КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

РОЗРОБКА ІГРОВОГО 2D РУШІЯ З ВИКОРИСТАННЯМ OPENGL

Виконав студент 4-го курсу:
Капелянович Максим Володимирович

(підпис)

Науковий керівник:
асистент, кандидат технічних наук
Федорус Олексій Мстиславович

(підпис)

Консультант:
асистент, кандидат фізико-математичних наук
Криволап Андрій Володимирович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«____» _____ 202_ р.,
протокол № ___
Завідувач кафедри
Нікітченко М.С.

(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи: 56 сторінок, 21 ілюстрація, 2 таблиці, 21 джерело посилань.

Об’єкт роботи: ігровий рушій для розробки відеоігор. Предметом роботи є реалізація системи на C++ з використанням OpenGL.

Мета роботи: створення системи для спрощення процесу розробки відеоігор.

Методи розроблення: аналіз існуючих систем, вивчення наявних підходів до розробки ігор, розробка системи.

Інструменти розроблення: мова програмування - C++ та GLSL, CLion – інтегроване середовище розробки (IDE) для мови C++, GLFW - Graphics Library Framework, бібліотека, яка надає OpenGL все необхідне, для рендерингу контенту на екран, GLEW - OpenGL Extension Wrangler Library, бібліотека, яка дає однаковий API до функції OpenGL, незалежно від платформи.

Результати роботи: було досліджено наявні ігрові рушії, способи їх реалізації, переваги та недоліки, розроблено власний ігровий рушій, функціоналу якого вже достатньо для створення простих ігор. Для демонстрації можливостей рушія було створено гру PacMan.

Висновки та пропозиції щодо розвитку об’єкта розроблення й доцільності продовження розробок: при подальшій розробці програмна система має багато варіантів розширення, які детально описані у висновках.

Ключові слова: OPENGL, GLSL, GLFW, ІГРОВИЙ РУШІЙ/ДВИГУН, C++, ШЕЙДЕРИ.

Зміст

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

GDB - GNU Debugger

STL - Standard Template Library

OpenGL - Open Graphics Library

GLSL - OpenGL Shading Library.

GLFW - Graphics Library Framework

GLEW - OpenGL Extension Wrangler Library

ШІ - штучний інтелект

API - Application Programming Interface

GPU - Graphics Processing Unit

CPU - Central Processing Unit

DPI - dots per inch

GLM - OpenGL Mathematics

NDC - normalized device coordinates

ВСТУП

Актуальність роботи та підстави для її виконання. У сучасному світі дуже вагоме місце займає індустрія розваг. Існує багато видів відпочинку, один із яких — відеоігри.

За останні десятиліття індустрія відеоігор перейшла від стереотипу “ігри тільки для дітей” до мільярдних прибутків та мільярдів граючих щодня людей. Індустрія росте і розвивається, тому їй потрібні засоби, які б спрощували процес розробки продуктів. Звісно існує дуже багато давно написаних і протестованих ігрових рушіїв, таких як, наприклад, Unreal Engine, яким користуються багато відомих і не тільки компаній (особливо коли з’явилась його офіційна безкоштовна версія). Але вони не ідеальні. Перша версія того ж Unreal написана більше двадцяти років тому, і хоч код багато разів модифікувався, стара неефективна архітектура дає про себе знати навіть зараз. А якщо постаратись, то можна навіть знайти незмінні шматки коду з найпершої версії рушія. Звісно, що з таким підходом архітектура системи далека від ідеальної. У деяких місцях зустрічається дублювання коду, різні класи мають однаковий функціонал з мінімальними змінами, але вони можуть бути дуже критичні і їх зміна може потягнути за собою зміни половини проекту.

Отже, **метою** роботи є розробка ігрового рушія на основі існуючих, з використанням сучасних підходів та шаблонів для розробки ігрових рушіїв. Для досягнення цієї мети було поставлено такі **завдання**:

- дослідити наявні ігрові рушії, способи їх реалізації;
- виявити переваги та недоліки;
- розробити власний ігровий рушій, враховуючи попередній пункт.

Об’єкти та методи розроблення. Об’єктом розроблення є ігровий рушій для розробки відеоігор. Предметом роботи є реалізація системи на C++ з використанням OpenGL.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ ІГРОВИХ РУШІЇВ

1.1 Ігровий рушій та його призначення

Ігровий рушій є основним програмним забезпеченням, необхідним для правильної роботи ігрової програми[1][2].

Розробники використовують ігрові рушії для побудови ігор для консолей, мобільних пристроїв та персональних комп’ютерів. Основна функціональність, яку зазвичай надає ігровий рушій, може включати механізм візуалізації ("візуалізатор") для 2D або 3D графіки, фізичний механізм або виявлення зіткнень (і відповідь на зіткнення), звук, сценарії, анімацію, штучний інтелект, мережу, потокове передавання, управління пам’яттю, потоками, підтримка локалізації, графік сцени та підтримка відео для кінематики. Реалізатори ігрових механізмів часто економлять на процесі розробки ігор, повторно використовуючи/адаптуючи, в основному, один і той же ігровий рушій для створення різних ігор або для допомоги в перенесенні ігор на кілька платформ.

У багатьох випадках ігрові рушії надають набір інструментів візуальної розробки на додаток до багаторазових програмних компонентів. Ці інструменти, як правило, надаються в інтегрованому середовищі розробки, що дозволяє спростити швидкий розвиток ігор на основі даних. Розробники ігрових механізмів часто намагаються максимально задовольнити потреби розробників ігор, розробляючи надійні програмні комплекти, що містять багато елементів, які розробнику ігор можуть знадобитися для побудови гри. Більшість наборів ігрових рушіїв забезпечують функціонал, що полегшує розробку, наприклад, графіка, звук, фізика та функції штучного інтелекту (ШІ). Ці ігрові рушії іноді називають "проміжним програмним забезпеченням", оскільки, як і в діловому сенсі цього терміну, вони забезпечують гнучку та багаторазову програмну платформу, яка забезпечує всі основні функціональні можливості, необхідні безпосередньо з коробки, для розробки гри при

одночасному зменшенні витрат, складності та часу виходу на ринок - усі найважливіші фактори у висококонкурентній галузі відеоігор. Одними з найпопулярніших сучасних ігрових рушіїв є Unity 3d, Unreal Engine, CryEngine та інші.

Як і інші типи проміжного програмного забезпечення, ігрові рушії, як правило, забезпечують абстракцію платформи, дозволяючи одній і тій самій грі працювати на різних платформах (включаючи ігрові консолі та персональні комп'ютери) з незначними, якщо такі є, змінами, внесеними у вихідний код гри. Часто програмісти розробляють ігрові рушії з архітектурою на основі компонентів, яка дозволяє замінити або розширити конкретні системи в рушії на більш спеціалізовані (а часто і дорожчі) компоненти ігрового проміжного програмного забезпечення. Деякі ігрові рушії містять серію слабо пов'язаних компонентів ігрового проміжного програмного забезпечення, які можна вибірково комбінувати для створення власного механізму, замість більш поширеного підходу до розширення або налаштування гнучкого інтегрованого продукту. Однак розширюваність залишається головним пріоритетом для ігрових рушіїв завдяки широкому спектру використання, для яких вони застосовуються. Незважаючи на специфіку назви "ігровий рушій", кінцеві користувачі часто переробляють ігрові рушії для інших типів інтерактивних додатків із графічними вимогами в режимі реального часу - наприклад, демонстраційні маркетингові демонстрації, візуалізація архітектури, симуляції тренувань, середовища моделювання та виробництво фільмів.

Деякі ігрові рушії надають лише можливості 3D-рендерингу в режимі реального часу, а не широкий спектр функціональних можливостей, необхідних іграм. Ці рушії покладаються на розробника ігор, щоб реалізувати решту цієї функціональності або зібрати її з інших компонентів ігрового проміжного програмного забезпечення. Ці типи рушіїв зазвичай називають "графічним рушієм", "механізмом рендерингу" або "рушієм 3D" замість більш охоплюючого терміна "ігровий рушій". Прикладами графічних механізмів є:

Crystal Space, Genesis3D, Irrlicht, OGRE, RealmForge, Truevision3D та Vision Engine. Сучасні ігрові чи графічні рушії, як правило, надають граф сцени - об’єктно-орієнтоване представлення 3D-ігрового світу, яке часто спрощує ігровий дизайн і може бути використано для більш ефективного відтворення величезних віртуальних світів.

У міру старіння технології компоненти рушія можуть застаріти або бути не достатніми для вимог проекту. Оскільки складність програмування абсолютно нового рушія може призвести до небажаних затримок (або вимагати повторного запуску проекту з самого початку), команда розробників рушіїв може вибрати оновлення свого існуючого рушія новими функціональними можливостями або компонентами.

1.2 Аналіз ігрових рушіїв

Зараз існує багато різноманітних ігрових рушіїв, які відрізняються за призначенням, зручністю та популярністю. Провести дослідження всіх або хоча б більшості існуючих рушіїв не є можливим, тому було вирішено взяти лише кілька популярних і детально їх вивчити.

Класифікувати рушії за популярністю можна декількома способами: провести опитування розробників щодо системи, яку вони використовують, або переглянути бази даних випущених ігор (наприклад, SteamDB) та визначити який рушій вони використовують.

Згідно з дослідженням [3] проведеним співробітниками Університету Шковде було проаналізовано 2 цифрові магазини ігор: Itch.io та Steam. Вибір магазинів ігор базувався на двох основних факторах: на даних звіту “The State of the Industry”, опублікований у співпраці з Game Developers Conference 2019 [4] про те, що Steam (47%) та Itch.io (18%) - це два найчастіше використовувані цифрові магазини для ігор на ПК, та на різниці вмісту між двома магазинами. Steam містить в собі спектр ігор, що складається з ігор як

великого, так і малого виробництва, тоді як Itch.io головним чином орієнтований на інді та розробників з проектами меншого масштабу.

В результаті проведеного дослідження були отримані дані по Itch.io (Таблиця 1) та Steam (Таблиця 2).

Таблиця 1. Ідентифіковані ігрові рушії в іграх, випущених в Itch.io [3]

Ігровий рушій	Кількість ігор	% від загальної кількості ігор
Unity	24200	47.3
Construct	6275	12.3
GameMaker	5643	11
Twine	3184	6.2
RPG Maker	1982	3.9
Bitsy	1683	3.3
PICO-8	1479	2.9
Unreal Engine	1458	2.8
Godot	1274	2.5
Ren`Py	1008	2
Інші рушії	2993	5.9
Всього	51179	

Таблиця 2. Ідентифіковані ігрові рушії в іграх, випущених в Steam [3]

Ігровий рушій	Кількість ігор	% від загальної кількості ігор
Unreal Engine	1726	23.9
Unity	889	12.3
Source	270	3.7
Cryengine	238	3.3
Gamebryo	215	3
IW	192	2.7
Anvil	166	2.3
id Tech	113	1.6
Essence	73	1
Clausewitz	68	0.9
Інші рушії	3266	45.3
Всього	7216	

Перетин 10 найкращих ігрових двигунів на Itch.io та списку ігрових двигунів Steam невеликий: тільки Unity та Unreal присутні в обох списках. Жоден інший движок із набору даних Steam взагалі не присутній у даних Itch.io. Тому для детального аналізу було вирішено обрати саме Unity та Unreal Engine.

1.3 Unity (компоненти для 2D розробки)

Unity - кросплатформовий інструмент для розробки відео ігор і програм, і рушій, на якому вони працюють (рис. 1). Створені за допомогою Unity програми працюють на настільних комп’ютерних системах, мобільних пристроях та гральних консолях у дво- та тривимірній графіці, та на пристроях віртуальної чи доповненої реальності. Unity дозволяє створювати програми із підтримкою як DirectX, так і OpenGL. [5]

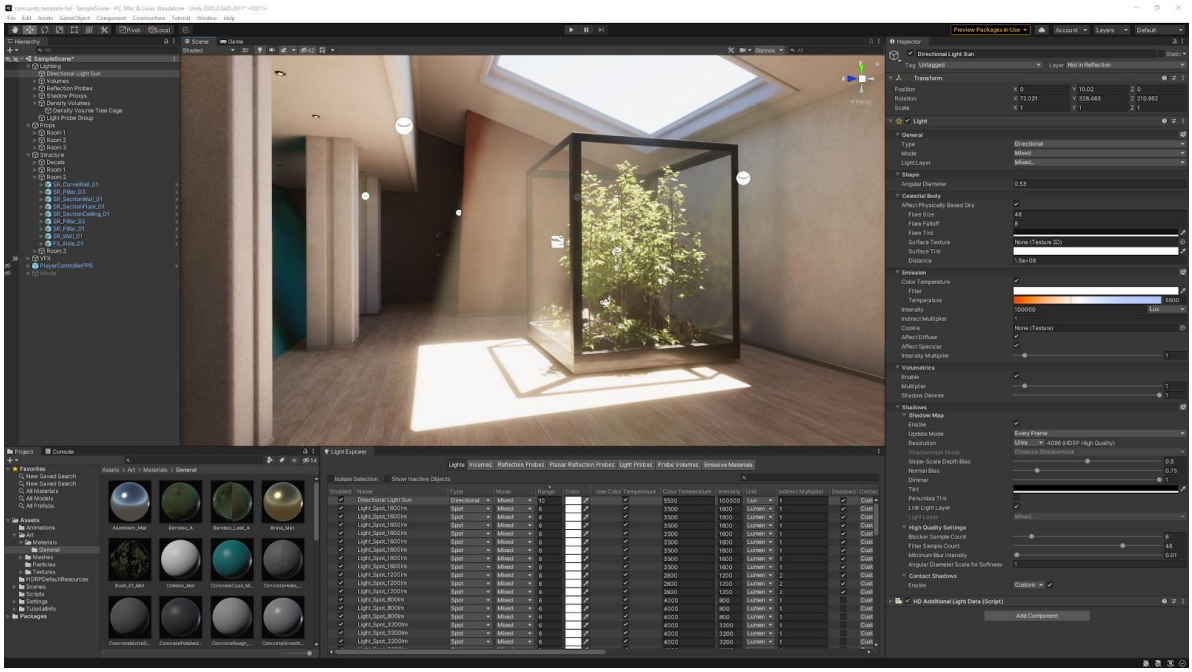

Рисунок 1. Вигляд вікна редактора Unity 3d

Програма-редактор Unity працює на Windows, macOS і Linux, а сам рушій може запускатися на 25 платормах, а саме iOS, Android, Tizen, Windows, Universal Windows Platform, Mac, Linux, WebGL, PlayStation 4, PlayStation Vita, Xbox One, 3DS, Oculus Rift, Google Cardboard, Steam VR, PlayStation VR, Gear VR, Windows Mixed Reality, Daydream, Android TV, Samsung Smart TV, tvOS, Nintendo Switch, Xbox Series X та Series S, PlayStation 5, Facebook Gameroom, Apple ARKit, Google ARCore, Vuforia, і Magic Leap.

