

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА**

**Факультет математики та інформатики  
Кафедра математичного моделювання**

**Розробка ігрового 3D рушія для платформи  
Windows**

**Кваліфікаційна робота  
Рівень вищої освіти – другий (магістерський)**

***Виконав:***

студент 6 курсу 601 групи  
спеціальності 122 – Комп'ютерні  
науки

**Селін Денис Сергійович**

***Керівник:***

доцент, канд. фіз.-мат. наук

**Юрченко Ігор Валерійович**

***До захисту допущено***

***на засіданні кафедри математичного моделювання***

***протокол № 5 від 26 листопада 2024 р.***

***Зав. кафедри \_\_\_\_\_ проф. Черевко І.М.***

**Чернівці – 2024**

# Анотація

Дана кваліфікаційна робота присвячена дослідженню та опануванню методів розробки ігрових рушіїв, шляхом написання ігрового рушія для платформи Windows з використанням мови програмування високого рівня C++, та сучасного графічного API Vulkan. Для моделювання фізики в даній роботі було застосовано бібліотеку Jolt Physics, а для написання ігрової логіки реалізована підтримка використання скриптової мови Lua.

**Ключові слова:** C++, Vulkan, Lua, Jolt Physics, Windows, ігровий рушій, розробка ігор

# Annotation

This qualification work is dedicated to the study and mastery of methods for developing game engines by creating a game engine for the Windows platform using the high-level programming language C++ and the modern Vulkan graphics API. The Jolt Physics library was utilized for physics simulation in this work, and support for the Lua scripting language was implemented for writing game logic.

**Keywords:** C++, Vulkan, Lua, Jolt Physics, Windows, game engine, game development

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів наукових досліджень інших авторів мають посилання на відповідне джерело.

\_\_\_\_\_ Д.С. Селін  
(підпис)

# Зміст

|                                                               |           |
|---------------------------------------------------------------|-----------|
| <b>Вступ .....</b>                                            | <b>5</b>  |
| <b>Розділ 1. Призначення та область використання .....</b>    | <b>6</b>  |
| 1.1 Використані технології для реалізації проекту .....       | 6         |
| 1.1.1 C++ .....                                               | 6         |
| 1.1.2 Vulkan .....                                            | 8         |
| 1.1.3 Lua .....                                               | 9         |
| 1.1.4 Sol2 .....                                              | 11        |
| 1.1.5 Jolt Physics .....                                      | 12        |
| 1.1.6 miniaudio .....                                         | 12        |
| 1.1.7 ImGUI .....                                             | 13        |
| 1.1.8 nlohmann/json .....                                     | 14        |
| 1.1.9 GLM .....                                               | 15        |
| <b>Розділ 2. Опис і обґрунтування програмних рішень .....</b> | <b>16</b> |
| 2.1 Система рендерингу .....                                  | 16        |
| 2.1.1 Моделі .....                                            | 16        |
| 2.1.2 API Vulkan .....                                        | 17        |
| 2.2 Аудіосистема .....                                        | 17        |
| 2.3 Система фізики .....                                      | 18        |
| 2.3.1 Система колізій .....                                   | 18        |
| 2.4 Система графічного інтерфейсу користувача .....           | 19        |
| 2.4.1 Система UI-елементів .....                              | 19        |
| 2.5 Система ігрової логіки .....                              | 20        |
| 2.6 Система рівнів .....                                      | 20        |
| 2.7 Система збережень .....                                   | 21        |
| <b>Розділ 3. Практична реалізація роботи .....</b>            | <b>22</b> |
| 3.1 Опис алгоритмів функціонування системи .....              | 22        |
| 3.1.1 Робота з користувацькими скриптами .....                | 22        |
| 3.1.1.1 Інтеграція API рушія .....                            | 22        |
| 3.1.1.2 Виклик користувацьких скриптів .....                  | 25        |
| 3.1.1.3 Додавання скрипту до об'єкту .....                    | 26        |
| 3.1.2 Робота з фізикою .....                                  | 28        |
| 3.1.3 Робота з файлом конфігурації .....                      | 30        |
| 3.1.4 Робота з аудіо файлами .....                            | 33        |
| 3.1.5 Робота з користувацьким вводом .....                    | 34        |
| 3.2 Опис програмного продукту .....                           | 35        |

|                                             |           |
|---------------------------------------------|-----------|
| 3.2.1 Основні можливості .....              | 35        |
| 3.2.2 Скріншоти програми.....               | 40        |
| <b>Висновки .....</b>                       | <b>43</b> |
| <b>Список використаної літератури .....</b> | <b>44</b> |
| <b>Додаток.....</b>                         | <b>45</b> |

# Вступ

У сучасному світі відеоігри стали не лише однією з найпопулярніших форм розваг, але й потужним інструментом для навчання, соціалізації та культурного вираження. З кожним роком галузь ігрової індустрії продовжує стрімко зростати, генеруючи мільярди доларів доходу, створюючи нові робочі місця та відкриваючи простір для інноваційних технологій. В основі будь-якої успішної гри лежить її рушій — програмна платформа, яка забезпечує технічну основу для відтворення графіки, фізики, звуку та взаємодії користувачів.

Сьогоднішні технології, такі як Vulkan, DirectX12, сучасні фізичні та звукові бібліотеки, відкривають нові горизонти у створенні реалістичного ігрового середовища. Розробка ігрового рушія не тільки дозволяє реалізувати інноваційні ідеї, але й сприяє глибшому розумінню архітектури програмного забезпечення, роботи з графічними та обчислювальними ресурсами, а також оптимізації ігрових систем. У результаті це дає можливість створювати ігри, які здатні не лише вражати гравців, але й задавати нові стандарти у галузі.

Саме тому метою своєї кваліфікаційної роботи я обрав дослідження методів сучасної розробки програмних платформ для створення відео ігор та власне сама розробка ігрового рушія, який би дозволяв розробникам без профільного досвіду створювати власні ігри без великих складнощів та за короткий проміжок часу. Серед програмних можливостей даного програмного продукту мною було заплановано реалізувати роботу з 3D графікою, аудіо файлами, фізикою та реалізувати можливість динамічно керувати станом гри за допомогою користувацьких скриптів.

В якості графічного API для роботи з графікою мною було вирішено обрати Vulkan API, робота симуляцією фізики відбувалася з використанням фізичного рушія Jolt Physics, в якості мови програмування для створення користувацьких скриптів мною було обрано мову Lua, а сам програмний продукт було розроблено з використанням мови програмування C++.

# Розділ 1. Призначення та область використання

У своїй кваліфікаційній роботі я розробляв ігровий рушій, який би надавав користувачам можливість створювати ігри для операційної системи Windows з використанням сучасної 3D графіки. Даний продукт має надавати користувачам зручний інтерфейс (платформу?) для створення динамічних ігор з використанням сучасної 3D графіки за короткий час, з залученням мінімальних зусиль та без потреби в ґрунтовних знаннях з програмування, адже ігрова логіка створюється за допомогою дуже простої скриптової мови Lua.

## 1.1 Використані технології для реалізації проекту

### 1.1.1 C++

Для побудови даного програмного продукту мною було використано потужну мову програмування високого рівня C++, яка об'єднує в собі як об'єктно-орієнтовану парадигму програмування, так і процедурну з функціональною парадигмою, дозволяючи вільно змішувати їх та застосовувати одночасно в одному проекті, завдяки чому досягається дуже високий рівень гнучкості й зручності у використанні.

Дана мова програмування є компіляційною, завдяки чому велика кількість помилок може бути знайдена компілятором ще на етапі збірки бінарного файлу, що значно спрощує використання даної мови й забезпечує високу надійність написаних програм.

Мова програмування C++ характеризується високою продуктивністю завдяки концепту "ти не платиш за те, що ти не використовуєш", а також завдяки здібностям компіляторів оптимізовувати вихідний код. Дана мова надає доступ до низки низькорівневих функцій, що дозволяє оптимізувати швидкість роботи розроблених програм та обсяги споживання пам'яті.

C++ є мовою яка швидко розвивається, з виходом нової версії кожні три роки, завдяки чому її можливості постійно зростають, як і зростає зручність та безпека та виразність для написання чистого та надійного коду. Завдяки високому рівню зворотної сумісності, в більшості випадків, код написаний на попередній версії мови, може бути без жодних змін запущений з використанням нового стандарту.

Серед особливостей мови програмування C++ варто виділити наступні:

- Розширені можливості контролю над пам'яттю. C++ надає користувачам повний контроль на виділенням, споживанням та звільненням пам'яті. Даний контроль може здійснюватися як через ручний менеджмент пам'яті завдяки операторам `new` та `delete`, які навіть можуть бути перевантажені для використання користувацьких розподільовачів пам'яті, так і через використання низькорівневих викликів API операційних систем (таких як `VirtualAlloc/VirtualFree` в операційній системі Windows). Також мова програмування C++ надає широкий вибір готових компонентів для роботи з пам'яттю, серед яких, наприклад, контейнери, що можуть інкапсулює керуванням пам'яті для об'єктів й за потреби можуть використовувати користувацькі аллокатори, для ще більшого контролю над пам'яттю, а також великий вибір розумних вказівників, що допомагаються ефективніше користуватися виділеними ресурсами та звільняти їх, коли вони більше не є потрібними, що значно знижує ризик, що дані ресурси будуть втрачені
- Наслідування від мови програмування C та сумісність. Початково мова програмування C++ створювалася Б'ярном Страуструпом як розширення мови програмування C, з більш сильною типізацією, для покращення безпеки використання, та з додаванням підтримки використання об'єктно-орієнтованої парадигми програмування. Завдяки цьому реалізується висока сумісність з кодом, написаним на мові C, наприклад таким як API операційних систем та великою кількістю інших бібліотек, серед яких, наприклад, сучасне низькорівневе графічне API Vulkan
- RAII. Resource Acquisition Is Initialization (виділення ресурсів це ініціалізація), це потужна концепція, яка лежить в основі більшості можливостей мови програмування C++. Згідно цієї концепції всі ресурси мають автоматично захоплюватися при створення об'єктів (на етапі ініціалізації) й автоматично звільнятися коли закінчується час життя даного об'єкту. В C++ це реалізується завдяки конструкторам та деструкторам об'єктів,

які дозволяються виконувати користувачський код з ініціалізації об'єкту при його створенні, та код з правильного звільнення всіх отриманих ресурсів при виході об'єкту з області видимості, що значно зменшує кількість помилок пов'язаних з неправильним керуванням ресурсами, та значно спрощує цей процес

- Шаблони. C++ надає потужний механізм шаблонів, що дозволяють писати узагальнений код, що може працювати з довільним типом даних, зокрема шаблони функцій та класів. Це дозволяє створювати узагальнені структури даних, наприклад такі як `std::vector` чи `std::map`, які використовують однаковий код для роботи з різними типами. Шаблони також є ключовим елементом для метапрограмування на C++, що дозволяє використовувати код, що генерується на етапі компіляції
- Поліморфізм. Мова програмування C++ дозволяє використовувати як динамічний поліморфізм через віртуальні функції, так і статичний, через використання шаблонів та перевантаження функцій. Це дозволяє робити код гнучкішим та більш адаптивним
- Багатопоточність. Починаючи зі стандарту C++11, C++ надає багаті інструменти для розробки багатопотокових застосунків, зокрема клас `std::thread`, різноманітні засоби синхронізації такі як `std::mutex`, `std::latch` чи `std::barrier`, що дозволяють реалізовувати паралельні та конкурентні програми
- Стандартна бібліотека шаблонів. Мова C++ має багату стандартну бібліотеку, що реалізує великий набір контейнерів, алгоритмів та функціональних об'єктів, завдяки чому використання та створення складних структур даних та алгоритмів є значно простішим та безпечнішим

### 1.1.2 Vulkan

Vulkan[1] це сучасне, низькорівневе графічне API, розроблений Khronos Group. Дане API забезпечує високопродуктивний доступ до графічного процесора (GPU) та надає можливість детально контролювати всі аспекти рендерингу та обробки розрахунків з використанням можливостей графічного процесора. Дане API було створено як еволюція графічного API OpenGL, щоб відповідати вимогам сучасної графіки та обробки даних в реальному часі. Основним призначенням

Vulkan є надання можливості для оптимізації роботи графічного рендерингу та обчислень на GPU з максимальним рівнем продуктивності для складних додатків, таких як ігри, віртуальна чи доповнена реальність, CAD-системи чи інші системи з симуляцією графіки в реальному часі.

Основними особливостями графічного API Vulkan є:

- Низький рівень доступу та контроль над апаратним забезпеченням. На відміну від OpenGL, Vulkan дозволяє розробникам працювати майже безпосередньо з апаратним забезпеченням, завдяки чому відкривається можливість самостійного керування пам'яттю, командними чергами та паралельними операціями, завдяки чому досягається максимальна продуктивність
- Мультиплатформність. Як і OpenGL, графічне API Vulkan доступне до використання серед великої кількості платформ, серед яких Windows, Linux, Android та інші. Це дозволяє створювати кросплатформенні додатки, які можна легко адаптувати під різноманітні операційні системи
- Менша кількість операцій виконуваних драйвером. Vulkan мінімізує навантаження на драйвери, перекладаючи частину роботи на розробників, завдяки чому зменшується час, що витрачається на обробку команд на стороні драйвер й підвищується продуктивність
- Підтримка багатопотоковості. Графічне API Vulkan було спроектовано з урахуванням можливості одночасного використання багатьох потоків, що дозволяє досягати кращої продуктивності на сучасних багатоядерних процесорах
- Менше абстракцій. Vulkan усуває значну частину абстракцій та оптимізації, що були присутніми в OpenGL й перекладає їхню реалізацію на розробників, завдяки чому повний контроль над усіма ресурсами та процесом рендерингу

### 1.1.3 Lua

Lua[4] це легка, швидка та високопродуктивна мова програмування, спроектована для вбудованих сценаріїв (скриптів) та інтеграції з іншими мовами.

Серед особливостей даної мови програмування можна виділити наступні:

- Легкість та висока швидкість. Мова програмування Lua відрізняється надзвичайною легкістю: інтерпретатор разом із базовою бібліотекою займає лише кілька сотень кілобайт пам'яті, що робить цю мову програмування оптимальним вибором для вбудованих систем чи програм, де обчислювальні ресурси є обмеженими
- Простий синтаксис. Lua володіє мінімалістичним та легким для вивчення синтаксисом, завдяки чому поріг входу для нових користувачів є дуже низьким, що є особливо корисним для написання скриптів, що далі інтегруються з іншими програмними засобами
- Надійність та легка інтеграція з C. Мова програмування Lua дає простий та прозорий інтерфейс для роботи з мовою C, що дозволяє розробникам з легкістю обмінюватися даними та викликами функцій між Lua та C, завдяки чому ця мова стала дуже популярною в якості вбудованого модуля для програм написаних з високопродуктивною головною частиною на C/C++, та можливістю розширення функціоналу за допомогою сценаріїв на Lua
- Підтримка автоматичного керування пам'яттю. Lua використовує автоматичний збирач сміття (garbage collector), що значно полегшує процес управління пам'яттю, особливо для досвідчених користувачів, що знижує кількість витоків ресурсів у додатках, що використовують Lua як вбудований інструмент, таким чином покращуючи їхню загальну надійність та продуктивність
- Портативність. Lua є кросплатформною мовою програмування і підтримується на різноманітних операційних системах та пристроях, що дозволяє запускати написані даною мовою скрипти на великій кількості різноманітних систем
- Гнучка система типів. Lua є динамічно типізованою мовою програмування з мінімальною кількістю базових типів, що робить її зручною для швидкого написання сценаріїв. Мова програмування Lua не містить окремих структур для масивів та словників й натомість використовує єдиний тип даних --- таблиці, що можуть використовуватися для різноманітних потреб. Вони підтримують як індексовані масиви, так і хеш-таблиці (асоціативні масиви), завдяки чому досягається високий рівень гнучкості, при використанні даної мови програмування

## 1.1.4 Sol2

sol2[5] — це сучасна бібліотека для інтеграції мови програмування Lua з мовою програмування C++, що забезпечує швидке і легке з'єднання між даними мовами, дозволяючи програм написаних на C++ безпечно і зручно використовувати скрипти написані на Lua. sol2 побудована поверх стандартного API Lua й пропонує більш зрозумілий, сучасний та безпечний інтерфейс, який значно спрощує виклики функцій, передачу даних і типів між C++ та Lua. Ця бібліотека особливо підходить для ігор та додатків, що вимагають налаштованих скриптів для конфігурації, ігрової логіки та автоматизації

Серед основних відмінностей даної бібліотеки варто виділити наступні:

- Зручність та читабельність коду. sol2 використовує синтаксис останніх стандартів мови програмування C++, завдяки чому вдається уникнути громіздкого, типонебезпечного та схильного до помилок традиційного Lua C API, побудованого на основі маніпуляцій зі стеком інтерпритатора. Дана бібліотека надає високорівневий інтерфейс для обробки Lua-об'єктів, функцій, мета таблиць та інших структур
- Безпечна робота з типами. Бібліотека sol2 підтримує автоматичне перетворення типів між мовами програмування Lua та C++, що знижує кількість помилок пов'язаних з трактуванням типів
- Висока продуктивність. Завдяки використанню сучасних можливостей мови програмування C++ та оптимізації, бібліотека sol2 забезпечує високу продуктивність при мінімальних накладних витратах на обробку, що оптимально підходить для використання скриптів написаних на мові Lua
- Високий рівень інтеграції. Бібліотека sol2 дозволяє дуже швидко та просто обмінюватися між мовами, як простими типами даних, так й складнішими, такими як класами. Завдяки чому будь-які користувацькі типи з C++ коду можуть бути переданими у скрипти на мові Lua, й те саме може відбуватися й у зворотному напрямку, завдяки чому досягається високий рівень інтеграції та зручності використання
- Гнучка підтримка функцій та лямбда-функцій. sol2 дозволяє легко передавати як звичайні функції, так і методи класів чи лямбда-функції у Lua оточення, де вони можуть бути викликані, як звичайні Lua функцій

### 1.1.5 Jolt Physics

Jolt Physics[3] — це сучасний двигун фізики реального часу для створення 2D та 3D симуляцій, орієнтований на високу продуктивність та ефективність. Він забезпечує інтерактивну фізику для ігрових та графічних додатків, з головним фокусом на точність розрахунків, продуктивність та зручність інтеграції. Дана бібліотека була розроблена для максимальної оптимізації використання багатоядерних процесорів й забезпечення високої швидкості виконання навіть для великих сцен з сотнями чи тисячами об'єктів одночасно.

Серед ключових особливостей цієї бібліотеки можна виділити наступні:

- Висока продуктивність. Завдяки тому, що даних фізичний двигун використовує сучасні можливості багатопроцесорної обробки, він має можливість рівномірно розподіляти навантаження між різними ядрами процесора, що значно збільшує ефективність обчислень, завдяки чому досягається висока оптимізація для роботи з даними в реальному часі з мінімальними затримками
- Зручне API. Бібліотека Jolt Physics надає розробникам зручне для використання API, яке легко налаштовується та дозволяє розробникам створювати власні типи об'єктів, інтегрувати власні нестандартні моделі взаємодії чи змінювати вже існуючі

### 1.1.6 miniaudio

miniaudio[2] — це легка, багатоплатформна аудіо-бібліотека, яка забезпечує простий та потужний API для роботи зі звуком. Дана бібліотека була розроблена для інтеграції з програмами, що потребують базової чи просунутої роботи з аудіо, однак без використання великих та складних аудіо-фреймворків. Дана бібліотека написана на мові програмування C, завдяки чому досягається висока зручність та легкість інтеграції з програмним кодом написаним мовою C++

Серед основних відмінностей бібліотеки `miniaudio` можна виділити наступні:

- Широкий асортимент вбудованих можливостей. Бібліотека `miniaudio` підтримує великий набір функціоналу, серед якого відтворення, запис, мікшування аудіо, додавання просунутих аудіо-ефектів, а також робота з різними форматами аудіо файлів
- Кросплатформність. Дана бібліотека була розроблена з урахуванням потреби кросплатформності, завдяки чому вона може використовувати нативні аудіо-API операційних систем й працювати на Windows, Linux, macOS, Android та IOS
- Зручне API. Бібліотека `miniaudio` надає зручне для використання API, завдяки чому вона може з легкістю застосовуватися як для простих задач на кшталт відтворення аудіофайлу, так і для більш складних завдань з багатоканальним мікшуванням та аудіо-ефектами

### 1.1.7 ImGUI

Бібліотека `ImGUI`[6] (Immediate Mode Graphical User Interface) надає широкі можливості для створення графічних інтерфейсів користувача в режимі негайного рендерингу (`immediate mode`). Дану бібліотеку було розроблено для інтерактивних додатків, таких як редактори, ігрові рушії тощо. Головною особливістю `ImGUI` є можливість швидко та просто створювати UI-елементи, які оновлюватимуться в реальному часі, без необхідності керування станом елементів між кадрами, завдяки цьому, а також тому, що дана бібліотека написана на C++ та має високу продуктивність, вона є ідеальним варіантом для застосування в складних програмних продуктах, де необхідно динамічно змінювати інтерфейс користувача без накладних витрат на підтримку та управління станом ієрархії UI-елементів.

Основними особливостями бібліотеки `ImGUI` є:

- Рендеринг в режимі реального часу. `ImGUI` працює в режимі `Immediate mode rendering`, що означає, що кожен UI-елемент рендериться й оновлюється відразу ж після його виклику. Це відрізняється від стандартних `retained mode` бібліотек, де інтерфейс будується як об'єктна структура, яку необхідно

постійно підтримувати в актуальному стані, що дозволяє уникнути накладних витрат на керування даними станами

- Легша інтеграція. Бібліотека ImGUI має підтримку вбудованої інтеграції з різноманітними рендеринг-системами та бібліотеками, серед яких є OpenGL, Vulkan, DirectX, Metal тощо, що робить її застосування універсальним для різних платформ та рушіїв
- Зручний у використанні API. ImGUI надає користувачам зручний та функціонально багатий API, завдяки якому можна легко створювати такі UI-елементи як: кнопки, слайдери, поля вводу, вікна, випадні списки, чек-бокси, елементи з деревоподібною ієрархією, елементи що підтримують переміщення та інші. Завдяки просунутому API, користувачам надається можливість як створювати власні UI-елементи, так і змінювати вже наявні стандартні
- Продуктивність. Бібліотека ImGUI є дуже швидкою в роботі, оскільки вона не потребує зберігання великої кількості UI об'єктів між кадрами, що робить дану бібліотеку оптимальним варіантом для застосування в складних програмах з динамічно-змінюваною графікою

### 1.1.8 nlohmann/json

Бібліотека `nlohmann/json`[7] є однією з найпопулярніших і широко використовуваних бібліотек для роботи з файлами у форматі JSON за допомогою мови програмування C++. Дана бібліотека надає зручний і простий API для парсингу, генерації та маніпулювання JSON-даними, зберігаючи зрозумілість та читабельність коду.

Серед основних відмінностей даної бібліотеки варто виділити наступні:

- Легкість у використанні. Бібліотека `nlohmann/json` використовує сучасний стандарт C++, завдяки чому вдається реалізувати інтуїтивно зрозумілий синтаксис. Внутрішня структура JSON-об'єктів реалізована за допомогою STL-контейнерів, що спрощує взаємодію з ними
- Легка інтеграція. Завдяки використанню шаблонів дана бібліотека підтримує автоматичну й зручну конвертацію між типами JSON і вбудованими типами C++. Також бібліотека надає можливості розробникам створювати власні

функції для перетворень, що дозволяє з легкістю як зберігати, так і завантажувати з JSON-формату користувацькі типи даних

- Широка функціональність. Бібліотека `nlohmann/json` надає розробникам широкий арсенал можливостей для взаємодії з даними у форматі JSON, серед який перетворення JSON-об'єктів у рядки, створення JSON-об'єктів з рядків чи файлів, можливість динамічно змінювати структуру JSON, додавати чи видаляти дані

### 1.1.9 GLM

Бібліотека GLM[8] — це потужна і зручна бібліотека для виконання математичних операцій, необхідних у графіці, фізичному моделюванні та ігровій розробці. Вона створена для забезпечення сумісності з мовою програмування шейдерів OpenGL Shader Language (GLSL), пропонуючи схожий синтаксис та набір функцій.

Серед ключових особливостей цієї бібліотеки можна виділити наступні:

- Сумісність з GLSL. GLM відтворює математичні концепції мови програмування GLSL, такі як вектори, матриці та кватерніони, з аналогічним синтаксисом та поведінкою, цим самим спрощуючи перехід між CPU-кодом і GPU-шейдерами.
- Широкий функціонал. Бібліотека GLM надає користувачам широкі функціональні можливості, серед який є операції на векторами та матрицями, проведення різних перетворень, таких як обертання, масштабування, трансляція, проведення тригонометричних обчислень, маніпуляції над кватерніонами
- Використання сучасного C++. Дана бібліотека використовує можливості сучасних стандартів C++, такі як шаблони, операції перевантаження та інші можливості C++ для забезпечення гнучкості та продуктивності

## Розділ 2. Опис і обґрунтування програмних рішень

### 2.1 Система рендерингу

Одним з найголовніших завдань для будь-якого ігрового рушія є відображення графічної інформації для гравця. Користувач даного ігрового рушія має мати можливість додавати графічні моделі до створюваного ним проекту, вільно розміщувати їх на сцені та модифікувати ключові характеристики даних моделей.

#### 2.1.1 Моделі

В якості підтримуваного графічного формату 3D моделей було обрано формат gltf, а саме його бінарний вигляд — glb. Даний сучасний формат файлів для передачі 3D-моделей, є оптимізованим для швидкого завантаження та рендерингу. Він був розроблений консорціумом Khronos Group і націлений на використання у ігрових рушіях для чого даний формат було спеціально спроектовано для зменшення обчислювальних витрат і спрощення процесу завантаження, з моделями та текстурами представленими у форматах, що є готовими для використання графічними API.

Користувач може імпортувати нові моделі у проект через меню експорту, після чого обрана модель буде завантаженою в сам ігровий рушій за допомогою бібліотеки fastgltf[9], а сама модель, якщо вона не знаходиться в папці проекту, буде скопійована в дану локацію. Під час збереження проекту, рушій зберігає інформацію про всі додані моделі, й автоматично завантажує їх при відкритті проекту.

Вже експортовані графічні моделі відображаються у вкладці Loaded Models, дозволяючи користувачу вільно додавати дані моделі до сцени, без необхідності знову їх завантажувати. Ігровий рушій оптимізовано таким чином, що інформація

про кожну модель (вершини, індекси, текстури) зберігається лише один раз, і всі однакові об'єкти на сцені використовують одну копію цих даних.

## 2.1.2 API Vulkan

Після запуску ігрового рушія, він створює вікно за допомогою бібліотеки SDL, після чого ініціалізує графічне API Vulkan, знаходить найвідповідніший GPU-девайс, що має підтримку версії Vulkan не нижче за 1.3, адже даний програмний продукт активно використовує функціонал, що був доданий в останніх версіях даного графічного API.

Рушій створює необхідні свопчейни (Swapchain), які відповідають за представлення графічного вмісту на екрані й керують кількома буферами відображення, надаючи змогу заповнювати один буфер, доки вміст іншого відображається на екрані, й таким чином уникаючи проблем з рваністю зображення, чи відображенням ще не готового кадру на екрані.

Далі створюються буфери команд, що надають створювати об'єкти з підготовленим набором команд, після чого використовувати їх без необхідності повторного створення, таким чином підвищуючи продуктивність та знижуючи навантаження на драйвер GPU, адже він заздалегідь знає набір команд, що будуть використовуватися.

Після ініціалізації, на кожній ітерації циклу відображення, проводиться обрахунок видимості кожного об'єкту та його відображення, якщо він потрапляє до кадру.

## 2.2 Аудіосистема

Аналогічно до завантаження моделей, користувач ігрового рушія може імпортувати аудіо файли у форматі wav, після чого вони будуть завантажені в

рушій за допомогою бібліотеки miniaudio, а сам аудіофайл доданий до папки ігрового проєкту, для можливості подальшого використання.

Користувач може керувати різними параметрами доданого аудіо файлу, наприклад такими як: гучність програвання, місце з якого дане програвання відбувається, чи є дане програвання зацикленним, тощо.

Ігровий рушій надає API для керування аудіо у скриптах, завдяки чому користувачі можуть вільно налаштовувати та змінювати поведінку аудіо в залежності від їхніх потреб.

## 2.3 Система фізики

Для моделювання фізики твердого тіла в даному програмному продукті використовується бібліотека Jolt Physics. Кожний додавання кожного ігрового об'єкта до сцени, для нього створюється фізичний обробник та інформація про даний об'єкт додається до фізичного рушія.

