_FAILURE, "Faield to
find supported format", true);
    }

```

```

uint32_t BDevice::LookForMemType(uint32_t Filter,
VkMemoryPropertyFlags VkMemProperties)
{
    VkPhysicalDeviceMemoryProperties
PhysDeviceMemProperties;
    vkGetPhysicalDeviceMemoryProperties(phDevice,
&PhysDeviceMemProperties);

    for (uint32_t iterator = 0; iterator <
PhysDeviceMemProperties.memoryTypeCount; iterator++)
    {
        if ((Filter & (1 << iterator)) &&
(PhysDeviceMemProperties.memoryTypes[iterator].propertyFl
ags & VkMemProperties) == VkMemProperties)
        {
            return iterator;
        }
    }
}

```

```

        BAOFATAL(EReportCodes::MEM_FAILURE, "Faield to find
suitable mem type", true);
}

```

```

void BDevice::CreateBBuffer(  VkDeviceSize  DeivceSize,
VkBufferUsageFlags    UsageFlags,    VkMemoryPropertyFlags
VkMemProperties,
                                VkBuffer                                &Buffer,
VkDeviceMemory &DeviceMemBuffer)
{
    VkBufferCreateInfo CreatedLocalBufferInformation{};

    CreatedLocalBufferInformation.usage = UsageFlags;
    CreatedLocalBufferInformation.sType =
VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    CreatedLocalBufferInformation.sharingMode =
VK_SHARING_MODE_EXCLUSIVE;
    CreatedLocalBufferInformation.size = DeivceSize;

    if (vkCreateBuffer(pDevice,
&CreatedLocalBufferInformation,    nullptr,    &Buffer) !=
VK_SUCCESS)
    {
        BAOFATAL(EReportCodes::VERTEX_FAILURE, "Faield to
create vertex buffer", true);
    }

    VkMemoryRequirements LocalMemoryRequirments;
    vkGetBufferMemoryRequirements(pDevice,    Buffer,
&LocalMemoryRequirments);

    VkMemoryAllocateInfo LocalMemAllocationInformation{};
    LocalMemAllocationInformation.sType =
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    LocalMemAllocationInformation.allocationSize =
LocalMemoryRequirments.size;
    LocalMemAllocationInformation.memoryTypeIndex =
LookForMemType(LocalMemoryRequirments.memoryTypeBits,
VkMemProperties);
}

```

```

        if (vkAllocateMemory(pDevice,
&LocalMemAllocationInformation, nullptr,
&DeviceMemBuffer) != VK_SUCCESS)
        {
            BAOFATAL(EReportCodes::VMEM_FAILURE, "Faield to
alloc vertex buf mem", true);
        }

        vkBindBufferMemory(pDevice, Buffer, DeviceMemBuffer,
0);
    }

VkCommandBuffer BDevice::StartExecutingSTimeCmds()
{
    VkCommandBufferAllocateInfo
LocalCommandBufferInformationAllocation{};
    LocalCommandBufferInformationAllocation.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    LocalCommandBufferInformationAllocation.commandPool =
cPool;

    LocalCommandBufferInformationAllocation.commandBufferCoun
t = 1;
    LocalCommandBufferInformationAllocation.level =
VK_COMMAND_BUFFER_LEVEL_PRIMARY;

    VkCommandBuffer LocalCommandBuffer;
    vkAllocateCommandBuffers(pDevice,
&LocalCommandBufferInformationAllocation,
&LocalCommandBuffer);

    VkCommandBufferBeginInfo
VkBufferBeginLookingForInformation{};
    VkBufferBeginLookingForInformation.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    VkBufferBeginLookingForInformation.flags =
VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

```

```

        vkBeginCommandBuffer(LocalCommandBuffer,
&VkBufferBeginLookingForInformation);

        return LocalCommandBuffer;
}

void      BDevice::StopExecutingSTimeCmds(VkCommandBuffer
VkBufferOfCommands)
{
    vkEndCommandBuffer(VkBufferOfCommands);