Ігрова логіка пишеться за допомогою мови C#, раніше також була можливість використовувати Boo та JavaScript, але розробники відмовились від їх підтримки.

Переваги: багатоплатформний, безкоштовний, простий для освоєння, має швидку компіляцію, детальну документацію, має велику бібліотеку відеоуроків, візуальне середовище розробки та модульну систему компонентів, за допомогою якої відбувається конструювання ігрових об’єктів.

Недоліки: обмеження візуального редактора при роботі з багатокомпонентними схемами, недостатній функціонал візуального редактора при роботі з 2D компонентами (наприклад, відсутні інструменти моделювання [6]), складне редагування шаблонів примірників (англ. Prefabs), проблеми оптимізації та продуктивності, великий об’єм пам’яті навіть невеликих ігор, незручна в користуванні система подій, обмежений функціонал дебагу в безкоштовній версії, для нормальної роботи потрібно встановити багато додаткових плагінів, часто платних.

1.4 Unreal Engine (компоненти для 2D розробки)

Unreal Engine - ігровий рушій, розроблюваний і підтримуваний компанією Epic Games (рис. 2) [7].

Перша гра, створена на цьому рушії - Unreal - з’явилася 1998 року. З тих пір різні версії цього ігрового рушія використали в більш ніж сотні ігор, серед яких Deus Ex, Lineage II, Thief: Deadly Shadows, Postal 2, а також у відомих ігрових серіях Unreal та Unreal Tournament від самої Epic Games. Пристосований у першу чергу для шутерів від першої особи, рушій використовувався й при створенні ігор інших жанрів.

Написаний мовою C++, рушій дозволяє створювати ігри для більшості операційних систем і платформ: Microsoft Windows, Linux, Mac OS і Mac OS

X, консолей Xbox, Xbox 360, PlayStation 2, PlayStation Portable, PlayStation 3, Wii, Dreamcast і Nintendo GameCube, iPod Touch та iPhone 3GS, webOS.

Рисунок 2. Вигляд вікна редактора Unreal Engine 4

Переваги: багатоплатформний, починаючи з 4 версії безкоштовний, наявне візуальне середовище розробки, наявні всі необхідні інструменти для повної розробки гри, наявна детальна документація та велика бібліотека відеоуроків, для програмування використовується мова C++, що позитивно впливає на продуктивність та швидкість роботи гри, є система візуального програмування Blueprints, яка дозволяє писати скрипти та створювати об’єкти без написання програмного коду, наявна велика кількість безкоштовних асетів, які без обмежень можна використовувати в комерційних продуктах, відкритий вихідний код рушія, який можна модифікувати, оптимізована мережева архітектура, підтримка всіх сучасних технологій рендерингу.

Недоліки: редактор важкий в освоєнні, потрібно мати досвід розробки, погана оптимізація системи Blueprints, заточений під мультиплеєрні шутери, “перевантаженість” класів, обмеження кількості параметрів, які може передати

делегат події (обмеження можливо обійти, але це потребує написання великої кількості коду).

Для розробки власного ігрового 2D рушія було вирішено створити систему компонентів як Unity. Також вирішено створити власну систему подій, яка була б простою, надійною та швидкою у використанні.

РОЗДІЛ 2. ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

2.1 CLion

CLion - кросплатформне інтегроване середовище розробки для мови C++ (рис. 3). [8]

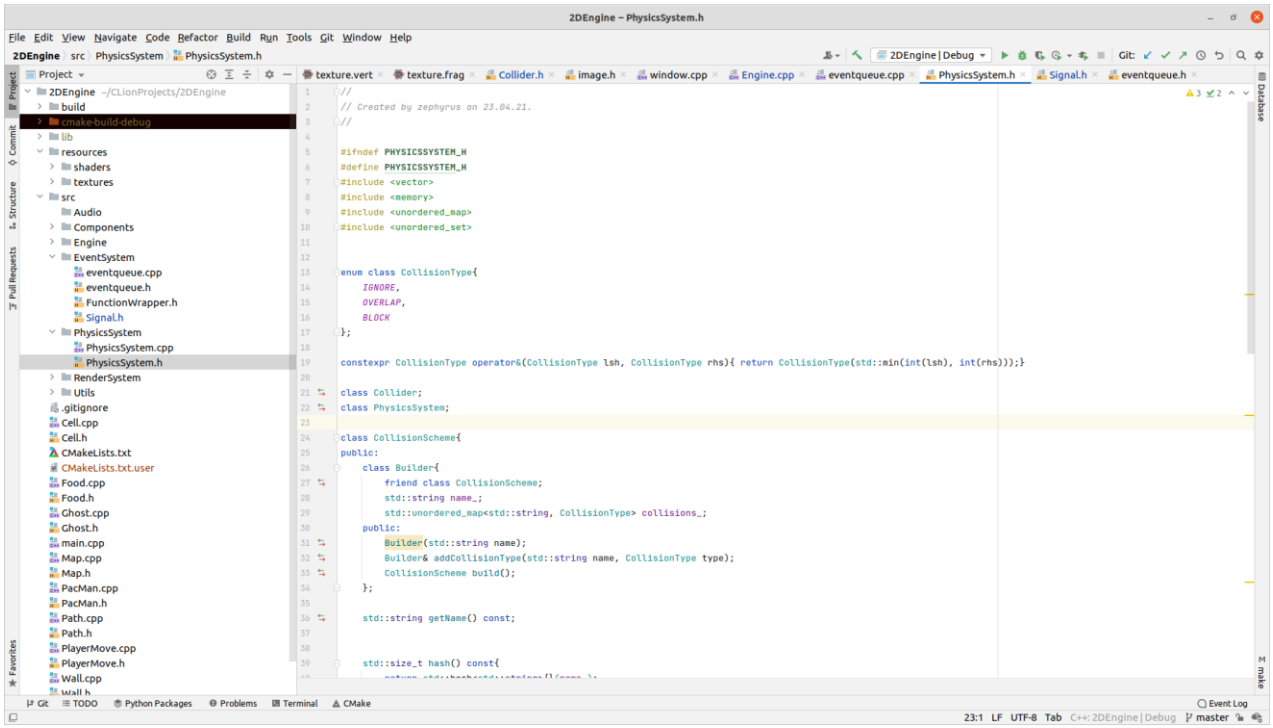

Рисунок 3. Вигляд редактора CLion

Основні його можливості:

- функціонал для автоматичного доповнення коду;
- форматування та слідування певному стилю коду;
- аналізатор коду Clang-Tidy;
- генерація функцій, конструкторів, сетерів та гетерів, операторів порівняння;
- підтримка Doxygen для документування функцій та класів;
- багатофункціональний зневаджувач з підтримкою GDB та LLDB;
- підтримка систем збірки CMake, Makefile, Gradle.

2.2 C++

C++ - мова програмування високого рівня [9], розроблена Б’ярном Страуструпом 1979 року з початковою назвою “C з класами”. Заснована на мові C. Має підтримку декількох парадигм програмування: об’єктно-орієнтованої, узагальненої та процедурної.

C++ було створено з орієнтацією на системне програмування та вбудоване програмне забезпечення, обмежене ресурсами, та великі системи, що вимагають продуктивності, ефективності та гнучкості використання. C++ також виявився корисним у багатьох інших сферах, головними вимогами яких є програмна інфраструктура та обмежені ресурси (наприклад, електронна комерція, веб-пошук або бази даних, телефонні комутатори або космічні зонди). [10]

C++ стандартизований Міжнародною організацією зі стандартизації (ISO), остання стандартна версія ратифікована та опублікована ISO у грудні 2020 року як ISO / IEC 14882: 2020 (неофіційно відома як C++20). Мова програмування C++ була спочатку стандартизована в 1998 році як ISO / IEC 14882: 1998, яка потім була змінена стандартами C++03, C++11, C++14 та C++17. Поточний стандарт C++20 доповнює їх новими функціями та розширеною стандартною бібліотекою. З 2012 року C ++ працює на трирічному графіку випуску, а C ++ 23 - наступний запланований стандарт. [11]

Стандарт C++ складається з двох частин: ядро мови та стандартна бібліотека. Стандартна бібліотека включає збірні типи (vectors, lists, maps, sets, queues, stacks, arrays, tuples), алгоритми (find, for_each, binary_search, random_shuffle тощо), засоби введення / виведення (iostream, для читання та запису в консоль та файли), бібліотека файлової системи, підтримка локалізації, розумні вказівники для автоматичного керування пам’яттю, підтримка регулярних виразів, багатопоточна бібліотека, підтримка атомарних операцій (що дозволяють читати або записувати змінну лише одним потоком), утиліти часу (вимірювання, отримання поточного часу тощо), систему

винятків, генератор випадкових чисел та дещо модифіковану версію стандартної бібліотеки C (для забезпечення сумісності з системою типів C++). [12]

Значна частина бібліотеки C++ базується на стандартній бібліотеці шаблонів (STL). Корисні інструменти, що надаються STL, включають контейнери як колекції об'єктів (наприклад, вектори та списки), ітератори, що забезпечують схожий на масив доступ до контейнерів, та алгоритми, що виконують такі операції, як пошук та сортування.

Крім того, наявні асоціативні масиви та множини. Тому, використовуючи шаблони, можна писати загальні алгоритми, які працюють з будь-яким контейнером або з будь-якою послідовністю, визначеною ітераторами. Як і в C, доступ до функцій бібліотеки здійснюється за допомогою директиви #include для включення стандартного заголовка. Стандартна бібліотека C++ забезпечує 105 стандартних заголовків, з яких 27 застаріли.

Стандарт включає STL, який спочатку був розроблений Олександром Степановим, який протягом багатьох років експериментував із загальними алгоритмами та контейнерами. Почавши з C++, він нарешті знайшов мову, де можна було створити загальні алгоритми (наприклад, сортування STL), які працюють навіть краще, ніж, наприклад, стандартна бібліотека мови C - qsort, завдяки таким функціям, як вбудовування та зв'язування часу компіляції замість вказівників функцій. Стандарт не позначає його як "STL", оскільки він є лише частиною стандартної бібліотеки, але цей термін все ще широко використовується, щоб відрізнити його від решти стандартної бібліотеки (потоки вводу / виводу, інтернаціоналізація, діагностика, підмножина бібліотеки C тощо). [13]

2.3 OpenGL

OpenGL (Open Graphics Library) - це міжмовна, кросплатформна програма інтерфейсу програмування (API) для візуалізації 2D та 3D векторної

графіки. API зазвичай використовується для взаємодії з графічним процесором (GPU) для досягнення апаратно прискореного візуалізації. [14]

Silicon Graphics, Inc. (SGI) розпочала розробку OpenGL у 1991 році та випустила його 30 червня 1992 року; додатки широко використовують її в областях автоматизованого проектування (САПР), віртуальної реальності, наукової візуалізації, візуалізація інформації, імітація польотів та відеоігри. З 2006 року OpenGL управляється неприбутковим технологічним консорціумом Khronos Group.

Специфікація OpenGL описує абстрактний API для малювання 2D та 3D графіки. Незважаючи на те, що API можна повністю реалізувати в програмному забезпеченні, він призначений для реалізації в основному або повністю в апаратному забезпеченні.