Користувач може створювати як динамічні об'єкти, тобто такі об'єкти, на які впливатиме фізична система, так і статичні об'єкти, тобто такі об'єкти, що фізичний рушій знатиме про їхнє існування на сцені, однак він ніяк не буде впливати на їхнє місцезнаходження. За потреби користувач може повністю вимкнути фізику для певного об'єкту, і як наслідок рушій фізики нічого не знатиме про даний об'єкт, не впливатиме на нього, а інші фізичні об'єкти не будуть з ним взаємодіяти.

### 2.3.1 Система колізій

Кожен динамічний об'єкт може вільно зіштовхуватися з іншими динамічними чи статичними об'єктами, що розміщені на даній сцені. Інформація про дану колізію зберігається ігровим рушієм весь час, доки ця колізія триває, й може бути отриманою через API рушія, яке він надає для скриптів.

## 2.4 Система графічного інтерфейсу користувача

Графічний інтерфейс в даному програмному продукті створено за допомогою бібліотеки ImGui. Користувачу надається праве бічне меню, яке розділене на дві частини:

- Верхня частина, яка в свою чергу також розділена на дві частини:
  - Вкладка зі списком ієрархії, який відображає об'єкти, що додані до даної сцени, такі як список об'єктів чи UI-елементів, завантажені аудіо-файли, скрипти, камеру, та список з усіма існуючими сценами
  - Вкладка Loaded Models, у якій відображаються всі імпортувати моделі
- Нижня частина, що також поділяється на дві вкладки
  - Вкладка з налаштуваннями обраного об'єкту, як-от налаштування графічної моделі, аудіо-файлу, скрипту, UI-елементу чи камери
  - Вкладка із загальними налаштуваннями проекту

Також користувачу надається верхнє меню, в якому доступні опції по завантаженню моделей, скриптів чи аудіо-файлів, створенню нових UI-елементів чи збереженню даного проекту. В цьому ж верхньому меню знаходиться кнопка для активації симуляції, яка відповідає за активацію фізичної системи та користувацьких скриптів. Після її активації, користувач може заховати бічне меню та деактивувати режим вільної камери, після чого керування камерою буде передано користувацьким скриптам.

### 2.4.1 Система UI-елементів

Користувач ігрового рушія може додавати до кожної сцени UI-елементи, вільно розміщувати їх та динамічно керувати ними через відповідне API, що надається для користувацьких скриптів. Кожен UI-елемент складається з головного вікна, в якому розміщуються інші елементи, такі як кнопки чи текст. Завдяки API користувач може змінювати положення даних елементів, їхній текст, чи отримувати стан натиску відповідної кнопки.

## 2.5 Система ігрової логіки

Даний ігровий рушій надає можливість створювати ігрову логіку за допомогою скриптів написаних мовою програмування Lua. Після додавання скрипту до ігрового проекту, ігровий рушій інтегрує в даний скрипт виклики користувацького API та необхідні структури даних, завдяки яким користувач може взаємодіяти з рушієм на різних об'єктах.

Скрипти можуть бути доданими до наступних елементів:

- Ігрових рівнів
- Графічних об'єктів
- UI-елементів

Також існує можливість використовувати скрипт, який існуватиме протягом всієї гри, даний скрипт може бути використаний для збереження чи обміну інформацією між іншими скриптами.

Кожен елемент може мати власну копію скрипта й таким чином власні, локальні для даного об'єкта, змінні. При додаванні скрипту до об'єкта, ігровий рушій інтегрує в скрипт спеціальну змінну, завдяки якій отримувати інформацію та керувати саме тим об'єктом, до якого він доданий.

Кожен елемент (графічна модель, аудіо файл, рівень, UI-елемент) має власний унікальний ідентифікатор, який використовується для його ідентифікації рушієм, та ім'я, яке має бути унікальним для кожного рівня. Для роботи з об'єктами за допомогою API рушія, користувач повинен використовувати ідентифікатор об'єкту.

## 2.6 Система рівнів

Користувач даного програмного засобу має можливість створювати необмежену кількість ігрових рівнів, кожен з яких може мати власні графічні

об'єкти, UI-елементи чи скрипти. Користувацькі скрипти можуть керувати завантаженням рівнів за допомогою наданих викликів API рушія. Вибрати головний рівень, тобто такий, що завантажуватиметься автоматично при відкритті гри, можна через меню налаштувань проекту.

## 2.7 Система збережень

Користувачі мають можливість зберегти створений проект за допомогою відповідної кнопки у меню, що знаходиться у верхній частині екрану. Після натискання даної кнопки, інформація про всі додані моделі, аудіо-файли, скрити, інформація про створені UI-елементи та рівнів, їхні налаштування та налаштування проекту будуть збережені у конфігураційний файл.

Також користувачам надається можливість експортувати створену гру, після чого рушій скопіює всі необхідні файли для роботи гри до спеціальної папки. При запуску виконуваного файлу з даної папки, буде завантажено основний ігровий рівень, без можливості активувати інтерфейс редактора

## Розділ 3. Практична реалізація роботи

### 3.1 Опис алгоритмів функціонування системи

#### 3.1.1 Робота з користувацькими скриптами

##### 3.1.1.1 Інтеграція API рушія

Після завантаження ігровим рушієм обраного користувачем скрипта, що реалізує ігрову логіку, рушій автоматично інтегрує до даного скрипта своє API, що робить можливим динамічну взаємодію з рушієм та об'єктами засобами мови програмування Lua.

API реалізується за допомогою лямда-функцій, що викликають необхідні функції рушія, чи його підсистем. Дана інтеграція реалізується наступним чином:

```
lua["playAudio"] = [this, &engine](const std::string&
id)
{
    auto objIt = engine.audioFiles.find(id);

    if (objIt != engine.audioFiles.end())
    {
        if (!objIt->second.isPlaying())
        {
            objIt->second.play();
        }
    }
};
```

Завдяки цьому користувач може викликати дану функцію у власному скрипті на мові Lua, та викликати дану функцію наступним чином:

```
function playMusic()
    audioId = getAudioId("main_music")
    playAudio(audioId)
end
```

Також рушій інтегрує типи мови C++ в Lua скрипти, реалізується це наступним чином:

```
lua.new_usertype<glm::vec3>("vec3",
    sol::constructors<glm::vec3(), glm::vec3(float),
glm::vec3(float, float, float)>(),
    "x", &glm::vec3::x,
    "y", &glm::vec3::y,
    "z", &glm::vec3::z);

lua["getObjectPosition"] = [this, &engine](const
std::string& id) -> glm::vec3
{
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);

    if (objIt != engine.currentLevel-
>loadedScenes.end())
    {
        return objIt->second.position;
    }

    return { 0, 0, 0 };
};
```

Завдяки цьому користувач може вільно ними користуватися при написанні власних скриптів для керування ігровою логікою:

```
function moveForward()
    position = getObjectPosition(id)
    setObjectPosition(id, position.x + 1, position.y,
position.z)
end
```

Також користувацькі скрипти можуть звертатися до головного скрипту рівня гри, для збереження там чи отримання певних даних. На стороні C++ це відбувається наступним чином:

```
lua["getStringValue"] = [this, &engine](const
std::string& name) -> std::string
{
    if (engine.currentProject.scriptId.empty())
    {
        return "";
    }

    return engine.currentProject.script-
>lua[name].get_or<std::string>("");
};
```

```
lua["setStringValue"] = [this, &engine](const
std::string& name, const std::string& value)
{
    if (!engine.currentProject.scriptId.empty())
    {
        engine.currentProject.script->lua[name] =
value;
    }
};
```

В скриптах на Lua ці дані можуть бути отриманими та використаними наступним чином:

```

function moveToNextLevel()
    currentLevel = getIntValue("currentLevel")
    setIntValue("currentLevel", currentLevel + 1)
    loadLevel("level "..currentLevel + 1)
end

```

### 3.1.1.2 Виклик користувацьких скриптів

Даний ігровий рушій підтримує виклик трьох функцій з користувацьких скриптів, а саме функції `init`, `step` та `stop`.

Функція `init` викликається при завантаженні нового рівня, якщо ігрова симуляція активована. Також на цьому етапі в змінну `thisObjectId` поміщається унікальний ідентифікатор кожного об'єкта, завдяки якому скрипт може ідентифікувати з яким об'єктом він працює. Це надає можливість одному скрипту керувати різними об'єктами одночасно:

```

if (!currentLevel->scriptId.empty() &&
    scriptSimulateionIsActive)
{
    currentLevel->script->lua["thisObjectId"] =
currentLevelId;
    currentLevel->script->init();
}

for (auto& [objId, obj] : currentLevel->loadedScenes)
{
    if (!obj.scriptId.empty())
    {
        obj.script->lua["thisObjectId"] = objId;
        obj.script->init();
    }
}

```

```
}
```

Кожного кадру, до етапу розрахунку фізики та відображення зображення, викликається функція `step` для кожного скрипту:

```
if (scriptSimulateionIsActive)
{
    for (auto& [id, ptr] : currentLevel->loadedScenes)
    {
        if (ptr.scriptId.empty() ||
!ptr.isScriptActive) continue;

        if (ptr.script)
        {
            ptr.script->step();
        }
    }

    for (auto& [id, ptr] : currentLevel->uiElements)
    {
        if (ptr.scriptId.empty()) continue;

        if (ptr.script)
        {
            ptr.script->step();
        }
    }
}
```

### 3.1.1.3 Додавання скрипту до об'єкту

Після завантаження скрипту через відповідне користувацьке меню, він може бути доданим до будь-якого об'єкта в панелі його налаштувань. Відбувається це наступним чином:

```
std::string scriptName = object.scriptId.empty() ? ""
: loadedScripts[object.scriptId].name;

if (ImGui::BeginCombo("Select script",
scriptName.c_str()))
{
    for (auto& [id, script] : loadedScripts)
    {
        if (ImGui::Selectable(script.name.c_str(), id
== object.scriptId))
        {
            object.scriptId = id;
            object.script->path = script.path;
            object.script->name = script.name;
            object.script->init();
        }
    }
    if (ImGui::Selectable("None",
object.scriptId.empty()))
    {
        object.scriptId = "";
        object.script.destroy();
    }
    ImGui::EndCombo();
}
```

Після чого обраний скрипт буде додано до даного об'єкту, й викликано при активації ігрової симуляції.

### 3.1.2 Робота з фізикою

При реєстрації колізії між двома об'єктами, для яких активована фізика, рушій зберігає інформацію про це, для можливості її використанні користувацькими скриптами, відбувається це наступним чином:

```
for (auto& [uuid, obj] : currentLevel->loadedScenes)
{
    if (obj.bodyId == Body1)
    {
        body1UUID = uuid;
    }
    if (obj.bodyId == Body2)
    {
        body2UUID = uuid;
    }
}
if (body1UUID.empty() || body2UUID.empty() ||
body1UUID == body2UUID)
{
    return;
}
collisionMap[body1UUID].insert(body2UUID);
collisionMap[body2UUID].insert(body1UUID);
```

При завантаженні кожного об'єкту розраховуються його межі, для подальшого використання фізичним рушієм:

```
auto sceneObject = loadedGLTFs[id];
std::filesystem::path path = sceneObject->path;
std::vector<Bounds> bounds;

for (auto& node : sceneObject->topNodes) {
```

```

MeshNode* meshNode =
dynamic_cast<MeshNode*>(node.get());
    if (meshNode) {
        for (auto& surface : meshNode->mesh-
>surfaces)
        {
            bounds.push_back(Bounds{
surface.bounds.origin, surface.bounds.sphereRadius,
surface.bounds.extents * meshNode->scale });
        }
    }
glm::vec3
minCorner(std::numeric_limits<float>::max());
glm::vec3
maxCorner(std::numeric_limits<float>::lowest());
for (const auto& bound : bounds) {
    glm::vec3 boundMin = bound.origin - bound.extents;
    glm::vec3 boundMax = bound.origin + bound.extents;

    minCorner = glm::min(minCorner, boundMin);
    maxCorner = glm::max(maxCorner, boundMax);
}
glm::vec3 newCenter = (minCorner + maxCorner) * 0.5f;
glm::vec3 newExtents = (maxCorner - minCorner) * 0.5f;
float newSphereRadius = glm::length(newExtents);
object.bounds = Bounds{ newCenter, newSphereRadius,
newExtents };
object.extents = newExtents;

```

При активації ігрової симуляції, та кожному кадрі відбувається розрахунок фізичної симуляції, відповідно до даних якої відбувається оновлення даних об'єктів для відображення на екрані:

```

if (isPhysicsRunning)

```

```

{
    physics_system.Update(1.0f/60.0f, cCollisionSteps,
&temp_allocator, &job_system);

    for (auto& [_, obj] : currentLevel->loadedScenes)
    {
        RVec3 position = body_interface-
>GetCenterOfMassPosition(obj.bodyId);
        Vec3 velocity = body_interface-
>GetLinearVelocity(obj.bodyId);
        Quat rotation = body_interface-
>GetRotation(obj.bodyId);
        obj.position = glm::vec3 { position.GetX(),
position.GetY(), position.GetZ() };
        ptr.linearVelocity = glm::vec3{ velocity.GetX(),
velocity.GetY(), velocity.GetZ() };
    }
}

```

### 3.1.3 Робота з файлом конфігурації

При відкритті проекту відбувається зчитування даних з файлу config.json. Відбувається це наступним чином:

```

std::fstream configFile("config.json");
json config = json::parse(configFile);
eng->currentProject.name = config["project"]["title"];
eng->currentProject.tempName = eng-
>currentProject.name;
SDL_SetWindowTitle(eng->_window, eng-
>currentProject.name.c_str());
eng->currentProject.path = path;

```

```

eng->currentProject.scriptId =
config["project"]["scriptId"];

for (const auto& [id, path] :
config["Scene"]["glTF"].items())
{
    eng->currentProject.loadedObjects[id] = path;
}

for (const auto& [id, object] :
config["Scene"]["audio"].items())
{
    eng->audioFiles[id] = object.template get
<AudioFile>();
    addSound(eng, eng->audioFiles[id]);
}

eng->currentProject.mainLevelId =
config["project"]["mainLevelId"];
eng->loadLevel(eng->currentProject.mainLevelId);
eng->mainCamera = config["Scene"]["camera"].template
get <Camera>();
loadGLTFObjects(eng);

```

Для перетворень JSON-об'єктів в C++ об'єкти, рушій реєструє функції-перетворювачі:

```

void from_json(const json& j, AudioFile& obj) {
    j.at("id").get_to(obj.id);
    j.at("name").get_to(obj.name);
    j.at("path").get_to(obj.path);
    j.at("isLooping").get_to(obj.isLooping);
    j.at("volume").get_to(obj.volume);
}

```

Збереження налаштувань поточного проекту відбувається схожим чином:

```
json config;
config["project"]["title"] = eng->currentProject.name;
config["project"]["mainLevelId"] = eng-
>currentProject.mainLevelId;
config["project"]["scriptId"] = eng-
>currentProject.scriptId;
for (const auto& [id, path] : eng-
>currentProject.loadedObjects)
{
    config["Scene"]["gltf"][id] = path;
}
for (const auto& [id, file] : eng->audioFiles)
{
    config["Scene"]["audio"][id] = file;
}
for (const auto& [id, object] : eng->levels)
{
    config["Scene"]["level"][id] = object;
}
for (const auto& [id, elem] : eng->loadedScripts)
{
    config["Scene"]["Script"][id] = elem;
}
config["Scene"]["camera"] = eng->mainCamera;
std::ofstream stream("config.json");
stream << std::setw(2) << config << std::endl;
```

Для перетворень C++ об'єктів у JSON об'єкти, рушій також використовує спеціальні функції:

```
void to_json(json& j, const AudioFile& obj) {
    j = json{
```

```

        {"id", obj.id},
        {"name", obj.name},
        {"path", obj.path},
        {"isLooping", obj.isLooping},
        {"volume", obj.volume}
    };
}

```

### 3.1.4 Робота з аудіо файлами

Рушій зберігає список доданих аудіо-файлів, кожен з яких може бути налаштованим як за допомогою графічного інтерфейсу користувача в редакторі, так і за допомогою користувацьких скриптів, що пишуться мовою програмування Lua.

Після того, як користувач вибере необхідний аудіо файл через меню файлового діалогу, завантаження даного файлу засобами бібліотеки `miniaudio` й додавання до списку відслідковуваних рушієм файлів, відбувається наступним чином:

```

std::filesystem::current_path(eng-
>currentProject.path);
std::string name =
std::filesystem::path(path).stem().string();
ma_sound sound;
AudioFile audioFile{
    .id =
uuids::to_string(uuids::uuid_system_generator{}()),
    .name = name
};
eng->audioFiles[audioFile.id] = audioFile;
ma_result result = ma_sound_init_from_file(&eng-

```

```
>audio_engine, path.c_str(),
MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_DECODE |
MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_STREAM, NULL,
NULL, &eng->audioFiles[audioFile.id].sound);
```

### 3.1.5 Робота з користувацьким вводом

Даний ігровий рушій для реєстрації користувацького вводу, такого як натискання клавіш чи рух мишою, використовує програмні засоби, що надаються бібліотекою SDL2.

Рушій зберігає асоціативну таблицю зі станом натиску кожної клавіші, та зберігає положення миші:

```
std::unordered_map<KeyType, bool> keys;
glm::vec2 mouseMotion;
```

Кожного кадру рушій оновлює ці дані, а також, якщо режим вільної камери не активовано, передає дані вводу для обробки камерою. Якщо режим вільного руху вимкнено, то керування камерою можливе лише через користувацькі скрипти.

```
while (SDL_PollEvent(&e) != 0)
{
    if (e.type == SDL_MOUSEBUTTONDOWN) {
        if (e.button.button == SDL_BUTTON_LEFT) {
            keys[KeyType::Mouse_Left] = true; }
        if (e.button.button == SDL_BUTTON_RIGHT) {
            keys[KeyType::Mouse_Right] = true; }
    }

    if (e.type == SDL_MOUSEMOTION) {
```

```

        mouseMotion.x += (float)e.motion.xrel;
        mouseMotion.y += (float)e.motion.yrel;
    }
    if (freeCamera)
    {
        mainCamera.processSDLEvent(e);
    }
    ImGui_ImplSDL2_ProcessEvent(&e);
}

```

## 3.2 Опис програмного продукту

### 3.2.1 Основні можливості

- Створення чи відкриття вже наявного проекту

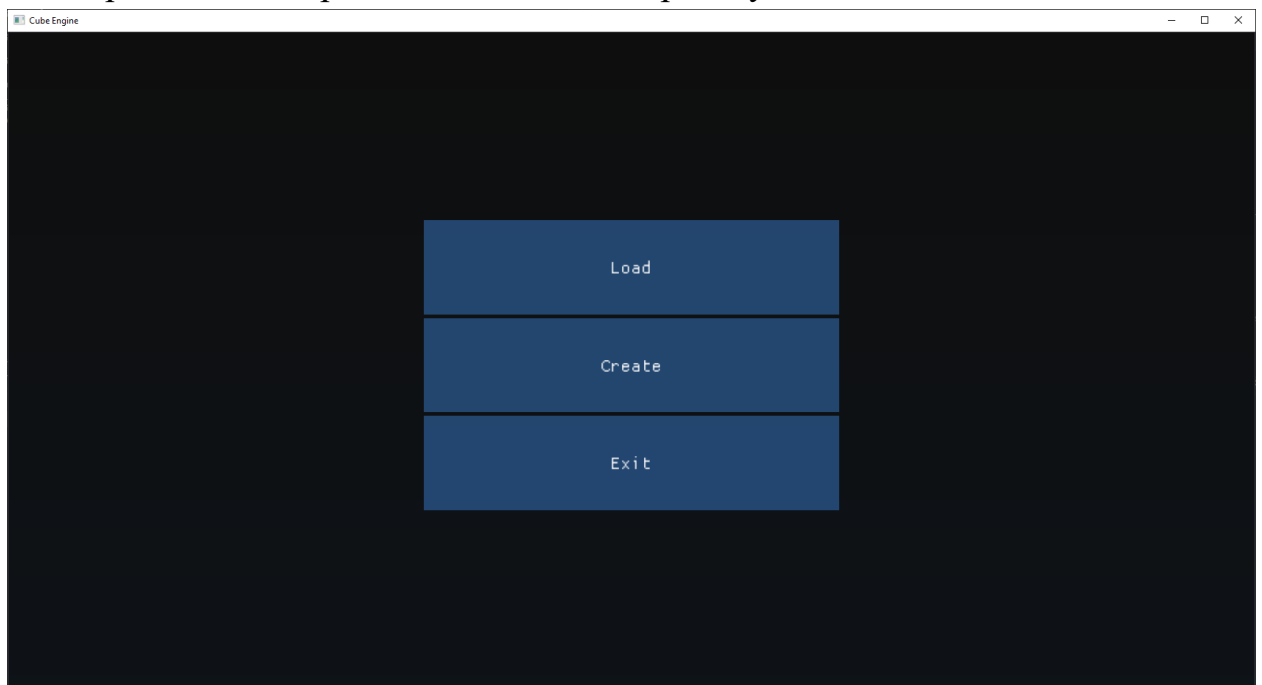


Рисунок 3.1 — Головне меню ігрового рушія

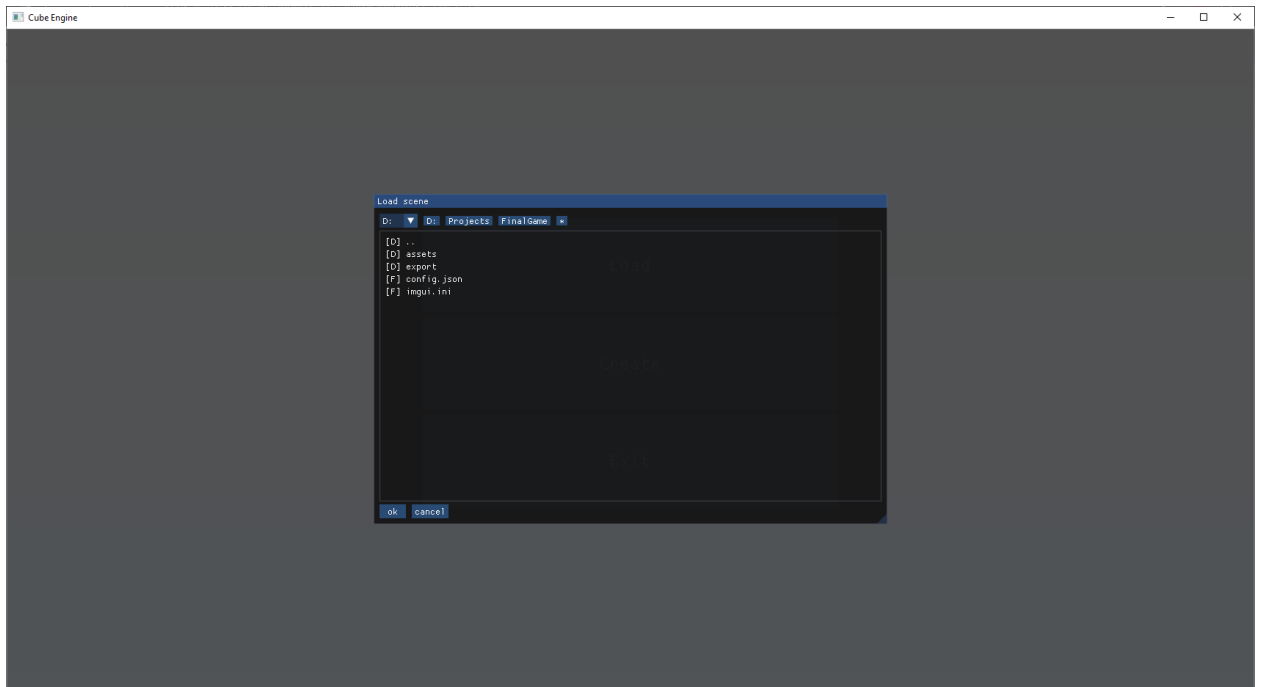


Рисунок 3.2 — Приклад обрання проекту для відкриття

- Додавання графічних моделей

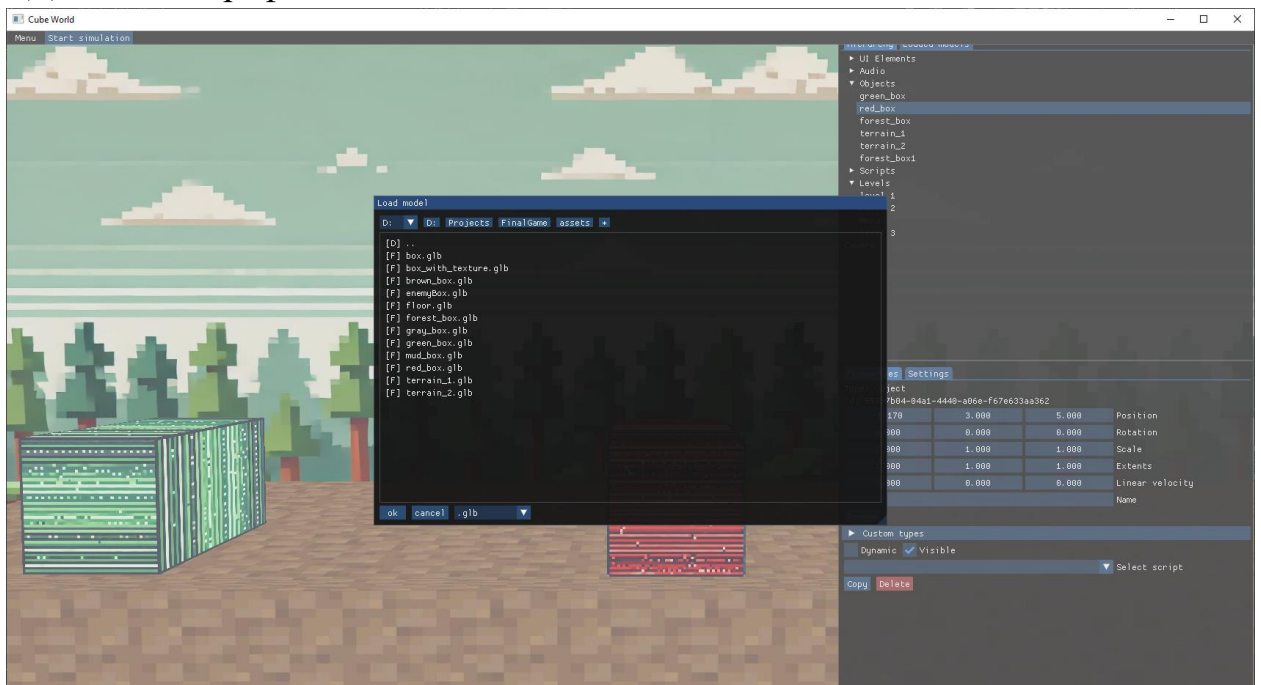


Рисунок 3.3 — Приклад завантаження моделі

- Налаштування характеристик об'єктів



Рисунок 3.4 — Приклад вкладки налаштувань об'єкта

- Налаштування характеристик аудіо-файлів



Рисунок 3.5 — Приклад вкладки налаштувань аудіо-файлу

- Створення та налаштування UI-елементів



Рисунок 3.6 — Приклад вкладки налаштувань UI-елементу

- Налаштування характеристик проекту



Рисунок 3.6 — Приклад вкладки налаштувань проекту

### 3.2.2 Скріншоти програми

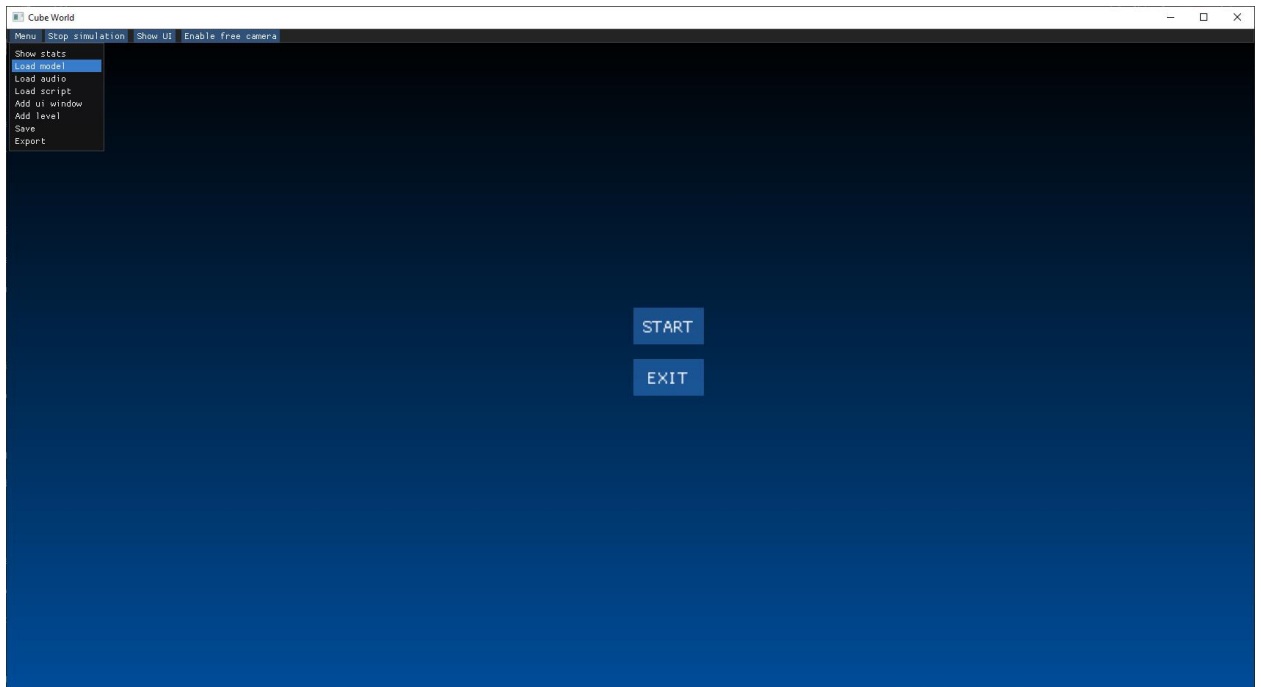


Рисунок 3.7 — Приклад верхнього меню та активованої симуляції із захованим бічним меню

