КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота
на здобуття освітнього рівня бакалавра

за спеціальністю 121 Інженерія програмного забезпечення на тему:

РОЗРОБКА МОБІЛЬНОГО ІГРОВОГО ДОДАТКУ "SOKOBAN" ДЛЯ ПЛАТФОРМИ ANDROID З ВИКОРИСТАННЯ ТЕХНОЛОГІЇ OPENGL

Виконав студент 4-го курсу
Микита ШКАРУПА _____
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Оксана ШКІЛЬНЯК _____
(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 інтелектуальних програмних систем
« 29 » травня 2023 р.,
протокол № 11
Завідувач кафедри
Олександр ПРОВОТАР _____
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 54 сторінка, 10 рисунків, 13 джерел посилання.

ANDROID, OPENGL, PIMPL, ІГРОВИЙ ДОДАТОК, ІГРОВИЙ МОБІЛЬНИЙ ЗАСТОСУНОК, 3D ГРА

Об’єктом розроблення є процес розробки та написання мобільного ігрового додатку з використанням технології OpenGL,

Метою роботи є створення мобільного ігрового додатку під ОС Android з використанням технології OpenGL, притримуючись приципу написання програми у вигляді PImpl (Pointer to IMPLementation).

Методи розробки: середовище розробки Android Studio, мова програмування C++, техніка написання програм PImpl,

Результати та їх новизна: розроблено мобільний ігровий додаток під ОС Android з власним рушієм гри.

Інформація щодо впровадження: Розроблений мобільний ігровий додаток може бути в майбутньому опублікований чи використовуватися у якості портфоліо. Реалізовану технологію рендерингу ігрових рівнів з використанням OpenGL можна буде впровадити в інші ігрові мобільні застосунки.

Взаємозв’язок з іншими роботами: Гра орієнтується та бере досвід з інших робіт у галузі "розробка мобільних ігрових додатків", беручи до уваги давно працюючи ігрові механіки та гейм дизайн.

Висновки та пропозиції щодо розвитку об’єкта розроблення й доцільності продовження розробок: Розробка ігрового додатку з власним рушієм з використанням технології OpenGL є нестандартним цікавим проєктом, який дає особливий досвід у написання програм. Створена гра має ряд можливостей для розвитку у майбутньому, таких як збільшення кількості рівнів у грі та вдосконалення інтерфейсу.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API – Application Programming Interface, прикладний програмний інтерфейс;

PImpl - Pointer to implementation;

Open GL SE - Open Graphics Library for Embedded Systems, відрита графічна бібліотека для вбудованих систем;

CGI - computer-generated imagery, створені комп'ютером зображення;

CAD - computer-aided design, система автоматизованого проєктування;

VR - virtual reality, виртуальная реальность;

GPU - Graphics processing unit, графічний процесор;

GLFW - Graphics Library Framework, фреймворк графічної бібліотеки;

SDL - Simple DirectMedia Layer, простий рівень DirectMedia;

GLUT - OpenGL Utility Toolkit, набір утиліт OpenGL;

GLSL - OpenGL Shading Language, мова затінення OpenGL;

ETC - Ericsson Texture Compression, стиснення текстури Ericsson;

SGI - Silicon Graphics, Inc;

ABI - Application Binary Interface, бінарний інтерфейс програми.

ВСТУП

Оцінка сучасного стану об’єкта розробки. У сучасному світі важко не помітити, наскільки сучасні технології стрімко увірвались у повсякденне життя людини та стали невід’ємною частиною сьогодення у всіх галузях життєдіяльності людини. Сучасні гаджети виконують дуже велику роль у житті людини від покупки автомобіля через банківський застосунок до розваг у вигляді ігрових застосунків. Дуже часто можна помітити, як більшість людей користуються ігровими додатками на свої смартфонах скрізь, будь то вагон метро, чи черга на касі чи просто інший вільний час.

Актуальність роботи та підстави для її виконання. В сучасному світі створюється багато різноманітних мобільних додатків для смартфонів, через зручність та компактність їх використання, завдяки чому вони стають все більш популярними, бо вони прогресивні, потужні та є вже невід’ємною частиною людської буденності. Важко уявити якийсь супермаркет чи банк, який би не мав свого мобільного додатку. Зараз можна сміливо сказати, що особливе місце в житті людей останнім часом займають ігрові додатки. Створюються цілі ігрові групи, змагання та челенджі, які присвячені мобільним ігровим додаткам. Ще одна причина, чому ігрові додатки мають актуальність на даний час, та матимуть її у майбутньому, це азарт, інша реальність, що відволікає від проблем, які кожен з нас має. Іноді це, навіть, якісь різноманітні комунікації з людьми, які знаходяться в іншій частині земної кулі. Саме ці підстави надихнули на те, аби розробити власний ігровий застосунок для смартфонів, а також паралельно з цим дослідити різноманітні методи розробок та різноманітні технології.

Мета й завдання роботи. Головною метою кваліфікаційної роботи є створення сучасної програми на мобільний пристрій з операційною системою Android, яка була створена компанією Google на базі ядра Linux, і є самою

розповсюдженою операційною системою для смартфонів в усьому світі. В програмі буде також використано OpenGL, залучаючи до неї вже давно працюючи ігрові механіки, які будуть реалізовані в ігровому додатку.

Об’єкт і методи дослідження або розроблення. Головним методом та підходом до розробки буде PImpl (Pointer to implementation). Програма має бути реалізована на мові програмування C++ та повинно бути використане середовище для програмування Android Studio. Також буде задіяний внутрішній емулятор мобільного телефону з операційною системою Android, для тестування додатку та перевірки правильності та коректності його роботи, відносно поставлених цілей.

Можливі сфери застосування. Головною сферою застосування програми є ігри, розваги та дозвілля, які наразі є доволі актуальними, популярними та користуються великим попитом серед потенційних користувачів на даний час, бо іноді це дуже допомагає людям розвантажити мозок, особливо під час воєнного часу. Об’єктивно, набутий досвід у написанні програм можна реалізувати при створенні інтерфейсу мобільних застосунків різної тематики, наприклад для музичного плеєра чи месенджера, але в першу чергу це може бути робота з іншими іграми чи додатками, які потребують 3D графіки.

Взаємозв'язок з іншими роботами. Отриманий досвід може виявитися корисним при впровадженні 3D графіки в існуючу вже програму або додаванні ігрових рушіїв в інших більш перспективних проєктах. Планується опублікувати готовий життєздатний, цікавий та конкурентноспроможний продукт на платформі Google Play.

РОЗДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА

1.1 Історія розвитку 3D графіки від комп'ютера до телефону

Галузі розвитку 3D графіки пройшли тернистий, довгий та важкий шлях від початку їх створення і до нашого сьогодення, де без неї важко уявити, будь яку сучасну мобільну чи комп’ютерну гру. Зі звичайних каркасних моделей до перехоплюючих подих шедеврів, які зараз намагаються видати найреалістичнішу картинку, яку іноді важко відрізнити від реальної. Всі ці сфери зазнали неймовірних змін протягом останніх десятиліть.

Отже, розглянемо головні моменти розвитку, починаючи з другої половини XX століття, а саме такі періоди: з 1960 до 1970 років; початок 1970-х років; 1980-ті роки; 1994 рік; початок 1998 року та початок XXI століття.

А тепер розглянемо окремо та більш детально кожен з цих періодів розвитку.

Період з 1960 до 1970 років. Початок комп’ютерної графіки був покладений ще в 60-х роках минулого століття, коли американський інформатик Айвен Сазерленд, якого ще називають “батьком” комп’ютерної графіки, розробив інноваційну програму, яка отримала назву Sketchpad, що з англійської мови перекладається, як альбом для ескизів або робот-кресляр. Так от, саме ця програма представила концепт взаємодії з графікою за допомогою пера та виступила прототипом сучасного графічного інтерфейсу, який ми всі вже звикли бачити на наших ноутбуках та персональних комп’ютерах.

Початок 1970-х років. Американський інформатик Едвін Кетмелл, а також співзасновних Pixar Animation Studios, представив світу текстурування. Текстурування - це особлива техніка, що включає в себе

надання поверхні тривимірної деталізації кольору, блиску або інших властивостей, для імітації, приміром, певного природного матеріалу. Ці блискучі нововведення заклали фундамент для розвитку одних з найважливіших галузей сучасності у 3D графіці, а саме каркасного моделювання.

Отже, каркасне моделювання у сімдесяті роки, в плані розвитку тривимірної графіки, стало важливим етапом у зв'язку з появою каркасних моделей. Вже згадані раніше Айвен Сазерленд та Девід Еванс стали засновниками цього важливого відкриття та невід'ємною частиною історії 3D графіки.

Що ж із себе представляли каркасні моделі? Вони включали в себе створення тривимірних об'єктів за допомогою з'єднання ліній, формуючи багатокутники, які стали формою та структурою для віртуальних моделей. Ця проста на перший погляд техніка стала вирішальною та прогресивною знахідкою, ба поява якої стала сходинкою для подальшого розвитку сфери комп'ютерної графіки.

1980-ті роки. У 1980-х роках відбувся вихід на сцену рендерингу (в перекладі з англійської мови - візуалізація, підмальовування), що надав помітний прогрес у розвитку тривимірній графіці. І відповідно, у свою чергу, це стало б неможливим без розвитку рендерингу. Визначальним моментом в цьому стало представлення методу затемнення за Фонгом, який був розроблений Буйєм Туонгом Фонгом. Затемнення за Фонгом допомогло в більш реалістичному представленні освітлення поверхонь тривимірних об'єктів, що покращило їхній зовнішній вигляд. Не можна не взяти до уваги паралельний розвиток обладнання та алгоритмів растеризації, що прискорили процес рендерингу та дозволили мати більш комплексні та детальні сцени в реальному часі. [1]

1994 рік. Це був великий рік в історії 3D графіки. Домашні комп'ютери та ігрові системи виходять на новий рівень розвитку та розповсюдженості, їхня потужність дозволяє запускати ігри, які потребують потужних процесорів, що дало поштовх до розвитку самих ігор та ігрової комп’ютерної індустрії в цілому. [2]

В 3D графіці існує такий метод, як затінення Гуро, він згладжує колірні переходи між ребрами багатокутників. Це дозволяє фігурам мати плавні кути навіть при невеликій кількості полігонів. Отже, тепер з'явилася можливість накладання текстур на 3D об'єкти. У тогочасі усім відома гра "Зоряні війни: Винищувач краваток" (Star Wars: Tie Fighter) (1994), стала першою грою, в якій було використано цей метод.

Початок 1998 року. Зрілість. Ера шейдерів та відеоприскорювачів. Девіз цієї епохи - більше полігонів, більше пост-ефектів, більше шейдерів, а от в останні роки почала набирати обертів популярності тенденція трасування променів. Нижче розглянемо коротко кожну з тенденцій.

Полігональність - ця тенденція передбачає розвиток саме моделей, більше уваги приділяється саме ним, вони стали більш деталізованими, з'явилося більше дрібних об'єктів, акцент з примітивної фігури був перенесений на деталі і реалістичну (кінематографічну) картинку. Завдяки більшій кількості полігонів можна було застосовувати більше текстур.

Щоб більше зрозуміти, про що мова, буде доречним зазначити що полігональність - це з’єднання трьох точок координат (X, Y та Z), які задані як вершини і з'єднати їх ребрами, то вийде трикутник, який в 3D моделюванні називають полігоном.

Пост-ефекти - ця тенденція втілює в собі накладення відблисків, віньєтка, інверсія, накладання тонів, градація кольорів та інші ефекти. Також, з фільмів походить "рендеринг по шарах/каналах" з можливістю збереження

кадрів з інформацією про глибину, рух та тіні в буфері. Якщо коротко, то пост ефекти - це додаткові фільтри, які накладаються на відеозображення.

Для більшої наглядності можна за приклад взяти всесвітньо відому комп'ютерну гру "Counter-Strike" американської компанії Valve Corporation), де є почервоніння екрану, або ж різні спалахи, тріщини та купа всього іншого, все це і є пост-ефекти. Вони додають більшої реалістичності тому, що відбувається в грі.

Шейдери - ця тенденція представляє з себе набір міні-програм, що визначають характеристики поверхні. З перших днів появи 3D моделювання, зовнішній вигляд моделі залежав від текстури, яка накладається на модель, тому при зміні світла, положення і тіней, сама ж поверхня не змінювала своїх характеристик і лишалась статичною без будь якої динаміки, що було не дуже природньо та реалістично. Тепер же, освітлення, тіні та ефекти стали динамічними і мінливими.

Трасування променів - це тенденційне рішення для створення реалістичного освітлення, віддзеркалень та тіней, що забезпечує вищий рівень реалістичності, ніж традиційні методи рендерингу.