API визначається як набір функцій, які може викликати програма-клієнт, поряд із набором іменованих цілочислових констант (наприклад, константа GL_TEXTURE_2D, що відповідає десятковому числу 3553). Хоча визначення функцій за зовнішнім виглядом подібні до визначень мови програмування C, вони не залежать від мови. Таким чином, OpenGL має безліч мовних прив’язок, серед яких найбільш вартими уваги є прив’язка JavaScript WebGL (API, заснований на OpenGL ES 2.0, для 3D-рендерингу з веб-браузера); C-прив’язки WGL, GLX і CGL; прив’язка C, надана iOS; та прив’язки Java та C, надані Android.

Окрім того, що OpenGL не залежить від мови, він також є кросплатформним. Специфікація нічого не говорить про те, як отримати та керувати контекстом OpenGL, залишаючи це деталлю основної системи вікон. З тієї ж причини OpenGL займається суто рендерингом, не надаючи API-інтерфейсів, пов’язаних із введенням, звуком чи вікном.

У OpenGL все знаходиться у тривимірному просторі, але екран або вікно - це 2D-масив пікселів, тому велика частина роботи OpenGL полягає у перетворенні всіх 3D-координат в 2D-пікселі, які вміщуються на вашому

екрані. Процесом перетворення 3D-координат у 2D-пікселі керує графічний конвеєр OpenGL. Графічний конвеєр можна розділити на кілька етапів (рис. 4), де кожен крок вимагає вихідних даних попереднього кроку як вхідних даних. [15] Усі ці кроки є вузькоспеціалізованими (вони мають одну конкретну функцію) і можуть бути легко виконані паралельно. Завдяки своїй паралельній сутності сучасні відеокарти мають тисячі невеликих обробних ядер для швидкої обробки ваших даних у графічному конвеєрі. Ядра обробки запускають невеликі програми на графічному процесорі для кожного кроку конвеєра. Ці невеликі програми називаються шейдерами.

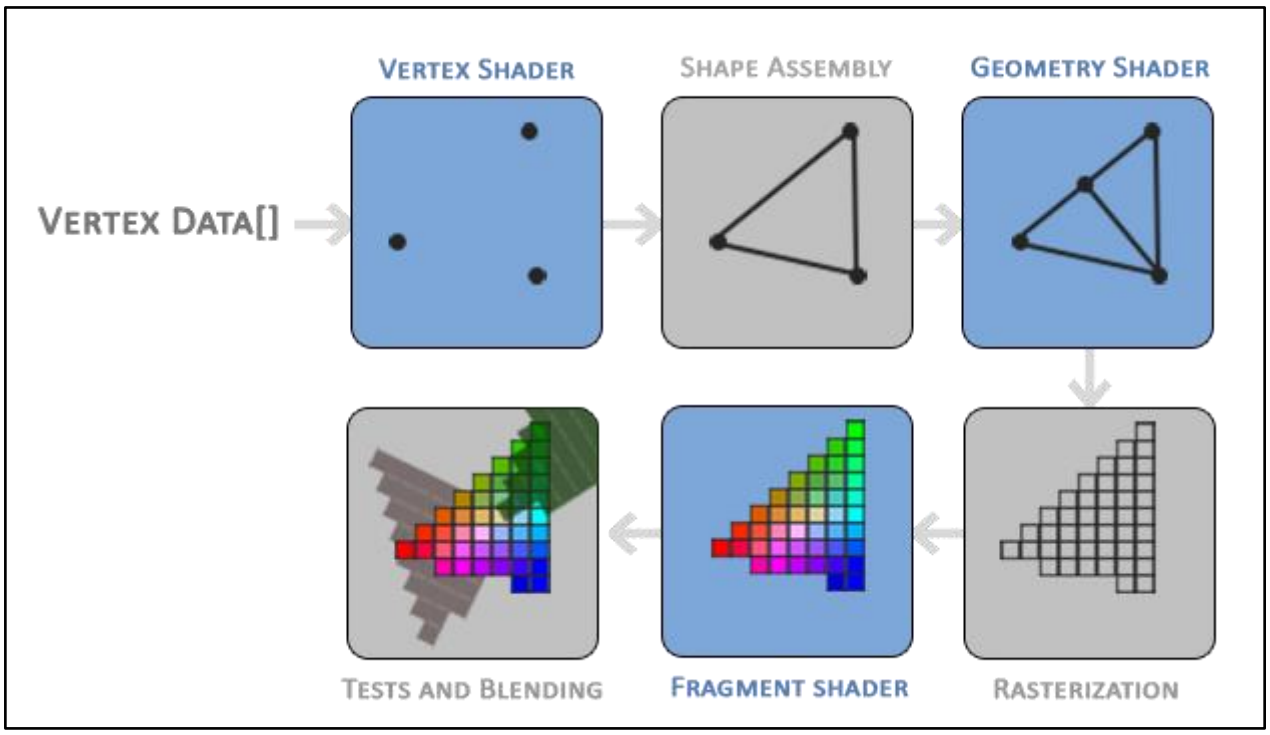

Рисунок 4. Графічний конвеєр OpenGL [15]

Деякі з цих шейдерів можуть бути налаштовані розробником, що дозволяє нам писати власні шейдери для заміни існуючих шейдерів за замовчуванням. Це дає нам набагато чіткіший контроль над певними частинами конвеєра, і оскільки вони працюють на графічному процесорі, вони

також можуть заощадити нам цінний час процесора. Шейдери написані мовою GLSL.

Перша частина конвеєра - це вершинний шейдер, який приймає в якості входу одну вершину. Основною метою вершинного шейдера є перетворення тривимірних координат у різні 3D-координати, він дозволяє виконати базову обробку атрибутів вершини. Перша частина конвеєра - це вершинний шейдер, який приймає в якості входу одну вершину. Основною метою вершинного шейдера є перетворення тривимірних координат у різні 3D-координати, він дозволяє виконати базову обробку атрибутів вершини.

Етап складання примітиву приймає як вхідні дані всі вершини (або вершину, якщо обрано GL_POINTS) з вершинного шейдера, який утворює примітив, і збирає всі точки в заданій примітивній формі; в цьому випадку трикутник.

Результат етапу складання примітивів передається в шейдер геометрії. Шейдер геометрії бере як вхід колекцію вершин, які утворюють примітив, і має здатність генерувати інші фігури, витягуючи нові вершини, утворюючи нові (або інші) примітиви. У рис. 4 він генерує другий трикутник із заданої фігури.

Потім результат геометричного шейдера передається на стадію растеризації, де він відображає результуючі примітиви до відповідних пікселів на кінцевому екрані, що дає можливість для використання шейдера фрагментів. Перед запуском шейдерів фрагментів виконується відсікання. Відсікання відкидає всі фрагменти, що перебувають поза оглядом, збільшуючи продуктивність.

Основною метою шейдера фрагментів є обчислення кінцевого кольору пікселя, і це, як правило, етап, коли виникають усі вдосконалені ефекти OpenGL. Зазвичай шейдер фрагментів містить дані про 3D-сцену, які він може використовувати для обчислення остаточного кольору пікселя (наприклад, джерело світла, тіні, колір світла тощо). Шейдер фрагментів не може змінити

положення фрагмента (x, y). Доступ до сусідніх фрагментів не дозволено. Значення, обчислені шейдером фрагментів, в кінцевому рахунку використовуються для оновлення пам’яті буфера кадрів або текстурної пам’яті, залежно від поточного стану OpenGL.

Після того, як будуть визначені всі відповідні значення кольору, кінцевий об’єкт пройде ще один етап, який називається етапом альфа-тесту та змішування. На цьому етапі перевіряється відповідне значення глибини (і трафарету) фрагмента і використовується, щоб перевірити, чи отриманий фрагмент знаходиться спереду або позаду інших об’єктів, і його слід відкидати відповідно. Етап також перевіряє наявність альфа-значень (альфа-значення визначають непрозорість об’єкта) і відповідно змішує об’єкти. Отже, навіть якщо вихідний колір пікселя обчислюється у шейдері фрагментів, кінцевий колір пікселя все одно може бути чимось зовсім іншим під час рендерінгу декількох трикутників.

Графічний конвеєр є досить складним цілим і містить багато налаштованих частин. Однак майже у всіх випадках нам доводиться працювати лише з вершинним та фрагментним шейдером. Шейдер геометрії є необов’язковим і зазвичай залишається за замовчуванням.

Існує також етап теселяції, який не зображений на рис. 4. Він виконується після вершинного шейдера. Шейдер управління теселяцією викликається для кожної вершини вихідного патча. Кожне виклик може читати атрибути будь-якої вершини у вхідних або вихідних патчах, але може писати лише атрибути для кожної вершини для відповідної вершини вихідного патча. Виклики шейдерів спільно створюють набір окремих патчів атрибутів вихідного патча. Після завершення всіх викликів шейдера управління теселяцією виводиться результат. Виклик шейдера управління теселяцією виконується здебільшого незалежно, з невизначеним відносним порядком виконання. Однак вбудований функціональний бар’єр можна використовувати для управління порядком виконання шляхом синхронізації викликів,

ефективно розподіляючи виконання шейдера управління теселяцією на набір фаз.

У сучасному OpenGL потрібно визначити принаймні власний шейдер вершин і фрагментів (на GPU немає шейдерів вершин / фрагментів за замовчуванням).

2.4 Допоміжні бібліотеки для роботи з OpenGL

GLM (OpenGL Mathematics) - бібліотека для OpenGL, що надає програмісту на C++ структури і функції, що дозволяють використовувати дані для OpenGL. Так як майже всі маніпуляції з трансформацією об’єктів в 3D або 2D просторі виконуються завдяки матрицям, то наявність GLM, яка надає набір оптимізованих функцій для роботи з матрицями 3x3 та 4x4, дуже допомагає. [16]

Одна з особливостей GLM полягає в тому, що його реалізація заснована на специфікації GLSL (OpenGL Shading Language).

Усередині бібліотека розділена на 2 частини:

- перша частина - glm.hpp - містить опис основних типів;
- друга частина - бібліотека розширень ext.hpp - підключає безліч корисних функцій, таких як rotate, translate, scale, lookAt і інші.

Хоча в документації пропонується підключати розширення окремо, вказуючи ту чи іншу бібліотеку. [17]

Приклад використання GLM разом із OpenGL:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
void foo(){
        glm::vec4 Position = glm::vec4(glm:: vec3(0.0f), 1.0f);
```