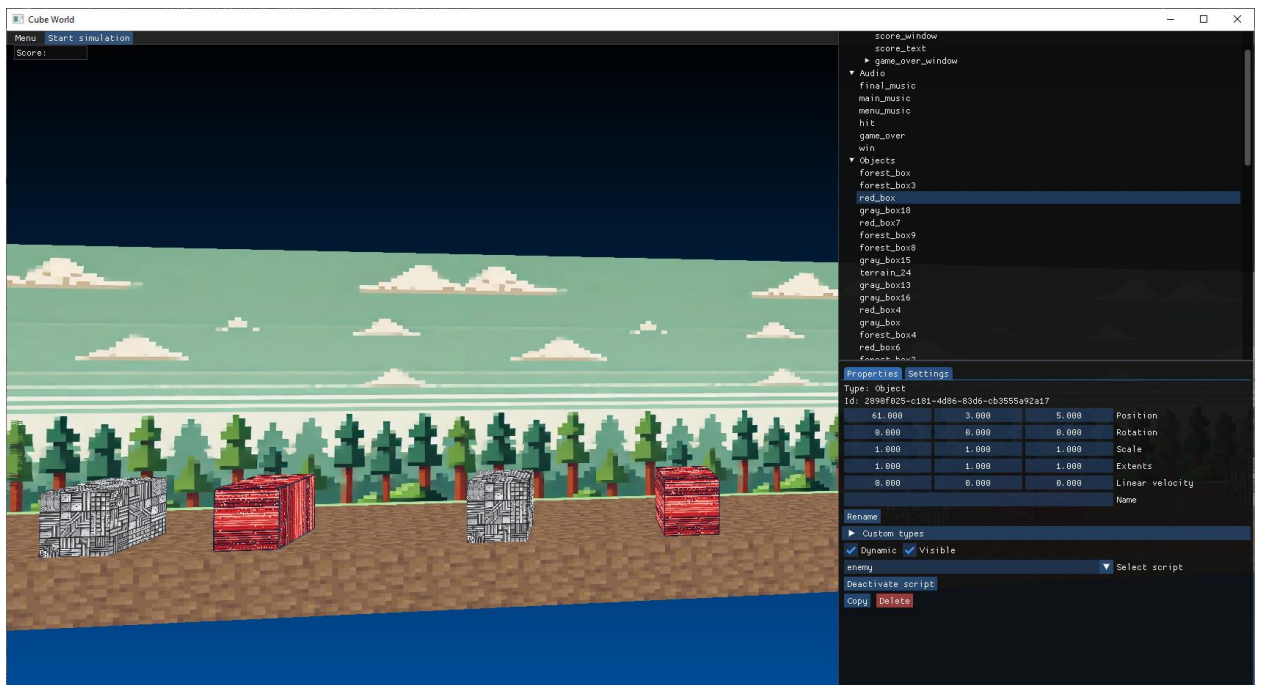


Рисунок 3.8 — Приклад ігрового редактора

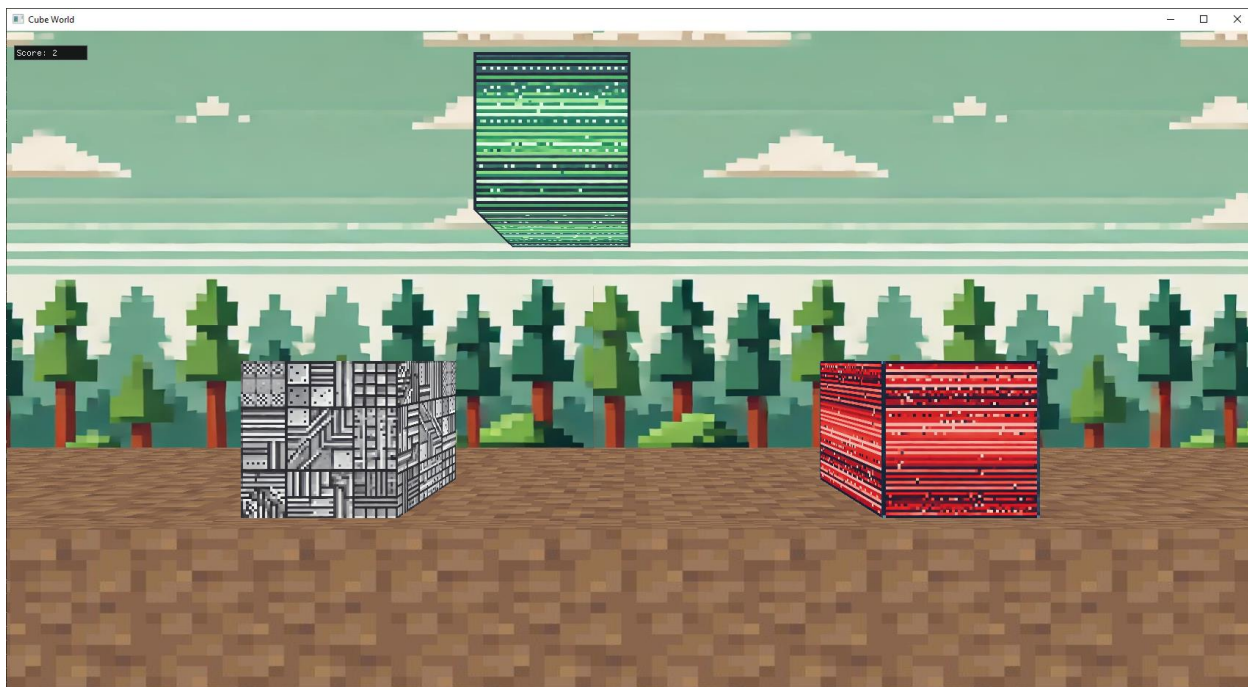


Рисунок 3.9 — Пример игры

## Висновки

В даній кваліфікаційній роботі я значно покращити свої знання з методів та технологій розробки сучасних ігрових рушіїв

Під час розробки програмного засобу, було створено ігровий рушій для операційної системи Windows, у якому реалізовано наступний функціонал:

- відображення та робота з графічними 3D моделями;
- робота з аудіо файлами;
- робота із симуляцією фізики твердого тіла;
- динамічне розширення функціоналу програмного засобу завдяки інтеграції скриптів написаних мовою Lua;
- створення та збереження ігрових проектів;

Розробка даного програмного продукту дозволила отримати нові компетенції в створенні програм з використанням графічного API Vulkan, та поглибити знання мов програмування C++ та Lua.

## Список використаної літератури

- [1] Vulkan API URL: <https://www.vulkan.org/>
- [2] miniaudio URL: <https://miniaud.io/index.html>
- [3] Jolt Physics URL: <https://jrouwe.github.io/JoltPhysics/index.html>
- [4] Lua URL: <https://www.lua.org/>
- [5] sol2 URL: <https://sol2.readthedocs.io/en/latest/>
- [6] Dear ImGui URL: <https://github.com/ocornut/imgui>
- [7] nlohmann/json URL: <https://json.nlohmann.me/>
- [8] GLM URL: <https://github.com/g-truc/glm>
- [9] fastglTF URL: <https://fastglTF.readthedocs.io/v0.8.x/>

# Додаток

## Файл logger.h

```
#pragma once
#include <spdlog/spdlog.h>

class Log {
public:
    static void init();
    inline static std::shared_ptr<spdlog::logger> getLogger() {
        return s_logger;
    }
private:
    static std::shared_ptr<spdlog::logger> s_logger;
};

#ifdef _DEBUG
#    define BK_TRACE(...)    ::Log::getLogger()->trace(__VA_ARGS__)
#    define BK_INFO(...)    ::Log::getLogger()->info(__VA_ARGS__)
#    define BK_WARN(...)    ::Log::getLogger()->warn(__VA_ARGS__)
#    define BK_ERROR(...)    ::Log::getLogger()->error(__VA_ARGS__)
#    define BK_CRITICAL(...) ::Log::getLogger()-
>critical(__VA_ARGS__)
#else
#    define BK_TRACE(...)
#    define BK_INFO(...)
#    define BK_WARN(...)
#    define BK_ERROR(...)
#    define BK_CRITICAL(...)
#endif
```

## Файл logger.cpp

```
#include "logger.h"
#include "spdlog/sinks/stdout_color_sinks.h"
#include <spdlog/sinks/basic_file_sink.h>

std::shared_ptr<spdlog::logger> Log::s_logger;

void Log::init() {
    std::vector<spdlog::sink_ptr> logSinks;
    logSinks.emplace_back(std::make_shared<spdlog::sinks::stdout_color_sink_mt>());
    logSinks.emplace_back(std::make_shared<spdlog::sinks::basic_file_sink_mt>("blockengine.log", true));

    logSinks[0]->set_pattern("%^[%T.%e] [%l]: %v%$");
    logSinks[0]->set_level(spdlog::level::info);
    logSinks[1]->set_pattern("[%Y-%m-%d %T.%f] [%l]: %v");
    logSinks[1]->set_level(spdlog::level::trace);

    s_logger = std::make_shared<spdlog::logger>("blockengine",
begin(logSinks), end(logSinks));
    spdlog::register_logger(s_logger);
    s_logger->set_level(spdlog::level::trace);
}
```

## Файл camera.h

```
#include "SDL_events.h"
#include <types.h>

enum class CameraDirection {
    Forward,
    Backward,
    Left,
    Right
};

class Camera {
public:
    glm::vec3 velocity;
    glm::vec3 position;
    float     pitch{0.f};
    float     yaw{0.f};
    bool      rightButtonPressed = false;

    glm::mat4 getViewMatrix() const;
    glm::mat4 getRotationMatrix() const;
    void      processSDLEvent(SDL_Event& e);
    void      move(CameraDirection direction, int count = 1);
    void      rotate(glm::vec2 rotation);
    void      update();
};
```

## Файл camera.cpp

```
#include "camera.h"
#include "glm/gtx/quaternion.hpp"
#include "glm/gtx/transform.hpp"

glm::mat4 Camera::getViewMatrix() const {
    glm::mat4 cameraTranslation = glm::translate(glm::mat4(1.f),
position);
    glm::mat4 cameraRotation    = getRotationMatrix();
    return glm::inverse(cameraTranslation * cameraRotation);
}

glm::mat4 Camera::getRotationMatrix() const {
    glm::quat pitchRotation = glm::angleAxis(pitch, glm::vec3(1.f,
0.f, 0.f));
    glm::quat yawRotation   = glm::angleAxis(yaw, glm::vec3(0.f, -
1.f, 0.f));
    return glm::toMat4(yawRotation) * glm::toMat4(pitchRotation);
}

void Camera::move(CameraDirection direction, int count) {
    glm::vec3 cameraDirection;
    switch (direction) {
        case CameraDirection::Forward: cameraDirection =
glm::vec3(0.f, 0.f, -1.f * count); break;
        case CameraDirection::Backward: cameraDirection =
glm::vec3(0.f, 0.f, 1.f * count); break;
        case CameraDirection::Left: cameraDirection = glm::vec3(-
1.f * count, 0.f, 0.f); break;
        case CameraDirection::Right: cameraDirection = glm::vec3(1.f
* count, 0.f, 0.f); break;
    }
    glm::mat4 cameraRotation = getRotationMatrix();
    position += glm::vec3(cameraRotation *
glm::vec4(cameraDirection * 0.5f, 0.f));
}

void Camera::rotate(glm::vec2 rotation) {
    yaw += rotation.x;
    pitch -= rotation.y;
}

void Camera::processSDLEvent(SDL_Event& e) {
```

```

if (e.type == SDL_KEYDOWN) {
    if (e.key.keysym.sym == SDLK_w) {
        velocity.z = -1;
    }
    if (e.key.keysym.sym == SDLK_s) {
        velocity.z = 1;
    }
    if (e.key.keysym.sym == SDLK_a) {
        velocity.x = -1;
    }
    if (e.key.keysym.sym == SDLK_d) {
        velocity.x = 1;
    }
}
if (e.type == SDL_KEYUP) {
    if (e.key.keysym.sym == SDLK_w) {
        velocity.z = 0;
    }
    if (e.key.keysym.sym == SDLK_s) {
        velocity.z = 0;
    }
    if (e.key.keysym.sym == SDLK_a) {
        velocity.x = 0;
    }
    if (e.key.keysym.sym == SDLK_d) {
        velocity.x = 0;
    }
}
if (e.type == SDL_MOUSEBUTTONDOWN) {
    if (e.button.button == SDL_BUTTON_RIGHT) {
        rightButtonPressed = true;
    }
}
if (e.type == SDL_MOUSEBUTTONUP) {
    if (e.button.button == SDL_BUTTON_RIGHT) {
        rightButtonPressed = false;
    }
}
if (e.type == SDL_MOUSEMOTION && rightButtonPressed) {
    yaw += (float)e.motion.xrel / 200.f;
    pitch -= (float)e.motion.yrel / 200.f;
}
}

```

```
void Camera::update() {  
    glm::mat4 cameraRotation = getRotationMatrix();  
    position                += glm::vec3(cameraRotation *  
glm::vec4(velocity * 0.5f, 0.f));  
}
```

## Файл descriptors.h

```
#pragma once

#include <deque>
#include <span>
#include <types.h>
#include <vector>

struct DescriptorLayoutBuilder {
    std::vector<VkDescriptorSetLayoutBinding> bindings;

    void addBinding(uint32_t binding,
VkDescriptorType type);
    void clear();
    VkDescriptorSetLayout build(VkDevice device, VkShaderStageFlags
shaderStages, void* pNext = nullptr,
    VkDescriptorSetLayoutCreateFlags flags = 0);
};

struct DescriptorWriter {
    std::deque<VkDescriptorImageInfo> imageInfos;
    std::deque<VkDescriptorBufferInfo> bufferInfos;
    std::vector<VkWriteDescriptorSet> writes;

    void writeImage(int binding, VkImageView image, VkSampler
sampler, VkImageLayout layout, VkDescriptorType type);
    void writeBuffer(int binding, VkBuffer buffer, size_t size,
size_t offset, VkDescriptorType type);

    void clear();
    void update_set(VkDevice device, VkDescriptorSet set);
};

struct DescriptorAllocator {
    struct PoolSizeRatio {
        VkDescriptorType type;
        float ratio;
    };

    VkDescriptorPool pool;

    void initPool(VkDevice device, uint32_t maxSets,
```

```

std::span<PoolSizeRatio> poolRatios);
    void clearDescriptors(VkDevice device);
    void destroyPool(VkDevice device);

    VkDescriptorSet allocate(VkDevice device, VkDescriptorSetLayout
layout);
};

struct DescriptorAllocatorGrowable {
public:
    struct PoolSizeRatio {
        VkDescriptorType type;
        float ratio;
    };

    void init(VkDevice device, uint32_t initialSets,
std::span<PoolSizeRatio> poolRatios);
    void clear_pools(VkDevice device);
    void destroyPool(VkDevice device);

    VkDescriptorSet allocate(VkDevice device, VkDescriptorSetLayout
layout, void* pNext = nullptr);
private:
    VkDescriptorPool getPool(VkDevice device);
    VkDescriptorPool createPool(VkDevice device, uint32_t setCount,
std::span<PoolSizeRatio> poolRatios);

    std::vector<PoolSizeRatio> ratios;
    std::vector<VkDescriptorPool> fullPools;
    std::vector<VkDescriptorPool> readyPools;
    uint32_t setsPerPool;
};

```

## Файл descriptors.cpp

```
#include "initializers.h"
#include <descriptors.h>

void DescriptorLayoutBuilder::addBinding(uint32_t binding,
VkDescriptorType type) {
    VkDescriptorSetLayoutBinding newbind{};
    newbind.binding = binding;
    newbind.descriptorCount = 1;
    newbind.descriptorType = type;
    bindings.push_back(newbind);
}

void DescriptorLayoutBuilder::clear() {
    bindings.clear();
}

VkDescriptorSetLayout DescriptorLayoutBuilder::build(VkDevice device,
VkShaderStageFlags shaderStages, void* pNext,

    VkDescriptorSetLayoutCreateFlags flags) {
    for (auto& b : bindings) {
        b.stageFlags |= shaderStages;
    }

    VkDescriptorSetLayoutCreateInfo info = {.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO};
    info.pNext = pNext;
    info.pBindings = bindings.data();
    info.bindingCount =
(uint32_t)bindings.size();
    info.flags = flags;
    VkDescriptorSetLayout set;
    VK_CHECK(vkCreateDescriptorSetLayout(device, &info, nullptr,
&set));
    return set;
}

void DescriptorAllocator::initPool(VkDevice device, uint32_t maxSets,
std::span<PoolSizeRatio> poolRatios) {
    std::vector<VkDescriptorPoolSize> poolSizes;
    for (PoolSizeRatio ratio : poolRatios) {
        poolSizes.push_back(
```

```

        VkDescriptorPoolSize{.type = ratio.type,
.descriptorCount = uint32_t(ratio.ratio * maxSets)}});
    }

    VkDescriptorPoolCreateInfo pool_info = {.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO};
    pool_info.flags = 0;
    pool_info.maxSets = maxSets;
    pool_info.poolSizeCount =
(uint32_t)poolSizes.size();
    pool_info.pPoolSizes = poolSizes.data();

    vkCreateDescriptorPool(device, &pool_info, nullptr, &pool);
}

void DescriptorAllocator::clearDescriptors(VkDevice device) {
    vkResetDescriptorPool(device, pool, 0);
}

void DescriptorAllocator::destroyPool(VkDevice device) {
    vkDestroyDescriptorPool(device, pool, nullptr);
}

VkDescriptorSet DescriptorAllocator::allocate(VkDevice device,
VkDescriptorSetLayout layout) {
    VkDescriptorSetAllocateInfo allocInfo = {.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO};
    allocInfo.pNext = nullptr;
    allocInfo.descriptorPool = pool;
    allocInfo.descriptorSetCount = 1;
    allocInfo.pSetLayouts = &layout;
    VkDescriptorSet ds;
    VK_CHECK(vkAllocateDescriptorSets(device, &allocInfo, &ds));
    return ds;
}

void DescriptorWriter::writeImage(int binding, VkImageView image,
VkSampler sampler, VkImageLayout layout,
                                VkDescriptorType type)
{
    VkDescriptorImageInfo& info =
        imageInfos.emplace_back(VkDescriptorImageInfo{.sampler =
sampler, .imageView = image, .imageLayout = layout});
    VkWriteDescriptorSet write = {.sType =

```

```

VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET};
    write.dstBinding          = binding;
    write.dstSet              = VK_NULL_HANDLE;
    write.descriptorCount     = 1;
    write.descriptorType      = type;
    write.pImageInfo          = &info;

    writes.push_back(write);
}

void DescriptorWriter::writeBuffer(int binding, VkBuffer buffer,
size_t size, size_t offset, VkDescriptorType type) {
    VkDescriptorBufferInfo& info =
        bufferInfos.emplace_back(VkDescriptorBufferInfo{.buffer =
buffer, .offset = offset, .range = size});
    VkWriteDescriptorSet write = {.sType =
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET};
    write.dstBinding          = binding;
    write.dstSet              = VK_NULL_HANDLE;
    write.descriptorCount     = 1;
    write.descriptorType      = type;
    write.pBufferInfo         = &info;

    writes.push_back(write);
}

void DescriptorWriter::clear() {
    imageInfos.clear();
    writes.clear();
    bufferInfos.clear();
}

void DescriptorWriter::update_set(VkDevice device, VkDescriptorSet
set) {
    for (VkWriteDescriptorSet& write : writes) {
        write.dstSet = set;
    }

    vkUpdateDescriptorSets(device, (uint32_t)writes.size(),
writes.data(), 0, nullptr);
}

void DescriptorAllocatorGrowable::init(VkDevice device, uint32_t
maxSets, std::span<PoolSizeRatio> poolRatios) {

```

```

        ratios.clear();

        for (auto r : poolRatios) {
            ratios.push_back(r);
        }
        VkDescriptorPool newPool = createPool(device, maxSets,
poolRatios);
        setsPerPool          = maxSets * 1.5;
        readyPools.push_back(newPool);
    }

void DescriptorAllocatorGrowable::clear_pools(VkDevice device) {
    for (auto p : readyPools) {
        vkResetDescriptorPool(device, p, 0);
    }
    for (auto p : fullPools) {
        vkResetDescriptorPool(device, p, 0);
        readyPools.push_back(p);
    }
    fullPools.clear();
}

void DescriptorAllocatorGrowable::destroyPool(VkDevice device) {
    for (auto p : readyPools) {
        vkDestroyDescriptorPool(device, p, nullptr);
    }
    readyPools.clear();
    for (auto p : fullPools) {
        vkDestroyDescriptorPool(device, p, nullptr);
    }
    fullPools.clear();
}

VkDescriptorPool DescriptorAllocatorGrowable::getPool(VkDevice
device) {
    VkDescriptorPool newPool;
    if (readyPools.size() != 0) {
        newPool = readyPools.back();
        readyPools.pop_back();
    } else {
        newPool = createPool(device, setsPerPool, ratios);
        setsPerPool = setsPerPool * 1.5;
        if (setsPerPool > 4092) {
            setsPerPool = 4092;

```

```

    }
}

return newPool;
}

VkDescriptorPool DescriptorAllocatorGrowable::createPool(VkDevice
device, uint32_t setCount,

    std::span<PoolSizeRatio> poolRatios) {
    std::vector<VkDescriptorPoolSize> poolSizes;
    for (PoolSizeRatio ratio : poolRatios) {
        poolSizes.push_back(
            VkDescriptorPoolSize{.type = ratio.type,
.descriptorCount = uint32_t(ratio.ratio * setCount)});
    }

    VkDescriptorPoolCreateInfo pool_info = {};
    pool_info.sType
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    pool_info.flags
    pool_info.maxSets
    pool_info.poolSizeCount
(uint32_t)poolSizes.size();
    pool_info.pPoolSizes

    VkDescriptorPool newPool;
    vkCreateDescriptorPool(device, &pool_info, nullptr, &newPool);
    return newPool;
}

VkDescriptorSet DescriptorAllocatorGrowable::allocate(VkDevice
device, VkDescriptorSetLayout layout, void* pNext) {
    VkDescriptorPool poolToUse = getPool(device);

    VkDescriptorSetAllocateInfo allocInfo = {};
    allocInfo.pNext
    allocInfo.sType
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    allocInfo.descriptorPool
    allocInfo.descriptorSetCount
    allocInfo.pSetLayouts
    VkDescriptorSet ds;
    VkResult result = vkAllocateDescriptorSets(device,

```

```

&allocInfo, &ds);

    if (result == VK_ERROR_OUT_OF_POOL_MEMORY || result ==
VK_ERROR_FRAGMENTED_POOL) {
        fullPools.push_back(poolToUse);
        poolToUse = getPool(device);
        allocInfo.descriptorPool = poolToUse;
        VK_CHECK(vkAllocateDescriptorSets(device, &allocInfo,
&ds));
    }
    readyPools.push_back(poolToUse);
    return ds;
}

```

## Файл engine.h

```
#pragma once

#include <types.h>

#include <deque>
#include <functional>
#include <set>
#include <span>
#include <string>
#include <unordered_map>
#include <vector>

#include <vk_mem_alloc.h>

#include <camera.h>
#include <descriptors.h>
#include <loader.h>
#include <pipelines.h>

#include <miniaudio.h>

#include <imgui.h>

#include "imfilebrowser.h"

#define SOL_ALL_SAFETIES_ON 1
#include <sol/sol.hpp>

struct MeshAsset;
namespace fastgltf {
struct Mesh;
}

struct DeletionQueue {
    std::deque<std::function<void()>> deletors;

    void push_function(std::function<void()>&& function) {
        deletors.push_back(function);
    }

    void flush() {
        for (auto it = deletors.rbegin(); it != deletors.rend();
```

```

it++) {
    (*it)();
}

    deletors.clear();
}
};

struct ComputePushConstants {
    glm::vec4 data1;
    glm::vec4 data2;
    glm::vec4 data3;
    glm::vec4 data4;
};

struct ComputeEffect {
    const char* name;

    VkPipeline          pipeline;
    VkPipelineLayout    layout;
    ComputePushConstants data;
};

struct RenderObject {
    uint32_t            indexCount;
    uint32_t            firstIndex;
    VkBuffer            indexBuffer;
    MaterialInstance* material;
    Bounds              bounds;
    glm::mat4           transform;
    VkDeviceAddress     vertexBufferAddress;
};

struct FrameData {
    VkSemaphore          _swapchainSemaphore,
    _renderSemaphore;
    VkFence              _renderFence;
    DescriptorAllocatorGrowable _frameDescriptors;
    DeletionQueue        _deletionQueue;
    VkCommandPool        _commandPool;
    VkCommandBuffer      _mainCommandBuffer;
};

constexpr unsigned int FRAME_OVERLAP = 2;

```

```

struct DrawContext {
    std::vector<RenderObject> OpaqueSurfaces;
    std::vector<RenderObject> TransparentSurfaces;
};

struct EngineStats {
    float frametime;
    int   triangle_count;
    int   drawcall_count;
    float mesh_draw_time;
};

struct GLTFMetallic_Roughness {
    MaterialPipeline opaquePipeline;
    MaterialPipeline transparentPipeline;

    VkDescriptorSetLayout materialLayout;

    struct MaterialConstants {
        glm::vec4 colorFactors;
        glm::vec4 metal_rough_factors;
        uint32_t  colorTexID;
        uint32_t  metalRoughTexID;
        uint32_t  pad1;
        uint32_t  pad2;
        glm::vec4 extra[13];
    };

    struct MaterialResources {
        AllocatedImage colorImage;
        VkSampler       colorSampler;
        AllocatedImage metalRoughImage;
        VkSampler       metalRoughSampler;
        VkBuffer        dataBuffer;
        uint32_t        dataBufferOffset;
    };

    DescriptorWriter writer;

    void build_pipelines(BlockEngine* engine);
    void clear_resources(VkDevice device);

    MaterialInstance write_material(VkDevice device, MaterialPass

```

```

pass, const MaterialResources& resources,

    DescriptorAllocatorGrowable& descriptorAllocator);
};

namespace JPH {
class BodyID;
}

struct MeshNode: public Node {
    std::shared_ptr<MeshAsset> mesh;

    virtual void Draw(const glm::mat4& topMatrix, DrawContext& ctx)
override;
};
struct TextureID {
    uint32_t Index;
};

struct TextureCache {
    std::vector<VkDescriptorImageInfo> Cache;
    std::unordered_map<std::string, TextureID> NameMap;
    TextureID AddTexture(const
VkImageView& image, VkSampler sampler);
};

class Script {
public:
    std::string path;
    std::string name;
    std::string id;
    sol::state lua;
    bool isLoaded = false;
    bool isActive = false;
    bool load(BlockEngine& engine);
    void init();
    void step();
    void stop();
};

struct Project {
    std::string name;
    std::string tempName;
    std::string path;

```

```

        std::unordered_map<std::string, std::string> loadedObjects;
        std::string                                mainLevelId;
        std::string                                scriptId;
        std::optional<Script>                      script;
};

class BlockEngine;

class Object: public IRenderable {
public:
    Object() = default;

    std::shared_ptr<LoadedGLTF> scene;
    glm::vec3                    position      = glm::vec3(1.0f);
    glm::vec3                    scale        = glm::vec3(1.0f);
    glm::vec3                    rotation     = glm::vec3(0.0f);
    glm::vec3                    extents      = glm::vec3(1.0f);
    glm::vec3                    linearVelocity = glm::vec3(0.0f);
    JPH::BodyID                  bodyId;
    std::string                  name;
    std::string                  tempName;
    Bounds                       bounds;
    bool                         isVisible = true;
    bool                         isDynamic = false;
    bool                         isActive  = false;
    std::string                  gltfuuid;
    std::string                  scriptId;
    bool                         isScriptActive = true;
    std::optional<Script>        script;
    std::set<std::string>        customTypes;
    std::string                  tempType;
    bool                         lockPositionX = false;
    bool                         lockPositionY = false;
    bool                         lockPositionZ = false;

    Object(const Object& other);
    Object& operator=(const Object& other);

    virtual void Draw(const glm::mat4& topMatrix, DrawContext&
ctx);
    void          setName(const std::string& newName);
    void          updateTransforms();
private:
    glm::vec3 prevPosition      = glm::vec3(1.0f);