    VkSubmitInfo VkLocalInformationForSumbittion{};
    VkLocalInformationForSumbittion.sType           =
VK_STRUCTURE_TYPE_SUBMIT_INFO;
    VkLocalInformationForSumbittion.commandBufferCount =
1;
    VkLocalInformationForSumbittion.pCommandBuffers    =
&VkBufferOfCommands;

    vkQueueSubmit(pGQueue, 1,
&VkLocalInformationForSumbittion, VK_NULL_HANDLE);
    vkQueueWaitIdle(pGQueue);

    vkFreeCommandBuffers(pDevice, cPool, 1,
&VkBufferOfCommands);
}

void      BDevice::BufferCopyFun(VkBuffer      VkSourceBuffer,
VkBuffer VkDestinationBuffer, VkDeviceSize VkDeviceSize)
{
    VkCommandBuffer      commandBuffer           =
StartExecutingSTimeCmds();

    VkBufferCopy copyRegion{};
    copyRegion.srcOffset = 0;
    copyRegion.dstOffset = 0;
    copyRegion.size = VkDeviceSize;
    vkCmdCopyBuffer(commandBuffer, VkSourceBuffer,
VkDestinationBuffer, 1, &copyRegion);
}

```

```

        StopExecutingSTimeCmds(commandBuffer);
    }

void    BDevice::BufferCopyToImageFun(VkBuffer    buffer,
VkImage image, uint32_t width, uint32_t height, uint32_t
layerCount)
{
    VkCommandBuffer    VkLocalCommandBuffer    =
StartExecutingSTimeCmds();

    VkBufferImageCopy VkBufferImageRegionLocal{};
    VkBufferImageRegionLocal.bufferOffset = 0;
    VkBufferImageRegionLocal.bufferImageHeight = 0;
    VkBufferImageRegionLocal.bufferRowLength = 0;

    VkBufferImageRegionLocal.imageSubresource.aspectMask
= VK_IMAGE_ASPECT_COLOR_BIT;

    VkBufferImageRegionLocal.imageSubresource.baseArrayLayer
= 0;
    VkBufferImageRegionLocal.imageSubresource.mipLevel =
0;
    VkBufferImageRegionLocal.imageSubresource.layerCount
= layerCount;

    VkBufferImageRegionLocal.imageOffset = {0, 0, 0};
    VkBufferImageRegionLocal.imageExtent    =    {width,
height, 1};

    vkCmdCopyBufferToImage(VkLocalCommandBuffer,    buffer,
image,    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,    1,
&VkBufferImageRegionLocal);
    StopExecutingSTimeCmds(VkLocalCommandBuffer);
}

void    BDevice::AssemblyImageWithInformationFun(const
VkImageCreateInfo    &VkImageInformation,
VkMemoryPropertyFlags VkMemProperties,

```

```

VkImage          &VkImg,
VkDeviceMemory &VkImgMem)
{
    if (vkCreateImage(pDevice,      &VkImageInformation,
        nullptr, &VkImg) != VK_SUCCESS)
    {
        BAOFATAL(EReportCodes::ASSEMBLY_FAILURE, "Failed
to assembly", true);
    }

    VkMemoryRequirements memRequirements;
    vkGetImageMemoryRequirements(pDevice,      VkImg,
    &memRequirements);

    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType          =
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex =
LookForMemType(memRequirements.memoryTypeBits,
VkMemProperties);

    if (vkAllocateMemory(pDevice, &allocInfo, nullptr,
    &VkImgMem) != VK_SUCCESS)
    {
        BAOFATAL(EReportCodes::IMEM_FAILURE, "Faield to
allocate image memory", true);
    }