```
glm::mat4 Model = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f));
Model = glm::rotate(Model,45,0,1,0);
glm::vec4 Transformed = Model * Position; }
```

GLFW - це багатоплатформна бібліотека з відкритим кодом для розробки OpenGL, OpenGL ES та Vulkan на робочому столі. Вона забезпечує простий API для створення вікон, контекстів і поверхонь, отримання вхідних даних та подій. GLFW написана мовою C та підтримує Windows, macOS, X11. [18]

Можливості:

- створює вікно та контекст OpenGL двома викликами функцій;
- підтримка OpenGL, OpenGL ES, Vulkan та пов’язані з ними опції, прапори та розширення;
- підтримка декількох вікон, декількох моніторів, високих DPI;
- підтримка клавіатури, миші, геймпада, введення часу та вікна подій за допомогою опитування або callback-функцій;
- підтримка кодування Unicode.

OpenGL Extension Wrangler Library (GLEW) - це міжплатформна бібліотека написана на мові C/C++ для завантаження розширень для OpenGL. GLEW забезпечує ефективні механізми виконання для визначення того, які розширення OpenGL підтримуються на цільовій платформі. Функціонал ядра та розширення OpenGL представлений в одному заголовковому файлі, який автоматично генерується з офіційного списку розширень. GLEW був протестований на різних операційних системах, включаючи Windows, Linux, Mac OS X, FreeBSD, Irix та Solaris. [19]

2.5 GLSL

OpenGL Shading Language (GLSL) - це мова затінення високого рівня з синтаксисом, заснованим на мові програмування C. Вона була створена

OpenGL ARB (OpenGL Architecture Review Board), щоб надати розробникам більше безпосереднього контролю над графічним конвеєром без необхідності використовувати мову збірки ARB чи мови, що стосується конкретного обладнання. [20]

Деякі переваги використання GLSL:

- Міжплатформна сумісність у багатьох операційних системах, включаючи Linux, macOS та Windows;
- Можливість писати шейдери, які можна використовувати на графічній карті будь-якого постачальника обладнання, що підтримує мову затінення OpenGL;
- Кожен постачальник обладнання включає компілятор GLSL у свій драйвер, що дозволяє кожному постачальнику створювати код, оптимізований для конкретної архітектури графічної карти.

GLSL містить ті самі оператори, що й оператори в C та C++, за винятком вказівників. Подібно до мови програмування C, GLSL підтримує цикли та розгалуження, наприклад: if-else, for, switch тощо. Рекурсія заборонена та перевіряється під час компіляції.

Підтримуються користувацькі функції та надаються вбудовані функції. Виробник відеокарти може оптимізувати вбудовані функції на апаратному рівні. Багато з цих функцій подібні до функцій в математичній бібліотеці мови програмування C, тоді як інші специфічні для графічного програмування. Більшість вбудованих функцій та операторів можуть працювати як на скалярах, так і на векторах (до 4 елементів), для одного або обох операндів. Поширеними вбудованими функціями, які надаються і зазвичай використовуються для графічних цілей, є: mix, smoothstep, normalize, inversesqrt, clamp, length, distance, dot, cross, reflect, refract, min та max. Надаються інші функції, такі як abs, sin, pow тощо, але ці також можуть оперувати векторними величинами, тобто pow(vec3(1.5, 2.0, 2.5), abs(vec3(0.1, -0.2, 0.3))). GLSL підтримує перевантаження функцій (як для вбудованих

функцій, так і для операторів, а також для визначених користувачем функцій), тому може бути кілька визначень функцій з однаковим іменем, що мають різну кількість параметрів або типів параметрів. Кожен з них може мати власний незалежний тип повернення.

GLSL визначає підмножину препроцесора C (CPP) у поєднанні зі своїми спеціальними директивами для визначення версій та розширень OpenGL. Вилучені з CPP частини - це частини, що стосуються імен файлів, таких як #include та __FILE __.

Розширення GL_ARB_shading_language_include реалізує можливість використання #include у вихідному коді, що дозволяє простіше обмін кодом та визначеннями між багатьма шейдерами без додаткової попередньої обробки вручну. Подібні розширення GL_GOOGLE_include_directive та GL_GOOGLE_cpp_style_line_directive існують для використання GLSL з Vulkan.

Шейдери GLSL не є окремими програмами; їм потрібна програма, яка використовує API OpenGL, який доступний на багатьох різних платформах (наприклад, Linux, macOS, Windows). Існують мовні прив’язки для C, C++, C#, JavaScript, Delphi, Java та багатьох інших.

Самі шейдери GLSL - це просто набір рядків, які передаються драйверу постачальника обладнання для компіляції з програми, використовуючи точки входу API OpenGL. Шейдери можна створювати на льоту з програми або читати як текстові файли, але вони повинні надсилатися драйверу у вигляді рядка.

Набір API, що використовуються для компіляції, зв’язку та передачі параметрів програмам GLSL, вказані у трьох розширеннях OpenGL і стали частиною основного OpenGL станом на OpenGL версії 2.0. API розширено за допомогою геометричних шейдерів у OpenGL 3.2, тесселяційних шейдерів у OpenGL 4.0 та обчислювальних шейдерів у OpenGL 4.3. Ці API OpenGL містяться в розширеннях:

- Вершинний шейдер;
- Фрагментний шейдер;
- Об’єктний шейдер;
- Геометричний шейдер;
- Теселяційний шейдер;
- Обчислювальний шейдер;

РОЗДІЛ 3. РЕАЛІЗАЦІЯ

3.1 Технічні вимоги до системи

Загальні вимоги

- система повинна бути абстрагована від платформи, для якої розробляється гра;
- рушій має бути поділений на підсистеми, які повинні бути максимально незалежними один від одного;
- для передачі даних між системами (та можливо всередині систем) має бути створена гнучка та зручна система сигналів (враховуючи переваги та недоліки проаналізованих рушіїв);
- на основі системи компонентів Unity розробити власну;
- розробити систему рендерингу об’єктів, використовуючи OpenGL;
- має бути статичний (або Singleton) клас, який буде керувати роботою рушія;
- кожна система повинна мати статичний (або Singleton) клас, який буде керувати її роботою;
- має бути коректно організоване спільне володіння об’єктами, для уникнення проблеми висячих посилань та витоку пам’яті;
- система повинна розроблятись із використанням ООП та сучасних шаблонів для програмування ігор.

Вимоги до системи сигналів

- система повинна зберігати функції, прив’язані до кожного сигналу;
- функції можуть бути як статичними, так і належати об’єкту. В другому випадку система повинна коректно працювати із не валідними об’єктами;

- система повинна вміти обробляти сигнали натискання клавіш клавіатури та мишки, при чому повинна бути підтримка обробки одночасного натискання кількох клавіш.

Вимоги до системи компонентів

- вся поведінка об’єктів повинна бути описана виключно в компонентах;
- для зберігання класів різних типів має бути створений базовий абстрактний клас, від якого наслідуватимуться всі компоненти;
- для коректного створення та ініціалізації компонентів конструктори класів мають бути приватними, а конструювання об’єктів повинно відбуватись або за допомогою статичної функції, або за допомогою шаблону Builder. [21]

Вимоги до системи фізики

- рух об’єктів повинен контролюватись системою фізики;
- має бути реалізована перевірка зіткнення об’єктів;
- має бути можливість задавати які об’єкти будуть перевірятись при зіткненні, а які ні;
- мають бути різні типи та схеми зіткнень;
- система повинна повідомляти інші системи про те, що відбулось зіткнення і надавати інформацію про об’єкти, які зіткнулись.

Вимоги до системи рендерингу

- повинна бути підтримка завантаження та відображення текстур, матеріалів;
- шейдери повинні коректно обчислювати позицію об’єкта в світі;
- має бути камера з ортогональною проекцією виду, яку можна буде прив’язувати до певного ігрового об’єкта, який керуватиме її переміщенням;
- система має бути оптимізована (не витрачати ресурси на рендеринг об’єктів, які знаходяться за межами камери).

3.2 Загальний огляд та принцип роботи

3.2.1 Класи Engine, Window, Scene

Під час розробки рушія було вирішено розділити всю систему на декілька згрупованих за призначенням частин:

- ядро рушія;
- система подій;
- система фізики;
- система рендеру;
- компоненти.

Для забезпечення безпечного використання пам’яті, зменшення кількості помилок, пов’язаних з “сирими” вказівниками, було вирішено використовувати розумні вказівники. Всі об’єкти-власники мають тип std::shared_ptr, а всі інші посилання на ті самі об’єкти мають тип std::weak_ptr. Std::weak_ptr є також типом повернення таких об’єктів. Для того, щоб уникнути ситуації захоплення вказівника this двома різними розумними вказівниками, всі класи, які можуть знаходитись в таких вказівниках, наслідуються від класу std::enable_shared-from_this<T> та мають приватний конструктор, створення об’єктів реалізоване з використанням статичної функції.

Основні класи системи, такі як, наприклад, PhysicsSystem або Engine було вирішено зробити статичними, а не за допомогою шаблону Singleton [21], для скорочення коду під час їх використання. Можливим був також варіант позбутись класів, а всі функції та дані перенести в відповідні простори імен, але при такому підході був би втрачений поділ коду по рівню доступу (private, protected, public). Крім того, ще можливим було б конфлікт імен, при використанні двох чи більше просторів імен, якби вони містили однакові сигнатури. Тому вибір було зроблено на користь статичних класів.

Центральним модулем рушія є Engine (рис. 5). Він відповідає за налаштування коректної роботи гри, а саме:

- ініціалізацію інших модулів;
- створення і налаштування вікна;
- створення і налаштування сцени;
- завантаження та звільнення ресурсів.

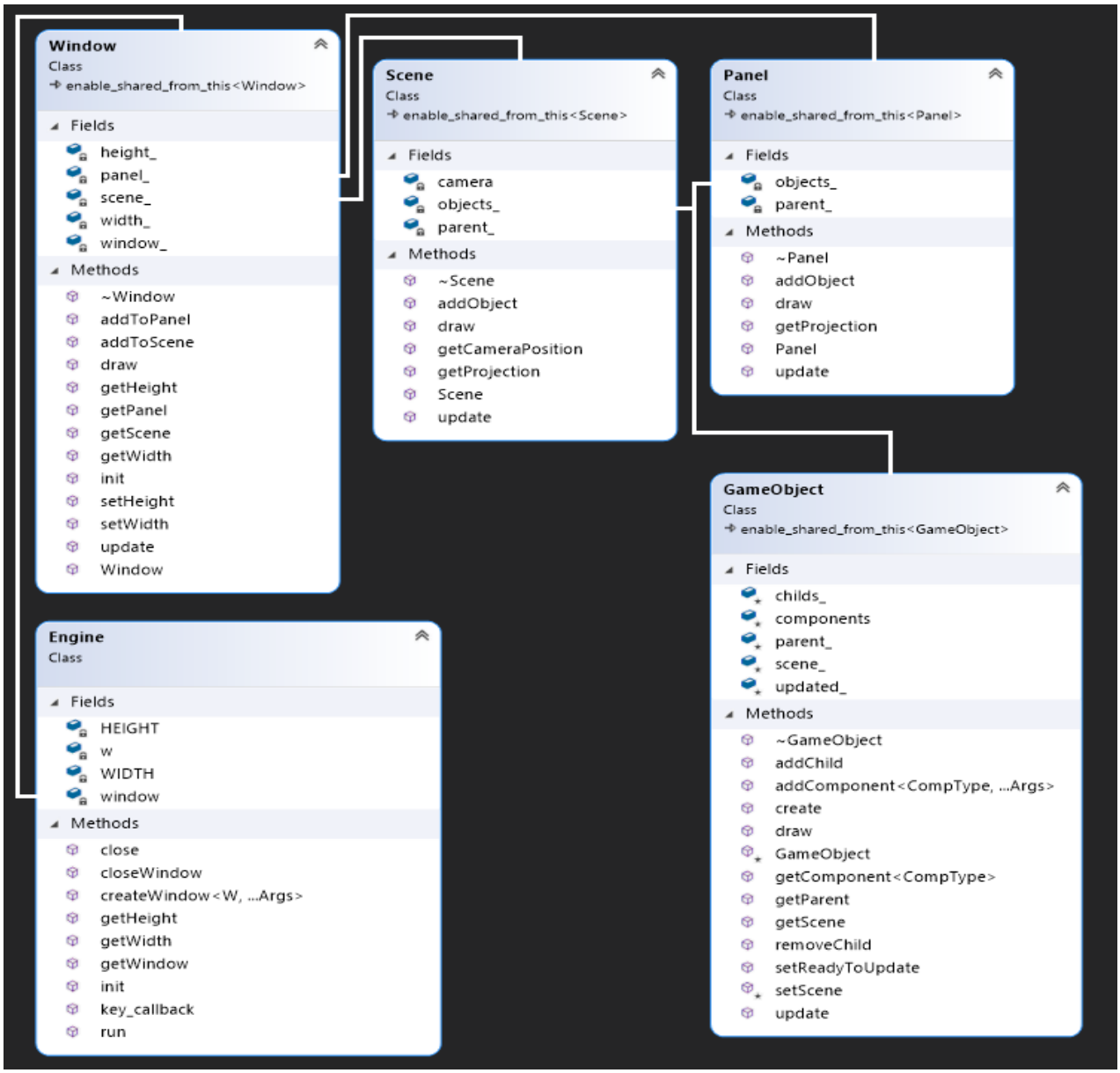

Рисунок 5. Діаграма класів ядра рушія

Ядро складається з таких класів:

- Engine — головний клас рушія. Керує головним циклом та роботою всіх інших модулів. Є статичним. Володіє вікном програми;
- Window — клас програмного вікна. Містить в собі об'єкт GLFWwindow (стандартний клас, з яким GLFW зв'язує події клавіатури та мишки). Також містить об'єкти сцени та панелі користувацького інтерфейсу;
- Scene — клас, який зберігає всі об'єкти, присутні в грі та відповідає за їх оновлення. Містить камеру, яка зберігається у вигляді std::unique_ptr, для того щоб інші об'єкти системи не могли впливати на роботу камери;
- Panel — клас, який містить всі об'єкти інтерфейсу. Їх позиція вказується відносно положення вікна і не змінюється при переміщенні камери;
- GameObject — основний клас ігрових об'єктів. Являє собою контейнер для компонентів, які описують його логіку, та дочірніх ігрових об'єктів. Сам по собі не містить ніякої поведінки.

Під час створення нової гри перш за все повинен бути ініціалізований сам рушій. В основному це робиться в функції main(). Користувачу потрібно виконати такі кроки:

1. Створити вікно програми за допомогою функції шаблонної Engine::createWindow(), якій треба передати тип вікна, що буде створено. Вона створить як і об'єкт GLFWwindow, так і стандартний об'єкт вікна рушія. Можна створити свій власний клас вікна, який повинен наслідувати стандартний. Єдина віртуальна функція — update(). Якщо вона перевизначається, обов'язково в тілі функції має бути доданий виклик стандартної реалізації цієї функції.
2. Запустити процес ініціалізації рушія та його підсистем. Для цього використовується функція Engine::init().
3. Завантажити рівень гри і всі необхідні об'єкти. Цей крок є опціональним, так як вибір рівня і його завантаження може відбуватись залежно від вибору гравця вже під час гри.

4. Викликати функцію Engine::run(), яка запускає основний цикл гри, де обробляються події клавіатури та мишки, обробка фізики, оновлення всіх об’єктів та їх рендеринг.

Під час процесу ініціалізації (рис. 6) рушій передає GLFW об’єкт вікна, до якого будуть прив’язані обробки подій, ініціалізує по черзі всі підсистеми, прив’язується подія натискання кнопки ESC до функції класу Engine. Ця кнопка відповідає за закриття вікна і зупинку рушія. Останньою йде ініціалізація GLEW, бібліотеки, яка спрощує запити і завантаження функції OpenGL для кожної платформи.