```

```

glm::vec3 prevScale          = glm::vec3(1.0f);
glm::vec3 prevRotation       = glm::vec3(0.0f);
glm::vec3 prevLinearVelocity = glm::vec3(0.0f);
};

struct AudioFile {
    std::string id;
    std::string name;
    std::string path;
    ma_sound      sound;
    bool          isLooping = false;
    float         volume    = 1.0f;

    uint64_t getCursorPosition();
    uint64_t getLengthInFrames();
    void      setCursorPosition(uint64_t position);
    float      getCursorPositionInSeconds();
    float      getLengthInSeconds();
    void      play();
    void      stop();
    void      stopWithFadeOut(uint64_t fadeInMilliseconds);
    void      pause();
    bool      isPlaying();
    void      setLooping(bool loop);
    void      setVolume(float newVolume);
    void      destroy();

    int tempFadeInMilliseconds = 3000;
};

enum UiType {
    None,
    Text,
    Button,
    Window,
};

class UiElement {
public:
    UiType          type =
    UiType::None;
    std::string     text;
    std::string     name;
    glm::vec2       position =

```

```

glm::vec2(64, 64);
    glm::vec2                                size      =
glm::vec2(100, 50);
    float                                    fontSize =
1.0f;
    std::string                             id;
    std::optional<std::reference_wrapper<UiElement>> parent;
    bool                                    isVisible
= true;
    bool                                    pressed
= false;
    bool
useCenterPosition = false;
    bool                                    useBackground
= false;
    bool                                    showTitleBar
= true;
    std::unordered_map<std::string, UiElement> children;
    std::string                             scriptId;
    std::optional<Script>                   script;

    void display();

    std::string tempName;
    void        rename(BlockEngine* eng);
    std::string tempText;
    void        changeText();
private:
    void displayText();
    void displayButton();
    void displayWindow();
};

enum class selectedObjectType {
    None,
    GLTF,
    Audio,
    UiElement,
    Camera,
    Script,
    Level,
    GLTFModel
};

```

```

enum class KeyType;

class Level {
public:
    std::string name;
    std::string id;
    std::unordered_map<std::string, Object> loadedScenes;
    std::unordered_map<std::string, UiElement> uiElements;
    std::string tempName;

    std::string scriptId;
    std::optional<Script> script;
};

class BlockEngine {
public:
    void
loadLevel(std::string id);
    std::string
createLevel();
    bool
createProject();
    void
createObject(std::string id);
    std::unordered_map<std::string, std::set<std::string>>
collisionMap;
    void addCollision(const JPH::BodyID& Body1,
const JPH::BodyID& Body2);
    void removeCollision(const JPH::BodyID& Body1,
const JPH::BodyID& Body2);
    void initKeyMap();
    void processInput();
    void cameraInput();
    static BlockEngine& Get();
    void searchGameFiles();
    void init();
    void cleanup();
    void draw();
    void draw_main(VkCommandBuffer cmd);
    void draw_imgui(VkCommandBuffer cmd, VkImageView
targetImageView);
    void draw_geometry(VkCommandBuffer cmd);
    void run();
    bool draw_menu();

```

```

void                drawHierarchy();
void                drawGLTFPModels();
void                drawProperties();
void                drawGLTFProperties();
void                drawAudioProperties();
void                drawUIiindowProperties();
void                drawUiProperties();
void                drawCameraProperties();
void                drawScriptProperties();
void                drawLevelProperties();
void                drawGLTFModelProperties();
void                drawStats();
void                drawMenuBar();
void                drawFileDialog();
void                drawSettings();
void                exportProject();
void                drawTop();
void                drawBottom();
void                loadScript(const std::string& path);
void                update_scene();
GPUMeshBuffers      uploadMesh(std::span<uint32_t> indices,
std::span<Vertex> vertices);
FrameData&          get_current_frame();
FrameData&          get_last_frame();
AllocatedBuffer      create_buffer(size_t allocSize,
VkBufferUsageFlags usage, VmaMemoryUsage memoryUsage);
AllocatedImage       create_image(VkExtent3D size, VkFormat
format, VkImageUsageFlags usage, bool mipmapped = false);
AllocatedImage       create_image(void* data, VkExtent3D size,
VkFormat format, VkImageUsageFlags usage,
                                bool mipmapped =
false);
void
immediate_submit(std::function<void(VkCommandBuffer cmd)>&&
function);
void                destroy_image(const AllocatedImage& img);
void                destroy_buffer(const AllocatedBuffer&
buffer);

std::unordered_map<std::string, Level>
levels;
std::string
currentLevelId;
Level*

```

```

currentLevel = nullptr;
    std::unordered_map<std::string, AudioFile>
audioFiles;
    ma_engine
audio_engine;
    std::unordered_map<KeyType, bool>
keys;
    glm::vec2
mouseMotion;
    bool
needsExit                = false;
    bool
cameraExternalControlled = false;
    Project
currentProject;
    VkExtent2D
_windowExtent{1700, 900};
    struct SDL_Window*
_window{nullptr};
    VkInstance
_instance;
    VkDebugUtilsMessengerEXT
_debug_messenger;
    VkPhysicalDevice
_chosenGPU;
    VkDevice
_device;
    VkQueue
_graphicsQueue;
    uint32_t
_graphicsQueueFamily;
    AllocatedBuffer
_defaultGLTFMaterialData;
    FrameData
_frames[FRAME_OVERLAP];
    VkSurfaceKHR
_surface;
    VkSwapchainKHR
_swapchain;
    VkFormat
_swapchainImageFormat;
    VkExtent2D
_swapchainExtent;
    VkExtent2D

```

```

_drawExtent;
    VkDescriptorPool
_descriptorPool;
    DescriptorAllocator
globalDescriptorAllocator;
    VkPipeline
_gradientPipeline;
    VkPipelineLayout
_gradientPipelineLayout;
    std::vector<VkImage>
_swapchainImages;
    std::vector<VkImageView>
_swapchainImageViews;
    VkDescriptorSet
_drawImageDescriptors;
    VkDescriptorSetLayout
_drawImageDescriptorLayout;
    DeletionQueue
_mainDeletionQueue;
    VmaAllocator
_allocator;
    VkDescriptorSetLayout
_gpuSceneDataDescriptorLayout;
    GLTFMetallic_Roughness
metalRoughMaterial;
    AllocatedImage
_drawImage;
    AllocatedImage
_depthImage;
    VkFence
_immFence;
    VkCommandBuffer
_immCommandBuffer;
    VkCommandPool
_immCommandPool;
    AllocatedImage
_whiteImage;
    AllocatedImage
_blackImage;
    AllocatedImage
_greyImage;
    AllocatedImage
_errorCheckerboardImage;
    VkSampler

```

```

_defaultSamplerLinear;
    VkSampler
_defaultSamplerNearest;
    TextureCache
texCache;
    GPUMeshBuffers
rectangle;
    DrawContext
drawCommands;
    GPUSceneData
sceneData;
    Camera
mainCamera;
    EngineStats
stats;
    bool
_isInitialized{false};
    int
_frameNumber{0};
    selectedObjectType
selectedObject = selectedObjectType::None;
    std::string
selectedObjectUuid;
    std::string
selectedObjectParentUuid;
    std::string
selectedScriptUuid;
    std::vector<ComputeEffect>
backgroundEffects;
    int
currentBackgroundEffect{0};
    bool
isGameInstance = false;
    std::unordered_map<std::string, Script>
loadedScripts;
    bool
showStats = false;
    bool
isPhysicsRunning = false;
    bool
scriptSimulateionIsActive = false;
    bool
hideUi = false;
    bool

```

```

freeCamera                = true;
    std::string
engineExecPath;
    std::unordered_map<std::string, std::shared_ptr<LoadedGLTF>>
loadedGLTFs;
    std::vector<std::shared_ptr<LoadedGLTF>>
brickadiaScene;
    bool
resize_requested{false};
    bool
freeze_rendering{false};
    ImGui::FileBrowser
mainFileDialog;
private:
    void init_vulkan();
    void init_swapchain();
    void create_swapchain(uint32_t width, uint32_t height);
    void resize_swapchain();
    void destroy_swapchain();
    void init_commands();
    void init_pipelines();
    void init_background_pipelines();
    void init_descriptors();
    void init_sync_structures();
    void init_renderables();
    void init_imgui();
    void init_default_data();
};

```

## Файл engine.cpp

```
#include "engine.h"
#include "descriptors.h"
#include "images.h"
#include "loader.h"
#include "logger.h"
#include <initializers.h>
#include <types.h>

#include <SDL.h>

#include <SDL_vulkan.h>

#include "VkBootstrap.h"

#include "imgui_impl_sdl2.h"
#include "imgui_impl_vulkan.h"
#include "imgui_stdlib.h"
#include <imgui.h>

#include "imfilebrowser.h"

#include <glm/gtx/transform.hpp>

#define VMA_IMPLEMENTATION
#include "vk_mem_alloc.h"

#include <miniaudio.h>

#include <Jolt/Jolt.h>

#include <filesystem>
#include <fstream>
#include <limits>

#include <nlohmann/json.hpp>
using json = nlohmann::json;

#define SOL_ALL_SAFETIES_ON 1
#include <sol/sol.hpp>

#define UUID_SYSTEM_GENERATOR
#include <stduuid/uuid.h>
```

```

#include <Jolt/Core/Factory.h>
#include <Jolt/Core/JobSystemThreadPool.h>
#include <Jolt/Core/TempAllocator.h>
#include <Jolt/Physics/Body/BodyActivationListener.h>
#include <Jolt/Physics/Body/BodyCreationSettings.h>
#include <Jolt/Physics/Collision/Shape/BoxShape.h>
#include <Jolt/Physics/Collision/Shape/SphereShape.h>
#include <Jolt/Physics/PhysicsSettings.h>
#include <Jolt/Physics/PhysicsSystem.h>
#include <Jolt/RegisterTypes.h>

#include <zip.h>

std::vector<char> readFileToBuffer(const std::string& filename) {
    std::ifstream file(filename, std::ios::binary);
    if (!file) {
        throw std::runtime_error("Could not open file: " +
filename);
    }
    return
std::vector<char>((std::istreambuf_iterator<char>(file)),
std::istreambuf_iterator<char>());
}

void writeFileFromBuffer(const std::string& filename, const
std::vector<char>& buffer) {
    std::ofstream file(filename, std::ios::binary);
    if (!file) {
        throw std::runtime_error("Could not open file for writing:
" + filename);
    }
    file.write(buffer.data(), buffer.size());
}

void extractZipArchive(const std::string& path) {
    const std::string archiveName = "data.bin";
    std::filesystem::current_path(path);
    int error = 0;
    zip_t* zipArchive = zip_open(archiveName.c_str(), ZIP_RDONLY,
&error);
    if (!zipArchive) {
        BK_ERROR("Failed to open archive: {}", archiveName);
        return;
    }
}

```

```

    }
    std::filesystem::create_directory(path /
std::filesystem::path("assets"));
    std::filesystem::current_path(path /
std::filesystem::path("assets"));
    zip_int64_t numEntries = zip_get_num_entries(zipArchive, 0);
    for (zip_int64_t i = 0; i < numEntries; ++i) {
        const char* filename = zip_get_name(zipArchive, i, 0);
        if (!filename) {
            BK_ERROR("Failed to get the name of file in archive
index: {}", i);
            continue;
        }
        BK_INFO("Extracting file: {}", filename);
        zip_file_t* file = zip_fopen_index(zipArchive, i, 0);
        if (!file) {
            BK_ERROR("Failed to open file in archive: {}",
filename);
            continue;
        }
        std::vector<char> buffer;
        zip_stat_t fileStat;
        if (zip_stat_index(zipArchive, i, 0, &fileStat) == 0) {
            buffer.resize(fileStat.size);
            zip_fread(file, buffer.data(), buffer.size());
        } else {
            BK_ERROR("Failed to get file stats for: {}",
filename);
            zip_fclose(file);
            continue;
        }
        zip_fclose(file);
        try {
            writeFileFromBuffer(filename, buffer);
            BK_INFO("Extracted file: {}", filename);
        } catch (const std::exception& e) {
            BK_ERROR("Error writing file {}: {}", filename,
e.what());
        }
    }
    std::filesystem::current_path(path);
    zip_close(zipArchive);
    BK_INFO("Archive extracted successfully: {}", archiveName);
}

```

```

void createArchive(const std::string& path) {
    std::filesystem::create_directory(path /
std::filesystem::path("export"));
    std::filesystem::current_path(path /
std::filesystem::path("export"));
    std::string archiveName = "data.bin";
    int error = 0;
    zip_t* zipArchive = zip_open(archiveName.c_str(),
ZIP_CREATE | ZIP_TRUNCATE, &error);
    if (!zipArchive) {
        BK_ERROR("Failed to create archive: {}", archiveName);
        return;
    }
    std::filesystem::path assetsPath = path /
std::filesystem::path("assets");
    std::vector<std::vector<char>> buffers;
    for (auto const& dir_entry :
std::filesystem::directory_iterator{assetsPath}) {

        buffers.emplace_back(readFileToBuffer(dir_entry.path().string()
));

        zip_source_t* source = zip_source_buffer(zipArchive,
buffers.back().data(), buffers.back().size(), 0);
        if (!source) {
            BK_ERROR("Failed to create zip source for file: {}",
dir_entry.path().string());
            continue;
        }
        if (zip_file_add(zipArchive,
dir_entry.path().filename().string().c_str(), source,
ZIP_FL_ENC_UTF_8) < 0) {
            BK_ERROR("Failed to add file to archive: {}",
dir_entry.path().string());
            zip_source_free(source);
            continue;
        }
        BK_INFO("Added file to archive: ",
dir_entry.path().string());
    }
    if (zip_close(zipArchive) < 0) {
        BK_ERROR("Failed to close archive: {}", archiveName);
    } else {

```

```

        BK_INFO("Archive created successfully: ", archiveName);
    }
    std::filesystem::current_path(path);
}

void BlockEngine::exportProject() {
    createArchive(currentProject.path());
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("engine.exe"),
                        std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path(currentProject.name + ".exe"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("effects.spv"),
                        std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("effects.spv"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("effects2.spv"),
                        std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("effects2.spv"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("fragment.spv"),
                        std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("fragment.spv"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("lua.dll"),
                        std::filesystem::current_path() /
std::filesystem::path("export") /

```

```

std::filesystem::path("lua.dll"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("SDL2.dll"),
                                std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("SDL2.dll"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("vertex.spv"),
                                std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("vertex.spv"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("zip.dll"),
                                std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("zip.dll"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("zlib1.dll"),
                                std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("zlib1.dll"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(engineExecPath) /
std::filesystem::path("bz2.dll"),
                                std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("bz2.dll"),

std::filesystem::copy_options::overwrite_existing);
    std::filesystem::copy(std::filesystem::path(currentProject.path

```

```

) / std::filesystem::path("config.json"),
                                std::filesystem::current_path() /
std::filesystem::path("export") /

std::filesystem::path("config.json"),

std::filesystem::copy_options::overwrite_existing);
}

using namespace JPH::literals;
using namespace JPH;
using namespace std;

#ifdef JPH_ENABLE_ASSERTS
static bool AssertFailedImpl(const char* inExpression, const char*
inMessage, const char* inFile, uint inLine) {
    cout << inFile << ":" << inLine << ": (" << inExpression << "
" << (inMessage != nullptr ? inMessage : "") << endl;
    return true;
};
#endif

namespace Layers {
static constexpr ObjectLayer NON_MOVING = 0;
static constexpr ObjectLayer MOVING     = 1;
static constexpr ObjectLayer NUM_LAYERS = 2;
};

class ObjectLayerPairFilterImpl: public ObjectLayerPairFilter {
public:
    virtual bool ShouldCollide(ObjectLayer inObject1, ObjectLayer
inObject2) const override {
        switch (inObject1) {
            case Layers::NON_MOVING: return inObject2 ==
Layers::MOVING;
            case Layers::MOVING:      return true;
            default:                  JPH_ASSERT(false); return false;
        }
    }
};

namespace BroadPhaseLayers {
static constexpr BroadPhaseLayer NON_MOVING(0);
static constexpr BroadPhaseLayer MOVING(1);

```

```

static constexpr uint NUM_LAYERS(2);
};

class BPLayerInterfaceImpl final: public BroadPhaseLayerInterface {
public:
    BPLayerInterfaceImpl() {
        mObjectToBroadPhase[Layers::NON_MOVING] =
BroadPhaseLayers::NON_MOVING;
        mObjectToBroadPhase[Layers::MOVING] =
BroadPhaseLayers::MOVING;
    }
    virtual uint GetNumBroadPhaseLayers() const override {
        return BroadPhaseLayers::NUM_LAYERS;
    }
    virtual BroadPhaseLayer GetBroadPhaseLayer(ObjectLayer inLayer)
const override {
        JPH_ASSERT(inLayer < Layers::NUM_LAYERS);
        return mObjectToBroadPhase[inLayer];
    }

#if defined(JPH_EXTERNAL_PROFILE) || defined(JPH_PROFILE_ENABLED)
    virtual const char* GetBroadPhaseLayerName(BroadPhaseLayer
inLayer) const override {
        switch ((BroadPhaseLayer::Type)inLayer) {
            case (BroadPhaseLayer::Type)BroadPhaseLayers::NON_MOVING:
return "NON_MOVING";
            case (BroadPhaseLayer::Type)BroadPhaseLayers::MOVING:
return "MOVING";
            default:
JPH_ASSERT(false); return "INVALID";
        }
    }
#endif
private:
    BroadPhaseLayer mObjectToBroadPhase[Layers::NUM_LAYERS];
};

class ObjectVsBroadPhaseLayerFilterImpl: public
ObjectVsBroadPhaseLayerFilter {
public:
    virtual bool ShouldCollide(ObjectLayer inLayer1,
BroadPhaseLayer inLayer2) const override {
        switch (inLayer1) {
            case Layers::NON_MOVING: return inLayer2 ==

```

```

BroadPhaseLayers::MOVING;
        case Layers::MOVING:         return true;
        default:                     JPH_ASSERT(false); return false;
    }
}

};

class MyContactListener: public ContactListener {
public:
    MyContactListener(BlockEngine& eng): m_engine(eng) {}
    virtual ValidateResult OnContactValidate(const Body& inBody1,
const Body& inBody2, RVec3Arg inBaseOffset,
                                                                    const
CollideShapeResult& inCollisionResult) override {
        return ValidateResult::AcceptAllContactsForThisBodyPair;
    }
    virtual void OnContactAdded(const Body& inBody1, const Body&
inBody2, const ContactManifold& inManifold,
                                                                    ContactSettings&
ioSettings) override {
        m_engine.addCollision(inBody1.GetID(), inBody2.GetID());
    }
    virtual void OnContactPersisted(const Body& inBody1, const
Body& inBody2, const ContactManifold& inManifold,
                                                                    ContactSettings&
ioSettings) override {}
    virtual void OnContactRemoved(const SubShapeIDPair&
inSubShapePair) override {
        m_engine.removeCollision(inSubShapePair.GetBody1ID(),
inSubShapePair.GetBody2ID());
    }
private:
    BlockEngine& m_engine;
};

void BlockEngine::addCollision(const JPH::BodyID& Body1, const
JPH::BodyID& Body2) {
    std::string body1UUID;
    std::string body2UUID;
    for (auto& [uuid, obj] : currentLevel->loadedScenes) {
        if (obj.bodyId == Body1) {
            body1UUID = uuid;
        }
        if (obj.bodyId == Body2) {

```

```

        body2UUID = uuid;
    }
}
if (body1UUID.empty() || body2UUID.empty()) {
    return;
}
if (body1UUID == body2UUID) {
    return;
}
collisionMap[body1UUID].insert(body2UUID);
collisionMap[body2UUID].insert(body1UUID);
BK_TRACE("Collision between: {} and {}", body1UUID, body2UUID);
}

void BlockEngine::removeCollision(const JPH::BodyID& Body1, const
JPH::BodyID& Body2) {
    std::string body1UUID;
    std::string body2UUID;
    for (auto& [uuid, obj] : currentLevel->loadedScenes) {
        if (obj.bodyId == Body1) {
            body1UUID = uuid;
        }
        if (obj.bodyId == Body2) {
            body2UUID = uuid;
        }
    }
    if (body1UUID.empty() || body2UUID.empty()) {
        return;
    }
    if (body1UUID == body2UUID) {
        return;
    }
    if (collisionMap.find(body1UUID) != collisionMap.end() &&
collisionMap[body1UUID].contains(body2UUID)) {
        collisionMap[body1UUID].erase(body2UUID);
    }
    if (collisionMap.find(body2UUID) != collisionMap.end() &&
collisionMap[body2UUID].contains(body1UUID)) {
        collisionMap[body2UUID].erase(body1UUID);
    }
    BK_TRACE("Collision ended between: {} and {}", body1UUID,
body2UUID);
}

```

```

class MyBodyActivationListener: public BodyActivationListener {
public:
    virtual void OnBodyActivated(const BodyID& inBodyID, uint64
inBodyUserData) override {}
    virtual void OnBodyDeactivated(const BodyID& inBodyID, uint64
inBodyUserData) override {}
};

void Object::setName(const std::string& newName) {
    name      = newName;
    tempName = newName;
}
BodyInterface* body_interface = nullptr;
enum class KeyType {
    A          = 1,
    B          = 2,
    C          = 3,
    D          = 4,
    E          = 5,
    F          = 6,
    G          = 7,
    H          = 8,
    I          = 9,
    J          = 10,
    K          = 11,
    L          = 12,
    M          = 13,
    N          = 14,
    O          = 15,
    P          = 16,
    Q          = 17,
    R          = 18,
    S          = 19,
    T          = 20,
    U          = 21,
    V          = 22,
    W          = 23,
    X          = 24,
    Y          = 25,
    Z          = 26,
    Num_1      = 27,
    Num_2      = 28,
    Num_3      = 29,
    Num_4      = 30,

```

```

    Num_5      = 31,
    Num_6      = 32,
    Num_7      = 33,
    Num_8      = 34,
    Num_9      = 35,
    Num_0      = 36,
    Enter      = 37,
    Escape     = 38,
    Backspace  = 39,
    Tab        = 40,
    Space      = 41,
    Up         = 42,
    Down       = 43,
    Left       = 44,
    Right      = 45,
    Mouse_Left = 46,
    Mouse_Right = 47,
    NUBMER_OF_KEYS
};

void BlockEngine::initKeyMap() {
    for (int i = 1; i < (int)KeyType::NUBMER_OF_KEYS; i++) {
        keys[KeyType(i)] = false;
    }
}

uint64_t AudioFile::getCursorPosition() {
    uint64_t currentFrame = 0;
    ma_result result;
    result = ma_sound_get_cursor_in_pcm_frames(&sound,
&currentFrame);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to getCursorPosition: {}", name);
    }
    return currentFrame;
}

uint64_t AudioFile::getLengthInFrames() {
    uint64_t length = 0;
    ma_result result;
    result = ma_sound_get_length_in_pcm_frames(&sound, &length);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to getLengthInFrames: {}", name);
    }
}

```

```

        return length;
    }
void AudioFile::setCursorPosition(uint64_t position) {
    ma_result result;
    result = ma_sound_seek_to_pcm_frame(&sound, position);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to setCursorPosition: {}", name);
    }
}

float AudioFile::getCursrorPositionInSeconds() {
    float currentPosition = 0;
    ma_result result;
    result = ma_sound_get_cursor_in_seconds(&sound,
&currentPosition);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to getCursrorPositionInSeconds: {}",
name);
    }
    return currentPosition;
}

float AudioFile::getLengthInSeconds() {
    float length = 0;
    ma_result result;
    result = ma_sound_get_length_in_seconds(&sound, &length);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to getLengthInSeconds: {}", name);
    }
    return length;
}

void AudioFile::play() {
    ma_result result;
    result = ma_sound_start(&sound);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to play: {}", name);
    }
}

void AudioFile::stop() {
    ma_result result;
    result = ma_sound_stop(&sound);
    setCursorPosition(0);
}

```

```

        if (result != MA_SUCCESS) {
            BK_WARN("Failed to stop: {}", name);
        }
    }

void AudioFile::stopWithFadeOut(uint64_t fadeInMilliseconds) {
    ma_sound_set_fade_in_milliseconds(&sound, -1, 0,
    fadeInMilliseconds);
}

void AudioFile::pause() {
    ma_result result;
    result = ma_sound_stop(&sound);
    if (result != MA_SUCCESS) {
        BK_WARN("Failed to pause: {}", name);
    }
}

bool AudioFile::isPlaying() {
    return ma_sound_is_playing(&sound);
}

void AudioFile::setLooping(bool loop) {
    ma_sound_set_looping(&sound, loop);
}

void AudioFile::setVolume(float newVolume) {
    volume = newVolume;
    ma_sound_set_volume(&sound, volume);
}

void AudioFile::destroy() {
    stop();
    ma_sound_uninit(&sound);
}

JPH::BodyID copyBody(const Object& object) {
    if ((object.bodyId.IsInvalid())) {
        return JPH::BodyID();
    }

    BodyID          bodyId      = object.bodyId;
    const auto&      extents     = object.bounds.extents;

```

```

        const auto          position    = RVec3(object.position.x,
object.position.y, object.position.z);
        const auto          rotation    =
glm::quat(glm::radians(object.rotation));
        ObjectLayer         layer       = object.isDynamic ?
Layers::MOVING : Layers::NON_MOVING;
        EMotionType         motionType = object.isDynamic ?
EMotionType::Dynamic : EMotionType::Static;
        BodyCreationSettings settings(new
BoxShapeSettings(Vec3(extents.x, extents.y, extents.z)), position,
                                Quat(rotation.x,
rotation.y, rotation.z, rotation.w), motionType, layer);
        settings.mPosition      = body_interface-
>GetPosition(bodyId);
        settings.mRotation      = body_interface-
>GetRotation(bodyId);
        settings.mLinearVelocity = body_interface-
>GetLinearVelocity(bodyId);
        settings.mAngularVelocity = body_interface-
>GetAngularVelocity(bodyId);
        EActivation activation    = object.isDynamic ?
EActivation::Activate : EActivation::DontActivate;
        return body_interface->CreateAndAddBody(settings, activation);
    }

void deleteObject(Object& obj) {
    if (obj.bodyId.IsInvalid() || !body_interface-
>IsAdded(obj.bodyId)) {
        return;
    }
    body_interface->RemoveBody(obj.bodyId);
    body_interface->DestroyBody(obj.bodyId);
}

void Script::init() {
    if (!isActive) {
        return;
    }
    lua["init"]();
}

void Script::step() {
    if (!isActive) {
        return;
    }

```

```

        }
        lua["step"]();
    }
    void Script::stop() {
        if (!isActive) {
            return;
        }
        lua["stop"]();
    }

    void createPhysicsBody(Object& object);

    void BlockEngine::loadLevel(std::string id) {
        if (currentLevel != nullptr) {
            for (auto& [_, obj] : currentLevel->loadedScenes) {
                deleteObject(obj);
            }
            currentLevel->script->stop();
        }
        currentLevel = &levels[id];
        currentLevelId = id;
        for (auto& [_, obj] : currentLevel->loadedScenes) {
            createPhysicsBody(obj);
        }
        if (!currentLevel->scriptId.empty() &&
scriptSimulateionIsActive) {
            currentLevel->script->lua["thisObjectId"] =
currentLevelId;
            currentLevel->script->init();
        }
        for (auto& [objId, obj] : currentLevel->loadedScenes) {
            if (!obj.scriptId.empty()) {
                obj.script->lua["thisObjectId"] = objId;
                obj.script->init();
            }
        }
    }

    std::string BlockEngine::createLevel() {
        std::string id =
uuids::to_string(uuids::uuid_system_generator{}());
        Level level;
        level.name = id;
        level.tempName = id;
    }