XXI століття. Поява перших мобільних 3D ігр, які з'явилися приблизно у 2003 році, але мобільні пристрої, які здатні належним чином підтримувати 3D графіку, з'явилися трохи пізніше, тому 3D ігри на мобільних пристроях стали широко доступними лише у 2005 році. Додайте також до цього всього величезну кількість виробників телефонів та інших пристроїв, що з'явилися на ринку, і стане зрозуміло, наскільки складніше було розробляти контент, який би працював на різних платформах.

OpenGL ES, підмножина графічного API OpenGL, орієнтована та розроблена спеціально на вбудовані системи, такі як мобільні телефони, також була запущена в 2003 році, але, звичайно, не була прийнята відразу для всіх мобільних пристрої та інших гаджетів. Таким чином, протягом деякого часу

більшість мобільних ігор все ще продовжували створюватися за допомогою API, орієнтованих тільки на центральний процесор.

У 2006 року, коли ARM® придбала Falanx і ARM Mali™ стала реальністю, наступні ітерації OpenGL ES почали зосереджуватися саме на функціональності 3D-графіки, а графічні процесори в мобільних пристроях ставали все більш поширеними, а згодом і більш потужними. Оскільки виробники усвідомлювали потребу в просунутій обробці графіки, а популярність мобільних ігор, їх різноманітність та кількість ставала все більшою, а зацікавленість в них та вимоги ставали більш прискіпливий. Відповідно саме тоді настав час для доповнення для OpenGL ES 3.0, додався постпроцесорний HDR, а саме додалась глибина різкості та відтінки.

Технології зменшення пропускної здатності, такі як *адаптивне масштабоване стиснення текстур* (ASTC) від ARM, також стали доступними і забезпечили величезні покращення, підвищивши якість та різноманітність ігор, які могли підтримуватися на мобільних пристроях. Найперший телефон на базі Mali був випущений у 2007 році. Це був LG KU990, або "Viewty", а гра яка була на ньому називалась “Space Command Platoon”, також була розроблена компанією LG на замовлення. Для того часу, це був передовий та проривний контент майбутнього, але дуже багато змінилося у цій технологічній галузі за наступне десятиліття, від моменту його випуску.

З того часу, Mali 400 став першим багатоядерним графічним процесором і допоміг зробити проривну революцію в мобільних іграх до мейнстріму.

Отже, коли стало очевидним, що маючи багато ядерний процесор, можна дозволити одному ядру обробляти більшість необхідних даних для сцени у грі, відповідно додаткові ядра можна використовувати, для того щоб додати складності та реалістичності тій сцені, яка в даний момент обробляється одним із ядер процесору.

Довгий час залишаючись найпопулярнішим мобільним графічним процесором для мобільних пристроїв з надто низьким енергоспоживанням, Mali-400 був першим у довгій лінійці графічних технологій від ARM, що постійно вдосконалювалися в своєму напрямку розробок. Адже сьогодні ARM Mali Multimedia Suite включає в себе не тільки графічні процесори з ультра низьким енергоспоживанням, а й доволі найпродуктивніший процесор Mali G71. Саме цей процесор став значним в еволюції мобільних ігор. Також ці процесори стали базовою основою для наступного покоління не тільки сучасних смартфонів, а й дали поштовх для пристроїв віртуальної реальності, як от, наприклад, VR ігри.

Далі розглянемо, що ж саме з себе уявляє найновіша технологія у світі ігор - *VR ігри,* чим вони різняться від звичайних мобільних та комп'ютерних ігор. Дана технологія є однією з найбільш вимогливих у порівнянні з іншими (мобільні телефони, планшети, комп'ютери, різні мобільні пристрої тощо). В чому ж полягає її примхлива вимогливість? А в тому, що вона потребує виконання відразу подвійних вимог до обчислювальної потужності через те, що має необхідність рендерингу контенту для кожного ока окремо.

Отже, проаналізувавши усю вище наведену інформацію, можна впевнено стверджувати, що впровадження подібних технологій на мобільних пристроях було б неможливим без постійно зростаючої обчислювальної потужності графічних процесорів.

Але прогрес технологій не стоїть на місці і навіть за межами VR, мобільні ігри продовжують розвиватись, їх складність іноді сягає високих вершин та вимог, оскільки споживачі прагнуть більшого від своїх мобільних пристроїв, а в переважній більшості саме від своїх смартфонів. Користувача вже більше не влаштовує простота всесвітньо відомої мобільної гри "Snake". Зараз натомість користувач вже очікує консольної якості, AAA-ігор на мобільних пристроях.

Цей ефект перебазовування консольних і комп'ютерних ігор на мобільні пристрої та які вже стали невід’ємною частиною будь якого смартфона чи планшета, також сприяє тому, що підвищуються вимоги до обробки даних та їх величезних обсягів.

Підсумовуючи, можна стверджувати, що основною частиною задач зараз перед розробниками мобільних ігор стає, не тільки забезпечення підтримки найновішого та найякіснішого контенту на мобільних пристроях, а й обов’язковий моніторинг вимог до цих ігор. Адже, для прикладу такі ігри як “Jurassic World” або “Unkilled” є одними з найскладніших мобільних ігор на сьогодні.

У зв’язку зі збільшенням кількості контенту зростає не тільки він, а також й досвід самих розробників.

Запуск API Vulkan від Khronos Group у 2016 році ще більше підняв рівень розробки мобільних ігор на новий рівень. API нижчого рівня, такі як Vulkan дозволяє розробнику набагато краще контролювати, на якому саме обладнанні буде виконуватися ті чи інші частини коду. Хоча іноді, така додаткова потужність може бути ризикованою, оскільки збільшується ймовірність того, що щось піде не так, що може мати неприємні наслідки. Тому розробнику треба постійно враховувати та моніторити усі ці можливі ризики та несподіванки. Але якщо відкинути можливі ризики, це також означає, що можлива більш детальна оптимізація, що є тільки великим плюсом для усіх розробників мобільних ігор. Бо таким чином, це може забезпечити доволі підвищену продуктивність та також збільшити тривалість часу для автономної роботи.

Оскільки сьогодні тенденція складності ігор тільки зростає, відповідно розробникам ігор потрібно постійно йти в ногу з цією тенденцією, та як кажуть, тримати руку на пульсі. І схоже на те, що ця тенденція йде до того, що у вимогах та очікуваннях до наступних поколінь графічних процесорів буде більша продуктивність, збалансованість з чудовою енергоефективністю в

порівняння з існуючими процесорами. Все це швидше за все призведе до того, щоб можливо було максимізувати досвід користувача і мінімізувати витрати, як для різноманітних виробників цього напряму так і для розробників.

Отже, повертаючись до процесорів Mali, можна побачити, що використовуючи весь набір мультимедійних процесорів Mali. Який як пам'ятаємо, складається з графічних процесорів, відео процесорів та дисплейних процесорів. Але виробники та розробники роблять ще більше неможливе, створюючи найвимогливіший контент перетворюючи його у реальність. Це великий стрибок від звичайних мобільних пристроїв до супер смартфонів. [3]

1.2 Огляд основних принципів 3D

1.2.1 Що таке рендер і як він працює

Рендер або візуалізація – це процес створення фінального зображення або послідовності із зображень на основі двовимірних або тривимірних даних. Даний процес відбувається з використанням комп'ютерних програм та часто супроводжується важкими технічними обчисленнями, з якими спокійно справляються потужності комп'ютера або окремі його комплектуючі частини.

Процес рендерингу так чи інакше присутній в різних сферах професійної діяльності яка має потребу у цьому процесі, будь то кіноіндустрія, індустрія відеоігор або ж банальний відеоблогінг. Найчастіше, рендер є останнім або передостаннім етапом в роботі над проектом який створюється розробником. Після цього робота вважається завершеною, але іноді має потребу у невеликій постобробці. Також варто відзначити, що нерідко рендером називають не сам процес рендерингу, а скоріше вже завершений етап даного процесу або його

підсумковий результат. Найчастіше, коли в процесі якоїсь із пов’язаних індустрій кажуть про рендер, то скоріше за все мають на увазі рендеринг у саме 3D графіці. Приставка “3D” в назві вказує на те що це спосіб створення рендеру, який і використовує тримірні об’єкти, які були створені на комп’ютері за допомогою спеціалізованих комп’ютерних програмах, які розробленні для 3D моделювання. Якщо казати простими тезами, то в кінці кінців отримується все одно 2D зображення.

Як вже зазначалося вище у даному розділі цієї роботи, рендеринг – один з найскладніших технічних етапів, адже саме під час рендеринга, йдуть складні математичні обчислення, які виконуються движком (рушієм) рендеру. На цьому наступному етапі, рушій переводить математичні дані про сцену у фінальне 2D зображення. Під час процесу, йде перетворення 3d геометрії, текстур та світлових даних сцени, в об’єднану інформацію про кольорове значення кожного пікселя у 2D зображенні. Відповідно, не важко зрозуміти, що движок (рушій) на основі наявних у нього даних, прораховує те, яким кольором повинен бути забарвлений кожен піксель зображення для отримання комплексної, гарної та закінченої картинки.

Далі для повноти розкриття теми рендерингу, розглянемо його основні типи, на які він поділяється.

Основні типи рендерингу:

Рендеринг у реальному часі (Real-Time Rendering).

Даний тип рендерингу використовується в тих додатках, які вимагають інтерактивного та динамічного візуального зворотного зв'язку в реальному часі, так для прикладу, у різноманітних відеоіграх та додатковій віртуальній реальності. Візуалізація в реальному часі фокусується на досягненні високої частоти кадрів і швидкості відклику.

Пост рендер (Pre-rendering)

Автономний рендеринг використовується в таких додатках, як кіновиробництво та архітектурна візуалізація, де першочерговим завданням постає візуальна якість, а час обчислень є менш важливим. Він передбачає використання більш складних і досконалих алгоритмів рендерингу для створення високо реалістичних і деталізованих зображень або анімації.

Рендер може виконуватися за допомогою різних алгоритмів і підходів, таких як растеризація та трасування променів.

Растеризація

Ще одна найпоширеніша техніка, що використовується при рендерингу в реальному часі, коли сцена перетворюється в серію пікселів й рендериться на основі геометричної інформації.

Трасування променів, з іншого боку, імітує шлях світлових променів для визначення кольору та освітлення об'єктів, що призводить до більш точного освітлення та віддзеркалення.

З огляду на викладену інформацію про рендеринг та його типи, об’єктиво та впевнено можна зазначити, що загалом він відіграє вирішальну та значну роль у створенні візуально привабливих та CGI у широкому спектрі застосувань, включаючи фільми, відеоігри, симуляції, архітектурну візуалізацію, дизайн продуктів і багато чого іншого.

1.2.2 Чому саме Open GL ES 3.0

У рендерингу є дві основні бібліотеки для програмування це OpenGL і DirectX. Порівняння між OpenGL і DirectX є суб'єктивним і залежить від конкретних вимог і переваг конкретного проєкту або програми. І OpenGL, і DirectX обидва є графічні API, які створені та використовуються для схожих

цілей, але мають деякі відмінні один від одного характеристики. Розглянемо кілька моментів, на які слід звернути увагу при їх порівнянні.

Сумісність з різними платформами

OpenGL відомий своєю кросплатформенною підтримкою, що дозволяє розробникам написати код, який працює на різних операційних системах. Він доступний як для Windows, так і для macOS й Linux, а також інших різних мобільних платформ.

З іншого боку, DirectX в першу чергу розроблений для систем на базі Windows, хоча були докладені зусилля, щоб перенести підмножину DirectX (DirectX 12) на інші платформи за допомогою таких проектів, як DirectX 12 Ultimate. Через цей головний параметр для створення мобільного ігрового додатка, для програми була обрана саме OpenGL, це був не якийсь осмислений або розсудливим вибір, це був вибір в якості вимушеної міри. Враховується саме той факт, що на мобільних пристрої основною бібліотекою є саме OpenGL.

Доступність та підтримка.

DirectX розробляється та підтримується компанією Microsoft, а це однозначно вказує на те що, він тісно інтегрований з операційною системою Windows. Він також забезпечує всебічну підтримку мультимедійних та ігрових функцій. В свою чергу, OpenGL, є відкритим стандартом і реалізується різними виробниками апаратного та програмного забезпечення. Хоча підтримка OpenGL зазвичай доступна на багатьох платформах, рівень підтримки драйверів й сумісності функцій може відрізнятися залежно від платформи та графічного обладнання. Але в цій роботі нас не дуже сильно цікавить операційна система Windows.

Набір функцій і продуктивність.

Обидва API пропонують поодібні функції рендерингу, але звісно ж між ними можуть бути відмінності в процесі, як реалізовані певні функції. Так історично склалось, що DirectX відомий своєю потужною підтримкою мультимедійних функцій, в які включені обробка звуку та вводу.

В свою чергу, OpenGL набула свою популярність завдяки своїй ефективністю у виконанні завдань певного спектру рендерингу, наприклад, таких як обробка великої кількості викликів жеребкування. Варто зазначити й той важливий факт, що DirectX 12 і Vulkan (низькорівневий API, який є подібним до DirectX 12) пропонують більш тонкий контроль над апаратними ресурсами та можуть забезпечити доволі кращу продуктивність у певних сценаріях порівняно з високорівневими API, такими як обраний нами OpenGL.