```
void Engine::init() {
    glfwMakeContextCurrent(window);
    PhysicsSystem::init();
    EventQueue::init(window);
    RenderSystem::init(window);

    EventQueue::addFunctionForKey(GLFW_KEY_ESCAPE, function: { closeWindow });

    // Set this to true so GLEW knows to use a modern approach to retrieving function pointers and extensions
    glewExperimental = GL_TRUE;
    // Initialize GLEW to setup the OpenGL Function pointers
    glewInit();

}
```

Рисунок 6. Функція ініціалізації рушія

Клас вікна містить сцену (на якій відбувається весь процес гри) та панель графічного інтерфейсу користувача. При створенні вікна потрібно обов’язково вказати його ширину та висоту, які потім передаються дочірнім елементам та використовуються для задання області рендерингу. Після створення, об’єкт вікна має бути ініціалізованим, саме тоді створюються дочірні сцена та панель. Такий підхід зумовлений використанням класу std::enable_shared_from_this<T>, який не дозволяє отримати розумний вказівник в конструкторі, так як повинен бути вже хоча б один такий вказівник.

Клас вікна має функції для динамічного додавання та видалення об’єктів на сцену та панель. Також в стандартному класі оголошені функції update() (для оновлення дочірніх об’єктів) та draw() (для рендерингу дочірніх об’єктів), при чому update() є віртуальною, тобто її можна перевизначити в класі-нащадку, додавши необхідну логіку.

Щоб вказати рушієві, що треба використовувати для створення вікна саме клас-нащадок, треба вказати потрібний тип для функції createWindow<T>().

Класи Scene та Panel являють собою контейнери для ігрових об’єктів. В них присутні відповідні функції для додавання та видалення об’єктів. Також є аналогічні функціям класу Window::update() та draw().

3.2.2 GameObject

GameObject один із головних класів рушія, який являє собою контейнер для дочірніх об’єктів та компонентів, які описують його поведінку. Єдиний клас, яким може бути представлено ігровий об’єкт на сцені та елемент інтерфейсу вікна.

Конструктор класу закритий для доступу іншими класами, так як процес створення не тривіальний. Натомість для створення об’єкта наявна статична функція create<T>(), яка приймає батьківський об’єкт (якщо такий наявний, за замовчуванням — nullptr) та позицію об’єкта в сцені (за замовчуванням це початкова точка координат). В самій функції створюється розумник вказівник для об’єкта, що забезпечує унікальне володіння над this та додається компонент Location.

Додавання самих компонентів відбувається з допомогою шаблонної функції create<T>(): параметр шаблону позначає тип компонента, який має бути створений. Аргументи конструктора компонента також передаються у функцію. Під час створення нового компонента перевіряється чи немає вже

такого компонента в цього об’єкта, чи немає інших компонентів деякого підтипу (Collider, Drawable) і, якщо нічого не було знайдено, то буде такий компонент буде створено. Передані параметри передаються в конструктор за допомогою “perfect forwarding”. Новостворений компонент відразу зберігається в контейнері, а потім приводиться то типу, який було передано як параметр шаблону (за допомогою функції std::dynamic_pointer_cast<T>()) і повертається з функції. Це дозволяє відразу налаштовувати компонент без додатково звернення до функції пошуку.

Шаблонні функції пошуку та видалення працюють схожим способом: за допомогою range-based for цикл проходить по всьому контейнеру, кожен із компонентів за допомогою функції std::dynamic_pointer_cast<T>() приводиться до переданого типу, якщо приведення неможливе, то буде повернений nullptr, що можна перевірити в умовному операторі. Якщо компонент було знайдено, то він або повертається, або видаляється, залежно від функції.

Ігрові об’єкти також мають дещо змінений процес оновлення. В класі є булева змінна, яка позначає чи був оновлений цей об’єкт на поточному кадрі. Якщо вона дорівнює false, то функція update() виконується без перешкод, інакше її виконання відразу переривається. Це зроблено для того, щоб об’єкт не оновлювався декілька разів протягом одного кадру, що можливо при деяких умовах. На початку кожного кадру для всіх об’єктів викликається функція setReadyToUpdate(), після чого йде безпосереднє їх оновлення.

3.3 Система сигналів

Однією з основних завдань під час розробки рушія була зберегти його модульну структуру з мінімальними (бажано нульовими) залежностями модулів один від одного. Для передачі інформації між підсистемами було вирішено створити спеціальний механізм зв’язку — сигнали (рис. 7).

Основним класом є Signal, допоміжний — FunctionWrapper. Для обробки подій клавіатури та мишки створено окремий клас — EventQueue.

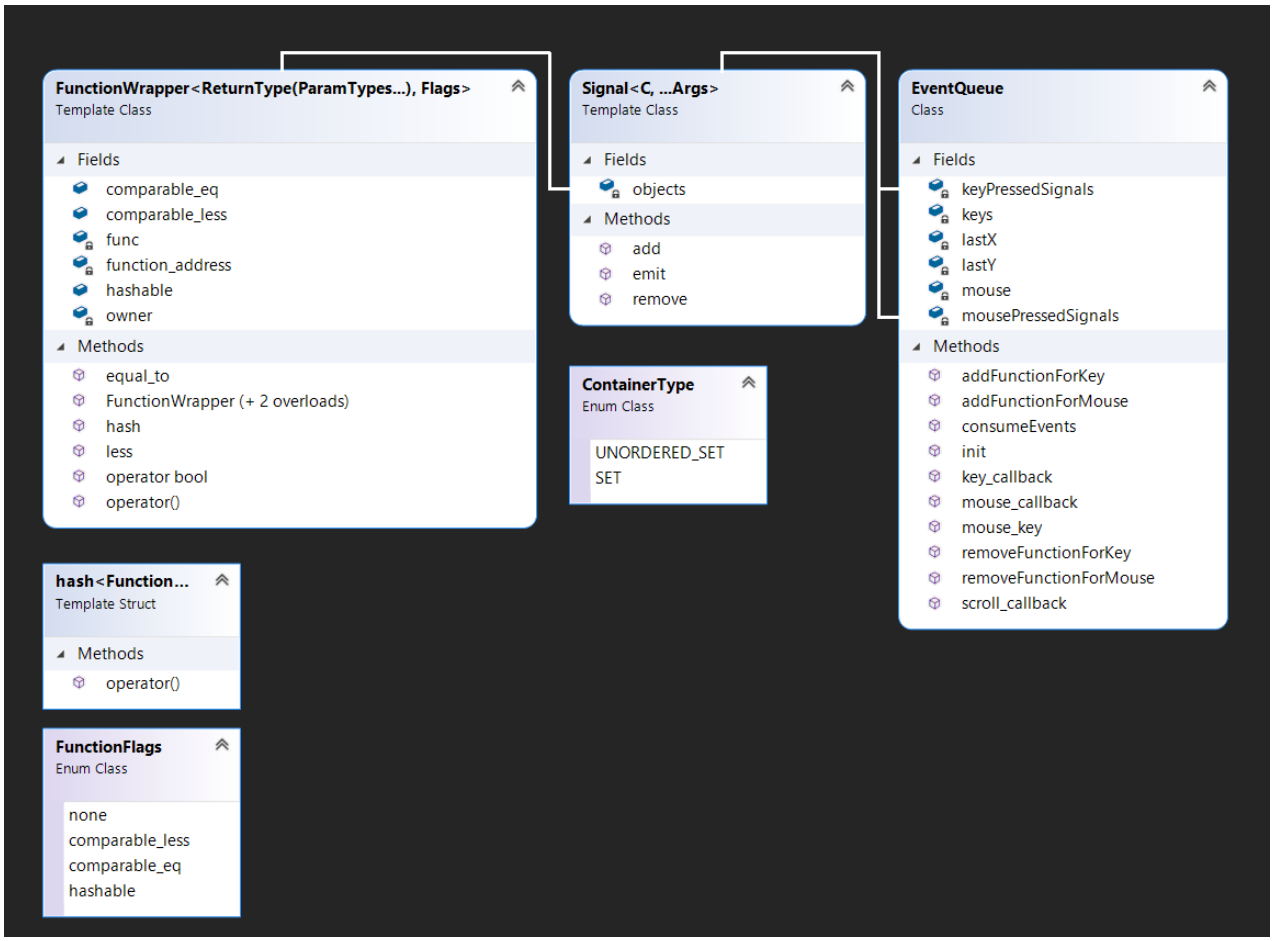

Рисунок 7. Діаграма класів системи сигналів

3.3.1 Signal

Клас Signal повинен вирішити проблему зв’язку різних модулів системи. За основу принципу його роботи взято шаблон дизайну — Observer [21]. Шаблонні параметри, які потрібно вказати при створенні сигналу — це типи аргументів, які передаються цим сигналом, а відповідно і типи аргументів функції, яка може очікувати сигнал. Результат виконання таких функцій ніде не використовується, тому сигнал може передаватись лише функціям з типом повернення void.

Для збереження функцій в сигналі використовуються дві варіації стандартного контейнеру “множина”: std::set та std::unordered_set. Перевагою його використання є унікальність елементів, тому зникає небезпека додавання однієї тієї самої функції більше одного разу, що могло б мати непередбачувані наслідки. Сортувати функції в більшості випадків не потрібно і немає користі, тому за замовчуванням обрано контейнер std::unordered_set. Тип контейнера теж може бути вказаний як параметр шаблону класу Signal. Це створює проблему, так як std::set теж шаблонний клас, а отже повинна вказуватись повна специфікація типу. А параметр для std::set залежить від переданих типів в Signal, яких може бути змінна кількість. Для їх передачі використовується синтаксична конструкція мови C++ - variadic template. Але вони можуть вказуватись лише останніми в списку параметрів та не можуть використовуватись до моменту свого фактичного оголошення. Тому тип контейнеру передається у вигляді константи, яка потім в коді на етапі компіляції використовується для визначення фактичного типу контейнера. Визначення відбувається за допомогою функції std::conditional_t з type_traits. Всі необхідні проміжні типи зберігаються за допомогою оголошення синонімів типів (рис. 8).

```
enum class ContainerType{
    UNORDERED_SET,
    SET
};

template<ContainerType C, typename ...Args>
class Signal{
public:
    using CallableObject = void(Args...);
    using hashable_function_t = FunctionWrapper<CallableObject,
        FunctionFlags::hashable | FunctionFlags::comparable_eq | FunctionFlags::comparable_less>;
    using Container = std::conditional_t<C == ContainerType::UNORDERED_SET,
        std::unordered_set<hashable_function_t>, std::set<hashable_function_t>>;

private:
    Container objects;
```

Рисунок 8. Вивід типів класу Signal

Інша проблема, яка виникла — зберігання функції в контейнері. Сама по собі функція в C++ - вказівник. Для вказівників не визначені оператори “менше”, “рівно”, та відсутнє обчислення хешу, які потрібні для зберігання в множину. Стандартна бібліотека C++ має реалізовану обгортку над функціями і лямбдами — std::function, але і вона не має перелічених операцій. Тому вирішенням проблеми було лише створення власного класу, який би мав всі потрібні функції — FunctionWrapper. В кінцевому варіанті класу Signal в контейнері зберігаються об’єкти цього типу.

Клас також має функції для додавання і видалення функцій в контейнер, функція сповіщення, коли трапляється якась подія, та функція очищення контейнеру від функцій, які належать якому об’єкту та цей об’єкт вже був видалений.

3.3.2 FunctionWrapper

FunctionWrapper - шаблонний клас обгортки функції, який має всі операції порівняння та хешування. При його створенні мають бути вказані тип повернення функції, типи її аргументів та опції, які включають кожну з трьох перелічених вище функцій. Тобто можна налаштувати функцію для зберігання в звичайному std::set, але забрати можливість збереження в std::unordered_set.

Клас має два конструктори: один для збереження звичайних функцій, другий для збереження функцій, які належать якому об’єкту/класу. Якщо передається функція класу, то передається і об’єкт, з якого вона буде викликана, який потім зберігається. А інакше зберігається адреса самої функції.

Для забезпечення коректного зберігання функцій обох типів було використано std::function. Але напряму в нього не можна зберегти функцію, яка була передана в конструктор, тому спочатку створюється лямбда, яка просто викликає передану функцію, а потім ця лямбда зберігається в

std::function. Для оптимізації передача параметрів в лямду з допомогою rvalue (так як більше ніде ці параметри не потрібні), а для передачі з лямбди у функцію використовується так званий “perfect forwarding” (рис. 9). Але виникла проблема при відсутності переданих параметрів (функція нічого не приймає). Для її вирішення було використано constexpr можливості C++, а саме оператор if constexpr, який може бути обчислений під час компіляції. В його умові відбувається перевірка на кількість параметрів в переданому паку, якщо вони відсутні, то і відсутня функція std::forward<T> (рис. 9).

```
FunctionWrapper(ReturnType(*function)(ParamTypes...))
{
    if constexpr(sizeof...(ParamTypes) == 0){
        func = [function]() -> ReturnType
        {
            return function();
        };
    }else {
        func = [function](ParamTypes&&... args) -> ReturnType
        {
            return function(std::forward<ParamTypes>(args)...);
        };
    }

    function_address = (size_t) function;
}
```

Рисунок 9. Приклад конструктора для статичної функції

Для підтримки хешування C++ вимагає створення окремої шаблонної структури std::hash<T>. В структурі мають бути визначені типи argument_type (тип об’єкта, який хешується) та result_type (тип хешу). Також для власне хешування має бути перевантажений operator(). Приклад структури хешування, створеної для класу FunctionWrapper (рис. 10). Для зручності обчислення хешу реалізовано в самому класі FunctionWrapper.