```

```

        level.id          = id;
        levels[id]        = std::move(level);
        return id;
    }

    bool Script::load(BlockEngine& engine) {
        lua.open_libraries(sol::lib::base, sol::lib::io,
        sol::lib::math, sol::lib::table);
        lua.safe_script_file(path);
        BK_INFO("Lua file loaded: {}", path);
        try {
            lua.new_usertype<glm::vec3>("vec3",

                sol::constructors<glm::vec3(), glm::vec3(float),
            glm::vec3(float, float, float)>(),

                "x", &glm::vec3::x,
            "y", &glm::vec3::y, "z", &glm::vec3::z);

            lua.new_usertype<glm::vec2>("vec2",
            sol::constructors<glm::vec2(), glm::vec2(float), glm::vec2(float,
            float)>(),

                "x", &glm::vec2::x,
            "y", &glm::vec2::y);

            lua["getObjectId"] = [this, &engine](const std::string&
            name) -> std::string {
                for (auto& [id, obj] : engine.currentLevel-
            >loadedScenes) {
                    if (obj.name == name) {
                        return id;
                    }
                }

                return "";
            };

            lua["getAudioId"] = [this, &engine](const std::string&
            name) -> std::string {
                for (auto& [id, obj] : engine.audioFiles) {
                    if (obj.name == name) {
                        return id;
                    }
                }
            }
        }
    }

```

```

        return "";
    };

    lua["getUiWindowId"] = [this, &engine](const std::string&
name) -> std::string {
        for (auto& [id, obj] : engine.currentLevel-
>uiElements) {
            if (obj.name == name) {
                return id;
            }
        }

        return "";
    };

    lua["setUiWindowVisability"] = [this, &engine](const
std::string& id, bool isVisible) {
        auto objIt = engine.currentLevel-
>uiElements.find(id);
        if (objIt != engine.currentLevel->uiElements.end()) {
            objIt->second.isVisible = isVisible;
        }
    };

    lua["setUiWindowChildText"] = [this, &engine](const
std::string& id, const std::string& childName,

    const std::string& text) {
        auto objIt = engine.currentLevel-
>uiElements.find(id);
        if (objIt != engine.currentLevel->uiElements.end()) {
            for (auto& child : objIt->second.children) {
                if (child.second.name == childName) {
                    child.second.text = text;
                }
            }
        }
    };

    lua["setUiWindowChildPosition"] = [this, &engine](const
std::string& id, const std::string& childName, float x,

    float y) {
        auto objIt = engine.currentLevel-

```

```

>uiElements.find(id);
    if (objIt != engine.currentLevel->uiElements.end()) {
        for (auto& child : objIt->second.children) {
            if (child.second.name == childName) {
                child.second.position = {x, y};
            }
        }
    }
};

lua["setUiWindowChildSize"] = [this, &engine](const
std::string& id, const std::string& childName, float width,

    float height) {
    auto objIt = engine.currentLevel-
>uiElements.find(id);
    if (objIt != engine.currentLevel->uiElements.end()) {
        for (auto& child : objIt->second.children) {
            if (child.second.name == childName) {
                child.second.size = {width, height};
            }
        }
    }
};

lua["getUiWindowVisability"] = [this, &engine](const
std::string& id) -> bool {
    auto objIt = engine.currentLevel-
>uiElements.find(id);
    if (objIt != engine.currentLevel->uiElements.end()) {
        return objIt->second.isVisible;
    }
    return false;
};

lua["getUiButtonState"] = [this, &engine](const
std::string& id, const std::string& buttonName) -> bool {
    auto objIt = engine.currentLevel-
>uiElements.find(id);
    if (objIt != engine.currentLevel->uiElements.end()) {
        for (auto& button : objIt->second.children) {
            if (button.second.name == buttonName) {
                return button.second.pressed;
            }
        }
    }
};

```

```

        }
    }
    return false;
};

lua["closeGame"] = [this, &engine]() {
    engine.needsExit = true;
};

lua["playAudio"] = [this, &engine](const std::string& id)
{
    auto objIt = engine.audioFiles.find(id);
    if (objIt != engine.audioFiles.end()) {
        if (!objIt->second.isPlaying()) {
            objIt->second.play();
        }
    }
};

lua["playAudioFromStart"] = [this, &engine](const
std::string& id) {
    auto objIt = engine.audioFiles.find(id);
    if (objIt != engine.audioFiles.end()) {
        if (objIt->second.isPlaying()) {
            objIt->second.stop();
        }
        objIt->second.play();
    }
};

lua["stopAudio"] = [this, &engine](const std::string& id)
{
    auto objIt = engine.audioFiles.find(id);
    if (objIt != engine.audioFiles.end()) {
        objIt->second.stop();
    }
};

lua["setObjectPosition"] = [this, &engine](const
std::string& id, float x, float y, float z) {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
{

```

```

        objIt->second.position = {x, y, z};
        objIt->second.updateTransforms();
    }
};

lua["getObjectPosition"] = [this, &engine](const
std::string& id) -> glm::vec3 {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {
        return objIt->second.position;
    }
    return {0, 0, 0};
};

lua["setObjectRotation"] = [this, &engine](const
std::string& id, float x, float y, float z) {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {
        objIt->second.rotation = {x, y, z};
        objIt->second.updateTransforms();
    }
};

lua["getObjectVelocity"] = [this, &engine](const
std::string& id) -> glm::vec3 {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {
        return objIt->second.linearVelocity;
    }
    return {0, 0, 0};
};

lua["addObjectVelocity"] = [this, &engine](const
std::string& id, float x, float y, float z) {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {

```

```

        objIt->second.linearVelocity += glm::vec3{x, y,
z};

        body_interface->AddLinearVelocity(objIt-
>second.bodyId, Vec3(x, y, z));
    }
};

lua["setObjectVelocity"] = [this, &engine](const
std::string& id, float x, float y, float z) {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {
        objIt->second.linearVelocity += glm::vec3{x, y,
z};

        body_interface->SetLinearVelocity(objIt-
>second.bodyId, Vec3(x, y, z));
    }
};

lua["getKeyState"] = [this, &engine](int key) -> bool {
    return engine.keys[static_cast<KeyType>(key)];
};

lua["getObjectCollisions"] = [this, &engine](const
std::string& id) -> std::vector<std::string> {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    std::vector<std::string> collisions;
    if (objIt != engine.currentLevel->loadedScenes.end()
&& engine.collisionMap.contains(id)) {
        for (auto& uuid : engine.collisionMap[id]) {
            collisions.push_back(uuid);
        }
    }
    return collisions;
};

lua["getObjectCustomTypes"] = [this, &engine](const
std::string& id) -> std::vector<std::string> {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    std::vector<std::string> types;

```

```

        if (objIt != engine.currentLevel->loadedScenes.end())
        {
            for (auto& type : objIt->second.customTypes) {
                types.push_back(type);
            }
        }
        return types;
    };

    lua["lockObjectPositionX"] = [this, &engine](const
std::string& id) {
        auto objIt = engine.currentLevel-
>loadedScenes.find(id);
        if (objIt != engine.currentLevel->loadedScenes.end())
        {
            objIt->second.lockPositionX = true;
        }
    };

    lua["lockObjectPositionY"] = [this, &engine](const
std::string& id) {
        auto objIt = engine.currentLevel-
>loadedScenes.find(id);
        if (objIt != engine.currentLevel->loadedScenes.end())
        {
            objIt->second.lockPositionY = true;
        }
    };

    lua["lockObjectPositionZ"] = [this, &engine](const
std::string& id) {
        auto objIt = engine.currentLevel-
>loadedScenes.find(id);
        if (objIt != engine.currentLevel->loadedScenes.end())
        {
            objIt->second.lockPositionZ = true;
        }
    };

    lua["unlockObjectPositionX"] = [this, &engine](const
std::string& id) {
        auto objIt = engine.currentLevel-
>loadedScenes.find(id);
        if (objIt != engine.currentLevel->loadedScenes.end())

```

```

{
    objIt->second.lockPositionX = false;
}
};

lua["unlockObjectPositionY"] = [this, &engine](const
std::string& id) {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {
        objIt->second.lockPositionY = false;
    }
};

lua["unlockObjectPositionZ"] = [this, &engine](const
std::string& id) {
    auto objIt = engine.currentLevel-
>loadedScenes.find(id);
    if (objIt != engine.currentLevel->loadedScenes.end())
    {
        objIt->second.lockPositionZ = false;
    }
};

lua["getCameraPosition"] = [this, &engine]() -> glm::vec3
{
    return engine.mainCamera.position;
};

lua["setCameraPosition"] = [this, &engine](float x, float
y, float z) {
    engine.mainCamera.position = glm::vec3{x, y, z};
    engine.mainCamera.update();
};

lua["getCameraYaw"] = [this, &engine]() {
    return engine.mainCamera.yaw;
};

lua["setCameraYaw"] = [this, &engine](float yaw) {
    engine.mainCamera.yaw = yaw;
    engine.mainCamera.update();
};

```

```

lua["getCameraPitch"] = [this, &engine]() {
    return engine.mainCamera.pitch;
};

lua["setCameraPitch"] = [this, &engine](float pitch) {
    engine.mainCamera.pitch = pitch;
    engine.mainCamera.update();
};

lua["loadLevel"] = [this, &engine](const std::string&
name) {
    for (auto& [id, obj] : engine.levels) {
        if (obj.name == name) {
            engine.loadLevel(id);
            return;
        }
    }
};

lua["getStringValue"] = [this, &engine](const std::string&
name) -> std::string {
    if (engine.currentProject.scriptId.empty()) {
        return "";
    }
    return engine.currentProject.script-
>lua[name].get_or<std::string>("");
};

lua["setStringValue"] = [this, &engine](const std::string&
name, const std::string& value) {
    if (!engine.currentProject.scriptId.empty()) {
        engine.currentProject.script->lua[name] =
value;
    }
};

lua["getFloatValue"] = [this, &engine](const std::string&
name) -> float {
    if (engine.currentProject.scriptId.empty()) {
        return 0.0f;
    }
    return engine.currentProject.script-
>lua[name].get_or(0.0f);
};

```

```

};

lua["setFloatValue"] = [this, &engine](const std::string&
name, float value) {
    if (engine.currentProject.scriptId.empty()) {
        return;
    }
    engine.currentProject.script->lua[name] = value;
};

lua["getIntValue"] = [this, &engine](const std::string&
name) -> int {
    if (engine.currentProject.scriptId.empty()) {
        return 0;
    }
    return engine.currentProject.script-
>lua[name].get_or(0);
};

lua["setIntValue"] = [this, &engine](const std::string&
name, int value) {
    if (engine.currentProject.scriptId.empty()) {
        return;
    }
    engine.currentProject.script->lua[name] = value;
};

lua["getBoolValue"] = [this, &engine](const std::string&
name) -> bool {
    if (engine.currentProject.scriptId.empty()) {
        return false;
    }
    return engine.currentProject.script-
>lua[name].get_or(false);
};

lua["setBoolValue"] = [this, &engine](const std::string&
name, bool value) {
    if (engine.currentProject.scriptId.empty()) {
        return;
    }
    engine.currentProject.script->lua[name] = value;
};

```

```

        lua["destroyObject"] = [this, &engine](const std::string&
id) {
            auto objIt = engine.currentLevel->loadedScenes.find(id);
            if (objIt != engine.currentLevel->loadedScenes.end())
            {
                objIt->second.isVisible = false;
                deleteObject(objIt->second);
            }
        };

        lua["KeyTable"] = lua.create_table_with("A", 1, "B", 2,
"C", 3, "D", 4, "E", 5, "F", 6, "G", 7, "H", 8, "I", 9,

        "J", 10, "K", 11, "L", 12, "M", 13, "N", 14, "O", 15, "P", 16,
"Q", 17,

        "R", 18, "S", 19, "T", 20, "U", 21, "V", 22, "W", 23, "X", 24,
"Y", 25,

        "Z", 26, "Space", 41, "Up", 42, "Down", 43, "Left", 44,
"Right", 45);

        isLoading = true;
        isActive = true;
    } catch (const sol::error& e) {
        BK_ERROR(std::string(e.what()));
    }
    return isLoading;
}

std::string createName(const std::unordered_map<std::string, Object>&
objects, const std::string& name) {
    auto findName = [&objects](const std::string& name) {
        for (auto& [_ , obj] : objects) {
            if (obj.name == name) {
                return true;
            }
        }
        return false;
    };
    std::string newName = name;
    int i = 1;
    while (findName(newName)) {

```

```

        newName = name + std::to_string(i);
        i++;
    }
    return newName;
}

```

```

Object::Object(const Object& other) {
    scene      = other.scene;
    position    = other.position;
    scale       = other.scale;
    rotation    = other.rotation;
    if (scene) {
        deleteObject(*this);
        bodyId = copyBody(other);
    }
    name        = other.name;
    bounds      = other.bounds;
    isVisible    = other.isVisible;
    isDynamic    = other.isDynamic;
    isActive     = other.isActive;
    gltfuuid     = other.gltfuuid;
    customTypes  = other.customTypes;
    scriptId     = other.scriptId;
    isScriptActive = other.isScriptActive;
    if (!scriptId.empty() && other.script) {
        script      = Script();
        script->path = other.script->path;
        script->name = other.script->name;
    }
}

```

```

Object& Object::operator=(const Object& other) {
    if (this == &other) {
        return *this;
    }
    scene      = other.scene;
    position    = other.position;
    scale       = other.scale;
    rotation    = other.rotation;
    if (scene) {
        deleteObject(*this);
        bodyId = copyBody(other);
    }
}

```

```

        name          = other.name;
        bounds        = other.bounds;
        isVisible     = other.isVisible;
        isDynamic     = other.isDynamic;
        isActive      = other.isActive;
        gltfuuid      = other.gltfuuid;
        customTypes   = other.customTypes;
        scriptId      = other.scriptId;
        isScriptActive = other.isScriptActive;
        if (!scriptId.empty() && other.script) {
            script      = Script();
            script->path = other.script->path;
            script->name = other.script->name;
        }
        return *this;
    }

void createPhysicsBody(Object& object) {
    const auto& extents    = object.bounds.extents;
    const auto  position   = RVec3(object.position.x,
object.position.y, object.position.z);
    const auto  rotation   =
glm::quat(glm::radians(object.rotation));
    ObjectLayer layer      = object.isDynamic ? Layers::MOVING :
Layers::NON_MOVING;
    EMotionType motionType = object.isDynamic ?
EMotionType::Dynamic : EMotionType::Static;
    object.linearVelocity  = glm::vec3(0.0f);
    BodyCreationSettings settings(new
BoxShapeSettings(Vec3(extents.x, extents.y, extents.z)), position,
                                Quat(rotation.x,
rotation.y, rotation.z, rotation.w), motionType, layer);
    EActivation      activation = object.isDynamic ?
EActivation::Activate : EActivation::DontActivate;
    object.bodyId    = body_interface-
>CreateAndAddBody(settings, activation);
}

void BlockEngine::createObject(std::string id) {
    Object      object;
    auto        sceneObject = loadedGLTFs[id];
    std::filesystem::path path    = sceneObject->path;
    object.name      = path.stem().string();
    object.scene     = sceneObject;
}

```

```

std::vector<Bounds> bounds;
for (auto& node : sceneObject->topNodes) {
    MeshNode* meshNode = dynamic_cast<MeshNode*>(node.get());
    if (meshNode) {
        for (auto& surface : meshNode->mesh->surfaces) {
            bounds.push_back(Bounds{surface.bounds.origin,
surface.bounds.sphereRadius,

            surface.bounds.extents * meshNode->scale});
        }
    }
}
glm::vec3 minCorner(std::numeric_limits<float>::max());
glm::vec3 maxCorner(std::numeric_limits<float>::lowest());
for (const auto& bound : bounds) {
    glm::vec3 boundMin = bound.origin - bound.extents;
    glm::vec3 boundMax = bound.origin + bound.extents;
    minCorner = glm::min(minCorner, boundMin);
    maxCorner = glm::max(maxCorner, boundMax);
}
glm::vec3 newCenter = (minCorner + maxCorner) * 0.5f;
glm::vec3 newExtents = (maxCorner - minCorner) * 0.5f;
float newSphereRadius = glm::length(newExtents);
object.bounds = Bounds{newCenter, newSphereRadius,
newExtents};
object.extents = newExtents;
const auto& extents = object.bounds.extents;
const auto rotation =
glm::quat(glm::radians(object.rotation));
const uuids::uuid uuid = uuids::uuid_system_generator{}();
object.setName(createName(currentLevel->loadedScenes,
object.name));
object.gltfuuid = id;
currentLevel->loadedScenes[uuids::to_string(uuid)] = object;
}

void loadObject(BlockEngine* eng, const std::string_view sw) {
    std::filesystem::current_path(eng->currentProject.path);
    TRS trs;
    auto loadFile = loadGltf(eng, sw, trs);
    if (loadFile.has_value()) {
        auto sceneObject =
loadFile.value();
        std::filesystem::path path = sw;

```

```

        const uuids::uuid      gltfUUID          =
uuids::uuid_system_generator{}();
        sceneObject->uuid      =
uuids::to_string(gltfUUID);
        eng->loadedGLTFs[uuids::to_string(gltfUUID)] =
sceneObject;
        std::filesystem::path newPath = eng->currentProject.path /
std::filesystem::path("assets") / path.filename();
        if (!std::filesystem::exists(eng->currentProject.path /
std::filesystem::path("assets"))) {
            std::filesystem::create_directory(eng-
>currentProject.path / std::filesystem::path("assets"));
        }
        if (!std::filesystem::exists(newPath)) {
            std::filesystem::copy(path, newPath);
        }
        sceneObject->name = newPath.stem().string();
        sceneObject->path = (std::filesystem::path("assets") /
path.filename()).string();
        eng-
>currentProject.loadedObjects[uuids::to_string(gltfUUID)] =
        (std::filesystem::path("assets") /
path.filename()).string();
        eng->createObject(uuids::to_string(gltfUUID));
    }
}

void addBodyToObject(Object& object) {
    const auto&      extents      = object.bounds.extents;
    const auto      position      = RVec3(object.position.x,
object.position.y, object.position.z);
    const auto      rotation      =
glm::quat(glm::radians(object.rotation));
    ObjectLayer      layer        = object.isDynamic ?
Layers::MOVING : Layers::NON_MOVING;
    EMotionType      motionType   = object.isDynamic ?
EMotionType::Dynamic : EMotionType::Static;
    BodyCreationSettings settings(new
BoxShapeSettings(Vec3(extents.x, extents.y, extents.z)), position,
Quat(rotation.x,
rotation.y, rotation.z, rotation.w), motionType, layer);
}

void loadGLTFObjects(BlockEngine* eng) {

```

```

    for (auto& [uuid, path] : eng->currentProject.loadedObjects) {
        TRS trs;
        auto loadFile = loadGltf(eng, path, trs);
        if (loadFile.has_value()) {
            auto sceneObject = loadFile.value();
            sceneObject->uuid = uuid;
            sceneObject->name =
std::filesystem::path(path).stem().string();
            sceneObject->path = path;
            eng->loadedGLTFs[uuid] = sceneObject;
            for (auto& [_ , level] : eng->levels) {
                for (auto& [_ , obj] : level.loadedScenes) {
                    if (obj.gltfuuid == uuid) {
                        obj.scene = sceneObject;
                    }
                }
            }
        }
    }
}

void setObjectMotionType(Object& object, EMotionType motionType) {
    ObjectLayer layer = motionType == EMotionType::Dynamic ?
Layers::MOVING : Layers::NON_MOVING;
    if (motionType == EMotionType::Dynamic) {
        body_interface->SetObjectLayer(object.bodyId, layer);
        body_interface->ActivateBody(object.bodyId);
    } else {
        body_interface->SetObjectLayer(object.bodyId, layer);
        body_interface->DeactivateBody(object.bodyId);
    }
}

void setBodyActivationState(Object& object, EActivation activation) {
    if (activation == EActivation::Activate) {
        body_interface->ActivateBody(object.bodyId);
    } else {
        body_interface->DeactivateBody(object.bodyId);
    }
    body_interface->SetLinearVelocity(object.bodyId, Vec3(0.0f, -
5.0f, 0.0f));
}

void Object::updateTransforms() {

```

```

        if (position != prevPosition) {
            body_interface->SetPosition(bodyId, Vec3(position.x,
position.y, position.z), EActivation::DontActivate);
            prevPosition = position;
        }
        if (scale != prevScale) {
            bounds.extents = extents * scale;
            body_interface->SetShape(
                bodyId, BoxShapeSettings(Vec3(bounds.extents.x,
bounds.extents.y, bounds.extents.z)).Create(), true,
                EActivation::DontActivate);
            prevScale = scale;
        }
        if (rotation != prevRotation) {
            auto rotationQ = glm::quat(glm::radians(rotation));
            body_interface->SetRotation(bodyId, Quat(rotationQ.x,
rotationQ.y, rotationQ.z, rotationQ.w),

            EActivation::DontActivate);
            prevRotation = rotation;
        }
        if (linearVelocity != prevLinearVelocity) {
            body_interface->SetLinearVelocity(bodyId,
Vec3(linearVelocity.x, linearVelocity.y, linearVelocity.z));
            prevLinearVelocity = linearVelocity;
        }
    }
}

void loadAudio(BlockEngine* eng, const std::string& path) {
    std::filesystem::current_path(eng->currentProject.path);
    std::string name = std::filesystem::path(path).stem().string();
    ma_sound sound;
    AudioFile audioFile{.id =
uuids::to_string(uuids::uuid_system_generator{}()), .name = name};
    eng->audioFiles[audioFile.id] = audioFile;
    ma_result result = ma_sound_init_from_file(&eng-
>audio_engine, path.c_str(),

        MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_DECODE |

        MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_STREAM,

        NULL, NULL, &eng->audioFiles[audioFile.id].sound);
    if (result != MA_SUCCESS) {

```

```

        BK_ERROR("Failed to load audio file: {}", path);
        return;
    }
    std::filesystem::path currentPath = path;
    std::filesystem::path newPath = eng->currentProject.path /
std::filesystem::path("assets") / currentPath.filename();
    if (!std::filesystem::exists(eng->currentProject.path /
std::filesystem::path("assets"))) {
        std::filesystem::create_directory(eng->currentProject.path
/ std::filesystem::path("assets"));
    }
    if (!std::filesystem::exists(newPath)) {
        std::filesystem::copy(path, newPath);
    }
    eng->audioFiles[audioFile.id].path =
(std::filesystem::path("assets") / currentPath.filename()).string();
}

void addSound(BlockEngine* eng, AudioFile& audioFile) {
    ma_result result = ma_sound_init_from_file(&eng->audio_engine,
audioFile.path.c_str(),

MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_DECODE |

MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_STREAM,

NULL,

NULL, &audioFile.sound);
    if (result != MA_SUCCESS) {
        BK_ERROR("Failed to load audio file: {}", audioFile.path);
        return;
    }
}

void renderUiElements(BlockEngine* eng) {
    for (auto& [_ , element] : eng->currentLevel->uiElements) {
        if (!element.parent) {
            element.display();
        }
    }
}

void UiElement::display() {
    if (!isVisible) {
        return;
    }
}

```

```

    }
    switch (type) {
    case UiType::Text:    displayText(); break;
    case UiType::Button: displayButton(); break;
    case UiType::Window: displayWindow(); break;
    default:              break;
    }
}

ImVec2 getCenterPosition() {
    return ImVec2(ImGui::GetIO().DisplaySize.x / 2,
ImGui::GetIO().DisplaySize.y / 2);
}

void UiElement::displayWindow() {
    if (useCenterPosition) {
        ImVec2 center = getCenterPosition();
        ImGui::SetNextWindowPos(ImVec2(center.x + position.x,
center.y + position.y), ImGuiCond_Always);
    } else {
        ImGui::SetNextWindowPos(ImVec2(position.x, position.y),
ImGuiCond_Always);
    }
    ImGui::SetNextWindowSize(ImVec2(size.x, size.y),
ImGuiCond_Always);
    ImGuiWindowFlags window_flags = ImGuiWindowFlags_NoResize |
ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoScrollbar |

    ImGuiWindowFlags_NoScrollWithMouse |
ImGuiWindowFlags_NoCollapse |

    ImGuiWindowFlags_AlwaysAutoResize |
ImGuiWindowFlags_NoNavInputs |

    ImGuiWindowFlags_NoNavFocus;
    if (!useBackground) {
        window_flags |= ImGuiWindowFlags_NoBackground;
    }
    if (!showTitleBar) {
        window_flags |= ImGuiWindowFlags_NoTitleBar;
    }
    ImGui::Begin(name.c_str(), nullptr, window_flags);
    for (auto& [_ , child] : children) {
        child.display();
    }
}

```

```

    }
    ImGui::End();
}

void UiElement::displayButton() {
    if (!parent) {
        return;
    }
    ImVec2 buttonSize = ImVec2(size.x, size.y);
    ImGui::SetWindowFontScale(fontSize);
    if (parent->get().useCenterPosition) {
        ImVec2 center = getCenterPosition();
        ImGui::SetCursorScreenPos(ImVec2(center.x + position.x +
parent->get().position.x + position.x,
center.y +
parent->get().position.y + position.y));
    } else {
        ImGui::SetCursorScreenPos(ImVec2(parent->get().position.x
+ position.x, parent->get().position.y + position.y));
    }
    if (ImGui::Button(text.c_str(), buttonSize)) {
        pressed = true;
    }
    ImGui::SetWindowFontScale(1.0f);
}

void UiElement::displayText() {
    if (!parent) {
        return;
    }
    ImGui::SetWindowFontScale(fontSize);
    if (parent->get().useCenterPosition) {
        ImVec2 center = getCenterPosition();
        ImGui::SetCursorScreenPos(ImVec2(center.x + position.x +
parent->get().position.x + position.x,
center.y +
parent->get().position.y + position.y));
    } else {
        ImGui::SetCursorScreenPos(ImVec2(parent->get().position.x
+ position.x, parent->get().position.y + position.y));
    }
    ImGui::Text(text.c_str());
    ImGui::SetWindowFontScale(1.0f);
}

```

```

std::string createUiElementName(BlockEngine* eng, const std::string&
name) {
    auto findName = [eng](const std::string& name) {
        for (auto& [_ , elem] : eng->currentLevel->uiElements) {
            if (elem.name == name) {
                return true;
            }
            for (auto& [_ , child] : elem.children) {
                if (child.name == name) {
                    return true;
                }
            }
        }
        return false;
    };
    std::string newName = name;
    int i = 1;
    while (findName(newName)) {
        newName = name + std::to_string(i);
        i++;
    }
    return newName;
}

void UiElement::rename(BlockEngine* eng) {
    if (!tempName.empty()) {
        tempName = createUiElementName(eng, tempName);
        name = tempName;
    }
}

void UiElement::changeText() {
    if (!tempText.empty()) {
        text = tempText;
    }
}

UiElement createUiElement(BlockEngine* eng, const std::string& name)
{
    const std::string uuid =
uuids::to_string(uuids::uuid_system_generator{}());
    UiElement element;
    element.name = createUiElementName(eng, name);
}

```

```

        element.text      = name;
        element.tempName = name;
        element.tempText = name;
        element.id        = uuid;
        return element;
    }
constexpr bool bUseValidationLayers = true;
using namespace std;
BlockEngine* loadedEngine = nullptr;
BlockEngine& BlockEngine::Get() {
    return *loadedEngine;
}
void BlockEngine::init() {
    assert(loadedEngine == nullptr);
    loadedEngine = this;
    Log::init();
    engineExecPath = std::filesystem::current_path().string();
    SDL_Init(SDL_INIT_VIDEO);
    SDL_WindowFlags window_flags =
(SDL_WindowFlags)(SDL_WINDOW_VULKAN | SDL_WINDOW_RESIZABLE);

    _window = SDL_CreateWindow("Block Engine",
SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
_windowExtent.width,
                                _windowExtent.height,
window_flags);
    init_vulkan();
    init_swapchain();
    init_commands();
    init_sync_structures();
    init_descriptors();
    init_pipelines();
    init_default_data();
    init_renderables();
    init_imgui();
    _isInitialized = true;
    mainCamera.velocity = glm::vec3(0.f);
    mainCamera.position = glm::vec3(0.0f, 50.0f, 100.0f);
    mainCamera.pitch    = 0;
    mainCamera.yaw      = 0;
    ma_result result    = ma_engine_init(nullptr, &audio_engine);
    assert(result == MA_SUCCESS);
}