Екосистема розробки

Дана екосистема, що оточує DirectX, як правило, тісно інтегрована з інструментами та технологіями на базі Windows. Даний факт може бути перевагою саме для тих розробників, які постійно працюють тільки на платформах Windows. Таким чином вони мають можливість використовувати такі корисні інструменти для роботи, як Visual Studio та специфічні для DirectX бібліотеки.

В свою чергу, OpenGL будучи відкритим стандартом, має більш ширшу екосистему, яка в себе включає підтримку різних мов програмування, різних фреймворків та різноманітних інструментів.

В кінцевому рахунку, вибір між OpenGL та DirectX залежить все ж таки від деяких факторів, в першу чергу, цільова платформа, специфіка вимог до проекту, загальний досвід команди розробників та особисті уподобання розробників.

Отже, приймаючи рішення щодо бібліотек, варто завжди враховувати об’єм, який охоплює підтримка платформи, функції які можуть бути

доступними, вимоги до продуктивності та екосистему розробки. Окрім всього вище зазначеного, слід завжди пам'ятати найважливише, що графічні API продовжують розвиватися й будуть розвиватися надалі. Також, нові стандарти, а саме Vulkan і DirectX 12, зазвичай пропонують набагато просунутіші функції та різні можливості продуктивності, якщо провести порівняння зі їхніми попередниками. Та як вже зазначалось вище вже не раз, для мобільного додатку вимушено буде використовуватись саме бібліотека OpenGL.

1.2.3 Що таке Open GL?

OpenGL - це кросплатформний та багатомовний програмний інтерфейс, який використовується для рендерингу як 2D комп'ютерної графіки, так і для 3D графіки, який як відомо працює з GPU. *OpenGL* надає певний набір функцій та потрібних команд, які дозволяють розробникам мати певну взаємодію з графічним обладнанням, а також створювати інтерактивні графічні додатки. На початку 1990-х років розроблений компанією Silicon Graphics Inc. OpenGL став стандартним графічним API. Він використовувався в доволі широкому секторі додатків, які включали в себе відеоігри, CAD, VR, наукову візуалізацію та багато всякого іншого. Різні ключові та базові особливості, можливості OpenGL включають також і кросплатформену сумісність: OpenGL був розроблений як незалежний від будь якої платформи, відповідно його завжди можна використовувати на різноманітних операційних системах, включаючи Windows, macOS, Linux та мобільні платформи, такі як Android та iOS.

Що саме використовэ та OpenGL включає в себе:

а) GPU, для можливості прискорення задач рендерингу, що забезпечує доволі ефективний та високопродуктивний рендеринг графіки;

б) конвеєрну модель рендерингу, завдяки якій геометричні операції та операції рендерингу можуть виконуватися у декілька підходів (етапів), включаючи в себе обробку вершин, збірку примітивів, растеризацію та обробку фрагментів;

в) функціональність для рендерингу як 2D, так і 3D графіки, включаючи примітиви, такі як точки, лінії, трикутники, текстури та моделі затінення;

г) розширення, які є додатковими можливостями та функціями, що можуть бути додані до основного API. Дані розширення надають розробникам простий доступ до розширених графічних можливостей, таких як шейдери, вдосконалені методи відображення текстур тощо;

ґ) використовується разом з іншими бібліотеками та фреймворками, а саме GLFW, SDL та GLUT, для обробки вікон, введення та інших завдань, пов'язаних з графічним програмуванням.

Важливо зазначити, що станом на вересень 2021 року остання стабільна версія OpenGL - це OpenGL 4.6. Також варто зазначити, що в останні роки набули значної популярності найновіші графічні API, такі як Vulkan та DirectX 12, які можуть запропонувати більш низькорівневий доступ до апаратного забезпечення, а також покращену продуктивність порівняно з OpenGL.

1.2.3.1 Що таке Open GL ES 3.0

OpenGL для вбудованих систем 3.0 - це API для рендерингу графіки, розроблений виключно для вбудованих систем, таких як мобільні пристрої, планшети та ігрові консолі. Дана підмножина ширшого стандарту OpenGL, є оптимізованою для середовищ з обмеженими ресурсами. Вона ідеально підходить для використання та створення сучасних мобільних ігрових додатків

широкого спектру застосування, також ідеально підходить під мету цієї кваліфікаційної роботи. [4][5]

OpenGL ES 3.0 представляє декілька вагомих покращень та поліпшень, а також деякі нові функції, якщо брати за порівняння його попередника, OpenGL ES 2.0.

Доцільно також буде розглянути й їх відмінності, отже першою буде підтримка програмованих шейдерів, які дозволяють розробникам писати власні не шаблонні вершини та інші фрагментні шейдери на мові програмування GLSL. Дана відмінність забезпечує більшу гнучкість та дозволяє створювати розширені графічні ефекти.

В OpenGL ES 3.0 також додано й підтримку мультидискретизация, що значно покращує візуальну якість зображень за рахунок зменшення нерівних країв або аліасингу. Саме це надає змогу отримати більш плавну і красиву графіку та більш кращі методи згладжування.

Також в OpenGL ES версії 3.0 додано і таку відмінність, як підтримку форматів стиснення текстур, таких як ETC2 і EAC, для того аби зменшити обсяг пам'яті та зменшити вимоги до пропускної здатності для текстур.

Ще одна відмінність полягає у введенні розширеної можливості текстурування. До таких можливостей відносяться: масиви текстур, пошук текстур у вершинних шейдерах, безшовні кубічні карти та текстури, що не є ступенем двійки. Всі ці перелічені можливості дозволяють створювати складніші та реалістичніші ефекти текстурування.

Наступна відмінність OpenGL ES 3.0 полягає у представлені запитів на оклюзію, ці запити дозволяють розробникам визначати видимість об'єктів у сцені. Дана можливість може бути корисна при використані оптимізації рендерингу, пропускаючи рендеринг невидимих об'єктів.

І ще одна, останн, відмінність полягає у тому, що було додано підтримку шейдерів яка була створена для обчислень на GPU. Ця підтримка шейдерів дозволяє виконувати об’ємні обчислення, які необхідні для загального призначення виключно на графічному процесорі. Саме це дозволяє розробникам відтепер завантажувати відповідні завдання на GPU. До таких завдань в даному випадку відносяться такі: фізичне моделювання, обробка зображень, яка має вагоме значення для підвищення продуктивності.

З огляду на вище наведену інформацію, можна зробити висновки такого ракурсу.

По-перше, в цілому, OpenGL ES версії 3.0 значно розширив багато можливостей для програмування вбудованої графіки.

По-друге, надав розробникам можливість більшу гнучкість і дозволивши їм створювати візуально приголомшливі та ефективні графічні додатки на широкому спектрі пристроїв.

1.2.3.2 Історія Open GL

Вісімдесяті роки минулого століття вважають початком історії виникнення стандартів (бібліотеки) OpenGl. Доволі відома у професійних колах американська компанія Silicon Graphics Inc (SGI), яка була (і надалі продовжує бути) виробником графічних терміналів, вирішила зробити власне програмне забезпечення. Правда. це програмне забезпечення повинно було бути спеціалізованим та вміти працювати з широким сектором графічного обладнання. Для початку вісімдесятих років двадцятого століття це був справжній виклик, який ніс у собі ряд великих ризиків. Написання такого специфічного програмного забезпечення коштувало багато зусиль для компанії, а також великих капітальних вкладень. Адже, командра розробників

компанії розробляли власні інтерфейси та власні драйвера для кожного обладнання окремо. Саме ці нюанси і збільшували витрати на проєкт, що розроблявся, а разом з витратами і збільшувались відповідно й ризики. [6]

У результаті рішення розробки власного програмного забезпечення, компанія SGI на початку 1990-х років випустила свій IRIS GL, що зробило їх лідерами у сфері 3D-графіки для робочих графічних станцій. Одночасно, їх програмний продукт став галузевим стандартом, який використовувався набагато ширше, це якщо порівнювати з іншим конкурентним програмним продуктом тогодення. На той час це був Programmer's Hierarchical Interactive Graphics System (PHIGS). Ця конкурентна система була розроблена на відкритих стандартах.

Саме такого результату було достигнуто компанією SGI, завдяки декільком важливим параметрам в їх програмному забезпеченні: по перше, воно вважалось простим у використанні, по-друге, воно підтримувало миттєвий режим рендеринга.

На противагу цим особливостям, конкуренти на той час не мали таких переваг перед IRIS GL.

Але так тривало не дуже довго і конкуренти компанії SGI, такі як: американська компанія програмного забезпечення Sun Microsystems, американська технологічна компанія Hewlett-Packard (HP) та американська електронна корпорація International Business Machines Corporation, більш відома, як IBM, теж з часом змогли розробити та вивести на світовий ринок своє власне 3D обладнання. Це обладнання підтримувало розширення стандарту PHIGS, що зробило зазначені компанії більш перспективними та конкурентно спроможними на світовому ринку. Відповідно саме через це компанія SGI була вимушена випустити свою власну версію IRIS GL з відкритим вихідним кодом, як публічний загальнодоступний

стандарт. Компанія надала цьому стандарту назву - OpenGL, наразі доволі відому у спеціалізованих сферах.

Однак вихід нового стандарту, був не дуже звичний та зручний для більшості існуючих клієнтів компанії SGI. А власне, саме перехід з одного стандарту, такого як IRIS GL на новий стандарт, такий як OpenGL викликав багато труднощів та проблем. Адже цей перехід вимагає значних капитальних вкладень від компанії та доволі немаленьких інвестицій. До зазначених вище причин, також додавався і той факт, що IRIS GL також мав функції API, які взагалі не мали відношення до Window System та Sun's NeWS. Тож і бібліотеки IRIS GL були непридатні для того, аби стати відкритими. Існували певні проблеми з ліцензуванням та патентами.

Всі ці доволі різноманітні, але доволі впливові фактори вимагали від американської компанії SGI, все таки продовжувати підтримувати просунуті та запатентовані програмні API Iris Inventor і Iris Performer. В той же самий час, поки йшла підтримка запатентованих раніше графічних програм, вже дороблялась та “дозрівала” довгоочікувана конкурентоспроможна підтримка для OpenGL.

Одним з обмежень IRIS GL було те, що він надавав доступ лише до функцій, які підтримувалися базовим апаратним забезпеченням. Якщо ж графічне обладнання не підтримувало функцію нативно (тобто мова програмування була відміна одна від одної), то тоді програма не могла її використовувати. Однако, OpenGL подолав і цю проблему, надавши програмну реалізацію функцій, які не підтримуються апаратним забезпеченням. Саме ця реалізація дозволила графічним програмам використовувати просунуту графіку, при цьому маючі відносно малопотужні системи. Також, OpenGL запровадив стандарт доступу до апаратного забезпечення. Тобто, переклав відповідальність за розробку програм апаратного інтерфейсу (драйверів

пристроїв) на самих виробників цього обладнання. В той же час ще відбулось делегування функції роботи з вікнами базовій операційній системі.

В світі існувало (відповідно й досі існує) доволі багато різноманітних видів графічного обладнання, і спроба примусити все це “різномовне” графічне обладнання “говорити” та “розуміти” одну мову у такий спосіб, який запровадила компанії SGI за допомогою OpenGL, було великим проривом та мало неабиякий вплив на розвиток цієї галузі взагалому. І все це було досягнуто, завдяки тому, що для всіх розробників була надана платформа іншого, доволі вищого рівня для розробки 3D програмного забезпечення.

На початку дев’яностких років минулого століття компанія SGI все ж таки ініціювала та згодом і очолила, створення ради з перегляду усієї архітектури OpenGL. В цій раді брали участь групи дотичних до цього компаній, які згодом підтримуватимуть та розширюватимуть специфікацію OpenGL у сучасному майбутньому.

У 1994 році компанія SGI мала ідею та зробила певну спробу, щоб розробити та випустити продукт під назвою "OpenGL++". Планувалось, що він буде в себе включати такі елементи, як API графіки сцен, на основі їхньої технології Performer. Навіть, була поширена специфікація серед кількох зацікавлених в цьому продукті замовників, але ця ідея так і не перетворилася на готовий продукт.

Вже у 1996 році багатонаціональна корпорація комп’ютерних технологій Microsoft випустила Direct3D, який з часом став головним конкурентом OpenGL. Понад 50 розробників ігор підписали відкритий лист до корпорації Microsoft, який був опублікований 12 червня 1997 року. У цьому відкритому листі розробники закликали компанію продовжувати активно підтримувати OpenGL.

Вже 17 грудня 1997 року корпорація Microsoft і компанія SGI ініціювали новий проєкт Fahrenheit. Даний проєкт був спільною роботою двох структур, мета якого була, уніфікувати інтерфейси OpenGL і Direct3D. А також планувалось додавання API для роботи зі сценографією.