```
namespace std
{
    <T,FunctionFlags Flags> struct hash<FunctionWrapper<T, Flags>>
    {
        using argument_type [[maybe_unused]] = FunctionWrapper<T, Flags>;
        using result_type [[maybe_unused]] = std::size_t;
        result_type operator()(const argument_type &f) const
        {
            return f.hash();
        }
    };
}
```

Рисунок 10. Реалізація структури std::hash<T> для класу FunctionWrapper

3.3.3 EventQueue

EventQueue — статичний клас, який містить сигнали для кожної кнопки клавіатури та мишки. Код кнопки та відповідний сигнал зберігаються у вигляді словника. Сигнал для кнопки клавіатури не передає ніякої інформації пов'язаним функціям, а сигнал для кнопки мишки передає координати у момент натискання.

Для того, щоб система коректно працювала вона має бути ініціалізована при запуску рушія. Під час ініціалізації зв'язуються події класу GLFWwindow з функціями системи. Потім створюються сигнали та зв'язуються з відповідними кнопками.

Кожна з функцій, яка приймає від GLFW події, не передає подію напряму в відповідний сигнал. Це викликало б проблему можливості обробки лише одного натискання на кадр. Вирішення цієї проблеми можливе завдяки використання шаблону — Event Queue [21]. Його суть полягає в зберіганні стану кожної кнопки і подальшої обробки всіх подій за один раз.

В класі оголошені функції для додавання та видалення функцій до сигналу для відповідної кнопки. Таким чином можна, наприклад, тимчасово забирати в гравця можливість керувати своїм персонажем.

3.4 Компоненти

Головною особливістю системи є компонентна система (рис. 11). Вся поведінка кожного об’єкту в грі міститься в наборі декількох його компонентів.

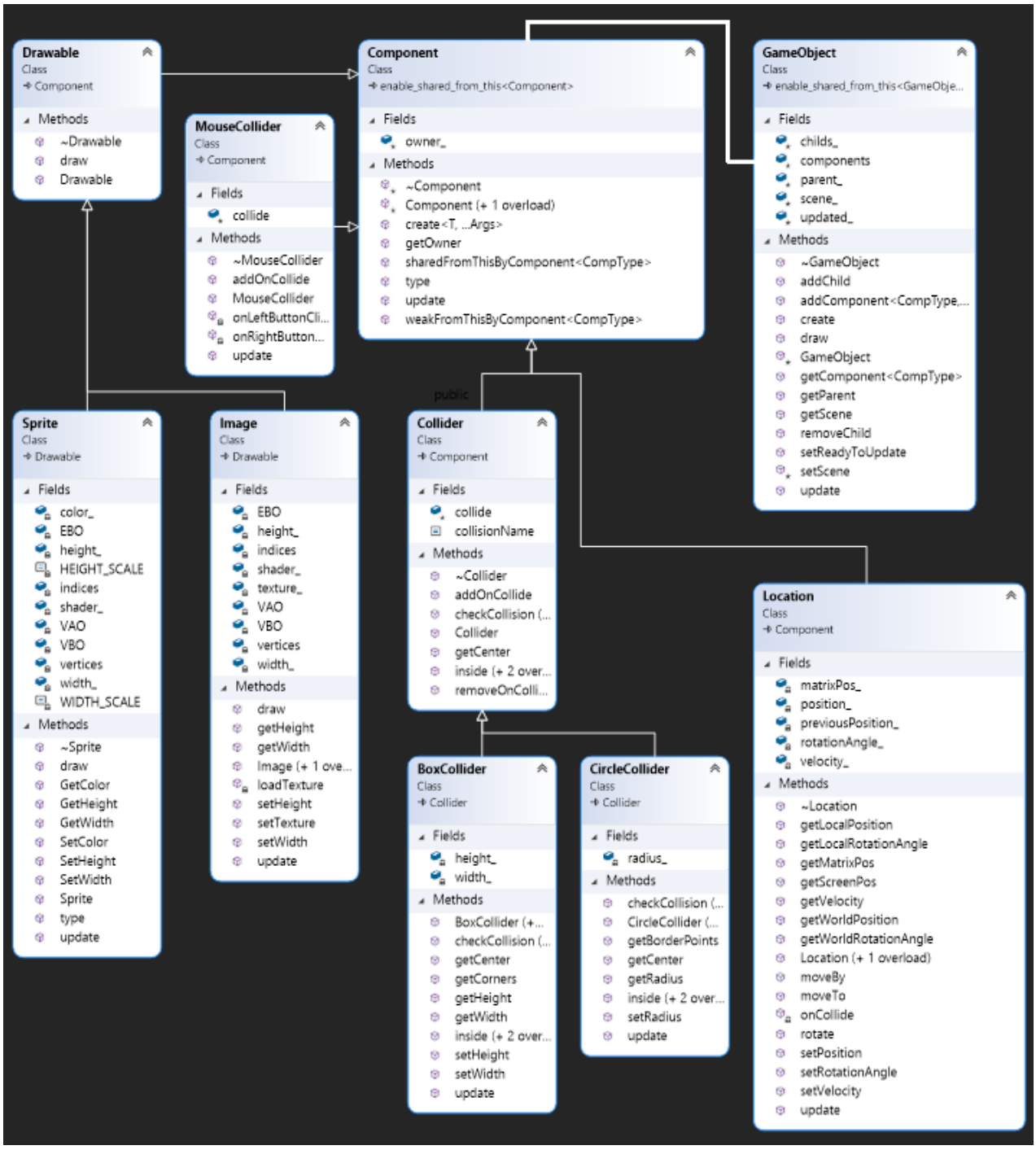

Рисунок 11. Діаграма класів системи компонентів

Приклади наявних компонентів:

- CircleCollider, BoxCollider;
- Location;
- Image;
- MouseCollider та інші.

Щоб мати можливість зберігати компоненти всіх видів в одному контейнері, було створено абстрактний клас Component, який повинні наслідувати інші. По суті, клас являється компонентом, якщо він наслідується від Component, який в свою чергу наслідується від std::enable_shared_from_this, що дозволяє краще контролювати об’єкт.

При створенні будь-якого компоненту має бути вказаний об’єкт, якому він буде належати. Так само як і в GameObject, конструктор не може бути викликаний іншими класами, а для створення компонента наявна спеціальна шаблонна функція create<T>(), в яку також передаються параметри конструктора. В тілі функції створюється розумний вказівник із об’єктом, який відразу ініціалізується. Дефолтний конструктор без параметрів видалений, так як компонент без ігрового об’єкта не може існувати.

В класі наявні спеціальні допоміжні шаблонні функції для отримання розумних вказівників: weakFromThisByComponent<T>() та sharedFromThisByComponent<T>(), які для приведення об’єкта до коректного типу використовують функцію std::dynamic_pointer_cast<T>().

Кожен із класів, які наслідуються від Component, повинні перевизначити чисто віртуальні функції init() та update(), так як їх реалізація повністю залежить від типу компонента.

Головний компонент, який повинен бути у всіх ігрових об’єктів та додається автоматично при їх створенні - Location. Містить позицію об’єкта відносно батьківського (або сцени, якщо батько відсутній), швидкість, попередню позицію (для системи фізики) та кут повороту.

Наявні всі функції, через які можна задати або отримати перелічені вище дані, у відносному або абсолютному вигляді.

Якщо компоненту задана швидкість, то процес руху буде супроводжуватись перевірками на колізію. Якщо потрібно просто перемістити об’єкт в задану точку або на задану відстань, незважаючи на можливу колізію, то для цього існують спеціальні функції moveTo() та moveBy().

3.5 Система фізики

Для обробки фізики було створено ряд класів (рис. 12), які пов’язані статичним класом PhysicsSystem. В ньому є контейнер для всіх компонентів типу Collider, які додаються автоматично при створенні.

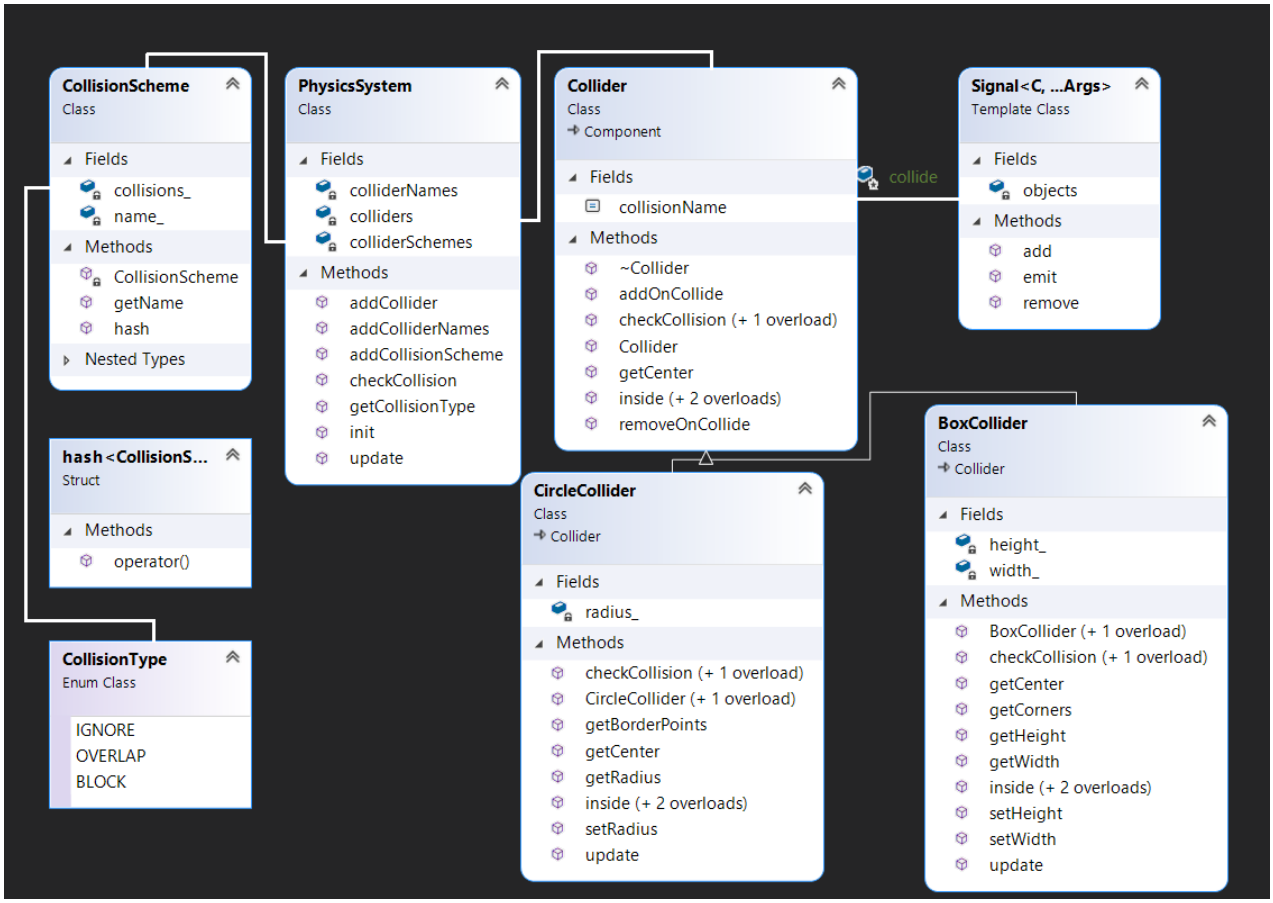

Рисунок 12. Діаграма класів системи фізики

Перевірка колізій відбувається після кожного переміщення будь-якого об’єкта, а не один раз на кадр, що не є оптимальним. З іншого боку, для оптимізації було введено поняття “тип колізії” та “схема колізій”. Типів колізії існує три:

- ignore — колізія між об’єктами не перевіряється;
- overlap — колізія тільки посилає сигнал, про те що вона відбулась;
- block — колізія посилає сигнал і зупиняє об’єкт.

Тип колізії між об’єктами з різними власними типами вираховується так як наведено на рис. 13.

	Ignore	Overlap	Block
Ignore	Ignore	Ignore	Ignore
Overlap	Ignore	Overlap	Overlap
Block	Ignore	Overlap	Block

Рисунок 13. Типи колізії

Схеми колізії необхідні для того, щоб розділити об’єкти на групи. Наприклад, всі статичні об’єкти, які ніколи не будуть рухатись під дією фізики, віднесені до групи “static”, а всі об’єкти, які можуть рухатись попередньо віднесені до групи “dynamic”. Кожній групі вказується, який тип колізії вона матиме з індивідуально іншими групами. Наприклад, “static” блокує всі наявні групи, а “dynamic” сама з собою має тип “overlap”. Якщо тип колізії для групи не вказується, то за замовчуванням він буде “block”.

Групи колізії можуть бути не лише стандартні, користувач також може створювати власні групи, їх кількість не обмежена. Для цього був створений спеціальний клас Builder. Для створення нової групи має бути задана хоча б її назва. Спеціальна функція addCollisionType() приймає назву групи, з якою буде колізія, та її тип. Так наприклад, можна створити групу для об'єктів типу "рослинність", з якою гравець не матиме колізії, а вона сама матиме вплив фізичної системи.

Для зберігання схем колізії в класі PhysicsSystem найкращим контейнером виявився std::unordered_set, але він вимагає наявності функцій рівності та хешування. Тому для класу CollisionScheme була створена структура std::hash<T>, в якій задано обов'язкові типи та визначено operator().

Для додавання класу в систему фізики було створено клас абстрактний Collider, та його нащадки: BoxCollider та CircleCollider.

Клас Collider має набір чистих віртуальних функцій для виявлення колізії з двома наявними реалізаціями:

- bool checkCollision(const std::weak_ptr<BoxCollider>& other);
- bool checkCollision(const std::weak_ptr<CircleCollider>& other)

Також в класі є аналогічні функції для перевірки чи є один об'єкт всередині іншого, та функція, яка перевіряє чи знаходиться передана точка всередині об'єкта. Всі ці функції реалізуються в нащадках залежно від їх типу.

Для повідомлення системи про наявну колізію клас містить спеціальний сигнал, який передає об'єкт, з яким відбулась колізія. Для додавання або видалення функції в сигнал існують спеціальні функції. При створенні будь-якого з нащадків класу в сигнал автоматично додається функція компоненти Location::onCollide(), яка зупинить об'єкт, якщо тип колізії "block".

3.6 Система рендерингу

Для відображення об’єкта на екран потрібно передати шейдерам координати його вершин, колір, текстуру, інформацію про світло тощо. Роботу з цими операціями було винесено в окрему систему (рис. 14).

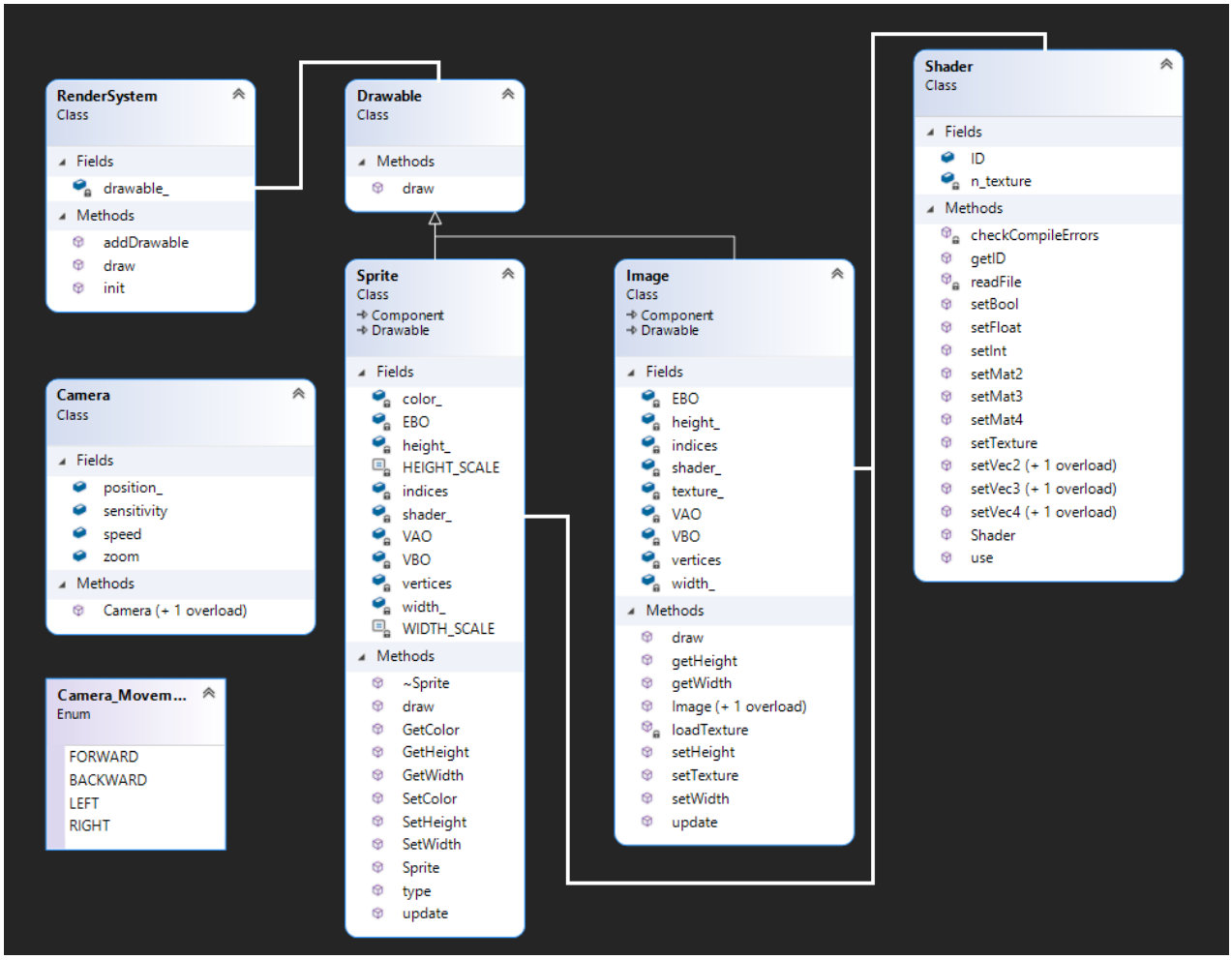

Рисунок 14. Діаграма класів системи рендерингу

3.6.1 Drawable

Абстрактний клас Drawable потрібний для визначення компоненти як такої, що буде відображена на екрані. При чому кожен об’єкт може мати тільки один такий компонент. Кожен клас, який наслідується від Drawable повинен перевизначати чисту віртуальну функцію draw(). Стандартних таких існує 2:

- Sprite — використовується для заливки об’єкта фоном;

- Image — використовується для відображення текстур.

Принцип визначення області, яка буде відображена, в цих двох компонентів однаковий. OpenGL може працювати тільки з трикутниками, тому навіть якщо програма повинна відобразити квадрат, він повинен складатись з двох трикутників. Є два способи задання таких трикутників.

В першому вершини трикутників задаються так:

```
GLfloat vertices[] = {
        0.5f,  0.5f, 0.0f,  // Top Right
        0.5f, -0.5f, 0.0f,  // Bottom Right
        -0.5f,  0.5f, 0.0f,  // Top Left
        -0.5f, -0.5f, 0.0f,  // Bottom Left
        -0.5f,  0.5f, 0.0f,  // Top Left
        0.5f, -0.5f, 0.0f  // Bottom Right
};
```

Позиції вершин трикутників розміщуються послідовно, тобто вершини з першої по третю формують один трикутник, а з четвертої по шосту — другий.

Центр квадрата розміщується в точці (0,0,0), а його висота і ширина задані рівними одиниці. При рендерингу ці параметри змінюються відповідно до заданих. Оскільки OpenGL працює тільки з тривимірним простором, то потрібно вказати і z координату, яка буде рівною нулю. Відповідно глибина трикутників буде однаковою і від буде виглядати двовимірним.

Після визначення вершинних даних потрібно передати їх в перший етап графічного конвеєра: в вершинний шейдер. Це робиться в такий спосіб: виділяється пам’ять на GPU, куди будуть збережені наші вершинні дані, далі потрібно вказати OpenGL як він повинен інтерпретувати передані йому дані і вказати GPU кількість переданих даних. Потім вершинний шейдер обробить таку кількість вершин, скільки йому повідомлено.

Керування цією пам’яттю здійснюється через, так звані, об’єкти вершинного буфера (vertex buffer objects (VBO)) як на рисунку 15, які можуть зберігати велику кількість вершин в пам’яті GPU. Перевага використання таких об’єктів буфера, в тому що можна посилати в відеокарту велику кількість наборів даних за один раз, без необхідності відправляти по одній вершині за раз. Передача даних з CPU на GPU досить повільна, тому краще намагатися відправляти якомога більше даних за один раз. Але як тільки дані виявляться в GPU, вершинний шейдер отримає їх практично миттєво. [15]

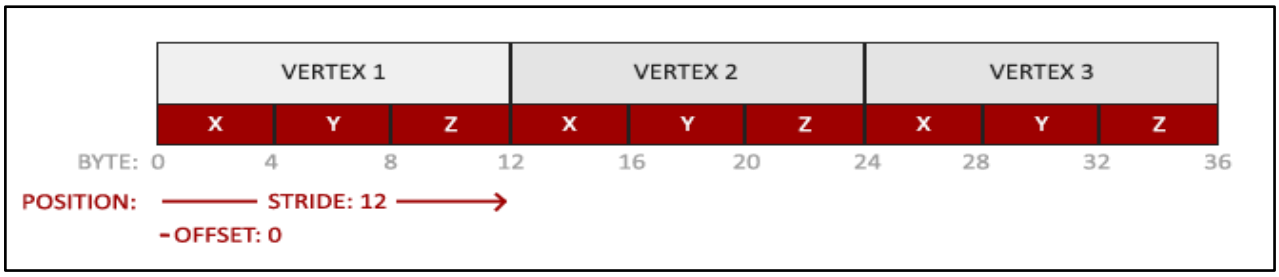

Рисунок 15. Формат вершинного буфера [15]

- Інформація про позицію зберігається в 32 бітному (4 байта) значенні з плаваючою точкою;
- Кожна позиція формується з 3 значень;
- Не існує ніякого роздільника між наборами з 3 значень. Такий буфер називається щільно упакованим;
- Перше значення в переданих даних - це початок буфера.

Другий спосіб передачі даних — використання element buffer objects (EBO). Він дозволяє вказувати не всі вершини трикутників, а лише вершини квадрата, а потім вказати яка з них до яких трикутників відноситься. EBO - це буфер, на кшталт VBO, але він зберігає індекси, які OpenGL використовує, щоб вирішити якусь вершину намалювати. Це називається рендеринг за індексами (indexed drawing). Масив вершин при такому підході задається так:

```
GLfloat vertices[] = {
```

```
        0.5f,  0.5f, 0.0f,  // Top Right
        0.5f, -0.5f, 0.0f,  // Bottom Right
        -0.5f, -0.5f, 0.0f,  // Bottom Left
        -0.5f,  0.5f, 0.0f   // Top Left
};
GLuint indices[] = {
        0, 1, 3,  // First Triangle
        1, 2, 3   // Second Triangle
};
```

Подальше налаштування майже таке саме як і для першого способу, з тією лише різницею, що тепер потрібно враховувати також і другий масив. Для розробки компонентів системи було використано другий спосіб.

Процес рендерингу спрайта та картинки відрізняється мінімально. Спочатку треба вирахувати абсолютні координати центру об’єкта, потім із сцени отримати матрицю проєкції, після чого створити матрицю моделі, транслювати її на позицію центра об’єкта, розширити до потрібних значень ширини та висоти і повернути на задану кількість градусів (у абсолютній величині). Далі отримані матриці передаються в шейдерну програму, а також колір (для спрайтів) або текстура (для картин). Останнім іде безпосередньо процес виклику функції рендерингу.

3.6.2 Шейдери

В системі було використано два типи шейдерів: вершинний та фрагментний, по два на кожен тип компонента Drawable (рис. 16 — 19).

```
#version 330 core
layout (location = 0) in vec3 position;

uniform mat4 model;
uniform mat4 projection;

void main()
{
    gl_Position = projection * model * vec4(position, 1.0f);
}
```

Рисунок 16. Вершинний шейдер для Sprite

```
#version 330 core

uniform vec3 color;

void main() {
    gl_FragColor = vec4(color, 1.0f);
}
```

Рисунок 17. Фрагментний шейдер для Sprite

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texture;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 projection;

void main() {
    gl_Position = projection * model * vec4(position, 1.0f);
    TexCoords = texture;
}
```

Рисунок 18. Вершинний шейдер для Image

```
#version 330 core
in vec2 TexCoords;

uniform sampler2D tex;

void main() {
    gl_FragColor = vec4(vec3(texture(tex, TexCoords)), 1.f);
}
```

Рисунок 19. Фрагментний шейдер для Image

Обидва вершинні шейдери приймають координати вершини, але в шейдері з текстурами присутні ще й текстурні координати, які потрібні, щоб визначити яка саме точка текстури відповідає пікселю об’єкта. В тілі функції main відбувається множення отриманих раніше матриць та координат вершини. Отриманий результат буде координатою у NDC (normalized device coordinates).

Після того, як вершинні координати будуть оброблені в вершинних шейдери, вони повинні бути нормалізовані в NDC, який представляє з себе маленький простір, де x, y і z координати знаходяться в проміжку від -1.0 до 1.0. Будь-які координати, які виходять за цю межу будуть відкинуті і не відобразяться на екрані.

Отримані NDC координати будуть потім перетворені в координати екранного простору через Viewport з використанням даних, наданих через виклик glViewport(). Множення матриць якраз і виконує операцію нормалізації. Координати екранного простору потім трансформуються у фрагменти і подаються на вхід фрагментного шейдеру.