    if (vkBindImageMemory(pDevice, VkImg, VkImgMem, 0) !=
VK_SUCCESS)
    {
        BAOFATAL(EReportCodes::IMEMBIND_FAILURE, "Faield
to bind image memmory", true);
    }
}

```

ДОДАТОК Б
Слайди презентації

Національний університет «Запорізька політехніка»
Кафедра програмних засобів

ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО РУШІЯ ДЛЯ WINDOWS

Виконав:
с.т. групи КНТ-129

Керівник:
зав.кафедрою, професор

Богдан БЕНДО

Сергій СУББОТІН

1

Рисунок Б.1 – Слайд 1

Інформація про роботу

- Об'єкт дослідження: процес створення ігрового рушія.
- Мета роботи: виявити, дослідити та продивитись процес створення ігрового рушія. Дослідити архітектуру ігрового рушія, типи та призначення структурних компонентів. Створити свій особистий ігровий рушій
- Завдання на дипломну роботу:
 - 1) Аналіз предметної області
 - 2) Аналіз та порівняння існуючих рішень
 - 3) Вивчення та аналіз літератури
 - 4) Проектування програмного застосунку
 - 5) Створення програмного застосунку

2

Рисунок Б.2 – Слайд 2

ПОРІВНЯННЯ ІГРОВИХ РУШІВ				
	Unreal Engine 4	Unreal Engine 5	Unity	Godot Engine
Графічний рушій	Unreal Engine 4 rendering architecture	Unreal Engine 5 renderer	Unity Engine	Godot Engine rendering engine
Основна мова програмування	C++	C++	C#	C++
Набір векторних інструкцій	SSE, SSE-2, SSE-3, SSE-4, AVX, AVX-2	SSE, SSE-2, SSE-3, SSE-4, AVX, AVX-2, AVX-512	SSE, AVX, AVX-2	SSE-2, SSE-3