У 1998 році до проєкту Fahrenheit приєдналася й американська компанія Hewlett-Packard. Спочатку цей проєкт давав деякі надії на впорядкування всесвіту інтерактивних API для інтерактивної 3D комп'ютерної графіки.

Але згодом, через певні фінансові обмеження в компанії SGI, стратегічні причини в корпорації Microsoft, а також та загальну відсутність підтримки всередині самої індустрії, проєкт було таки припинено у 1999 році.

У XXI століття OpenGL більше не перебуває в активній розробці. Якщо між 2001 і 2014 роками специфікація OpenGL оновлювалася переважно щороку. Два випуски протягом року (3.1 і 3.2) відбулися у 2009 році, а цілих три (3.3, 4.0 та 4.1) відбулися у наступному році (2010), що показувало позитивну динаміку розвитку. Що не можна сказати за наступні специфікації. Отже, остання специфікація OpenGL 4.6 була випущена у 2017 році, майже, після трирічної перерви. Та і то вона обмежилася включенням одинадцяти існуючих розширень ARB і EXT до базового профілю.

На такий перебіг у специфікаціях в OpenGL також дуже вплинуло й подія, яка відбулася у липні 2006 року. Вище згадувана рада з перегляду архітектури OpenGL проголосувала за передачу контролю над стандартом OpenGL API компанії Khronos Group. Відбулась подія не на користь стандарту OpenGL.

Так, активна розробка OpenGL була припинена, перевага в цьому напрямку була надана API Vulkan, який був випущений у 2016 році. API Vulkan під час початкової розробки мав кодову назву glNext.

У 2017 році компанія Khronos Group оголосила, про те що OpenGL ES не матиме більше нових версій й відтоді активно зосередилася на розробці Vulkan та інших передових технологіях.

Але як наслідок, не можна сказати, що це погано якось вплинуло на розвиток комп'ютерної графіки, скоріше позитивно. Адже, певні можливості, пропоновані сучасними графічними процесорами, наприклад, трасування променів, не підтримуються OpenGL.

1.3 Чому саме мова програмування C++

Мова програмування C++ (читається як - сі плюс плюс) - це мова, якій вже близько 40 років, яка й досі все ще залишається однією з найпопулярніших мов у всесвіті програмування. Однією із головних переваг даної мови є оптимізація, швидко дія, тому на неї все ще залишається суттєвий попит, особливо коли мова йде про специфікацію програмного забезпечення та про розробку мобільних та комп'ютерних ігор. Вона доволі широко й давно використовується, і відповідно сумісна з багатьма мовами програмування. Не дивлячись на доволі солідний вік для мови програмування, багато розробників сьогодні віддають перевагу C++. Тому вона досі користується великим попитом у різноманітних галузях розробників. [7]

Крім того, велика кількість компанії, які мають топ рівень у своїй сфери, до тепер шукають співробітників з високим рівнем досвіду розуміння та застосування програмування на C++ .

Але все ж таки, розглянемо основні причини чому була обрана саме мова програмування C++ для цієї кваліфікаційної роботи.

Отже:

а) продуктивність та швидкість;

Як було зазначено до цього мова С++ має високу швидкодію, через те що дозволяє низькорівневі маніпуляції з пам'яттю та прямий доступ до апаратного забезпечення. Саме це робить її придатною та привабливою для ресурсоб’ємних додатків, таких як розробка комп’ютерних або мобільних ігор. Також в ній приваблює і вбудовані системи та високопродуктивні обчислення.

б) *підходить для великих проєктів*;

Хоча мобільний ігровий додаток не є великим проєктом й здавалося було б краще і логічніше використовувати мову програмування Java (читається як джава). Але написання програми (роботи) на С++ дає значний поштовх до майбутнього розвитку. А також наштовхує на амбіційні плани для подальшого переростання програми у більш крупніший проєкт, а також доцільний та корисний досвід для майбутньої роботи.

в) *кар'єрні можливості, досвід, та підтримка спільноти С++;*

Як було зазначено вище, написання програми дає можливості здобути досвід для написання (кодування) майбутніх робіт, а в перспективі не тільки для власного досвіду й для кар'єри в цілому, та й для проекту. Також мова С++ більше підтримується спільнотою, через що знаходження розв'язання тих чи інших задач під час написання коду, є більш простою задачею.

г) контроль;

Мова програмування С++ дає дуже багато інструментів та високий рівень контролю над системними ресурсами та апаратним забезпеченням. Це дозволяє покращити оптимізацію для певних задач та потреб С++. Пропонує такі можливості, як ручне керування пам'яттю, вбудована збірка та тонкий контроль над часом життя об'єктів.

г) сумісність;

Мова C++ широко підтримується на різних платформах та архітектурах. Її можна компілювати та запускати на різних операційних системах, включаючи Windows, macOS, Linux та інших системах. А це добре підходить під наші задачі та цілі, а також для розробки мобільного ігрового додатку.

д) гнучкість;

C++ - це мультипарадигмальна мова, яка підтримує як процедурний, так і об'єктно-орієнтований стилі програмування. Вона надає широкий спектр можливостей, включаючи шаблони, перевантаження операторів та поліморфізм, що дозволяє розробникам писати виразний та багаторазовий код.

е) стандартна бібліотека;

C++ постачається з багатою стандартною бібліотекою, яка надає широкий спектр функціональних можливостей, включаючи структури даних, алгоритми, операції вводу/виводу та підтримку паралелізму. Ця бібліотека, разом з бібліотеками Boost C++ Libraries, пропонує широку екосистему готових до використання компонентів.

1.4 Чому саме Pimpl

Головною ідеєю Pimpl є техніка написання програмного забезпечення, яке приховує деталі реалізації класів від його користувачів, а також ця ідея використовується в мові програмування C++. [8][9][10]

Внутрішня частина програми є прихованою, але не дивлячись на це, Pimpl дозволяє вам надати чистий загальнодоступний інтерфейс, зберігаючи деталі реалізації приватними. Таким чином, інтерфейс відокремлюється від іншої програми. Це допомагає зменшити кількість

залежностей. І також, мінімізувати залежності під час компіляції та ізолювати зміни в реалізації. Таким чином, це забезпечує кращу абстракцію та приховування інформації.

Саме це розділення дає переваги у компіляції, бо якщо внутрішні класи змінюються, то не треба перекомпоновувати заголовки, а також інтерфейсні класи. Все це допомагає у бінарній сумісності, з тою ж самою ситуацією, коли можна надати нову версію бібліотеки, не вимагаючи від розробників перекомпілювати свій код.

Ідиома Pimpl може допомогти підтримувати стабільність ABI. Приховуючи деталі реалізації, можна змінювати внутрішнє представлення класу, не впливаючи на бінарну сумісність. Це особливо важливо при створенні бібліотек, які мають бути сумісними з декількома компіляторами або платформами.

Також деталі реалізації обмежуються файлом реалізації, що зменшує кількість заголовків, які потрібно включати в інші файли. Це може допомогти мінімізувати залежності заголовків і скоротити час збирання за рахунок зменшення кількості одиниць перекладу, які потрібно перекомпілювати, коли змінюються деталі реалізації.

Важливо зазначити, що ідиома Pimpl додає додатковий рівень непрямолінійності та може спричинити певні нагальні витрати на продуктивність через додаткове виділення пам'яті та непрямолінійність, пов'язану з цим. Але її слід використовувати розсудливо, а її переваги слід порівнювати з додатковою складністю, яку вона вносить у кодову базу.

Pimpl особливо корисна при роботі з великими класами, класами, що часто змінюються, або коли ви прагнете покращити інкапсуляцію та приховування інформації.

Це потрібно для програми задля експерименту з кодом програми, для більш детального налаштування рендерингу, що б зайвий раз не змінювати заголовки програми.

Мобільні ігрові додатки змінювалися не один раз, тож треба зазначити що в нього був перехід з 2D у 3D графіку, що знову таки показує вигоду ідіоми Pimpl у написанні цього проєкту. Також цей метод був обраний для його дослідження та отримання нового досвіду, який може бути корисним у подальшому для саморозвитку та кар'єри.

Треба зазначити також, що у цій ідіомі дуже зручне розділення класів, що не раз було корисно, а також у банальній зручності.

1.5 Головні патерни програми

1.5.1 Патерн “Singleton”

Патерн “Singleton” або в перекладі на українську мову - шаблон “Одинак”. [11]

“Singleton” - це патерн (шаблон) проєктування головною ідеєю якого є те, що клас може мати тільки один екземпляр, та не може мати ніяких більше його копій. Також, він надає глобальну точку доступу до нього. Таким чином цей патерн надає нам беззаперечну гарантію того, що клас буде одним єдиним для всієї написаної програми, щось на кшталт, як база даних.

Розглянемо випадки, коли патерн “Singleton” може бути корисний. Отже, по-перше, коли наявність декількох екземплярів класу може спричинити проблеми. Для прикладу, можна взяти деякі конфлікти у спільних ресурсах або наприклад, суперечливі дані.

По-друге, патерн “Singleton” зазвичай використовують, коли конче потрібен один спільний ресурс. При цьому, доступ до даного ресурсу завжди можна отримати з різних частин програми. Ця можливість добре об'єднується та корелюється принципом Pimpl, який був розглянутий у попередніх розділах цієї роботи.

Патерн “Singleton” можна побачити не тільки у всесвіті програмування або інших дотичних до програмування галузях, його можна побачити й в реальному житті. Як приклад, цього патерну у реальному житті - Верховна Рада України, яке є одною єдиною законодавчою владою, яка має певну структуру й не має більше аналогів (копій) в Україні. Також, як приклад, це може бути законодавчий або виконавчий органи влади будь якої іншої країни світу.

Одна із ключових особливостей при написанні даного класу - це саме команда *getInstance*. Адже, увесь клас повністю повинен бути прихованим від клієнта. А виклик даного методу являє собою один єдиний існуючий спосіб, аби отримати об'єкт зазначеного класу. Ще також один із способів надання глобальної точки доступу до цього класу.

Але, як і будь що в цьому світі, патерн “Singleton” може мати і недоліки, що теж важливо зазначити. Так, наприклад, він може ускладнювати процес тестування та супровід. Й також може потенційно створювати тісні зв'язки в коді.

Тому патерн “Singleton”, як і всі інші патерни, слід використовувати з розумом. Використовувати його у програмуванні треба тоді, коли це дійсно необхідно.

1.5.2 Патерн “Observer”

Патерн “Observer” або в перекладі на українську мову шаблон “Спостерігач”. [12]

Патерн “Observer” - це такий принцип написання класу, який може зберігати список, як отримав назву - список “підписок”. Він включає в себе перелік інших класів, які використовуються для надсилання інформації та зміни його стану. Також цей список можна назвати як “один-до-багатьох”. Даний патерн дозволяє усівм створеним у класі об'єктам спілкуватися та синхронізуватися один з одним у слабко пов'язаний спосіб.

Можна, навіть, навести аналогію цього патерну з нашого реального життя, це може бути банальне оформлення підписки на отримання журналів. Ба правда зараз, це доволі не дуже часто можна зустріти у сучасному світ. Адже, зараз більше користуються попитом підписки на усякі стрімінгові канали в інтернеті.

Але так чи інакше, все ж повернемося до прикладу умовної підписки на свіжі випуски, якогось журналу. Отже, маючі підписку, клієнку більше не потрібно йти до магазину або газетного кіоску, аби дізнатися, чи вже вийшов черговий номер бажаного журналу. На заміну зазначеного процесу, саме видавництво журналу тепер надсилатиме нові випуски, як тільки вони вийдуть з друку, прямо поштою до підписника додому.

Отже, маючи перелік підписників, в якому зазначені такі дані про підписника як: його прізвище та ім’я, його адреса, куди саме надсилати журнал, й відповідно назва журналу на який оформлена підписка, видавництво робить розсилку. Підписник також може в будь-який момент відмовитися від підписки, й тоді видавництво перестане надсилати йому журнал й той відповідно перестане до нього надходити.

Далі будемо розглядати ключових учасників, які присутні у патерні “Observer”.

Отже, *учасниками патерну ”Observer” є:*

1) *Об'єкт*.

Також відомий як “Observable” або “Publisher”. Даний об'єкт, завжди який має валідний список своїх залежних спостерігачів та повідомляє їх про будь-які зміни стану.

2) С*постерігач.*

Також відомий як абонент або слухач. Це є інтерфейс або абстрактний клас, який визначає певний перелік для певних об'єктів, які будуть повідомлені про зміну стану суб'єкта.

3) *Конкретний об'єкт.*

А саме, це реалізація об'єкта. Об’єкт підтримує стан та надсилає сповіщення спостерігачам, які є зареєстрованими.

4) *Конкретний спостерігач.*

Реалізація інтерфейсу спостерігача. Спостерігач реєструється з конкретним суб'єктом для отримання необхідних сповіщень та реалізує логіку оновлення, якої саме потребує.