```

```

void BlockEngine::init_default_data() {
    std::array<Vertex, 4> rect_vertices;
    rect_vertices[0].position = {0.5, -0.5, 0};
    rect_vertices[1].position = {0.5, 0.5, 0};
    rect_vertices[2].position = {-0.5, -0.5, 0};
    rect_vertices[3].position = {-0.5, 0.5, 0};
    rect_vertices[0].color     = {0, 0, 0, 1};
    rect_vertices[1].color     = {0.5, 0.5, 0.5, 1};
    rect_vertices[2].color     = {1, 0, 0, 1};
    rect_vertices[3].color     = {0, 1, 0, 1};
    rect_vertices[0].uv_x      = 1;
    rect_vertices[0].uv_y      = 0;
    rect_vertices[1].uv_x      = 0;
    rect_vertices[1].uv_y      = 0;
    rect_vertices[2].uv_x      = 1;
    rect_vertices[2].uv_y      = 1;
    rect_vertices[3].uv_x      = 0;
    rect_vertices[3].uv_y      = 1;
    std::array<uint32_t, 6> rect_indices;
    rect_indices[0] = 0;
    rect_indices[1] = 1;
    rect_indices[2] = 2;
    rect_indices[3] = 2;
    rect_indices[4] = 1;
    rect_indices[5] = 3;
    rectangle        = uploadMesh(rect_indices, rect_vertices);
    uint32_t white    = glm::packUnorm4x8(glm::vec4(1, 1, 1, 1));
    _whiteImage =
        create_image((void*)&white, VkExtent3D{1, 1, 1},
VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_USAGE_SAMPLED_BIT);
    uint32_t grey     = glm::packUnorm4x8(glm::vec4(0.66f, 0.66f,
0.66f, 1));
    _greyImage = create_image((void*)&grey, VkExtent3D{1, 1, 1},
VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_USAGE_SAMPLED_BIT);
    uint32_t black    = glm::packUnorm4x8(glm::vec4(0, 0, 0, 0));
    _blackImage =
        create_image((void*)&black, VkExtent3D{1, 1, 1},
VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_USAGE_SAMPLED_BIT);
    uint32_t          magenta =
glm::packUnorm4x8(glm::vec4(1, 0, 1, 1));
    std::array<uint32_t, 16 * 16> pixels;
    for (int x = 0; x < 16; x++) {
        for (int y = 0; y < 16; y++) {
            pixels[y * 16 + x] = ((x % 2) ^ (y % 2)) ? magenta :

```

```

black;
    }
}
_errorCheckerboardImage =
    create_image(pixels.data(), VkExtent3D{16, 16, 1},
VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_USAGE_SAMPLED_BIT);
    VkSamplerCreateInfo sampl = {.sType =
VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO};
    sampl.magFilter = VK_FILTER_NEAREST;
    sampl.minFilter = VK_FILTER_NEAREST;
    vkCreateSampler(_device, &sampl, nullptr,
&_defaultSamplerNearest);
    sampl.magFilter = VK_FILTER_LINEAR;
    sampl.minFilter = VK_FILTER_LINEAR;
    vkCreateSampler(_device, &sampl, nullptr,
&_defaultSamplerLinear);
}

void BlockEngine::cleanup() {
    if (_isInitialized) {
        vkDeviceWaitIdle(_device);
        if (currentLevel != nullptr) {
            currentLevel->loadedScenes.clear();
        }
        for (auto& frame : _frames) {
            frame._deletionQueue.flush();
        }
        _mainDeletionQueue.flush();
        destroy_swapchain();
        vkDestroySurfaceKHR(_instance, _surface, nullptr);
        vkDestroyDevice(_device, nullptr);
        vkb::destroy_debug_utils_messenger(_instance,
_debug_messenger);
        vkDestroyInstance(_instance, nullptr);
        for (auto& [_ , obj] : audioFiles) {
            obj.destroy();
        }
        SDL_DestroyWindow(_window);
    }
}

void BlockEngine::init_background_pipelines() {
    VkPipelineLayoutCreateInfo computeLayout{};
    computeLayout.sType =

```

```

VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    computeLayout.pNext          = nullptr;
    computeLayout.pSetLayouts    = &_amp;drawImageDescriptorLayout;
    computeLayout.setLayoutCount = 1;
    VkPushConstantRange pushConstant{};
    pushConstant.offset          = 0;
    pushConstant.size            =
sizeof(ComputePushConstants);
    pushConstant.stageFlags      =
VK_SHADER_STAGE_COMPUTE_BIT;
    computeLayout.pPushConstantRanges = &pushConstant;
    computeLayout.pushConstantRangeCount = 1;
    VK_CHECK(vkCreatePipelineLayout(_device, &computeLayout,
nullptr, &_gradientPipelineLayout));
    VkShaderModule gradientShader;
    if (!vkutil::load_shader_module("effects.spv", _device,
&gradientShader)) {
        BK_ERROR("Error when building the compute shader");
    }
    VkShaderModule skyShader;
    if (!vkutil::load_shader_module("effects2.spv", _device,
&skyShader)) {
        BK_ERROR("Error when building the compute shader");
    }
    VkPipelineShaderStageCreateInfo stageinfo{};
    stageinfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    stageinfo.pNext = nullptr;
    stageinfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
    stageinfo.module = gradientShader;
    stageinfo.pName = "main";
    VkComputePipelineCreateInfo computePipelineCreateInfo{};
    computePipelineCreateInfo.sType =
VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
    computePipelineCreateInfo.pNext = nullptr;
    computePipelineCreateInfo.layout = _gradientPipelineLayout;
    computePipelineCreateInfo.stage = stageinfo;
    ComputeEffect gradient;
    gradient.layout = _gradientPipelineLayout;
    gradient.name = "gradient";
    gradient.data = {};
    gradient.data.data1 = glm::vec4(0, 0, 0, 1);
    gradient.data.data2 = glm::vec4(0, 0.3f, 0.6f, 1);
    VK_CHECK(

```

```

        vkCreateComputePipelines(_device, VK_NULL_HANDLE, 1,
&computePipelineCreateInfo, nullptr, &gradient.pipeline));
        computePipelineCreateInfo.stage.module = skyShader;
        ComputeEffect sky;
        sky.layout      = _gradientPipelineLayout;
        sky.name        = "sky";
        sky.data        = {};
        sky.data.data1 = glm::vec4(0.1, 0.2, 0.4, 0.97);
        VK_CHECK(vkCreateComputePipelines(_device, VK_NULL_HANDLE, 1,
&computePipelineCreateInfo, nullptr, &sky.pipeline));
        backgroundEffects.push_back(gradient);
        backgroundEffects.push_back(sky);
        vkDestroyShaderModule(_device, gradientShader, nullptr);
        vkDestroyShaderModule(_device, skyShader, nullptr);
    }

void BlockEngine::draw_main(VkCommandBuffer cmd) {
    ComputeEffect& effect =
backgroundEffects[currentBackgroundEffect];
        vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_COMPUTE,
effect.pipeline);
        vkCmdBindDescriptorSets(cmd, VK_PIPELINE_BIND_POINT_COMPUTE,
_gradientPipelineLayout, 0, 1, &_drawImageDescriptors,
                                0, nullptr);
        vkCmdPushConstants(cmd, _gradientPipelineLayout,
VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(ComputePushConstants),
                                &effect.data);
        vkCmdDispatch(cmd, std::ceil(_drawExtent.width / 16.0),
std::ceil(_drawExtent.height / 16.0), 1);
        VkRenderingAttachmentInfo colorAttachment =
attachment_info(_drawImage.imageView, nullptr,
VK_IMAGE_LAYOUT_GENERAL);
        VkRenderingAttachmentInfo depthAttachment =
depth_attachment_info(_depthImage.imageView,
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL);
        VkRenderingInfo renderInfo = rendering_info(_drawExtent,
&colorAttachment, &depthAttachment);
        vkCmdBeginRendering(cmd, &renderInfo);
        auto start = std::chrono::system_clock::now();
        draw_geometry(cmd);
        auto end = std::chrono::system_clock::now();
        auto elapsed =
std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        stats.mesh_draw_time = elapsed.count() / 1000.f;

```

```

        vkCmdEndRendering(cmd);
    }

void BlockEngine::draw_imgui(VkCommandBuffer cmd, VkImageView
targetImageView) {
    VkRenderingAttachmentInfo colorAttachment =
attachment_info(targetImageView, nullptr, VK_IMAGE_LAYOUT_GENERAL);
    VkRenderingInfo          renderInfo      =
rendering_info(_windowExtent, &colorAttachment, nullptr);
    vkCmdBeginRendering(cmd, &renderInfo);
    ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), cmd);
    vkCmdEndRendering(cmd);
}

void BlockEngine::draw() {
    VK_CHECK(vkWaitForFences(_device, 1,
&get_current_frame()._renderFence, true, 1'000'000'000));
    get_current_frame()._deletionQueue.flush();
    get_current_frame()._frameDescriptors.clear_pools(_device);
    uint32_t swapchainImageIndex;
    VkResult e = vkAcquireNextImageKHR(_device, _swapchain,
1'000'000'000, get_current_frame()._swapchainSemaphore,
                                nullptr,
&swapchainImageIndex);
    if (e == VK_ERROR_OUT_OF_DATE_KHR) {
        resize_requested = true;
        return;
    }
    _drawExtent.height = std::min(_swapchainExtent.height,
_drawImage.imageExtent.height) * 1.f;
    _drawExtent.width  = std::min(_swapchainExtent.width,
_drawImage.imageExtent.width) * 1.f;
    VK_CHECK(vkResetFences(_device, 1,
&get_current_frame()._renderFence));
    VK_CHECK(vkResetCommandBuffer(get_current_frame()._mainCommandB
uffer, 0));
    VkCommandBuffer          cmd          =
get_current_frame()._mainCommandBuffer;
    VkCommandBufferBeginInfo cmdBeginInfo =
command_buffer_begin_info(VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT
);
    VK_CHECK(vkBeginCommandBuffer(cmd, &cmdBeginInfo));
    transitionImage(cmd, _drawImage.image,
VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_GENERAL);

```

```

        transitionImage(cmd, _depthImage.image,
VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL);
        draw_main(cmd);
        transitionImage(cmd, _drawImage.image, VK_IMAGE_LAYOUT_GENERAL,
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL);
        transitionImage(cmd, _swapchainImages[swapchainImageIndex],
VK_IMAGE_LAYOUT_UNDEFINED,
                                VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);

        VkExtent2D extent;
        extent.height = _windowExtent.height;
        extent.width  = _windowExtent.width;
        copyImageToImage(cmd, _drawImage.image,
_swapchainImages[swapchainImageIndex], _drawExtent,
_swapchainExtent);
        transitionImage(cmd, _swapchainImages[swapchainImageIndex],
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
                                VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL);
        draw_imgui(cmd, _swapchainImageViews[swapchainImageIndex]);
        transitionImage(cmd, _swapchainImages[swapchainImageIndex],
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
                                VK_IMAGE_LAYOUT_PRESENT_SRC_KHR);

        VK_CHECK(vkEndCommandBuffer(cmd));
        VkCommandBufferSubmitInfo cmdinfo =
command_buffer_submit_info(cmd);
        VkSemaphoreSubmitInfo waitInfo =
semaphore_submit_info(VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT
_KHR,

                                get_current_frame()._swapchainSemaphore);
        VkSemaphoreSubmitInfo signalInfo =

        semaphore_submit_info(VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT,
get_current_frame()._renderSemaphore);
        VkSubmitInfo2 submit = submit_info(&cmdinfo, &signalInfo,
&waitInfo);
        VK_CHECK(vkQueueSubmit2(_graphicsQueue, 1, &submit,
get_current_frame()._renderFence));
        VkPresentInfoKHR presentInfo = present_info();
        presentInfo.pSwapchains = &_swapchain;
        presentInfo.swapchainCount = 1;
        presentInfo.pWaitSemaphores =
&get_current_frame()._renderSemaphore;
        presentInfo.waitSemaphoreCount = 1;
        presentInfo.pImageIndices = &swapchainImageIndex;

```

```

        VkResult presentResult =
vkQueuePresentKHR(_graphicsQueue, &presentInfo);
        if (e == VK_ERROR_OUT_OF_DATE_KHR) {
            resize_requested = true;
            return;
        }
        _frameNumber++;
    }

bool is_visible(const RenderObject& obj, const glm::mat4& viewproj) {
    std::array<glm::vec3, 8> corners{
        glm::vec3{1, 1, 1},
        glm::vec3{1, 1, -1},
        glm::vec3{1, -1, 1},
        glm::vec3{1, -1, -1},
        glm::vec3{-1, 1, 1},
        glm::vec3{-1, 1, -1},
        glm::vec3{-1, -1, 1},
        glm::vec3{-1, -1, -1},
    };
    glm::mat4 matrix = viewproj * obj.transform;
    glm::vec3 min = {1.5, 1.5, 1.5};
    glm::vec3 max = {-1.5, -1.5, -1.5};
    for (int c = 0; c < 8; c++) {
        glm::vec4 v = matrix * glm::vec4(obj.bounds.origin +
(corners[c] * obj.bounds.extents), 1.f);
        v.x = v.x / v.w;
        v.y = v.y / v.w;
        v.z = v.z / v.w;
        min = glm::min(glm::vec3{v.x, v.y, v.z}, min);
        max = glm::max(glm::vec3{v.x, v.y, v.z}, max);
    }
    if (min.z > 1.f || max.z < 0.f || min.x > 1.f || max.x < -1.f
|| min.y > 1.f || max.y < -1.f) {
        return false;
    } else {
        return true;
    }
}

void BlockEngine::searchGameFiles() {
    auto currentPath = std::filesystem::current_path();
    for (const auto& entry :
std::filesystem::directory_iterator(currentPath)) {

```

```

        if (entry.path().filename().string() == "config.json") {
            isGameInstance = true;
            return;
        }
    }
}

void BlockEngine::draw_geometry(VkCommandBuffer cmd) {
    std::vector<uint32_t> opaque_draws;
    opaque_draws.reserve(drawCommands.OpaqueSurfaces.size());

    for (int i = 0; i < drawCommands.OpaqueSurfaces.size(); i++) {
        if (is_visible(drawCommands.OpaqueSurfaces[i],
sceneData.viewproj)) {
            opaque_draws.push_back(i);
        }
    }

    AllocatedBuffer gpuSceneDataBuffer =
        create_buffer(sizeof(GPUSceneData),
VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, VMA_MEMORY_USAGE_CPU_TO_GPU);
    get_current_frame()._deletionQueue.push_function([=, this]() {
destroy_buffer(gpuSceneDataBuffer); });
    GPUSceneData* sceneUniformData =
(GPUSceneData*)gpuSceneDataBuffer.allocation->GetMappedData();
    *sceneUniformData = sceneData;
    VkDescriptorSetVariableDescriptorCountAllocateInfo
allocArrayInfo{
        .sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_I
NFO,
        .pNext = nullptr};
    uint32_t descriptorCounts = texCache.Cache.size();
    allocArrayInfo.pDescriptorCounts = &descriptorCounts;
    allocArrayInfo.descriptorSetCount = 1;
    VkDescriptorSet globalDescriptor =
        get_current_frame()._frameDescriptors.allocate(_device,
_gpuSceneDataDescriptorLayout, &allocArrayInfo);
    DescriptorWriter writer;
    writer.writeBuffer(0, gpuSceneDataBuffer.buffer,
sizeof(GPUSceneData), 0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER);
    if (texCache.Cache.size() > 0) {
        VkWriteDescriptorSet arraySet{.sType =

```

```

VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET};
    arraySet.descriptorCount = texCache.Cache.size();
    arraySet.dstArrayElement = 0;
    arraySet.descriptorType =
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    arraySet.dstBinding = 1;
    arraySet.pImageInfo = texCache.Cache.data();
    writer.writes.push_back(arraySet);
}
writer.update_set(_device, globalDescriptor);
MaterialPipeline* lastPipeline = nullptr;
MaterialInstance* lastMaterial = nullptr;
VkBuffer lastIndexBuffer = VK_NULL_HANDLE;
auto draw = [&](const RenderObject& r)
{
    if (r.material != lastMaterial) {
        lastMaterial = r.material;
        if (r.material->pipeline != lastPipeline) {
            lastPipeline = r.material->pipeline;
            vkCmdBindPipeline(cmd,
VK_PIPELINE_BIND_POINT_GRAPHICS, r.material->pipeline->pipeline);
            vkCmdBindDescriptorSets(cmd,
VK_PIPELINE_BIND_POINT_GRAPHICS, r.material->pipeline->layout, 0, 1,

                &globalDescriptor, 0, nullptr);

            VkViewport viewport = {};
            viewport.x = 0;
            viewport.y = 0;
            viewport.width = (float)_drawExtent.width;
            viewport.height = (float)_drawExtent.height;
            viewport.minDepth = 0.f;
            viewport.maxDepth = 1.f;
            vkCmdSetViewport(cmd, 0, 1, &viewport);
            VkRect2D scissor = {};
            scissor.offset.x = 0;
            scissor.offset.y = 0;
            scissor.extent.width = _drawExtent.width;
            scissor.extent.height = _drawExtent.height;
            vkCmdSetScissor(cmd, 0, 1, &scissor);
        }

        vkCmdBindDescriptorSets(cmd,
VK_PIPELINE_BIND_POINT_GRAPHICS, r.material->pipeline->layout, 1, 1,

```

```

        &r.material->materialSet, 0, nullptr);
    }
    if (r.indexBuffer != lastIndexBuffer) {
        lastIndexBuffer = r.indexBuffer;
        vkCmdBindIndexBuffer(cmd, r.indexBuffer, 0,
VK_INDEX_TYPE_UINT32);
    }
    GPUdrawPushConstants push_constants;
    push_constants.worldMatrix = r.transform;
    push_constants.vertexBuffer = r.vertexBufferAddress;

    vkCmdPushConstants(cmd, r.material->pipeline->layout,
VK_SHADER_STAGE_VERTEX_BIT, 0,

sizeof(GPUdrawPushConstants), &push_constants);

    stats.drawcall_count++;
    stats.triangle_count += r.indexCount / 3;
    vkCmdDrawIndexed(cmd, r.indexCount, 1, r.firstIndex, 0, 0);
};
stats.drawcall_count = 0;
stats.triangle_count = 0;
for (auto& r : opaque_draws) {
    draw(drawCommands.OpaqueSurfaces[r]);
}

for (auto& r : drawCommands.TransparentSurfaces) {
    draw(r);
}
drawCommands.OpaqueSurfaces.clear();
drawCommands.TransparentSurfaces.clear();
}

namespace glm {
void to_json(json& j, const vec3 obj) {
    j = json{
        {"x", obj.x},
        {"y", obj.y},
        {"z", obj.z}
    };
}

void from_json(const json& j, vec3& obj) {

```

```

        j.at("x").get_to(obj.x);
        j.at("y").get_to(obj.y);
        j.at("z").get_to(obj.z);
    }

    void to_json(json& j, const vec2 obj) {
        j = json{
            {"x", obj.x},
            {"y", obj.y}
        };
    }

    void from_json(const json& j, vec2& obj) {
        j.at("x").get_to(obj.x);
        j.at("y").get_to(obj.y);
    }
} // namespace glm

    void to_json(json& j, const Bounds& obj) {
        j = json{
            {"origin",      obj.origin      },
            {"extents",     obj.extents     },
            {"sphereRadius", obj.sphereRadius}
        };
    }

    void from_json(const json& j, Bounds& obj) {
        j.at("origin").get_to(obj.origin);
        j.at("extents").get_to(obj.extents);
        j.at("sphereRadius").get_to(obj.sphereRadius);
    }

    void to_json(json& j, const Object& obj) {
        j = json{
            {"name",          obj.name          },
            {"isVisible",     obj.isVisible     },
            {"isDynamic",     obj.isDynamic     },
            {"isActive",      obj.isActive      },
            {"bounds",        obj.bounds        },
            {"position",      obj.position      },
            {"rotation",      obj.rotation      },
            {"scale",         obj.scale         },
            {"extents",       obj.extents       },
            {"linearVelocity", obj.linearVelocity},

```

```

        {"glTFuuid",      obj.glTFuuid      },
        {"customTypes",   obj.customTypes   },
        {"scriptId",      obj.scriptId      },
        {"isScriptActive", obj.isScriptActive}
    };
}

void from_json(const json& j, Object& obj) {
    j.at("name").get_to(obj.name);
    j.at("isVisible").get_to(obj.isVisible);
    j.at("isDynamic").get_to(obj.isDynamic);
    j.at("isActive").get_to(obj.isActive);
    j.at("bounds").get_to(obj.bounds);
    j.at("position").get_to(obj.position);
    j.at("rotation").get_to(obj.rotation);
    j.at("scale").get_to(obj.scale);
    j.at("extents").get_to(obj.extents);
    j.at("linearVelocity").get_to(obj.linearVelocity);
    j.at("glTFuuid").get_to(obj.glTFuuid);
    j.at("customTypes").get_to(obj.customTypes);
    j.at("scriptId").get_to(obj.scriptId);
    j.at("isScriptActive").get_to(obj.isScriptActive);
}

void to_json(json& j, const AudioFile& obj) {
    j = json{
        {"id",      obj.id      },
        {"name",     obj.name     },
        {"path",     obj.path     },
        {"isLooping", obj.isLooping},
        {"volume",   obj.volume   }
    };
}

void from_json(const json& j, AudioFile& obj) {
    j.at("id").get_to(obj.id);
    j.at("name").get_to(obj.name);
    j.at("path").get_to(obj.path);
    j.at("isLooping").get_to(obj.isLooping);
    j.at("volume").get_to(obj.volume);
}

void to_json(json& j, const UiElement& obj) {
    j = json{

```

```

        {"type",          obj.type          },
        {"text",         obj.text          },
        {"name",         obj.name          },
        {"position",     obj.position       },
        {"size",         obj.size          },
        {"fontSize",     obj.fontSize       },
        {"isVisible",    obj.isVisible      },
        {"showTitleBar", obj.showTitleBar   },
        {"children",     obj.children       },
        {"scriptId",     obj.scriptId       },
        {"useCenterPosition", obj.useCenterPosition},
        {"useBackgroud",  obj.useBackgroud  }
    };
}

void from_json(const json& j, UiElement& obj) {
    j.at("type").get_to(obj.type);
    j.at("text").get_to(obj.text);
    j.at("text").get_to(obj.tempText);
    j.at("name").get_to(obj.name);
    j.at("name").get_to(obj.tempName);
    j.at("position").get_to(obj.position);
    j.at("size").get_to(obj.size);
    j.at("fontSize").get_to(obj.fontSize);
    j.at("isVisible").get_to(obj.isVisible);
    j.at("showTitleBar").get_to(obj.showTitleBar);
    j.at("children").get_to(obj.children);
    j.at("scriptId").get_to(obj.scriptId);
    j.at("useCenterPosition").get_to(obj.useCenterPosition);
    j.at("useBackgroud").get_to(obj.useBackgroud);
}

void to_json(json& j, const Camera& obj) {
    j = json{
        {"velocity", obj.velocity},
        {"position", obj.position},
        {"pitch",    obj.pitch    },
        {"yaw",      obj.yaw      },
    };
}

void from_json(const json& j, Camera& obj) {
    j.at("velocity").get_to(obj.velocity);
    j.at("position").get_to(obj.position);
    j.at("pitch").get_to(obj.pitch);
    j.at("yaw").get_to(obj.yaw);
}

```

```

        obj.update();
    }

    void to_json(json& j, const Script& obj) {
        j = json{
            {"path", obj.path},
            {"name", obj.name},
            {"id",   obj.id   },
        };
    }

    void from_json(const json& j, Script& obj) {
        j.at("path").get_to(obj.path);
        j.at("name").get_to(obj.name);
        j.at("id").get_to(obj.id);
    }

    void to_json(json& j, const Level& obj) {
        j = json{
            {"id",          obj.id          },
            {"name",        obj.name        },
            {"loadedScenes", obj.loadedScenes},
            {"uiElements",  obj.uiElements  },
            {"scriptId",    obj.scriptId    },
        };
    }

    void from_json(const json& j, Level& obj) {
        j.at("id").get_to(obj.id);
        j.at("name").get_to(obj.name);
        j.at("loadedScenes").get_to(obj.loadedScenes);
        j.at("uiElements").get_to(obj.uiElements);
        j.at("scriptId").get_to(obj.scriptId);
    }

    bool openProject(const std::string& path, BlockEngine* eng) {
        bool jsonFound = false;
        for (const auto& entry :
            std::filesystem::directory_iterator(path)) {
            if (entry.path().filename().string() == "config.json") {

                std::filesystem::current_path(entry.path().parent_path());
                std::fstream configFile("config.json");
                json          config          =

```

```

json::parse(configFile);
    eng->currentProject.name      =
config["project"]["title"];
    eng->currentProject.tempName = eng-
>currentProject.name;
    SDL_SetWindowTitle(eng->_window, eng-
>currentProject.name.c_str());
    eng->currentProject.path      = path;
    eng->currentProject.scriptId =
config["project"]["scriptId"];

    for (const auto& [id, path] :
config["Scene"]["gltf"].items()) {
        eng->currentProject.loadedObjects[id] = path;
    }
    for (const auto& [id, object] :
config["Scene"]["audio"].items()) {
        eng->audioFiles[id] = object.template
get<AudioFile>();
        addSound(eng, eng->audioFiles[id]);
    }
    for (const auto& [id, object] :
config["Scene"]["level"].items()) {
        eng->levels[id] = object.template get<Level>();
        for (auto& [elementId, element] : eng-
>levels[id].uiElements) {
            for (auto& [uiId, child] :
element.children) {
                child.parent = eng-
>levels[id].uiElements[elementId];
            }
        }
    }
    for (const auto& [id, object] :
config["Scene"]["Script"].items()) {
        eng->loadedScripts[id] = object.template
get<Script>();
        eng->loadedScripts[id].load(*eng);
    }
    for (auto& [_, level] : eng->levels) {
        for (auto& [_, object] : level.loadedScenes) {
            if (!object.scriptId.empty()) {
                auto script      = &eng-
>loadedScripts[object.scriptId];

```

```

        object.script      = Script();
        object.script->path = script->path;
        object.script->name = script->name;
        object.script->load(*eng);
        deleteObject(object);
    }
}
for (auto& [_, object] : level.uiElements) {
    if (!object.scriptId.empty()) {
        auto script      = &eng-
>loadedScripts[object.scriptId];
        object.script      = Script();
        object.script->path = script->path;
        object.script->name = script->name;
        object.script->load(*eng);
    }
}
if (!level.scriptId.empty()) {
    auto script      = &eng-
>loadedScripts[level.scriptId];
    level.script      = Script();
    level.script->path = script->path;
    level.script->name = script->name;
    level.script->load(*eng);
}
}
if (!eng->currentProject.scriptId.empty()) {
    auto script      = &eng-
>loadedScripts[eng->currentProject.scriptId];
    eng->currentProject.script      = Script();
    eng->currentProject.script->path = script-
>path;
    eng->currentProject.script->name = script-
>name;
    eng->currentProject.script->load(*eng);
}

eng->currentProject.mainLevelId =
config["project"]["mainLevelId"];
eng->loadLevel(eng->currentProject.mainLevelId);
eng->mainCamera = config["Scene"]["camera"].template
get<Camera>();

loadGLTFObjects(eng);

```

```

        return true;
    }
}
return false;
}

std::string newProjectName;
std::string newProjectPath;
bool        openProjectPopup  = false;
bool        newProjectCreated = false;

void drawProjectPopup() {
    ImVec2 center = ImGui::GetMainViewport()->GetCenter();
    ImGui::SetNextWindowPos(center, ImGuiCond_Appearing,
ImVec2(0.5f, 0.5f));
    if (ImGui::BeginPopupModal("Project name", NULL,
ImGuiWindowFlags_AlwaysAutoResize)) {
        ImGui::InputText("Name", &newProjectName);

        if (ImGui::Button("OK", ImVec2(120, 0))) {
            openProjectPopup = false;
            ImGui::CloseCurrentPopup();
        }
        ImGui::EndPopup();
    }
}

bool BlockEngine::createProject() {
    std::filesystem::current_path(newProjectPath);
    json config;
    config["project"]["title"]      = newProjectName;
    config["project"]["version"]    = "0.0.1";
    config["project"]["description"] = "A test game";
    config["project"]["author"]     = "Author";
    currentLevelId                  = createLevel();
    loadLevel(currentLevelId);
    currentProject.mainLevelId      = currentLevelId;
    config["project"]["mainLevelId"] = currentLevelId;
    std::ofstream stream("config.json");
    stream << std::setw(2) << config << std::endl;
    return true;
}

void saveConfig(BlockEngine* eng) {

```

```

        std::filesystem::current_path(eng->currentProject.path);
        json config;
        config["project"]["title"] = eng->currentProject.name;
        config["project"]["mainLevelId"] = eng-
>currentProject.mainLevelId;
        config["project"]["scriptId"] = eng-
>currentProject.scriptId;
        for (const auto& [id, path] : eng-
>currentProject.loadedObjects) {
            config["Scene"]["gltf"][id] = path;
        }
        for (const auto& [id, file] : eng->audioFiles) {
            config["Scene"]["audio"][id] = file;
        }
        for (const auto& [id, object] : eng->levels) {
            config["Scene"]["level"][id] = object;
        }
        for (const auto& [id, elem] : eng->loadedScripts) {
            config["Scene"]["Script"][id] = elem;
        }
        config["Scene"]["camera"] = eng->mainCamera;
        std::ofstream stream("config.json");
        stream << std::setw(2) << config << std::endl;
    }

bool BlockEngine::draw_menu() {
    SDL_Event e;
    bool bQuit = false;
    bool stop = false;
    ImGui::FileBrowser
fileDialog(ImGuiFileBrowserFlags_SelectDirectory);
    ImGui::FileBrowser
createFileDialog(ImGuiFileBrowserFlags_SelectDirectory |
ImGuiFileBrowserFlags_CreateNewDir);
    fileDialog.SetTitle("Load scene");
    while (!bQuit && !stop) {
        while (SDL_PollEvent(&e) != 0) {
            if (e.type == SDL_QUIT)
                bQuit = true;
            if (e.type == SDL_WINDOWEVENT) {
                if (e.window.event == SDL_WINDOWEVENT_RESIZED)
{
                    resize_requested = true;
                }
            }
        }
    }
}