Для роботи із шейдерними програмами було створено клас Shader. Йому в конструктор передаються шляхи до розміщення текстових файлів із вершинним, фрагментним та (опціонально) геометричним шейдерами. Далі відбувається зчитування необхідних файлів та компіляція шейдерних програм. Щоб передати дані в шейдер створений набір функцій, які приймають більшість підтримуваних типів даних GLSL. Для активації шейдерної програми необхідно викликати функцію use().

РОЗДІЛ 4. ПРИКЛАД ВИКОРИСТАННЯ ДЛЯ РОЗРОБКИ

Рушій було використано для розробки клона гри PacMan (рис. 20)

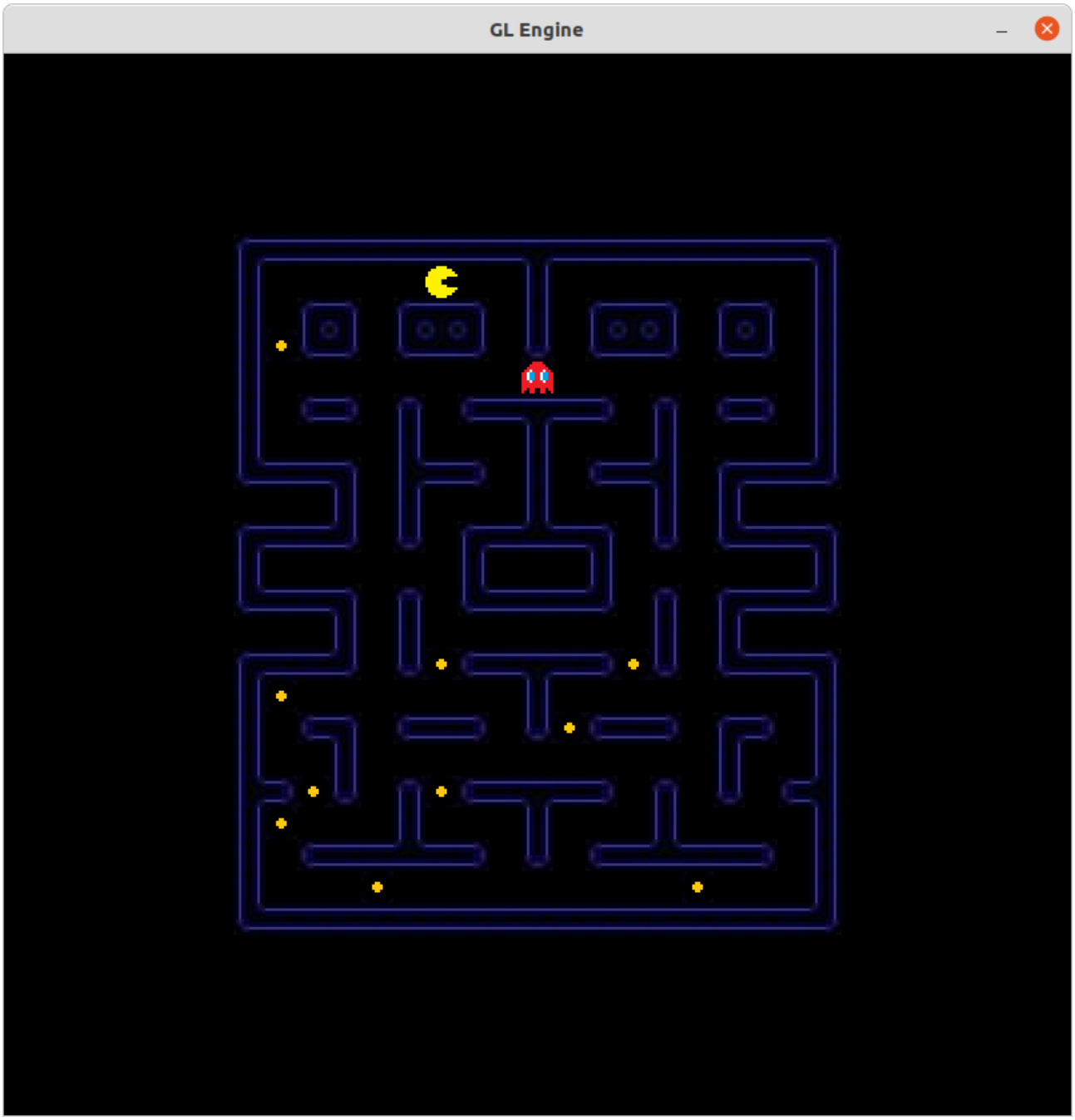

Рисунок 20. PacMan створений з використанням розробленої системи

В грі присутнє завантаження рівня з текстового рядка (теоретично його можна брати з файлу), мапа при цьому генерується динамічно, а текстури вибираються залежно від сусідніх клітин.

Вся мапа поділена на клітини, в яких може бути розміщений або гравець, або привид, або їжа, або стіна. І гравець, і привид для вибору шляху керуються алгоритмом A* (реалізовано також пошук в глибину та ширину, жадібний алгоритми).

Ціль привида: з'їсти гравця. Ціль гравця: з'їсти всю їжу і залишитись живим. Кожен раз коли гравець їсть їжу, в нього збільшується кількість очок, число виводиться в консоль.

Також з використанням рушія було створено клон гри Battlecity (рис. 21).

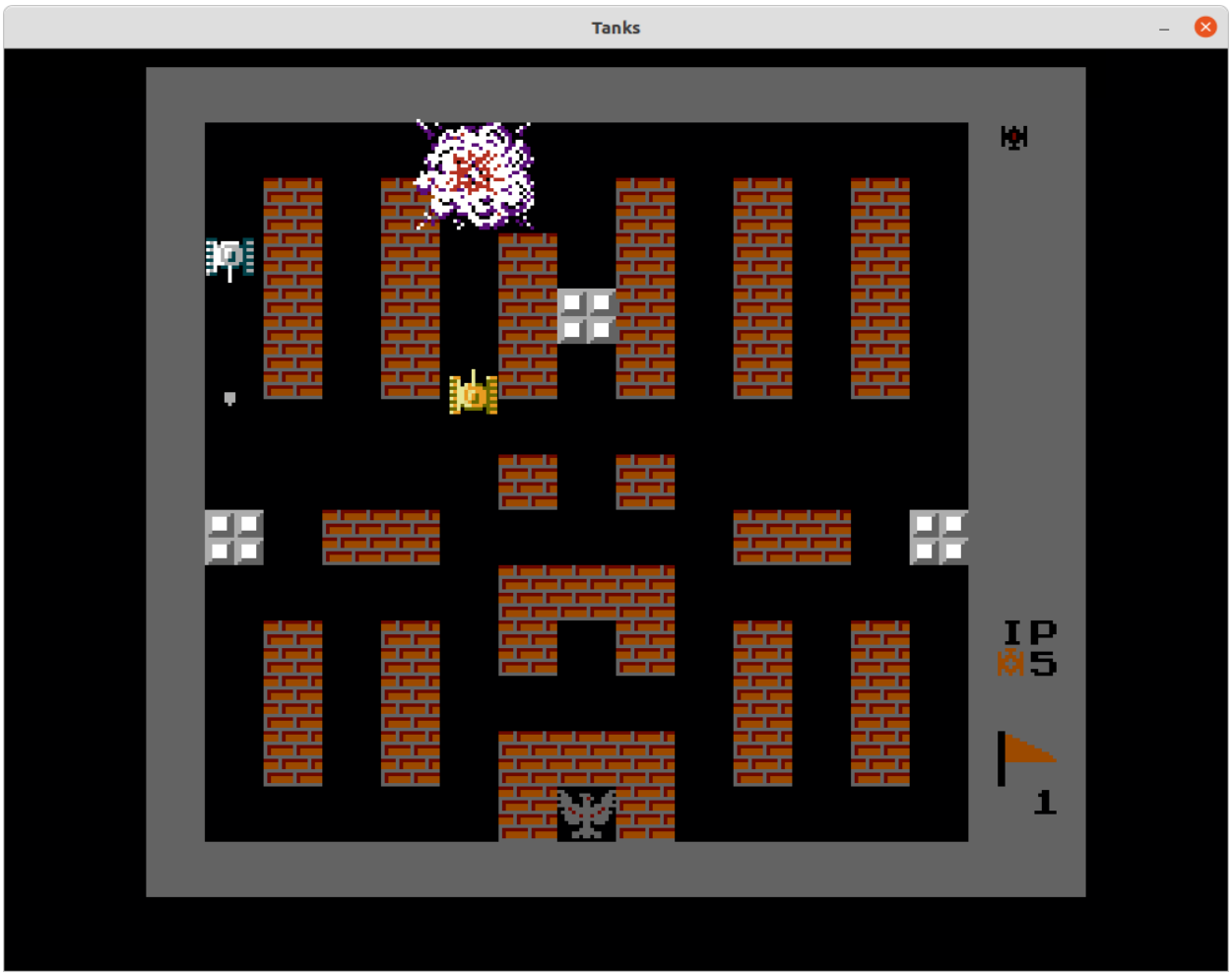

Рисунок 21. Battlecity, створений з використанням розробленої системи

В грі реалізовано підтримку багатьох рівнів, які завантажуються після перемоги в попередньому. Умовою перемоги на рівні є знищення всіх ворогів. Вороги спавняться у наперед заданих точках, при чому їх одночасна кількість на мапі не може перевищувати кількість спавнерів. Загальна кількість ворогів і їх типи наперед задані в налаштуваннях рівня. Гравець має декілька життів, які втрачає, якщо в нього влучить ворожий снаряд. Після втрати кожного життя гравець спавниться в стартовій точці. Якщо втрачено всі життя, або знищена база, то це означає, що гравець програв, після чого буде виведено відповідне повідомлення.

Типів ворогів існує 4, вони відрізняються силою, швидкістю та кількістю життів. Кожен тип має свою текстуру. Також реалізовано різні типи стін: сталева (не можна знищити), цегляна (знищується потраплянням одного снаряду), вода (не можна проїхати, проте снаряди можуть пролетіти), дерева (можуть приховати в собі об’єкти).

Для всіх ефектів, наприклад, поява ворогів та гравця, вибух, знищення, реалізовано відповідні анімації.

ВИСНОВКИ

Під час виконання даної кваліфікаційної роботи було досліджено наявні ігрові рушії, проаналізовано переваги та недоліки. На основі отриманих знань було розроблено власний ігровий рушій на мові C++ з використанням OpenGL. Головні його переваги:

- компонентна структура об’єктів;
- для взаємодії між об’єктами і підсистемами реалізована зручна система сигналів;
- гнучка система налаштування груп колізій об’єктів.

У випадку подальшої розробки система має такі можливості розширення:

- централізовану систему керування текстурами;
- зони колізій для оптимізації їх обробки;
- додавання автоматичної генерації карти для можливості пошуку шляху.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Gregory J. Game Engine Architecture / Jason Gregory., 2018. – 1240 с. – (3rd Edition).
2. What is a Game Engine? [Електронний ресурс]. – 2013. – Режим доступу до ресурсу: https://www.gamecareerguide.com/features/529/what_is_a_game_.php.
3. Toftedahl M. A Taxonomy of Game Engines and the Tools that Drive the Industry [Електронний ресурс] / M. Toftedahl, H. Engström – Режим доступу до ресурсу: http://www.digra.org/wp-content/uploads/digital-library/DiGRA_2019_paper_164.pdf.
4. GDC State of the Game industry 2019 [Електронний ресурс] – Режим доступу до ресурсу: https://reg.gdconf.com/gdc-state-of-game-industry-2019.
5. Unity 3D [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/index.html.
6. Unity — подводные камни разработки 2D игры [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/432946/.
7. Unreal Engine [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unrealengine.com/en-US/index.html.
8. CLion [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/ru-ru/clion/.
9. High-Level Language [Електронний ресурс] – Режим доступу до ресурсу: https://www.techopedia.com/definition/3925/high-level-language-hll.
10. C++ Applications [Електронний ресурс] – Режим доступу до ресурсу: https://www.stroustrup.com/applications.html.
11. History of C++ [Електронний ресурс] – Режим доступу до ресурсу: https://www.cplusplus.com/info/history/.
12. Stroustrup B. The C++ Programming Language (4th Edition) / Bjarne Stroustrup., 2013.

13. An Interview with A. Stepanov [Електронний ресурс] – Режим доступу до ресурсу: http://www.stlport.org/resources/StepanovUSA.html.
14. Segal M. The OpenGL Graphics System: A Specification [Електронний ресурс] / M. Segal, K. Akeley // Version 4. – 2010. – Режим доступу до ресурсу: https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf.
15. OpenGL tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://learnopengl.com.
16. OpenGL Mathematics [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/g-truc/glm.
17. OpenGL Mathematics [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/138731/.
18. GLFW [Електронний ресурс] – Режим доступу до ресурсу: https://www.glfw.org.
19. The OpenGL Extension Wrangler Library [Електронний ресурс] – Режим доступу до ресурсу: http://glew.sourceforge.net.
20. Rost R. OpenGL Shading Language / R. Rost, J. Kassenich, B. Lichtenbelt., 2004. – 416 с. – (3).
21. Nystrom R. Game Programming Patterns / Robert Nystrom., 2014. – 354 с.