1.5.3 Патерн “Composite”

Патерн “Composite” в перекладі на українську мову - шаблон “Композиція”. [13]

Отже, патерн “Composite” - це є патерн проєктування в об'єктно-орієнтованому програмуванні (ООП), який надає змогу обробляти конкретні

окремі об'єкти та групи об'єктів в однаковий спосіб. Шаблон “Композиція” має призначення, щоб мати змогу представляти ієрархічні структури об'єктів у такий спосіб, який би дозволяв клієнтам мати змогу для однакової взаємодії з конкретними окремими об'єктами та групами об'єктів. Цей патерн дає змогу як компонувати об'єкти у деревоподібні структури, так і працювати з ними в такий спосіб, ніби вони є окремими об'єктами.

Отже, за традицією можна навести приклад з реального життя - це може бути будь-яка ієрархія любої структури. Для прикладу, можемо взяти поліцейський департамент, або армію тощо. Можна умовно розділити на головного керівника, його під керівники (різноманітні зами по різним напрямкам), а під ними всі інші працівники, які теж мають певну ієрархію, але вони всі разом підлеглі для головного керівника. Відповідно, таким чином, головний керівник дає команду на запит, а далі цей запит йде по вертикалі вниз поки не зупиняється на виконавці даного запиту. При цьому запит може пройти через декілька верхівок ієрархії.

Використовуючи шаблон “Композиція”, з’являється можливість створювати доволі складні структури для об'єктів, які можна обробляти в один і той же спосіб. Клієнти мають змогу взаємодіяти зі структурами на доволі різних рівнях абстракції. При цьому, можуть розглядати їх як самостійні окремі об'єкти або як групи об'єктів. Цей патерн особливо дотепний у використанні, коли ви маєте рекурсивну структуру. От як, для прикладу, дерево, де ви хочете виконувати різноманітні операції над усією структурою або у вас є потреба працювати тільки виключно з окремими його підструктурами.

З огляду на вищезазначене, можна дійти певного висновку, що шаблон “Композиція” може сприяти не тільки гнучкості та масштабованості в

управлінні ієрархічних структур об’єктів, а і в їх простоті під час управління цих структур.

Для повноти висвітлення інформації про шаблон “Композиція”, слід також розглянути та коротко розповісти про кожну із основних частин, із яких складається даний патерн. А саме: компонент, складеник та листок.

Отже, далі коротко розглянемо кожну частину патерну окремо.

Основні частини патерну:

1. *Компонент.*

Загальний інтерфейс або так званий, абстрактний клас, який визначає певну поведінку. Цю поведінку повинні реалізовувати як окремі його об'єкти, так й групи об'єктів. Зазвичай він включає в себе такий перелік операцій, як: додавання, видалення, а також доступ до дочірніх компонентів.

2. *Складник.*

Представляє собою групу об'єктів або вузол у структурі дерева. Він реалізує інтерфейс Компонент і може містити колекцію дочірніх компонентів. Він надає методи для маніпулювання дочірніми компонентами, а також може визначати додаткову поведінку, специфічну для груп.

3) Листок.

Самостійний окремий об'єкт або вузол листа в структурі дерева. Листок реалізує інтерфейс *компонента* (одна із складових частин патерну “Composite”), але не має своїх дочірніх компонентів. Він визначає поведінку, характерну для окремих об'єктів.

1.5.4 Патерн Factory Method

Патерн “Factory Method” в перекладі на українську мову - шаблон “Фабричний метод” [14]

Даний шаблон проєктування, це шаблон, який надає інтерфейс для створення об'єктів. Але при цьому надає змогу своїм підкласам самостійно вирішувати, який саме клас при цьому створювати. Патерн “Factory Method” частіше за все використовується, в тому випадку, коли клас не може передбачити тип для об'єктів, які йому необхідно створити. Іноді ще цей патерн використовується коли клас хоче, щоб його підкласи визначали саме ті об'єкти, які вони самі створюють.

Основна концепція шаблону “Фабричний метод” полягає в інкапсуляції логіки створення об'єктів в окремому методі або класі. Це метод має однойменну назву патерна - фабричний метод. Даний метод (клас) потрібен для того аби створювати різноманітні об'єкти безпосередньо за допомогою конструктора. Даний спосіб, сприяє послабленню зв'язку між творцем та виробом, який нім створюється. Така тенденція відбувається із-за того, що творець залежить лише від абстрактного інтерфейсу створеного виробу, а не від конкретних його реалізацій.

Шаблон “Фабричний метод”, також сприяє більшій розширюваності класу. Таке сприяння дозволяє додавати нові підкласи без модифікації вже існуючих класів-створювачів. Патерн також добре відповідає принципу “відкритості-закритості”, оскільки дозволяє додавати нові типи виробів без модифікації клієнтського коду.

Патерн “Фабричний метод” надає прекрасну можливість не тільки делегувати відповідальність за створення об'єктів підкласам, а ще й надає їм гнучкість у створенні конкретно визначених екземплярів, зберігаючи при цьому спільний інтерфейс.

РОЗДІЛ 2 ПРАКТИЧНА ЧАСТИНА

2.1 Засоби розробки

Через те, що цільовою платформою програмного продукту, що розробляється, є Android, цілком логічно було обрати за інтегроване середовище розробки (IDE) Android Studio. Наразі використовується остання версія зазначеного IDE, а саме Android Studio Flamingo 2002.2.1.

Для сучасних пристроїв Android використовується OpenGL ES версії 3.0 і вище. Також зберігається зворотна сумісність, тобто за необхідності є можливість і досі використовувати OpenGL ES 2.0. Але для розробки кваліфікаційної роботи було обрано саме версію 3.0, що забезпечує сумісність з переважною більшістю сучасних приладів Android.

В процесі написання та налагодження програмного коду не використовувався фізичний пристрій. Натомість перевірка і налагодження відбувалась на емуляторі. Слід зазначити, що керування віртуальними пристроями, а саме створення та налаштування, дуже зручно реалізовано в IDE Android Studio. Використання емулятора значно спрощує процес розробки та налагодження коду на відміну від використання фізичного пристрою. Слід зазначити, що вагомим мінусом емулятора є відчутно менша продуктивність у порівнянні з фізичними пристроями. Проте використання емуляторів не потребує жодних фінансових інвестицій.

Отже, для розробки та демонстрації кваліфікаційної роботи було обрано емуляцію фізичного пристрою Pixel C, а застосована версія API 33 (Tiramisu).

2.2 Створення проєкту

Через той факт, що кінцевий програмний продукт є ігровим застосунком, для створення нового проєкту якнайкраще підходить опція: **New Project → Game Activity (C++)**. Цей шаблон проєкту передбачає наступне:

1. створення класу **MainActivity** що є нащадком бібліотечного **GameActivity** (Рисунок 2.1);
2. створення інтерфейсу нативної бібліотеки;
3. генерація допоміжного коду, що забезпечує весь необхідний зв’язок між Java-кодом застосунку та C++-кодом нативної бібліотеки.

```
import com.google.androidgamesdk.GameActivity;
public class MainActivity extends GameActivity {
        static {
                System.loadLibrary("sokoban");
        }
}
```

Рисунок 2.1 - Приклад коду MainActivity та завантаження нативної бібліотеки.

Слід зазначити, що застосування виключно засобів Java для розробки ігрового застосунку з використанням бібліотеки OpenGL (GLES) є цілком можливим та вельми природним для Android. Проте застосування нативної бібліотеки забезпечує максимальну продуктивність. В рамках даної кваліфікаційної роботи не передбачено демонстрацію переваги використання нативної бібліотеки у порівнянні із застосунком на Java, проте використання цього підходу є цілком виправданим з точки зору поглиблення знань на придбання досвіду.

2.3 Інтерфейс нативної бібліотеки

Інтерфейс нативної бібліотеки було створено засобами автоматичної генерації коду Android Studio. Його розміщено у файлі main.cpp за замовчуванням і він складається всього з трьох викликів:

- handle_cmd
- motion_event_filter_func
- android_main

Розглянемо цей інтерфейс більш детально.

Функція **handle_cmd** обробляє команди, надіслані до бібліотеки з боку Android. Вона приймає два аргументи: вказівник на програму (struct android_app *pApp), та команду, що необхідно обробити (int32_t cmd). Наразі з всього спектру можливих команд розглядається дві:

- **APP_CMD_INIT_WINDOW** – створення вікна. Бібліотека реагує на цю команду створенням об’єкта центрального класу AppCore, що створює решту компонентів та сервісів гри.
- **APP_CMD_TERM_WINDOW** – знищення вікна. Бібліотека викликає по цій команді робить вивантаження всіх створених ресурсів та очищує виділену пам’ять.

Решта команд ігнорується, але може бути розглянута в майбутньому за необхідності.

Функція **motion_event_filter_func** відповідає за фільтрацію подій та приймає єдиний параметр: константний вказівник на подію (const GameActivityMotionEvent *motionEvent). Наразі бібліотека відфільтровує тільки події із двох джерел: вказівника та джойстика.

- **AINPUT_SOURCE_CLASS_POINTER**
- **AINPUT_SOURCE_CLASS_JOYSTICK**

Як і у випадку функції **handle_cmd** решта джерел може бути додана в майбутньому за необхідності.

Функція **android_main** є головною точкою входу для нативної діяльності. Вона приймає вказівник на програму (**struct android_app *pApp**) та забезпечує виклик головного циклу застосунку. Структура головного циклу буде детально розглянута у пункті 2.4.

На Рисунку 2.2 відображено базову основу бібліотеки, а саме: підключення бібліотеки JNI (Java Native Interface), трьох файлів із кодом, що є автоматично

згенерованим засобами Android Studio, та, безпосередньо, три інтерфейсні функції, що було описано вище.

```
#include <jni.h>
#include <game-activity/GameActivity.cpp>
#include <game-text-input/gametextinput.cpp>
extern "C" {

#include <game-activity/native_app_glue/android_native_app_glue.c>

void handle_cmd(android_app *pApp, int32_t cmd) {
    // . . .
}

bool motion_event_filter_func(
                const GameActivityMotionEvent *motionEvent) {
    // . . .
}

void android_main(struct android_app *pApp) {
        // . . .
}
}  // extern "C"
```

Рисунок 2.2. – Вигляд main.cpp

2.4 Центральний клас застосунку AppCore

В центрі архітектури застосунку знаходиться клас **AppCore**. Цей клас відповідає за життєвий цикл всіх компонентів, таких як рендеринг, інтерфейс

користувача, різноманітні менеджери. Він обслуговує два інтерфейсні виклики, а саме **handle_cmd** та **android_main**. У першому випадку **AppCore** створює всі компоненти програми, або вивантажує їх та очищає пам’ять (в від команди, що було отримано). У другому випадку **AppCore** послідовно викликає свої три публічні методи.

1. handleInput - обробка подій вводу (клавіатура, миша, тачскрин тощо);
2. update - оновлення всіх внутрішніх компонентів програми;
3. render - рендеринг нового кадру.

Клас **AppCore** реалізує шаблон програмування *Singleton*. Він має закритий (приватний) конструктор і статичний метод **getInstance**. Саме виклик цього методу створює єдиний екземпляр класу і повертає посилання на цей екземпляр.

В середині **AppCore** клас мусить займатись опрацюванням більшості компонентів всієї гри (менеджер завдань, менеджер подій, ігровий менеджер, рендеринг, інтерфейс користувача). Для того, щоб прибрати із декларації класу згадку про всі перелічені вище компоненти, клас **AppCore** реалізовано за допомогою стратегії *PImpl*. Таким чином файл заголовка класу має тільки попереднє оголошення типу **AppCoreImpl** (*forward declaration*), а сам клас містить приватний вказівник на цей клас. Декларація класу **AppCoreImpl** та його реалізація винесені в cpp-файл. Підключення всіх необхідних файлів заголовків, що описують необхідні компоненти гри, також зроблено саме в cpp-файлі.

Таким чином клас **AppCore** перенаправляє всі зовнішні виклики безпосередньо класу **AppCore**, а підключення файлу заголовка **AppCore.h** не призводить до рекурсивного підключення інших файлів заголовків, які відповідають за решту компонентів гри. Це і є перевага використання стратегії *PImpl* - вся залаштункова робота прихована від користувача.

2.5 Основні компоненти та підсистеми проєкту

Серед всіх компонентів програми слід виділяти 5 основних, за життєвий цикл яких відповідає AppCore. Слід розуміти, що насправді відповідальність лежить на класі реалізації, тобто AppCoreImpl, але якщо розглядати це з точки зору користувача, який не знає тонкощів внутрішньої реалізації, то виглядає, наче саме AppCore є агрегатором основних компонентів. Діаграму класів зображено на Рисунку 2.3.