Підтримка методів вирівнювання даних у векторному просторі	Регуляризація патерну векторизації, модифікація "піп-лупів"	Регуляризація патерну векторизації, модифікація "піп-лупів"	Відсутні	Відсутні
Підтримка інструкцій розпаралелювання операцій	Ні	Так	Ні	Так

3

Рисунок Б.3 – Слайд 3

ПОРІВНЯННЯ МОВ ПРОГРАМУВАННЯ			
	C++	C#	Python
Параметр швидкодії	Високий	Середній	Низький
Складність	Висока	Середня	Низька
Наявність збирача сміття	Ні	Так	Так
Типізація	Статична	Статична	Динамічна
Рівень	Низький + Високий	Високий	Високий
Алокація пам'яті	Програмістом	Автоматична	Автоматична

4

Рисунок Б.4 – Слайд 4

ПОРІВНЯННЯ ГРАФІЧНИХ БІБЛІОТЕК		
	VULKAN	OpenGL
Орієнтованість на потоки	Так	Ні
Тип API	Низькорівневий	Використовуваний
Показник швидкодії	Високий	Залежить від імплементації
Підтримка асинхронної обробки інформації	Так	Ні
Рівень доступу до апаратних ресурсів	Прямий	Обмежений
Підтримка використання мультипотуку	Так	Ні
Переносимість без портування	Так	Так
Підтримка геометричних шейдерів	Так	Ні
Підтримка мультиплатформеної обробки графічної інформації	Так	Ні

5

Рисунок Б.5 – Слайд 5



6

Рисунок Б.6 – Слайд 6

Архітектура розробленого ігрового рушія

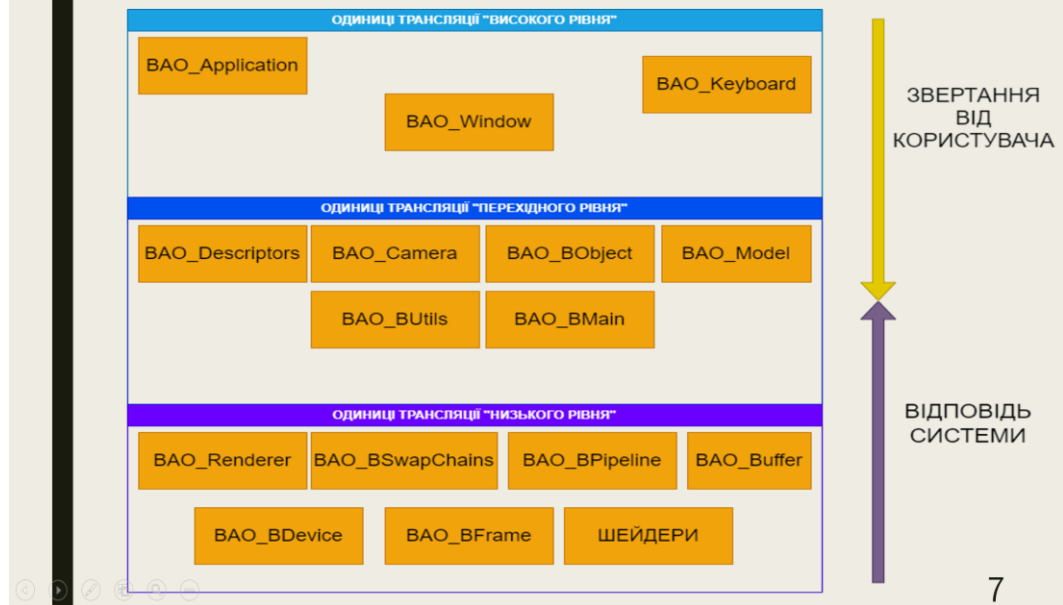


Рисунок Б.7 – Слайд 7

Схема потоку відображення інформації у розробленому застосунку

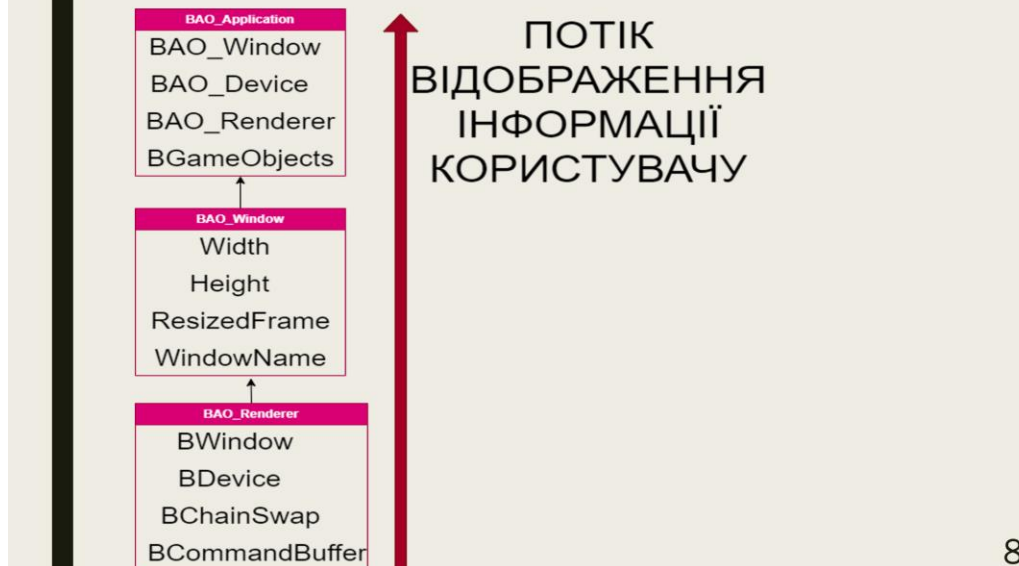
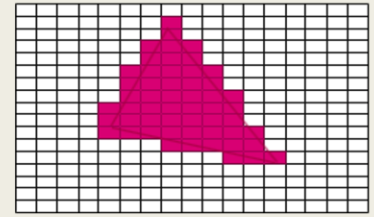


Рисунок Б.8 – Слайд 8

Частини рендеру

- Шейдер - шейдером називається примітивна програма для візуального визначення поверхні об'єкта