```

```

        if (e.window.event ==
SDL_WINDOWEVENT_MINIMIZED) {
            freeze_rendering = true;
        }
        if (e.window.event == SDL_WINDOWEVENT_RESTORED)
{
            freeze_rendering = false;
        }
    }
    if (e.type == SDL_KEYDOWN && e.key.keysym.sym ==
SDLK_ESCAPE) {
        bQuit = true;
    }
    ImGui_ImplSDL2_ProcessEvent(&e);
}
if (freeze_rendering)
    continue;
if (resize_requested) {
    resize_swapchain();
}
ImGui_ImplVulkan_NewFrame();
ImGui_ImplSDL2_NewFrame();
ImGui::NewFrame();

    const ImGuiViewport* main_viewport =
ImGui::GetMainViewport();
    ImGui::SetNextWindowPos(main_viewport->Pos,
ImGuiCond_Always);
    ImGui::SetNextWindowSize(main_viewport->Size,
ImGuiCond_Always);
    ImGuiWindowFlags window_flags =
        ImGuiWindowFlags_NoNav |
ImGuiWindowFlags_NoDecoration | ImGuiWindowFlags_NoMove;

    ImGui::Begin("Menu", nullptr, window_flags);
    ImGui::SetWindowFontScale(2.0f);
    ImVec2 buttonSize = ImVec2(main_viewport->Size.x / 3.0f,
main_viewport->Size.y / 7.0f);
    ImGui::SetCursorScreenPos(
        ImVec2((main_viewport->Size.x - buttonSize.x) / 2.0f,
(main_viewport->Size.y - buttonSize.y) / 3.0f));
    if (ImGui::Button("Load", buttonSize)) {
        openFileDialog.Open();
    }
}

```

```

        ImGui::SetCursorScreenPos(ImVec2((main_viewport->Size.x -
buttonSize.x) / 2.0f,

(main_viewport->Size.y - buttonSize.y) / 3.0f + buttonSize.y +
5.0f));
        if (ImGui::Button("Create", buttonSize)) {
            createFileDialog.Open();
        }

        ImGui::SetCursorScreenPos(ImVec2((main_viewport->Size.x -
buttonSize.x) / 2.0f,

(main_viewport->Size.y - buttonSize.y) / 3.0f + +buttonSize.y * 2 +
10.0f));
        if (ImGui::Button("Exit", buttonSize)) {
            bQuit = true;
        }
        ImGui::SetWindowFontScale(1.0f);
        openFileDialog.Display();
        createFileDialog.Display();
        if (fileDialog.HasSelected()) {
            stop = openProject(fileDialog.GetSelected().string(),
this);

            openFileDialog.ClearSelected();
        }
        if (createFileDialog.HasSelected()) {
            newProjectPath =
createFileDialog.GetSelected().string();
            openProjectPopup = true;
            newProjectCreated = true;
            ImGui::OpenPopup("Project name");
            createFileDialog.ClearSelected();
        }
        if (openProjectPopup) {
            drawProjectPopup();
        }
        if (newProjectCreated && !openProjectPopup) {
            stop = createProject();
            currentProject.name = newProjectName;
            currentProject.tempName = newProjectName;
            currentProject.path = newProjectPath;
            SDL_SetWindowTitle(_window,
currentProject.name.c_str());
        }

```

```

        ImGui::End();
        ImGui::Render();
        draw();
    }
    return bQuit;
}

void BlockEngine::drawUIiindowProperties() {
    const std::string& uuid = selectedObjectUuid;
    if (uuid.empty()) {
        return;
    }
    const auto& iter = currentLevel->uiElements.find(uuid);
    if (iter == currentLevel->uiElements.end()) {
        return;
    }
    auto& element = iter->second;
    bool deleteElement = false;
    ImGui::Text("Id: %s", uuid.c_str());
    ImGui::InputText("Name", &element.tempName);
    ImGui::SameLine();
    if (ImGui::Button("Rename")) {
        element.rename(this);
    }
    ImGui::DragFloat2("Position", (float*)&element.position, 0.1f);
    ImGui::DragFloat2("Size", (float*)&element.size, 0.01f, 0.01f,
100.0f);
    ImGui::Checkbox("Visible", &element.isVisible);
    ImGui::Checkbox("Show title bar", &element.showTitleBar);
    ImGui::Checkbox("Use center position",
&element.useCenterPosition);
    ImGui::Checkbox("Use backgroud", &element.useBackgroud);
    if (ImGui::Button("Add button")) {
        auto button =
createUiElement(this, "new button");
        button.type =
UiType::Button;
        button.parent =
element;
        currentLevel->uiElements[uuid].children[button.id] =
std::move(button);
    }
    ImGui::SameLine();
    if (ImGui::Button("Add text")) {

```

```

        auto text =
createUiElement(this, "new text");
        text.type =
UiType::Text;
        text.parent =
element;
        currentLevel->uiElements[uuid].children[text.id] =
std::move(text);
    }
    std::string scriptName = element.scriptId.empty() ? "" :
loadedScripts[element.scriptId].name;
    if (ImGui::BeginCombo("Select script", scriptName.c_str())) {
        for (auto& [id, script] : loadedScripts) {
            if (ImGui::Selectable(script.name.c_str(), id ==
element.scriptId)) {
                element.scriptId = id;
                element.script = Script{};
                element.script->path = script.path;
                element.script->name = script.name;
                element.script->load(*this);
            }
        }
        if (ImGui::Selectable("None", element.scriptId.empty())) {
            element.scriptId = "";
            element.script = Script{};
        }
        ImGui::EndCombo();
    }
    ImGui::PushStyleColor(ImGuiCol_Button,
(ImVec4) ImColor::HSV(0.0f, 0.6f, 0.6f));
    ImGui::PushStyleColor(ImGuiCol_ButtonHovered,
(ImVec4) ImColor::HSV(0.0f, 0.7f, 0.7f));
    ImGui::PushStyleColor(ImGuiCol_ButtonActive,
(ImVec4) ImColor::HSV(0.0f, 0.8f, 0.8f));
    if (ImGui::Button("Delete")) {
        deleteElement = true;
    }
    ImGui::PopStyleColor(3);
    if (deleteElement) {
        currentLevel->uiElements.erase(uuid);
        selectedObjectUuid = "";
        selectedObject = selectedObjectType::None;
    }
}

```

```

void BlockEngine::drawUiProperties() {
    if (selectedObjectParentUuid.empty()) {
        drawUiWindowProperties();
        return;
    }
    const std::string& parentUuid = selectedObjectParentUuid;
    if (parentUuid.empty()) {
        return;
    }
    const auto& parentIter = currentLevel->uiElements.find(parentUuid);
    if (parentIter == currentLevel->uiElements.end()) {
        return;
    }
    auto& parentElement = parentIter->second;
    const std::string& uuid = selectedObjectUuid;
    if (uuid.empty()) {
        return;
    }
    const auto& iter = parentElement.children.find(uuid);
    if (iter == parentElement.children.end()) {
        return;
    }
    auto& element = iter->second;
    bool deleteElement = false;
    ImGui::Text("Id: %s", uuid.c_str());
    ImGui::InputText("Name", &element.tempName);
    ImGui::SameLine();
    if (ImGui::Button("Rename")) {
        element.rename(this);
    }
    ImGui::DragFloat2("Position", (float*)&element.position, 0.1f);
    if (element.type == UiType::Button) {
        ImGui::DragFloat2("Size", (float*)&element.size, 0.01f,
0.01f, 100.0f);
    }
    ImGui::DragFloat("Font size", &element.fontSize, 0.01f, 0.01f,
5.0f);
    ImGui::InputText("Text", &element.tempText);
    ImGui::SameLine();
    if (ImGui::Button("Set text")) {
        element.changeText();
    }
}

```

```

        ImGui::Checkbox("Visible", &element.isVisible);
        ImGui::PushStyleColor(ImGuiCol_Button,
        (ImVec4)ImColor::HSV(0.0f, 0.6f, 0.6f));
        ImGui::PushStyleColor(ImGuiCol_ButtonHovered,
        (ImVec4)ImColor::HSV(0.0f, 0.7f, 0.7f));
        ImGui::PushStyleColor(ImGuiCol_ButtonActive,
        (ImVec4)ImColor::HSV(0.0f, 0.8f, 0.8f));
        if (ImGui::Button("Delete")) {
            deleteElement = true;
        }
        ImGui::PopStyleColor(3);
        if (deleteElement) {
            currentLevel->uiElements[parentUuid].children.erase(uuid);
            selectedObjectUuid = "";
            selectedObjectParentUuid = "";
            selectedObject = selectedObjectType::None;
        }
    }

void BlockEngine::drawAudioProperties() {
    const std::string& uuid = selectedObjectUuid;
    if (uuid.empty()) {
        return;
    }
    const auto& iter = audioFiles.find(uuid);
    if (iter == audioFiles.end()) {
        return;
    }
    auto& file = iter->second;
    ImGui::Text("Id: %s", uuid.c_str());
    ImGui::Text("Name: %s", file.name.c_str());
    if (ImGui::Button((file.isPlaying() ? "Stop" : "Start"))) {
        if (file.isPlaying()) {
            file.stop();
        } else {
            file.play();
        }
    }
    if (file.isPlaying()) {
        ImGui::SameLine();
        if (ImGui::Button("Pause")) {
            file.pause();
        }
    }
}

```

```

        if (ImGui::Button((file.isLooping ? "Unloop" : "Loop"))) {
            file.isLooping = !file.isLooping;
            ma_sound_set_looping(&file.sound, file.isLooping);
        }
        ImGui::Text("Duration: %f", file.getLengthInSeconds());
        ImGui::Text("Position: %f",
file.getCursorPositionInSeconds());
        int currentFrame = file.getCursorPosition();
        int tempCurrentFrame = currentFrame;
        ImGui::SliderInt("Position", &currentFrame, 0,
file.getLengthInFrames());
        if (currentFrame != tempCurrentFrame) {
            file.setCursorPosition(currentFrame);
        }
        float progress = 0;
        if (file.getLengthInFrames() > 0) {
            progress = (float)tempCurrentFrame /
(float)file.getLengthInFrames();
        }
        ImGui::ProgressBar(progress);
        float tempVolume = file.volume;
        ImGui::SliderFloat("Volume", &tempVolume, 0.0f, 10.0f);
        if (tempVolume != file.volume) {
            file.setVolume(tempVolume);
        }
        ImGui::PushStyleColor(ImGuiCol_Button,
(ImVec4) ImColor::HSV(0.0f, 0.6f, 0.6f));
        ImGui::PushStyleColor(ImGuiCol_ButtonHovered,
(ImVec4) ImColor::HSV(0.0f, 0.7f, 0.7f));
        ImGui::PushStyleColor(ImGuiCol_ButtonActive,
(ImVec4) ImColor::HSV(0.0f, 0.8f, 0.8f));
        if (ImGui::Button("Delete")) {
            file.destroy();
            audioFiles.erase(uuid);
        }
        ImGui::PopStyleColor(3);
    }

void BlockEngine::drawScriptProperties() {
    const std::string& uuid = selectedObjectUuid;
    if (uuid.empty()) {
        return;
    }
    const auto& iter = loadedScripts.find(uuid);

```

```

        if (iter == loadedScripts.end()) {
            return;
        }
        auto& object = iter->second;
        ImGui::Text("Name: %s", object.name.c_str());
        if (ImGui::Button((object.isActive ? "Stop" : "Activate"))) {
            if (scriptSimulateionIsActive && object.isActive) {
                object.stop();
            } else if (scriptSimulateionIsActive && !object.isActive)
            {
                object.init();
            }
            object.isActive = !object.isActive;
        }
        if (ImGui::Button("Reload")) {
            object.load(*this);

            for (auto& [_ , scene] : currentLevel->loadedScenes) {
                if (scene.scriptId == uuid) {
                    scene.script->load(*this);
                }
            }
        }
        ImGui::SameLine();
        ImGui::PushStyleColor(ImGuiCol_Button,
        (ImVec4)ImColor::HSV(0.0f, 0.6f, 0.6f));
        ImGui::PushStyleColor(ImGuiCol_ButtonHovered,
        (ImVec4)ImColor::HSV(0.0f, 0.7f, 0.7f));
        ImGui::PushStyleColor(ImGuiCol_ButtonActive,
        (ImVec4)ImColor::HSV(0.0f, 0.8f, 0.8f));
        if (ImGui::Button("Delete")) {
            loadedScripts.erase(uuid);
        }
        ImGui::PopStyleColor(3);
    }

void BlockEngine::drawLevelProperties() {
    const std::string& uuid = selectedObjectUuid;
    if (uuid.empty()) {
        return;
    }
    const auto& iter = levels.find(uuid);
    if (iter == levels.end()) {
        return;
    }

```

```

    }
    auto& object = iter->second;
    ImGui::InputText("Name", &object.tempName);
    if (ImGui::Button("Rename")) {
        object.name = object.tempName;
    }
    if (currentLevelId == uuid) {
        ImGui::Text("Current level");
    } else {
        if (ImGui::Button("Set as current")) {
            loadLevel(uuid);
        }
    }
    if (currentProject.mainLevelId == uuid) {
        ImGui::Text("Main level");
    } else {
        if (ImGui::Button("Set as main")) {
            currentProject.mainLevelId = uuid;
        }
    }
    std::string scriptName = object.scriptId.empty() ? "" :
loadedScripts[object.scriptId].name;
    if (ImGui::BeginCombo("Select script", scriptName.c_str())) {
        for (auto& [id, script] : loadedScripts) {
            if (ImGui::Selectable(script.name.c_str(), id ==
object.scriptId)) {
                object.scriptId = id;
                object.script = Script{};
                object.script->path = script.path;
                object.script->name = script.name;
                object.script->load(*this);
            }
        }
        if (ImGui::Selectable("None", object.scriptId.empty())) {
            object.scriptId = "";
            object.script = Script{};
        }
        ImGui::EndCombo();
    }
    ImGui::PushStyleColor(ImGuiCol_Button,
(ImVec4)ImColor::HSV(0.0f, 0.6f, 0.6f));
    ImGui::PushStyleColor(ImGuiCol_ButtonHovered,
(ImVec4)ImColor::HSV(0.0f, 0.7f, 0.7f));
    ImGui::PushStyleColor(ImGuiCol_ButtonActive,

```

```

(ImVec4) ImColor::HSV(0.0f, 0.8f, 0.8f));
    if (uuid != currentLevelId) {
        if (ImGui::Button("Delete")) {
            levels.erase(uuid);
        }
    }
    ImGui::PopStyleColor(3);
}

void BlockEngine::drawGLTFModelProperties() {
    const std::string& uuid = selectedObjectUuid;

    if (uuid.empty()) {
        return;
    }
    const auto& iter = loadedGLTFs.find(uuid);
    if (iter == loadedGLTFs.end()) {
        return;
    }
    auto& object = *iter->second;
    ImGui::Text("Name: %s", object.name.c_str());
    ImGui::Text("Path: %s", object.path.c_str());
    if (ImGui::Button("Create object")) {
        createObject(uuid);
    }
}

void BlockEngine::drawGLTFProperties() {
    const std::string& uuid = selectedObjectUuid;
    if (uuid.empty()) {
        return;
    }
    const auto& iter = currentLevel->loadedScenes.find(uuid);
    if (iter == currentLevel->loadedScenes.end()) {
        return;
    }
    auto& object = iter->second;
    bool deleteElement = false;
    ImGui::Text("Id: %s", uuid.c_str());
    ImGui::DragFloat3("Position", (float*)&object.position, 0.1f);
    ImGui::DragFloat3("Rotation", (float*)&object.rotation, 1.f);
    ImGui::DragFloat3("Scale", (float*)&object.scale, 0.01f, 0.01f,
100.0f);
    ImGui::DragFloat3("Extents", (float*)&object.bounds.extents,

```

```

0.01f, 0.01f, 100.0f);
    ImGui::DragFloat3("Linear velocity",
(float*)&object.linearVelocity, 0.01f, 0.01f, 100.0f);
    ImGui::InputText("Name", &object.tempName);
    if (ImGui::Button("Rename")) {
        object.setName(createName(currentLevel->loadedScenes,
object.tempName));
    }
    if (ImGui::CollapsingHeader("Custom types",
ImGuiTreeNodeFlags_None)) {
        for (auto& type : object.customTypes) {
            ImGui::Text(type.c_str());
        }
        ImGui::InputText("New type", &object.tempType);
        if (ImGui::Button("Add")) {
            if (!object.tempType.empty()) {
                object.customTypes.insert(object.tempType);
                object.tempType = "";
            }
        }
    }
    object.updateTransforms();
    bool isDynamic = object.isDynamic;
    if (ImGui::Checkbox("Dynamic", &isDynamic)) {
        if (isDynamic != object.isDynamic) {
            object.isDynamic = isDynamic;
            if (isDynamic) {
                setObjectMotionType(object,
EMotionType::Dynamic);
            } else {
                setObjectMotionType(object,
EMotionType::Static);
            }
        }
    }
    ImGui::SameLine();
    bool isVisible = object.isVisible;
    if (ImGui::Checkbox("Visible", &isVisible)) {
        if (isVisible != object.isVisible) {
            object.isVisible = isVisible;
        }
    }
    std::string scriptName = object.scriptId.empty() ? "" :
loadedScripts[object.scriptId].name;

```

```

    if (ImGui::BeginCombo("Select script", scriptName.c_str())) {
        for (auto& [id, script] : loadedScripts) {
            if (ImGui::Selectable(script.name.c_str(), id ==
object.scriptId)) {
                object.scriptId      = id;
                object.script         = Script{};
                object.script->path = script.path;
                object.script->name = script.name;
                object.script->load(*this);
            }
        }
        if (ImGui::Selectable("None", object.scriptId.empty())) {
            object.scriptId = "";
            object.script    = Script{};
        }
        ImGui::EndCombo();
    }
    if (!object.scriptId.empty()) {
        if (ImGui::Button((object.isScriptActive ? "Deactivate
script" : "Activate script"))) {
            object.isScriptActive = !object.isScriptActive;
        }
    }
    if (ImGui::Button("Copy")) {
        const uuids::uuid id  = uuids::uuid_system_generator{}();
        Object                copy = object;
        copy.setName(createName(currentLevel->loadedScenes,
copy.name));
        currentLevel->loadedScenes[uuids::to_string(id)] = copy;
        if (currentLevel-
>loadedScenes[uuids::to_string(id)].script) {
            currentLevel-
>loadedScenes[uuids::to_string(id)].script->load(*this);
        }
    }
    ImGui::SameLine();
    ImGui::PushStyleColor(ImGuiCol_Button,
(ImVec4) ImColor::HSV(0.0f, 0.6f, 0.6f));
    ImGui::PushStyleColor(ImGuiCol_ButtonHovered,
(ImVec4) ImColor::HSV(0.0f, 0.7f, 0.7f));
    ImGui::PushStyleColor(ImGuiCol_ButtonActive,
(ImVec4) ImColor::HSV(0.0f, 0.8f, 0.8f));
    if (ImGui::Button("Delete")) {
        deleteElement = true;
    }

```

```

    }
    ImGui::PopStyleColor(3);
    if (deleteElement) {
        deleteObject(object);
        currentLevel->loadedScenes.erase(uuid);
        selectedObjectUuid = "";
        selectedObject      = selectedObjectType::None;
    }
}

void BlockEngine::drawCameraProperties() {
    ImGui::DragFloat3("Position", (float*)&mainCamera.position,
0.1f);
    ImGui::DragFloat("Vertical rotation (pitch)",
(float*)&mainCamera.pitch);
    ImGui::DragFloat("Horizontal rotation (yaw)",
(float*)&mainCamera.yaw);
}

void BlockEngine::drawProperties() {
    switch (selectedObject) {
    case selectedObjectType::GLTF:
        ImGui::Text("Type: Object");
        drawGLTFProperties();
        break;
    case selectedObjectType::Audio:
        ImGui::Text("Type: Audio");
        drawAudioProperties();
        break;
    case selectedObjectType::UiElement:
        ImGui::Text("Type: UI element");
        drawUiProperties();
        break;
    case selectedObjectType::Camera:
        ImGui::Text("Type: Camera");
        drawCameraProperties();
        break;
    case selectedObjectType::Script:
        ImGui::Text("Type: Script");
        drawScriptProperties();
        break;
    case selectedObjectType::Level:
        ImGui::Text("Type: Level");
        drawLevelProperties();
    }
}

```

```

        break;
    case selectedObjectType::GLTFModel:
        ImGui::Text("Type: GLTF model");
        drawGLTFModelProperties();
        break;
    default: break;
}

}

void BlockEngine::drawGLTFModels() {
    for (auto& [uuid, file] : loadedGLTFs) {
        if (ImGui::Selectable(file->name.c_str(), uuid ==
selectedObjectUuid)) {
            selectedObjectUuid = uuid;
            selectedObject      = selectedObjectType::GLTFModel;
        }
    }
}

void BlockEngine::drawHierarchy() {
    if (ImGui::TreeNode("UI Elements")) {
        for (auto& [uuid, element] : currentLevel->uiElements) {
            if (ImGui::TreeNode(uuid.c_str(),
element.name.c_str())) {
                if (ImGui::Selectable(element.name.c_str(),
uuid == selectedObjectUuid)) {
                    selectedObjectParentUuid = "";
                    selectedObjectUuid       = uuid;
                    selectedObject            =
selectedObjectType::UiElement;
                }
                for (auto& [childUuid, childElement] :
element.children) {
                    if
(ImGui::Selectable(childElement.name.c_str(), childUuid ==
selectedObjectUuid)) {
                        selectedScriptUuid       = "";
                        selectedObjectParentUuid = uuid;
                        selectedObjectUuid       =
childUuid;
                        selectedObject          =
selectedObjectType::UiElement;
                    }
                }
            }
        }
    }
}

```

```

        ImGui::TreePop();
    }
}
ImGui::TreePop();
}
if (ImGui::TreeNode("Audio")) {
    for (auto& [uuid, file] : audioFiles) {
        if (ImGui::Selectable(file.name.c_str(), uuid ==
selectedObjectUuid)) {
            selectedObjectUuid = uuid;
            selectedObject      = selectedObjectType::Audio;
        }
    }
    ImGui::TreePop();
}
if (ImGui::TreeNode("Objects")) {
    for (auto& [uuid, object] : currentLevel->loadedScenes) {
        if (ImGui::Selectable(object.name.c_str(), uuid ==
selectedObjectUuid)) {
            selectedScriptUuid = "";
            selectedObjectUuid = uuid;
            selectedObject      = selectedObjectType::GLTF;
        }
    }
    ImGui::TreePop();
}
if (ImGui::TreeNode("Scripts")) {
    for (auto& [uuid, object] : loadedScripts) {
        if (ImGui::Selectable(object.name.c_str(), uuid ==
selectedObjectUuid)) {
            selectedObjectUuid = uuid;
            selectedObject      =
selectedObjectType::Script;
        }
    }
    ImGui::TreePop();
}
if (ImGui::TreeNode("Levels")) {
    for (auto& [uuid, object] : levels) {
        if (ImGui::Selectable(object.name.c_str(), uuid ==
selectedObjectUuid)) {
            selectedObjectUuid = uuid;
            selectedObject      = selectedObjectType::Level;
        }
    }
}

```

```

        }
        ImGui::TreePop();
    }
    if (ImGui::Selectable("Camera", selectedObject ==
selectedObjectType::Camera)) {
        selectedObjectUuid = "";
        selectedObject      = selectedObjectType::Camera;
    }
}

void BlockEngine::drawStats() {
    if (!showStats) {
        return;
    }
    ImGui::Begin("Stats", &showStats);
    ImGui::Text("frametime %f ms", stats.frametime);
    ImGui::Text("drawtime %f ms", stats.mesh_draw_time);
    ImGui::Text("triangles %i", stats.triangle_count);
    ImGui::Text("draws %i", stats.drawcall_count);
    ImGui::End();
}

void BlockEngine::drawMenuBar() {
    if (ImGui::BeginMainMenuBar()) {
        if (ImGui::BeginMenu("Menu")) {
            if (ImGui::MenuItem("Show stats")) {
                showStats = true;
            }
            if (ImGui::MenuItem("Load model")) {
                mainFileDialog.SetTitle("Load model");
                mainFileDialog.SetTypeFilters({".glb"});
                mainFileDialog.Open();
            }
            if (ImGui::MenuItem("Load audio")) {
                mainFileDialog.SetTitle("Load audio");
                mainFileDialog.SetTypeFilters({".wav"});
                mainFileDialog.Open();
            }
            if (ImGui::MenuItem("Load script")) {
                mainFileDialog.SetTitle("Load script");
                mainFileDialog.SetTypeFilters({".lua"});
                mainFileDialog.Open();
            }
        }
    }
}

```

```

        if (ImGui::MenuItem("Add ui window")) {
            auto element =
createUiElement(this, "new window");
            element.type =
UiType::Window;
            currentLevel->uiElements[element.id] =
std::move(element);
        }
        if (ImGui::MenuItem("Add level")) {
            createLevel();
        }
        if (ImGui::MenuItem("Save")) {
            saveConfig(this);
        }
        if (ImGui::MenuItem("Export")) {
            saveConfig(this);
            exportProject();
        }
        ImGui::EndMenu();
    }
    if (scriptSimulateionIsActive) {
        if (ImGui::Button("Stop simulation")) {
            scriptSimulateionIsActive = false;
            isPhysicsRunning = false;
        }
        if (ImGui::Button(hideUi ? "Show UI" : "Hide UI")) {
            hideUi = !hideUi;
        }
        if (ImGui::Button(freeCamera ? "Disable free camera"
: "Enable free camera")) {
            freeCamera = !freeCamera;
        }
    } else {
        if (ImGui::Button("Start simulation")) {
            scriptSimulateionIsActive = true;
            isPhysicsRunning = true;

            if (!currentLevel->scriptId.empty()) {
                currentLevel->script->lua["thisObjectId"]
= currentLevelId;

                currentLevel->script->init();
            }
        }
    }
}

```

```

        ImGui::EndMainMenuBar();
    }
}

void BlockEngine::drawFileDialog() {
    mainFileDialog.Display();
    if (mainFileDialog.HasSelected()) {
        std::filesystem::path path =
mainFileDialog.GetSelected().string();
        if (path.extension() == ".glb") {
            loadObject(this, path.string());
        } else if (path.extension() == ".wav") {
            loadAudio(this, path.string());
        } else if (path.extension() == ".lua") {
            loadScript(path.string());
        }
        mainFileDialog.ClearSelected();
    }
}

void BlockEngine::loadScript(const std::string& path) {
    std::filesystem::current_path(currentProject.path);
    std::string id =
uuids::to_string(uuids::uuid_system_generator{}());
    loadedScripts.emplace(id, Script());
    auto& script = loadedScripts[id];
    std::filesystem::path currentPath = path;
    std::filesystem::path newPath = currentProject.path /
std::filesystem::path("assets") / currentPath.filename();
    if (!std::filesystem::exists(currentProject.path /
std::filesystem::path("assets"))) {
        std::filesystem::create_directory(currentProject.path /
std::filesystem::path("assets"));
    }
    if (!std::filesystem::exists(newPath)) {
        std::filesystem::copy(path, newPath);
    }
    script.path = (std::filesystem::path("assets") /
currentPath.filename()).string();
    script.name = newPath.stem().string();
    script.load(*this);
}

void BlockEngine::drawSettings() {

```

```

ImGui::InputText("Name", &currentProject.tempName);
if (ImGui::Button("Rename")) {
    currentProject.name = currentProject.tempName;
    SDL_SetWindowTitle(_window, currentProject.name.c_str());
}
ImGui::Text("Main level: %s",
levels[currentProject.mainLevelId].name.c_str());
if (ImGui::Button("Activate audio engine")) {
    ma_engine_start(&audio_engine);
}
if (ImGui::Button("Deactivate audio engine")) {
    ma_engine_stop(&audio_engine);
}
if (ImGui::Button((isPhysicsRunning ? "Disable simulation" :
"Enable simulation"))) {
    isPhysicsRunning          = !isPhysicsRunning;
    scriptSimulateionIsActive = !scriptSimulateionIsActive;
    if (!currentLevel->scriptId.empty() &&
scriptSimulateionIsActive) {
        currentLevel->script->lua["thisObjectId"] =
currentLevelId;
        currentLevel->script->init();
    }
}
std::string scriptName = currentProject.scriptId.empty() ? "" :
loadedScripts[currentProject.scriptId].name;
if (ImGui::BeginCombo("Select script", scriptName.c_str())) {
    for (auto& [id, script] : loadedScripts) {
        if (ImGui::Selectable(script.name.c_str(), id ==
currentProject.scriptId)) {
            currentProject.scriptId    = id;
            currentProject.script      = Script{};
            currentProject.script->path = script.path;
            currentProject.script->name = script.name;
            currentProject.script->load(*this);
        }
    }
    if (ImGui::Selectable("None",
currentProject.scriptId.empty())) {
        currentProject.scriptId = "";
        currentProject.script   = Script{};
    }
    ImGui::EndCombo();
}