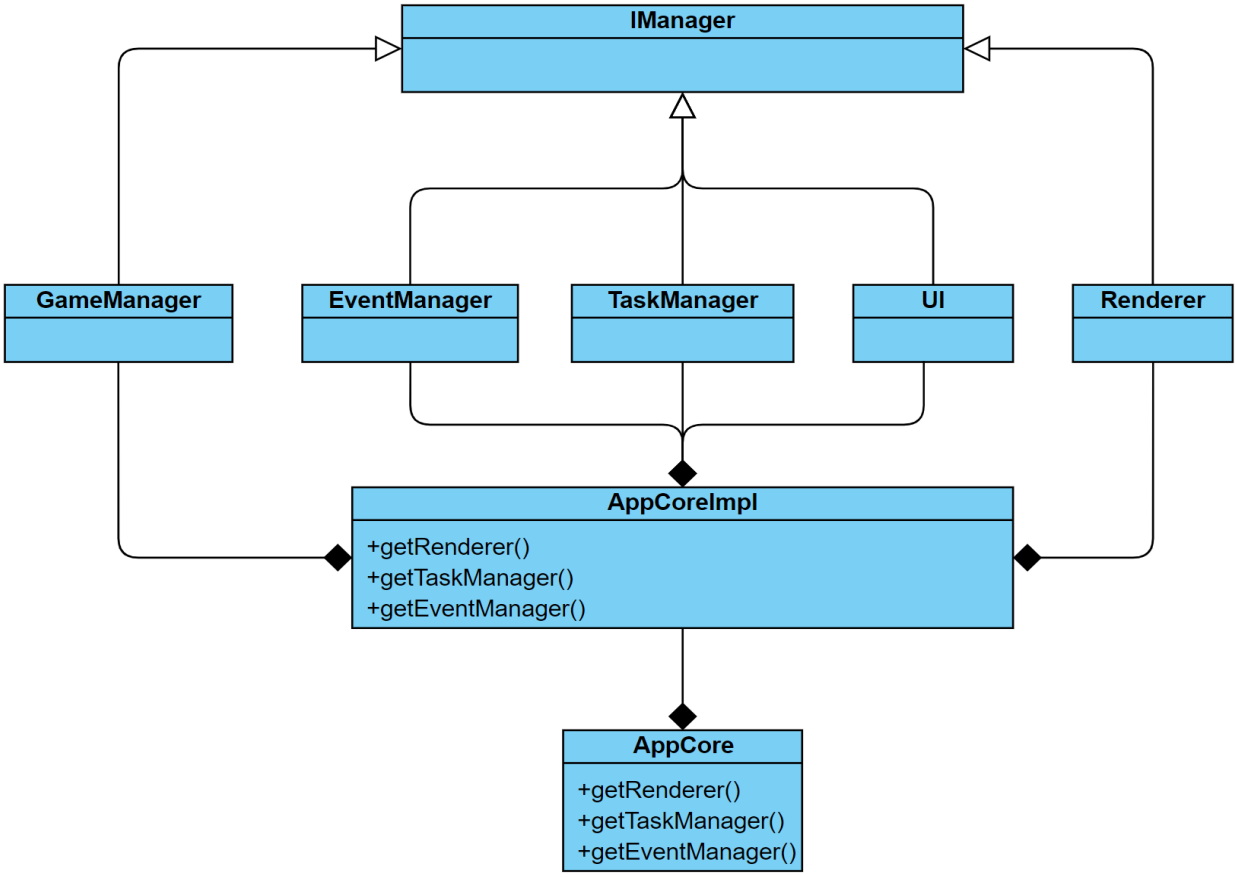

Рисунок 2.3. - Діаграма класів.

Базовим класом для п’яти основних компонентів є клас **IManager** (Рисунок 2.4). Насправді він не надає який-то певний інтерфейс, натомість він приховує конструктор копії та оператор присвоювання. По суті це є власна

реалізація широко відомого *класу boost::noncopyable*, але використання бібліотеки boost не передбачалось в рамках цього проєкту.

```
class IManager {
public:
   IManager() = default;

private:
   // Noncopyable
   IManager(const IManager&);
   IManager& operator=(const IManager&);
};
```

Рисунок 2.4. Клас IManager.

Повертаючись до Рисунку 2.3, слід зазначити, що відносини між компонентами та класом **AppCoreImpl** є композицією (*composition*), а не агрегацією (*aggregation*), тому що AppCoreImpl відповідає за життєвий цикл цих компонентів: він створює та видаляє кожен з них.

2.5.1 Менеджер подій

Менеджер подій – **EventManager** – відповідає за отримання подій від Android Application та сповіщення про подію, що відбулась, всіх своїх підписників. Реалізацію подібної взаємодії описує шаблон програмування “Спостерігач” (*Observer*). Наразі реалізовано роботу з двома класами подій: події миші та події клавіатури.

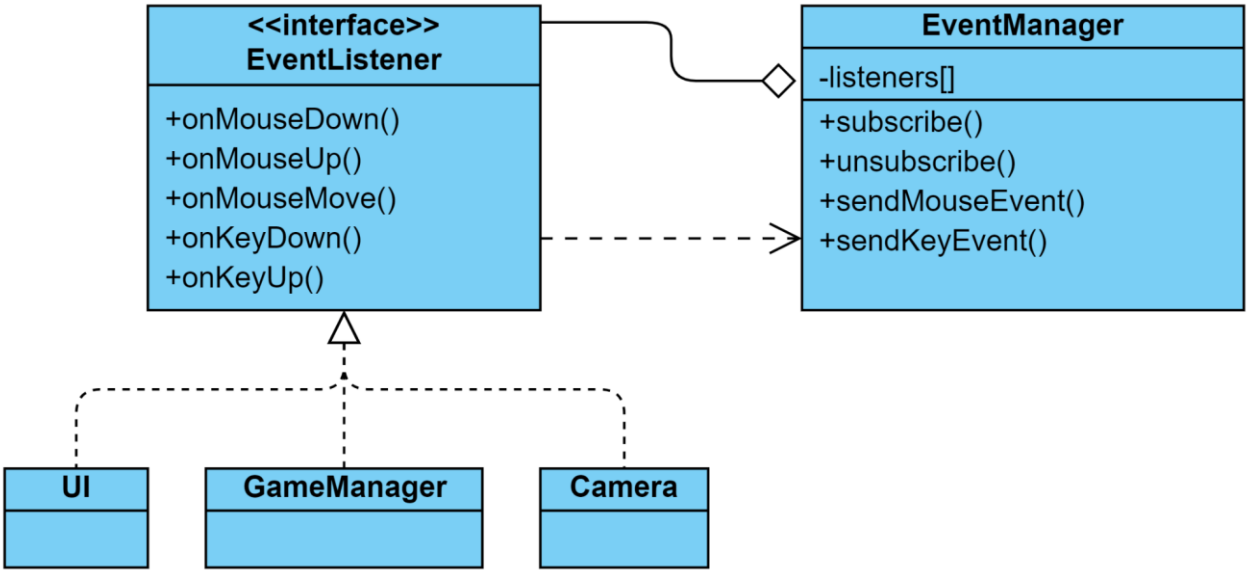

Рисунок 2.5. - EventManager.

Проте реалізація відрізняється від канонічної. Базовий клас **EventListener** сам підписує свій екземпляр на отримання подій у власному конструкторі, і відписує від отримання подій в деструкторі. Інакше кажучи, той, хто створює екземпляри нащадків **EventListener** вже є позбавлений від обов'язку підписувати щойно створені об'єкти самостійно. Таке легко зробити враховуючи те, що будь-хто може отримати доступ до **EventManager** звернувшись до сінглтону **AppCore**.

2.5.2 Менеджер завдань

Менеджер завдань – **TaskManager** (Рисунок 2.6) – відповідає за агрегацію завдань (об'єктів класу **Task**), постійне оновлення статусу цих завдань та їхню фіналізацію.

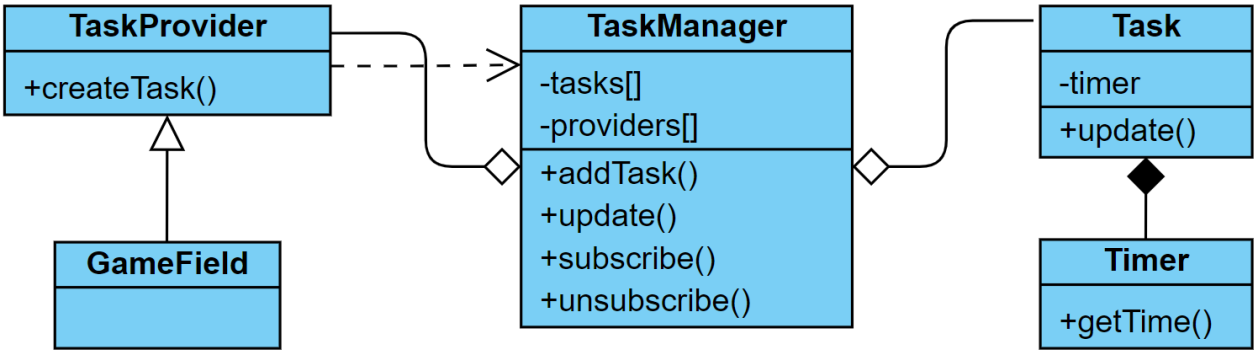

Рисунок 2.6. TaskManager

При створенні нового завдання вказується час (в секундах), протягом якого це завдання мусить існувати. Кожен раз, коли відбувається оновлення завдання (*update*), об’єкт класу **Task** отримує два параметри: число, що є результатом інтерполяції від 0 до 1, де 0 – це момент створення завдання, а 1 – це момент, коли час існування завдання вичерпано, та безпосередньо час (в секундах), що минув від моменту створення завдання. Операція фіналізації, про яку було зазначено вище, є такою ж самою операцією оновлення з єдиною відмінністю: якщо час існування завдання вичерпано, то функція update для об'єкта завдання буде викликана з такими аргументами: 1 замість реального числа інтерполяції, та час існування завдання замість реального часу, що минув від моменту створення завдання. Це необхідно для того, щоб нівелювати невеликі розбіжності для цих двох параметрів, тому що їх реальні значення будуть завжди трішки більші, ніж максимально очікувані. Таке відбувається через те, що фіналізація відбувається тоді, коли реальний час існування завдання стає більшим за той час, що було вказано при створенні. Після того, як завдання фіналізоване, воно видаляється із менеджера завдань.

Також додана можливість створення нескінченних завдань. Для цього при створенні слід передати -1 в якості часу існування завдання. У цьому випадку завдання ніколи не буде фіналізовано і видалено, але протягом всього терміну існування буде отримувати час (в секундах) що минув від моменту його створення.

Ще одним аспектом, що враховується при створенні завдання, є функція інтерполяції (Рисунок 2.7). Наразі реалізовано три функції: лінійна (*Linear*), експоненціальна (*Exponential*) та логарифмічна (*Logarithmic*). Система побудована таким чином, що додавання нових функції, що будуть задавати форму кривої, є абсолютно прозорим для системи і не потребує жодних додаткових втручань.

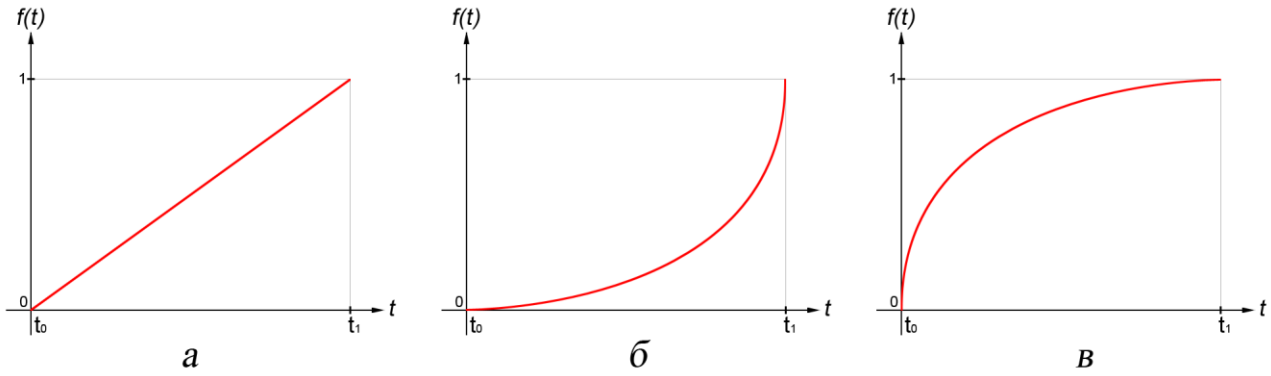

Рисунок 2.7. а) лінійна форма; б) експоненціальна форма; в) логарифмічна форма

Наразі завдання використовується для створення плавної анімації переміщення об'єктів на ігровому полі. Також в майбутньому буде використано для реалізації плавного анімованого переміщення камери над ігровим полем.

2.6 Ігрова підсистема

В цьому розділі розглянуто всі класи, що відповідають безпосередньо за ігровий процес: ігрові об’єкти, ігрове поле, менеджер, що всім керує, та фабрика, що створює об’єкти.