- Графічний пайплайн – послідовність етапів процесу відтворення графічних зображень, що відбувається у графічному процесорі комп'ютера

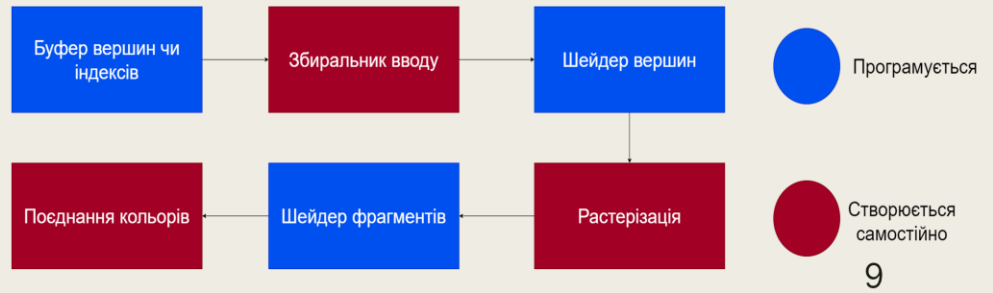
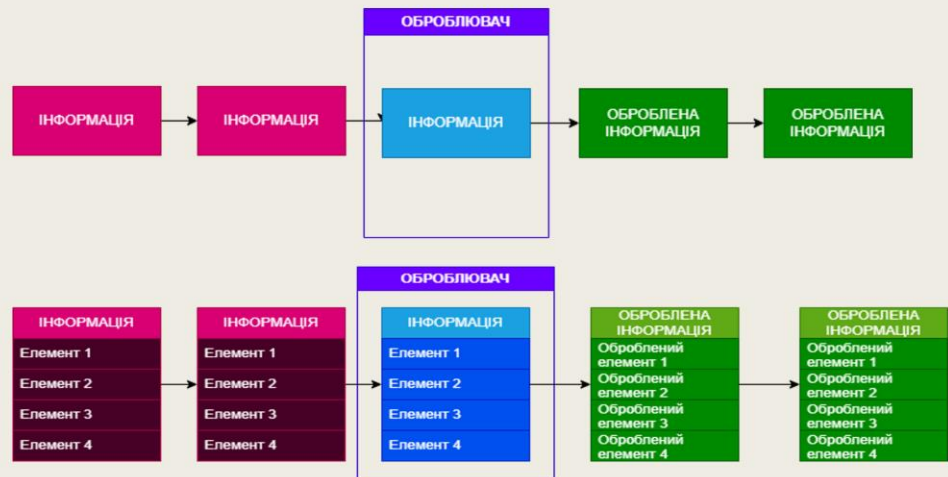


Рисунок Б.9 – Слайд 9

Схема роботи SIMD



10

Рисунок Б.10 – Слайд 10

Порівняння програмного рішення з кращим рушієм		
	BAO_Engine	Unreal Engine 5
Ціна	Безкоштовний	5% від доходу гри
Функціонал	Менший за Unreal Engine 5	Широкий
Підтримка RT	Так	Так
Вага	Відносно мала	Велика
Наявність бібліотек для роботи зі штучним інтелектом	Ні	Так
Наявність системи візуального скриптингу	Ні	Так
Основна мова програмування	C++	C++
Використання пам'яті ОЗП	Мала	Дуже велика
Підтримка можливості компіляції шейдерів у рантаймі	Так	Ні

11

Рисунок Б.11 – Слайд 11

Висновок

- Під час виконання поставленого завдання, було успішно розроблено програмний застосунок типу «ігровий рушій», що здатний приймати користувацький ввід та візуально відображати графічну інформацію пов'язану з об'єктами рендеру. Ігровий рушій справно працює під управлінням операційної системи Windows. Проте потрібно зазначити, що створений ігровий рушій має свої особисті як переваги, так і недоліки. До основних недоліків можна віднести катастрофічно малий функціонал у порівнянні з існуючими рішеннями, та відсутність широкого спектру оптимізаційних прийомів які присутні у Unreal Engine 5, що є найкращим ігровим рушієм виходячи з висновку першої частини дипломної роботи

12

Рисунок Б.12 – Слайд 12