2.6.1 Ігрові об’єкти та поле

Царина ігрових об’єктів – це класичний приклад для демонстрації принципів ООП. Маємо базовий клас **Object**, в якому об’явлено декілька абстрактних методів (**getId**, **getTexture**), та п’ять класів-нащадків, що реалізують ці методи (Рисунок 2.8).

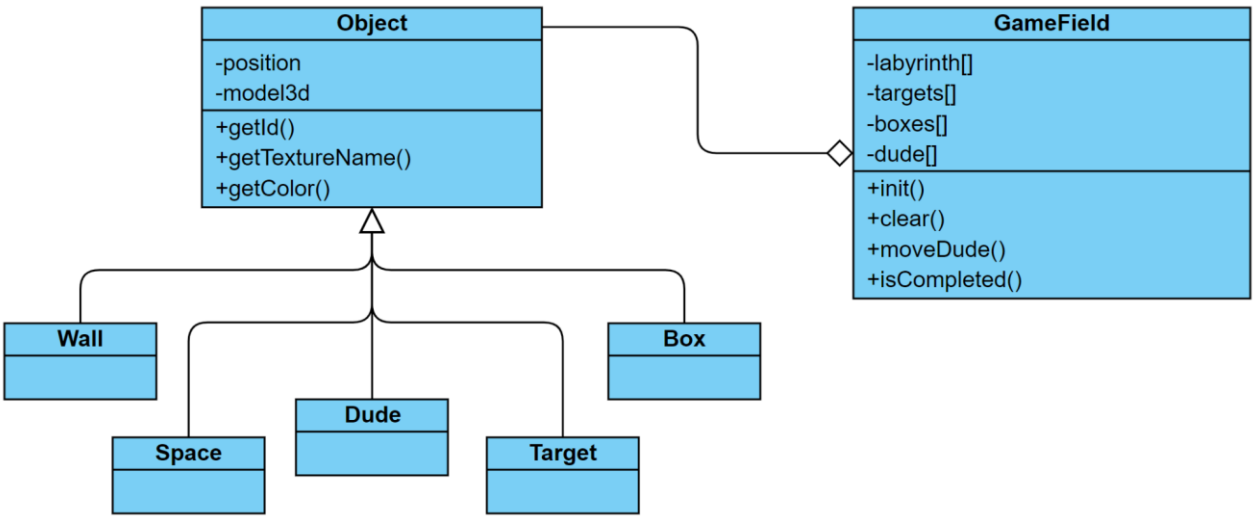

Рисунок 2.8. - Ігрові об’єкти та ігрове поле

Базовий клас інкапсулює в собі координату, де саме знаходиться об’єкт на ігровому полі, а також вказівник на 3D-модель, що буде намальовано графічною підсистемою. Віртуальні методи **getId**, **getTexture** та **getColor** застосовуються фабричним методом при створенні об’єктів. Кожен нащадок по своєму перевизначає реалізацію цих методів, тим самим в унікальний спосіб налаштовує власні характеристики.

Ігрове поле зберігає всі об’єкти в чотирьох масивах:

- *labyrinth* – містить об’єкти класів **Wall** та **Space**, що задає лабіринт, в якому знаходяться та пересуваються інші об’єкти;
- *targets* – містить тільки об’єкти класу **Target**;
- *boxes* – містить тільки об’єкти класу **Box**;
- *dude* – це хоча і є масив, але по факту завжди містить тільки один об’єкт класу **Dude**.

Може показатись дивним, що для зберігання одного об’єкта класу Dude використовується масив, а не звичайна змінна. Проте всі приватні методи класу, що виконують всю роботу, побудовані таким чином, щоб працювати з масивом, який передається як параметр. Саме звідси виникло таке архітектурне рішення. При

створенні ігрового поля перевіряється, що кількість об’єктів класу **Dude** мусить строго дорівнювати 1. В іншому випадку вважається, що дані не є валідними та подальше виконання програми не є можливим.

Клас ігрового поля має найважливіший метод **moveDude** що реалізує, власне, весь ігровий процес. Цей метод приймає двовимірний вектор, що задає бажаний напрямок переміщення персонажу, виконує перевірку, чи можливе таке переміщення, і виконує його, якщо немає перешкод для виконання. Це може змінити стан ігри з “процес триває” на “завдання виконано”. За цю перевірку відповідає метод **isCompleted**, який після кожного ходу перевіряє ігровий менеджер.

2.6.2 Фабрика об’єктів

Через те, що робота фабрики об’єктів ніяк не залежить від зовнішніх факторів, а в системі достатньо мати одну фабрику, щоб забезпечити процес створення, цей клас реалізує шаблон програмування *Singleton*. Таким чином клас фабрики **Factory** має приватний конструктор, статичний метод, що створює єдину копію класу та повертає посилання на цю копію, та всього один публічний метод **produce**, який виконує всю необхідну роботу зі створення об’єктів.

Слід зазначити, що метод **produce** реалізує шаблон програмування *Factory Method*. Цей метод приймає в якості аргументу *ObjectId* – енумератор чинних у грі класів об’єктів. Клас **Factory** містить асоційований масив так званих “творців”, кожен з яких відповідає за певний вид об'єкта. Ключем асоційованого масиву є *ObjectId*, тому вся робота зводиться до пошуку “творця” по ключу і передача йому команди на створення об’єктів. Кожен “творець“ завжди повертає два вказівники. У випадку об’єктів лабіринту, таких як **Wall** та **Space**, перший вказівник буде відповідати за створений об’єкт відповідного типу, а другий завжди буде порожнім. У решті випадків (**Dude**, **Box**, **Target**) перший вказівник завжди буде вказувати на щойно

створений об’єкт класу **Space**, а другий відповідно до того, який *ObjectId* було передано.

Така “двошарова” поведінка необхідна для того, щоб окремо створити й сам лабіринт (нижній шар), й ігрові об’єкти (верхній шар), причому оперуючи тільки одним параметром для обох шарів.

2.6.3 Ігровий менеджер

Цей клас створено для зв’язку ігрової механіки з рештою програми. **GameManager** агрегує в собі ігрове поле, відповідає за ініціалізацію процесів створення та очищення чергового рівня гри, перевірки та реакції на подію, що рівень завершено. З іншого боку **GameManager** приймає події вводу, джерелами яких є клавіатура, миша тощо.

2.7 Графічна підсистема

Центральний клас 2D/3D графічної підсистеми – це **Renderer**. Саме він відповідає за ту картинку, яку користувач бачить на екрані. Будь-які процеси в програмі, які хочуть показати користувачеві якусь інформацію мусять в той чи інший спосіб співпрацювати з рендерингом.

Після запуску програми створюється екземпляр класу **Renderer**, який одразу в конструкторі викликає приватний метод ініціалізації. Мусить пройти налаштування таких трьох сутностей:

- EGLDisplay;
- EGLSurface;
- EGLContext.

Коли ці сутності OpenGL успішно створені, можна переходити до малювання 2D зображень та 3D сцен.

2.7.1 Моделі

В програмі реалізовано базовий клас моделі **Model**, що агрегує два масиви: вертексів та індексів. Вертекси описують координати вершини в локальній системі координат об’єкта, містять текстурні координати або колір вершини об’єкта. Другий масив – масив індексів – містить інформацію про трикутники: набіг груп по три елементи, що є номерами вершин у вертексному масиві, і складає один трикутник. Відомий факт: навіть якщо ми хочемо намалювати прямокутник, ми мусимо намалювати два трикутники.

Також існує два класи-нащадки: Model2D та Model3D (Рисунок 2.9). Вони реалізують допоміжні механізми щодо моделей свого типу, а саме двовимірних спрайтів, що використовуються для створення інтерфейсу користувача, та тривимірних моделей, з яких складається ігровий світ (сцена).

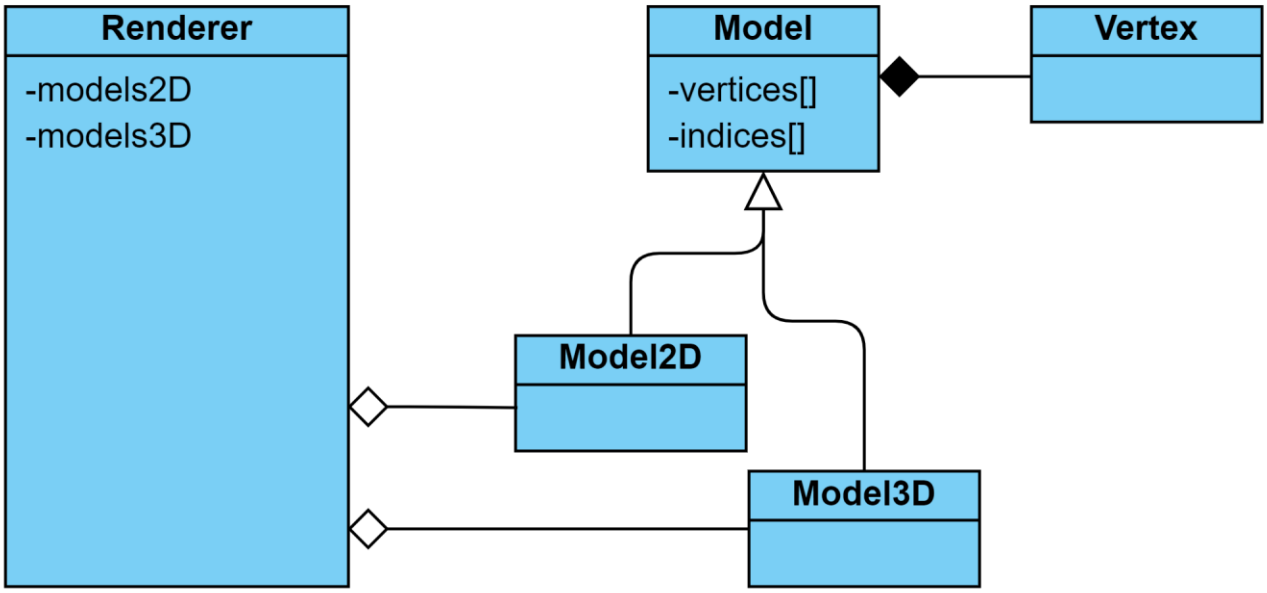

Рисунок 2.9. - 2D та 3D моделі.

Всі моделі, що створюються, мусять бути додані до відповідних колекцій рендерера, бо саме ці масиви є джерелом інформації для малювання. Інакше кажучи, якщо модель не додано в колекцію рендерера, вона ніколи не буде намальована, або

для цього прийдеться робити якісь штучні умови, писати додатковий код, та порушувати дизайн проєкту, що існує.

2.7.2 Шейдери

Для того, щоб відобразити модель на екрані, треба виконати спеціальну програму, яка називається шейдером (*shader*). Шейдери можуть створюватись на різних мовах, це залежить від того, яка бібліотека наразі використовується. Для цього проєкту це *GLSL – Graphic Library Shader Language.*

Як і у випадку з розподілом на двовимірні та тривимірні моделі, в даному проєкті розроблено окремі шейдери для цих типів моделей. В сучасних 3D рушіях набір шейдерів складається із десятків, а то і сотень різних програм, реалізована статична та динамічна генерація програмного коду шейдерів тощо. Наразі для поточного стану проєкту цілком вистачає двох шейдерів: **Shader_2D** та **Shader_3D**.

Обидва класи шейдерів працюють у вельми схожий спосіб: завантажують шейдер у вигляді тесту програми, компілюють його, та чекають на вхід модель, яку треба вивести на екран.

2.7.3 Матриці трансформації

Для того, щоб правильно відобразити 3D-об’єкт на екрані будь-якого пристрою, треба провести деяку математичну підготовку.

Що мається на увазі: кожна модель, по суті, являє собою набір вершин, що задані координатами у локальній системі координат. Але для того, щоб об'єкт виглядав на екрані природним чином, необхідно провести трансформацію цих координат із 3D простору у 2D простір екрана. А якщо взяти до уваги, що ми хочемо відображати не один об'єкт, а цілу сцену, що створено із багатьох 3D елементів, то такі перетворення координат стають ще більше об’ємними.

Для таких перетворень нам необхідно мати три матриці: матрицю перетворення локальних координат об'єкта у світові координати (або координати

сцени), матрицю перетворення камери та матрицю проєкції перспективи. Тобто матрицю трансформації можна вирахувати по формулі:

$$Transform = Projection * View * World \quad (1)$$

Світову матрицю перетворення можна отримати виходячи із наступної формули:

$$World = Translate * Rotate * Scale \quad (2)$$

Тобто спочатку треба створити три допоміжні матриці (переміщення, повороту та масштабу), а потім порахувати добуток цих матриць – це і буде світова матриця для певного об’єкта сцени.

На даному етапі розробки об’єкти на сцені не обертаються, тому матриця повороту є одиничною матрицею. Таким чином цей множник можна взагалі ігнорувати. А дві інші матриці будуються на базі інформації, яку безпосередньо містить ігровий об’єкт.

2.7.4 Математична бібліотека GLM

Для роботи з векторами та матрицями в проєкті використовується популярна бібліотека OpenGL Mathematics (GLM). Ця бібліотека широко використовується більше десятка років, є дуже популярною і потужною.

До прикладу, щоб збудувати світову матрицю із п. 2.7.3 можна використати дві бібліотечні функції: **glm::translate** та **glm::scale**. Для створення матриці перспективної проєкції також дуже зручно використати бібліотечну функцію **glm::perspective**, що приймає такі аргументи як: кут огляду камери (*FOV - Field of View*), відношення висоти екрана на його ширину, та дві відстані – до ближньої та задньої Z-площини проєкції.

Третій матриці перетворення – *View* – буде присвячено окремий параграф.

2.7.5 Камера

Камера в комп’ютерних іграх – це така математична сутність, яка відповідає за обчислення матриці перетворення *View*, що є однією із трьох матриць трансформацій (формула 1).

В залежності від потреб програмного застосунку та загальноприйнятих тенденцій UX (*User Experience*) та чи інша схема керування камерою буде максимально зручною для користувача. Тут не йдеться тільки про відеоігри, тут мається на увазі всі аспекти застосування 3D графіки: 3D-редактори, програми для створення відеоефектів для кінематографа, системи автоматизованого проєктування (САПР) та багато інших галузей.

Самою простою математичною моделлю, що описує камеру, є система чотирьох векторів: позиція камери та трійка векторів, що задають поворот матриці. На базі цих векторів будуються дві відповідні матриці: переміщення (*Translate*) та повороту (*Rotate*).

$$C_t = \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad C_r = \begin{bmatrix} T_x & T_y & T_z & 0 \\ U_x & U_y & U_z & 0 \\ R_x & R_y & R_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ct – матриця переміщення камери, *P* – вектор, що задає позицію камери (*Position*).

Cr – матриця оберту камери (*Camera Rotate*), T – вектор, спрямований на ціль камери (*Target*), U – вектор, спрямований вгору від камери (*Up*), R – вектор, спрямований праворуч від камери (*Right*).

У випадку, якщо ми повністю виключаємо так званий крен камери, достатньо всього двох векторів Target та Up, а третій вектор Right буде обчислено як векторний добуток перших двох.

Таким чином маємо формулу для обчислення матриці камери:

$$View = CamRorarte * CamTranslate \quad (3)$$

Отже, формул (1), (2) та (3) цілком достатньо, щоб відображати 3D сцену.

2.8 Інтерфейс користувача

Будь-яка програма, навіть найпростіша шкільна задача з інформатики, мусить мати хоча б мінімальний інтерфейс користувача - UI (*User Interface*). Запрошення строки вводу, пояснення до даних, що виводяться на екран. Тим більше інтерфейс інтерактивних графічних застосунків мусить відповідати очікуванням та вимогам користувача. Він мусить бути інтуїтивно зрозумілим та зручним у використанні.

2.8.1 Віджети

Базовий клас **Widget** є фундаментом для всіх об’єктів інтерфейсу: кнопок, рядків вводу, вікон тощо. В цьому проєкті реалізована тільки необхідна частина від всього різноманіття інтерфейсних віджетів: кнопка (*Button*), Рисунок (*Image*) та вікно (*Window*) (Рисунок 2.10).

Віконна графічна система реалізує шаблон програмування Компонувальник (*Composite*). Кожен елемент інтерфейсу може бути представлений як вказівник на батьківський клас. У кожного елемента можна отримати вектор вказівників на “дітей” та отримати вказівник на “батька”. Звісно, що тільки вікно поверне непорожній вектор, проте для уніфікованої маніпуляції з вказівниками на базовий клас для нас немає ніякої різниці, чим насправді є той чи інший об’єкт.

Базовий клас має деякі характеристики, а саме: вказівник на батьківський віджет, а також позиція та розмір. Клас вікна містить вектор вказівників на “дітей”, а також два методи: для додавання та видалення дітей.

Таким чином вікно може містити в собі інші вікна, тому що всі вони будуть зберігатись в батьківському об’єкті як вказівники на базовий клас.

Слід зазначити, що базовий клас не агрегує в собі інформації для рендерингу. Це пов’язано з тим, що деякі об’єкти складаються із декількох спрайтів (*Model2D*) і

має більш складну схему малювання. Тому кожен клас віджетів забезпечує свої унікальні налаштування рендерингу на етапі створення конкретного екземпляра.

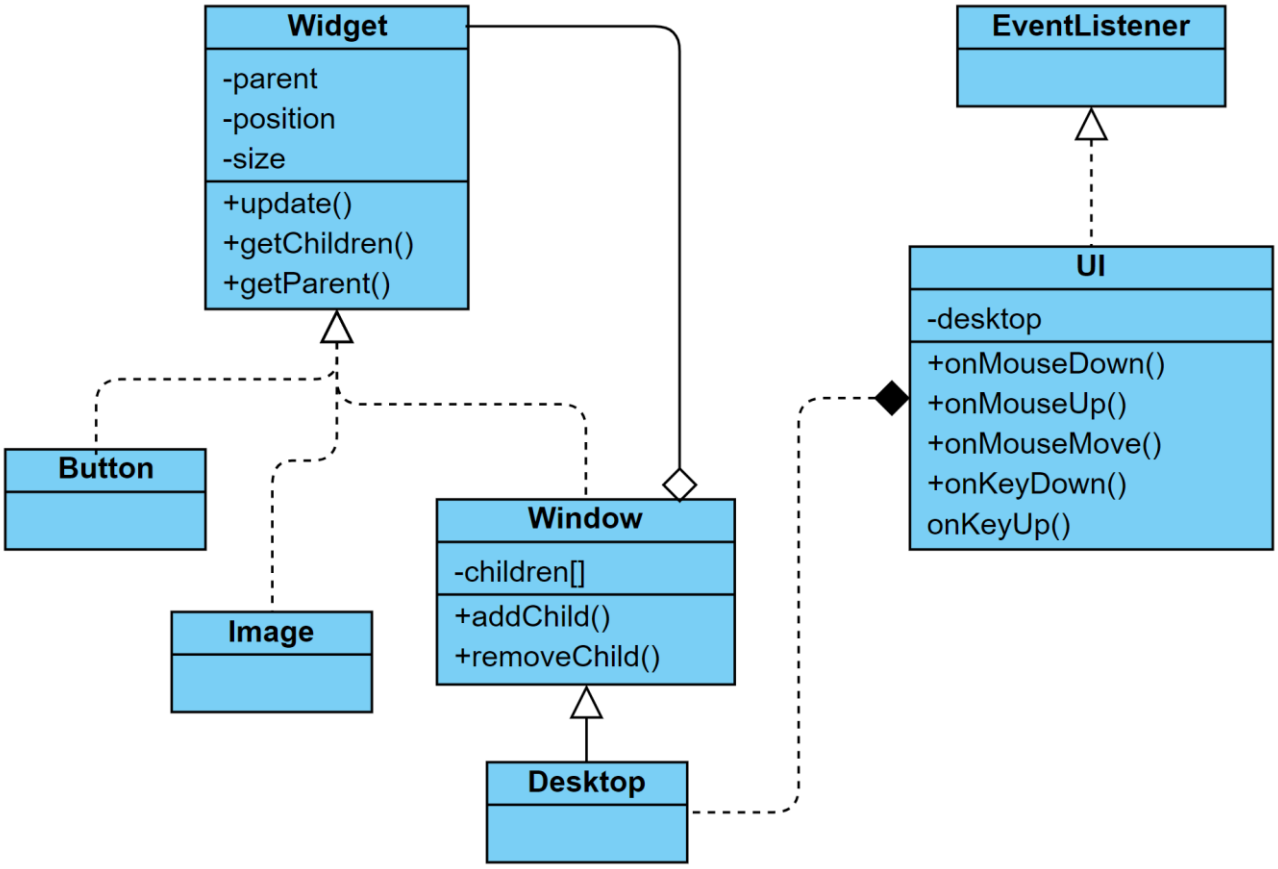

Рисунок 2.10. – Графічний інтерфейс користувача

2.8.2 Решта класів інтерфейсу користувача

Не є винятком в схемі класів віджетів і клас **Desktop**. Це нащадок класу **Window**, проте він має деякі особистості реалізації. До прикладу, при створенні екземпляра класу **Desktop** не передається вказівник на батьківський об’єкт, тому що цей об’єкт являє собою корінь всієї деревоподібної структури. Також **Desktop** є “прозорим” для рендерингу: взагалі не містить ніяких спрайтів для малювання, натомість малює всіх вложені об’єкти. І останнє: **Desktop** є “прозорим” для обробки подій.

Вказівник на екземпляр класу **Desktop** зберігається в класі **UI** – класі-менеджері всієї підсистеми інтерфейсу користувача.

Клас **UI** є нащадком від інтерфейсного класу **EventListener**, тобто він буде отримувати всі події, які наразі реалізовано в проєкті (клавіатура, миша). Слід зазначити, що екземпляр класу **UI** завжди знаходиться на першому місці у векторі **listeners[]** класу **EventManager** (п. 2.5.1), тому завжди перший отримує події. Механізм обробки подій в класі **EventManager** побудований таким чином, що коли черговий клієнт обробив подію самостійно (повернув *True*), то ця подія вважається опрацьованою і не передається наступним клієнтам. Однак якщо клієнт повертає *False*, подія передається на обробку наступному клієнту і далі по черзі.

Віджети не є нащадками **EventListener**, проте сам клас **UI** бере на себе відповідальність за пошук віджета, який обробить подію. Відбувається обхід дерева віджетів і перевіряється геометрія кожного віджета на предмет подій від миші та утримання фокуса вводу у випадку подій від клавіатури.

Розглянемо приклад. Припустимо, що подія **onMouseDown** відбувається над певним вікном, таким чином вважається, що це вікно забирає на себе цю подію і далі по ланцюжку клієнтів вона не передається. Винятком є вікно **Desktop**, про що було зазначено раніше: воно є “прозорим” для всіх подій. Хоча це вікно за розміром покриває всі 100% екрану і знаходиться на самій вершині віконної ієрархії, проте будь-яка подія вважається такою, що її не опрацьовано, якщо вложені вікна її не опрацювали. Таким чином сам **Desktop** ніяк не реагує на будь-які події самостійно.

ВИСНОВКИ

3D графіка є однією з важливих тем для відеоігор, а ігрові мобільні додатки ще довгий час матимуть актуальність.

В процесі роботи було розроблено мобільну гру під ОС Android з власним рушієм за допомогою застосування 3D графіки з використанням технології OpenGL та притримуючись приципу написання програми у вигляді PImpl (Pointer to IMPLementation).

В результаті було створено комп'ютерну гру, основною ціллю якої є переміщення агента по лабіринту та встановлення скринь у потрібне положення. Особливістю роботи стала реалізація 3D графіки та гри на мобільному пристрої з використанням C++. Написаний автором власний рушій гри та рендеринг може дозволити грі як окремий розвиток, так і бути впровадженим в інший мобільний ігровий проєкт.

Отримані результати є досить важливими та цікавими в області мобільних ігрових додатків.

Готовий розроблений додаток планується опублікувати на платформі Google Play.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Pharr M. Physically Based Rendering: From Theory to Implementation, 3rd Edition / M. Pharr, W. Jakob, G. Humphreys. – Morgan Kaufmann, 2016. – 1266 p.

2. Історія 3D графіки [Електронний ресурс] – Режим доступу до ресурсу: https://www.3dhorse.com/blogs/3d/history-of-3d-computer-graphics.

3. Історія 3D графіки [Електронний ресурс] – Режим доступу до ресурсу: https://techtoday.in.ua/news/evolyuciya-pikselya-malovidomi-fakti-pro-istoriyu-komp-yuternoyi-grafiki-50869.html

4. Що таке OpenGL [Електронний ресурс] – Режим доступу до ресурсу: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/

5. Що таке OpenGL [Електронний ресурс] – Режим доступу до ресурсу: http://www.opengl-tutorial.org/ru/beginners-tutorials/tutorial-2-the-first-triangle/

6. Історія OpenGL [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/OpenGL

7. Чому саме C++ [Електронний ресурс] – Режим доступу до ресурсу: https://www.simplilearn.com/tutorials/cpp-tutorial/learn-cpp

8. Чому саме Pimpl [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/pimpl-idiom-in-c-with-examples/

9. Чому саме Pimpl [Електронний ресурс] – Режим доступу до ресурсу: ресурсу: https://en.cppreference.com/w/cpp/language/pimpl

10. Чому саме Pimpl [Електронний ресурс] – Режим доступу до ресурсу: https://www.youtube.com/watch?v=lETcZQuKQBs

11. Патерн singleton [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/singleton

12. Патерн Observer [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/observer

13. Патерн Composite [Електронний ресурс] – Режим доступу до ресурсу:

https://refactoring.guru/uk/design-patterns/composite

14. Патерн Factory Method [Електронний ресурс] – Режим доступу до ресурсу:https://refactoring.guru/uk/design-patterns/factory-method