```

```

}

void BlockEngine::drawTop() {
    const ImGuiViewport* main_viewport = ImGui::GetMainViewport();
    ImVec2 size = ImVec2(main_viewport->Size.x / 3, main_viewport->Size.y / 2);
    ImVec2 position = ImVec2(main_viewport->Pos.x + main_viewport->Size.x - size.x, main_viewport->Pos.y);
    ImGui::SetNextWindowPos(position, ImGuiCond_Always);
    ImGui::SetNextWindowSize(size, ImGuiCond_Always);
    ImGuiWindowFlags window_flags =
        ImGuiWindowFlags_NoResize | ImGuiWindowFlags_NoMove |
        ImGuiWindowFlags_NoCollapse | ImGuiWindowFlags_NoTitleBar;
    ImGui::Begin("TopWindow", nullptr, window_flags);
    if (ImGui::BeginTabBar("TopTabBar")) {
        if (ImGui::BeginTabItem("Hierarchy")) {
            drawHierarchy();
            ImGui::EndTabItem();
        }
        if (ImGui::BeginTabItem("Loaded models")) {
            drawGLTFModels();
            ImGui::EndTabItem();
        }
    }
    ImGui::EndTabBar();
}
ImGui::End();
}

void BlockEngine::drawBottom() {
    const ImGuiViewport* main_viewport = ImGui::GetMainViewport();
    ImVec2 size = ImVec2(main_viewport->Size.x / 3, main_viewport->Size.y / 2);
    ImVec2 position = ImVec2(main_viewport->Pos.x + main_viewport->Size.x - size.x, main_viewport->Size.y - size.y);
    ImGui::SetNextWindowPos(position, ImGuiCond_Always);
    ImGui::SetNextWindowSize(size, ImGuiCond_Always);
    ImGuiWindowFlags window_flags =
        ImGuiWindowFlags_NoResize | ImGuiWindowFlags_NoMove |
        ImGuiWindowFlags_NoCollapse | ImGuiWindowFlags_NoTitleBar;
    ImGui::Begin("BottomWindow", nullptr, window_flags);
    if (ImGui::BeginTabBar("BottomTabBar")) {
        if (ImGui::BeginTabItem("Properties")) {
            drawProperties();
            ImGui::EndTabItem();
        }
    }
}

```

```

    }
    if (ImGui::BeginTabItem("Settings")) {
        drawSettings();
        ImGui::EndTabItem();
    }
    ImGui::EndTabBar();
}
ImGui::End();
}

void BlockEngine::processInput() {
    SDL_Event e;
    initKeyMap();
    mouseMotion = {0, 0};
    while (SDL_PollEvent(&e) != 0) {
        if (e.type == SDL_QUIT) {
            needsExit = true;
        }
        if (e.type == SDL_WINDOWEVENT) {
            if (e.window.event == SDL_WINDOWEVENT_RESIZED) {
                resize_requested = true;
            }
            if (e.window.event == SDL_WINDOWEVENT_MINIMIZED) {
                freeze_rendering = true;
            }
            if (e.window.event == SDL_WINDOWEVENT_RESTORED) {
                freeze_rendering = false;
            }
        }
        if (e.type == SDL_KEYDOWN && e.key.keysym.sym ==
SDLK_ESCAPE) {
            needsExit = true;
        }
        if (e.type == SDL_KEYDOWN) {
            if (e.key.keysym.sym == SDLK_a) {
                keys[KeyType::A] = true;
            }
            if (e.key.keysym.sym == SDLK_b) {
                keys[KeyType::B] = true;
            }
            if (e.key.keysym.sym == SDLK_c) {
                keys[KeyType::C] = true;
            }
            if (e.key.keysym.sym == SDLK_d) {

```

```

        keys[KeyType::D] = true;
    }
    if (e.key.keysym.sym == SDLK_e) {
        keys[KeyType::E] = true;
    }
    if (e.key.keysym.sym == SDLK_f) {
        keys[KeyType::F] = true;
    }
    if (e.key.keysym.sym == SDLK_g) {
        keys[KeyType::G] = true;
    }
    if (e.key.keysym.sym == SDLK_h) {
        keys[KeyType::H] = true;
    }
    if (e.key.keysym.sym == SDLK_i) {
        keys[KeyType::I] = true;
    }
    if (e.key.keysym.sym == SDLK_j) {
        keys[KeyType::J] = true;
    }
    if (e.key.keysym.sym == SDLK_k) {
        keys[KeyType::K] = true;
    }
    if (e.key.keysym.sym == SDLK_l) {
        keys[KeyType::L] = true;
    }
    if (e.key.keysym.sym == SDLK_m) {
        keys[KeyType::M] = true;
    }
    if (e.key.keysym.sym == SDLK_n) {
        keys[KeyType::N] = true;
    }
    if (e.key.keysym.sym == SDLK_o) {
        keys[KeyType::O] = true;
    }
    if (e.key.keysym.sym == SDLK_p) {
        keys[KeyType::P] = true;
    }
    if (e.key.keysym.sym == SDLK_q) {
        keys[KeyType::Q] = true;
    }
    if (e.key.keysym.sym == SDLK_r) {
        keys[KeyType::R] = true;
    }
}

```

```

if (e.key.keysym.sym == SDLK_s) {
    keys[KeyType::S] = true;
}
if (e.key.keysym.sym == SDLK_t) {
    keys[KeyType::T] = true;
}
if (e.key.keysym.sym == SDLK_u) {
    keys[KeyType::U] = true;
}
if (e.key.keysym.sym == SDLK_v) {
    keys[KeyType::V] = true;
}
if (e.key.keysym.sym == SDLK_w) {
    keys[KeyType::W] = true;
}
if (e.key.keysym.sym == SDLK_x) {
    keys[KeyType::X] = true;
}
if (e.key.keysym.sym == SDLK_y) {
    keys[KeyType::Y] = true;
}
if (e.key.keysym.sym == SDLK_z) {
    keys[KeyType::Z] = true;
}
if (e.key.keysym.sym == SDLK_0) {
    keys[KeyType::Num_0] = true;
}
if (e.key.keysym.sym == SDLK_1) {
    keys[KeyType::Num_1] = true;
}
if (e.key.keysym.sym == SDLK_2) {
    keys[KeyType::Num_2] = true;
}
if (e.key.keysym.sym == SDLK_3) {
    keys[KeyType::Num_3] = true;
}
if (e.key.keysym.sym == SDLK_4) {
    keys[KeyType::Num_4] = true;
}
if (e.key.keysym.sym == SDLK_5) {
    keys[KeyType::Num_5] = true;
}
if (e.key.keysym.sym == SDLK_6) {
    keys[KeyType::Num_6] = true;
}

```

```

    }
    if (e.key.keysym.sym == SDLK_7) {
        keys[KeyType::Num_7] = true;
    }
    if (e.key.keysym.sym == SDLK_8) {
        keys[KeyType::Num_8] = true;
    }
    if (e.key.keysym.sym == SDLK_9) {
        keys[KeyType::Num_9] = true;
    }
    if (e.key.keysym.sym == SDLK_RETURN) {
        keys[KeyType::Enter] = true;
    }
    if (e.key.keysym.sym == SDLK_ESCAPE) {
        keys[KeyType::Escape] = true;
    }
    if (e.key.keysym.sym == SDLK_BACKSPACE) {
        keys[KeyType::Backspace] = true;
    }
    if (e.key.keysym.sym == SDLK_SPACE) {
        keys[KeyType::Space] = true;
    }
    if (e.key.keysym.sym == SDLK_TAB) {
        keys[KeyType::Tab] = true;
    }
    if (e.key.keysym.sym == SDLK_UP) {
        keys[KeyType::Up] = true;
    }
    if (e.key.keysym.sym == SDLK_DOWN) {
        keys[KeyType::Down] = true;
    }
    if (e.key.keysym.sym == SDLK_LEFT) {
        keys[KeyType::Left] = true;
    }
    if (e.key.keysym.sym == SDLK_RIGHT) {
        keys[KeyType::Right] = true;
    }
}
if (e.type == SDL_MOUSEBUTTONDOWN) {
    if (e.button.button == SDL_BUTTON_LEFT) {
        keys[KeyType::Mouse_Left] = true;
    }
    if (e.button.button == SDL_BUTTON_RIGHT) {
        keys[KeyType::Mouse_Right] = true;
    }
}

```

```

        }
    }
    if (e.type == SDL_MOUSEMOTION) {
        mouseMotion.x += (float)e.motion.xrel;
        mouseMotion.y += (float)e.motion.yrel;
    }
    if (freeCamera) {
        mainCamera.processSDLEvent(e);
    }
    ImGui_ImplSDL2_ProcessEvent(&e);
}

void BlockEngine::cameraInput() {
    if (keys[KeyType::Mouse_Right]) {
        mainCamera.rotate(mouseMotion);
    }
    if (keys[KeyType::W]) {
        mainCamera.move(CameraDirection::Forward);
    }
    if (keys[KeyType::S]) {
        mainCamera.move(CameraDirection::Backward);
    }
    if (keys[KeyType::A]) {
        mainCamera.move(CameraDirection::Left);
    }
    if (keys[KeyType::D]) {
        mainCamera.move(CameraDirection::Right);
    }
}

void BlockEngine::run() {
    RegisterDefaultAllocator();
    JPH_IF_ENABLE_ASSERTS(AssertFailed = AssertFailedImpl);
    Factory::sInstance = new Factory();
    RegisterTypes();
    TempAllocatorImpl    temp_allocator(10 * 1024 * 1024);
    JobSystemThreadPool  job_system(cMaxPhysicsJobs,
cMaxPhysicsBarriers, thread::hardware_concurrency() - 1);
    const uint          cMaxBodies          = 1024;
    const uint          cNumBodyMutexes     = 0;
    const uint          cMaxBodyPairs       = 1024;
    const uint          cMaxContactConstraints = 1024;
    BPLayerInterfaceImpl broad_phase_layer_interface;

```

```

    ObjectVsBroadPhaseLayerFilterImpl
object_vs_broadphase_layer_filter;
    ObjectLayerPairFilterImpl
object_vs_object_layer_filter;
    PhysicsSystem physics_system;
    physics_system.Init(cMaxBodies, cNumBodyMutexes, cMaxBodyPairs,
cMaxContactConstraints, broad_phase_layer_interface,
object_vs_broadphase_layer_filter,
object_vs_object_layer_filter);
    MyBodyActivationListener body_activation_listener;
    physics_system.SetBodyActivationListener(&body_activation_liste
ner);
    MyContactListener contact_listener(*this);
    physics_system.SetContactListener(&contact_listener);
    body_interface = &physics_system.GetBodyInterface();
    searchGameFiles();
    bool needQuit = false;
    if (!isGameInstance) {
        needQuit = draw_menu();
        if (needQuit) {
            return;
        }
    } else {
        auto currentPath = std::filesystem::current_path();
        extractZipArchive(currentPath.string());
        openProject(currentPath.string(), this);
        isPhysicsRunning = true;
        scriptSimulateionIsActive = true;
        hideUi = true;
        freeCamera = false;
        if (std::filesystem::exists(currentPath /
std::filesystem::path("assets"))) {
            BK_INFO("Removing assets folder");
        }
        if (currentLevel != nullptr) {
            currentLevel->script->init();
        }
    }

    initKeyMap();

    while (!needsExit) {
        auto start = std::chrono::system_clock::now();
        processInput();
    }

```

```

    if (freeze_rendering)
        continue;
    if (resize_requested) {
        resize_swapchain();
    }
    ImGui_ImplVulkan_NewFrame();
    ImGui_ImplSDL2_NewFrame();
    ImGui::NewFrame();
    if (!isGameInstance) {
        drawStats();
        drawMenuBar();
        drawFileDialog();
    }
    if (!hideUi) {
        drawTop();
        drawBottom();
    }
    renderUiElements(this);
    ImGui::Render();
    if (scriptSimulateionIsActive) {
        for (auto& [id, ptr] : currentLevel->loadedScenes) {
            if (ptr.scriptId.empty() ||
!ptr.isScriptActive)
                continue;
            if (ptr.script) {
                ptr.script->lua["thisObjectId"] = id;
                ptr.script->step();
            }
        }
        for (auto& [id, ptr] : currentLevel->uiElements) {
            if (ptr.scriptId.empty())
                continue;
            if (ptr.script) {
                ptr.script->lua["thisObjectId"] = id;
                ptr.script->step();
            }
        }
    }

    update_scene();
    if (isPhysicsRunning) {
        const int cCollisionSteps = 1;
        physics_system.Update(1.0f / 60.0f, cCollisionSteps,
&temp_allocator, &job_system);

```

```

        for (auto& [_, ptr] : currentLevel->loadedScenes) {
            RVec3 position = body_interface-
>GetCenterOfMassPosition(ptr.bodyId);
            Vec3 velocity = body_interface-
>GetLinearVelocity(ptr.bodyId);
            Quat rotation = body_interface-
>GetRotation(ptr.bodyId);
            if (ptr.lockPositionX) {
                body_interface->SetPosition(ptr.bodyId,
Vec3(ptr.position.x, position.GetY(), position.GetZ()),

                EActivation::DontActivate);
            } else {
                ptr.position.x = position.GetX();
            }
            if (ptr.lockPositionY) {
                body_interface->SetPosition(ptr.bodyId,
Vec3(position.GetX(), ptr.position.y, position.GetZ()),

                EActivation::DontActivate);
            } else {
                ptr.position.y = position.GetY();
            }
            if (ptr.lockPositionZ) {
                body_interface->SetPosition(ptr.bodyId,
Vec3(position.GetX(), position.GetY(), ptr.position.z),

                EActivation::DontActivate);
            } else {
                ptr.position.z = position.GetZ();
            }
            ptr.linearVelocity = glm::vec3{velocity.GetX(),
velocity.GetY(), velocity.GetZ()};
        }
    }

    draw();
    auto end = std::chrono::system_clock::now();
    auto elapsed =
std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    stats.frameTime = elapsed.count() / 1000.f;
}
}

```

```

void BlockEngine::update_scene() {
    mainCamera.update();
    glm::mat4 view = mainCamera.getViewMatrix();
    glm::mat4 projection =
        glm::perspective(glm::radians(70.f),
        (float)_windowExtent.width / (float)_windowExtent.height, 10000.f,
        0.1f);
    projection[1][1] *= -1;
    sceneData.view = view;
    sceneData.proj = projection;
    sceneData.viewproj = projection * view;
    for (auto& [_ , ptr] : currentLevel->loadedScenes) {
        if (ptr.scene == nullptr)
            continue;
        ptr.Draw(glm::mat4{1.f}, drawCommands);
    }
}

```

```

AllocatedBuffer BlockEngine::create_buffer(size_t allocSize,
VkBufferUsageFlags usage, VmaMemoryUsage memoryUsage) {
    VkBufferCreateInfo bufferInfo = {};
    bufferInfo.sType =
VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.pNext = nullptr;
    bufferInfo.size = allocSize;
    bufferInfo.usage = usage;
    VmaAllocationCreateInfo vmaallocInfo = {};
    vmaallocInfo.usage = memoryUsage;
    vmaallocInfo.flags =
VMA_ALLOCATION_CREATE_MAPPED_BIT;
    AllocatedBuffer newBuffer;
    VK_CHECK(vmaCreateBuffer(_allocator, &bufferInfo,
&vmaallocInfo, &newBuffer.buffer, &newBuffer.allocation,
&newBuffer.info));

    return newBuffer;
}

```

```

AllocatedImage BlockEngine::create_image(VkExtent3D size, VkFormat
format, VkImageUsageFlags usage, bool mipmapped) {
    AllocatedImage newImage;
    newImage.imageFormat = format;
    newImage.imageExtent = size;
    VkImageCreateInfo img_info = image_create_info(format, usage,
size);
}

```

```

        if (mipmapped) {
            img_info.mipLevels =
static_cast<uint32_t>(std::floor(std::log2(std::max(size.width,
size.height)))) + 1;
        }
        VmaAllocationCreateInfo allocinfo = {};
        allocinfo.usage = VMA_MEMORY_USAGE_GPU_ONLY;
        allocinfo.requiredFlags =
VkMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
        VK_CHECK(vmaCreateImage(_allocator, &img_info, &allocinfo,
&newImage.image, &newImage.allocation, nullptr));
        VkImageAspectFlags aspectFlag = VK_IMAGE_ASPECT_COLOR_BIT;
        if (format == VK_FORMAT_D32_SFLOAT) {
            aspectFlag = VK_IMAGE_ASPECT_DEPTH_BIT;
        }
        VkImageViewCreateInfo view_info =
imageview_create_info(format, newImage.image, aspectFlag);
        view_info.subresourceRange.levelCount = img_info.mipLevels;
        VK_CHECK(vkCreateImageView(_device, &view_info, nullptr,
&newImage.imageView));
        return newImage;
    }

AllocatedImage BlockEngine::create_image(void* data, VkExtent3D size,
VkFormat format, VkImageUsageFlags usage,
bool
mipmapped) {
    size_t data_size = size.depth * size.width *
size.height * 4;
    AllocatedBuffer uploadbuffer =
        create_buffer(data_size, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
VMA_MEMORY_USAGE_CPU_TO_GPU);
    memcpy(uploadbuffer.info.pMappedData, data, data_size);
    AllocatedImage new_image = create_image(
        size, format, usage | VK_IMAGE_USAGE_TRANSFER_DST_BIT |
VK_IMAGE_USAGE_TRANSFER_SRC_BIT, mipmapped);
    immediate_submit([&](VkCommandBuffer cmd) {
        transitionImage(cmd, new_image.image,
VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);

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

```

```

        copyRegion.imageSubresource.aspectMask    =
VK_IMAGE_ASPECT_COLOR_BIT;
        copyRegion.imageSubresource.mipLevel      = 0;
        copyRegion.imageSubresource.baseArrayLayer = 0;
        copyRegion.imageSubresource.layerCount    = 1;
        copyRegion.imageExtent                    = size;
        vkCmdCopyBufferToImage(cmd, uploadbuffer.buffer,
new_image.image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1,
                                &copyRegion);
        if (mipmapped) {
            generateMipmaps(cmd, new_image.image,

            VkExtent2D{new_image.imageExtent.width,
new_image.imageExtent.height});
        } else {
            transitionImage(cmd, new_image.image,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,

            VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
        }
    });
    destroy_buffer(uploadbuffer);
    return new_image;
}

GPUMeshBuffers BlockEngine::uploadMesh(std::span<uint32_t> indices,
std::span<Vertex> vertices) {
    const size_t    vertexBufferSize = vertices.size() *
sizeof(Vertex);
    const size_t    indexBufferSize  = indices.size() *
sizeof(uint32_t);
    GPUMeshBuffers newSurface;
    newSurface.vertexBuffer = create_buffer(vertexBufferSize,

    VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT |

    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT,

    VMA_MEMORY_USAGE_GPU_ONLY);
    VkBufferDeviceAddressInfo deviceAddressInfo{.sType =
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO,

    .buffer = newSurface.vertexBuffer.buffer};

```

```

        newSurface.vertexBufferAddress =
vkGetBufferDeviceAddress(_device, &deviceAdressInfo);
        newSurface.indexBuffer =
            create_buffer(indexBufferSize,
VK_BUFFER_USAGE_INDEX_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
                        VMA_MEMORY_USAGE_GPU_ONLY);

        AllocatedBuffer staging =
            create_buffer(vertexBufferSize + indexBufferSize,
VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VMA_MEMORY_USAGE_CPU_ONLY);
        void* data = staging.allocation->GetMappedData();
        memcpy(data, vertices.data(), vertexBufferSize);
        memcpy((char*)data + vertexBufferSize, indices.data(),
indexBufferSize);
        immediate_submit([&](VkCommandBuffer cmd) {
            VkBufferCopy vertexCopy{0};
            vertexCopy.dstOffset = 0;
            vertexCopy.srcOffset = 0;
            vertexCopy.size = vertexBufferSize;
            vkCmdCopyBuffer(cmd, staging.buffer,
newSurface.vertexBuffer.buffer, 1, &vertexCopy);
            VkBufferCopy indexCopy{0};
            indexCopy.dstOffset = 0;
            indexCopy.srcOffset = vertexBufferSize;
            indexCopy.size = indexBufferSize;
            vkCmdCopyBuffer(cmd, staging.buffer,
newSurface.indexBuffer.buffer, 1, &indexCopy);
        });
        destroy_buffer(staging);
        return newSurface;
    }

FrameData& BlockEngine::get_current_frame() {
    return _frames[_frameNumber % FRAME_OVERLAP];
}

FrameData& BlockEngine::get_last_frame() {
    return _frames[( _frameNumber - 1) % FRAME_OVERLAP];
}

void BlockEngine::immediate_submit(std::function<void(VkCommandBuffer
cmd)>&& function) {
    VK_CHECK(vkResetFences(_device, 1, &_immFence));
    VK_CHECK(vkResetCommandBuffer(_immCommandBuffer, 0));
    VkCommandBuffer cmd = _immCommandBuffer;

```

```

        VkCommandBufferBeginInfo cmdBeginInfo =
command_buffer_begin_info(VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT
);
        VK_CHECK(vkBeginCommandBuffer(cmd, &cmdBeginInfo));
        function(cmd);
        VK_CHECK(vkEndCommandBuffer(cmd));
        VkCommandBufferSubmitInfo cmdinfo =
command_buffer_submit_info(cmd);
        VkSubmitInfo2          submit = submit_info(&cmdinfo,
nullptr, nullptr);
        VK_CHECK(vkQueueSubmit2(_graphicsQueue, 1, &submit,
_immFence));
        VK_CHECK(vkWaitForFences(_device, 1, &_immFence, true,
9'999'999'999));
    }

void BlockEngine::destroy_image(const AllocatedImage& img) {
    vkDestroyImageView(_device, img.imageView, nullptr);
    vmaDestroyImage(_allocator, img.image, img.allocation);
}

void BlockEngine::destroy_buffer(const AllocatedBuffer& buffer) {
    vmaDestroyBuffer(_allocator, buffer.buffer, buffer.allocation);
}

void BlockEngine::init_vulkan() {
    vkb::InstanceBuilder builder;
    auto                inst_ret =
builder.set_app_name("BlockEngine")

        .request_validation_layers(bUseValidationLayers)
                                .use_default_debug_messenger()
                                .require_api_version(1, 3, 0)
                                .build();

    vkb::Instance vkb_inst = inst_ret.value();
    _instance      = vkb_inst.instance;
    _debug_messenger = vkb_inst.debug_messenger;
    SDL_Vulkan_CreateSurface(_window, _instance, &_surface);
    VkPhysicalDeviceVulkan13Features features13{.sType =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_3_FEATURES};
    features13.dynamicRendering = true;
    features13.synchronization2 = true;
    VkPhysicalDeviceVulkan12Features features12{.sType =

```

```

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES};
    features12.bufferDeviceAddress          = true;
    features12.descriptorIndexing            = true;
    features12.descriptorBindingPartiallyBound = true;
    features12.descriptorBindingVariableDescriptorCount = true;
    features12.runtimeDescriptorArray        = true;
    vkb::PhysicalDeviceSelector selector{vkb_inst};
    vkb::PhysicalDevice      physicalDevice =
selector.set_minimum_version(1, 3)

.set_required_features_13(features13)

.set_required_features_12(features12)

.set_surface(_surface)

.select()

.value();
    vkb::DeviceBuilder deviceBuilder{physicalDevice};
    vkb::Device      vkbDevice      =
deviceBuilder.build().value();
    _device          = vkbDevice.device;
    _chosenGPU       =
physicalDevice.physical_device;
    _graphicsQueue   =
vkbDevice.get_queue(vkb::QueueType::graphics).value();
    _graphicsQueueFamily =
vkbDevice.get_queue_index(vkb::QueueType::graphics).value();
    VmaAllocatorCreateInfo allocatorInfo = {};
    allocatorInfo.physicalDevice      = _chosenGPU;
    allocatorInfo.device               = _device;
    allocatorInfo.instance             = _instance;
    allocatorInfo.flags                =
VMA_ALLOCATOR_CREATE_BUFFER_DEVICE_ADDRESS_BIT;
    vmaCreateAllocator(&allocatorInfo, &_allocator);
}

void BlockEngine::init_swapchain() {
    create_swapchain(_windowExtent.width, _windowExtent.height);
    VkExtent3D drawImageExtent = {_windowExtent.width,
_windowExtent.height, 1};
    _drawImage.imageFormat      = VK_FORMAT_R16G16B16A16_SFLOAT;
    _drawImage.imageExtent     = drawImageExtent;

```

```

        VkImageUsageFlags drawImageUsages{};
        drawImageUsages |=
VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
        drawImageUsages |=
VK_IMAGE_USAGE_STORAGE_BIT;
        drawImageUsages |=
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
        VkImageCreateInfo rimg_info =
image_create_info(_drawImage.imageFormat, drawImageUsages,
drawImageExtent);
        VmaAllocationCreateInfo rimg_allocinfo = {};
        rimg_allocinfo.usage =
VMA_MEMORY_USAGE_GPU_ONLY;
        rimg_allocinfo.requiredFlags =
VkMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
        vmaCreateImage(_allocator, &rimg_info, &rimg_allocinfo,
&_drawImage.image, &_drawImage.allocation, nullptr);
        VkImageViewCreateInfo rview_info =
        imageview_create_info(_drawImage.imageFormat,
        _drawImage.image, VK_IMAGE_ASPECT_COLOR_BIT);
        VK_CHECK(vkCreateImageView(_device, &rview_info, nullptr,
&_drawImage.imageView));
        _depthImage.imageFormat = VK_FORMAT_D32_SFLOAT;
        _depthImage.imageExtent = drawImageExtent;
        VkImageUsageFlags depthImageUsages{};
        depthImageUsages |=
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
        VkImageCreateInfo dimg_info =
image_create_info(_depthImage.imageFormat, depthImageUsages,
drawImageExtent);
        vmaCreateImage(_allocator, &dimg_info, &rimg_allocinfo,
&_depthImage.image, &_depthImage.allocation, nullptr);
        VkImageViewCreateInfo dview_info =
        imageview_create_info(_depthImage.imageFormat,
        _depthImage.image, VK_IMAGE_ASPECT_DEPTH_BIT);
        VK_CHECK(vkCreateImageView(_device, &dview_info, nullptr,
&_depthImage.imageView));
        _mainDeletionQueue.push_function([=]() {
            vkDestroyImageView(_device, _drawImage.imageView,
nullptr);
            vmaDestroyImage(_allocator, _drawImage.image,
_drawImage.allocation);
            vkDestroyImageView(_device, _depthImage.imageView,
nullptr);

```

```

        vmaDestroyImage(_allocator, _depthImage.image,
        _depthImage.allocation);
    });
}

void BlockEngine::create_swapchain(uint32_t width, uint32_t height) {
    vkb::SwapchainBuilder swapchainBuilder{_chosenGPU, _device,
    _surface};
    _swapchainImageFormat = VK_FORMAT_B8G8R8A8_UNORM;
    vkb::Swapchain vkbSwapchain =
        swapchainBuilder
            .set_desired_format(
                VkSurfaceFormatKHR{.format =
                _swapchainImageFormat, .colorSpace =
                VK_COLOR_SPACE_SRGB_NONLINEAR_KHR})
            .set_desired_present_mode(VK_PRESENT_MODE_FIFO_KHR)
            .set_desired_extent(width, height)

            .add_image_usage_flags(VK_IMAGE_USAGE_TRANSFER_DST_BIT)
            .build()
            .value();
    _swapchainExtent = vkbSwapchain.extent;
    _swapchain = vkbSwapchain.swapchain;
    _swapchainImages = vkbSwapchain.get_images().value();
    _swapchainImageViews = vkbSwapchain.get_image_views().value();
}

void BlockEngine::destroy_swapchain() {
    vkDestroySwapchainKHR(_device, _swapchain, nullptr);
    for (int i = 0; i < _swapchainImageViews.size(); i++) {
        vkDestroyImageView(_device, _swapchainImageViews[i],
        nullptr);
    }
}

void BlockEngine::resize_swapchain() {
    vkDeviceWaitIdle(_device);
    destroy_swapchain();
    int w, h;
    SDL_GetWindowSize(_window, &w, &h);
    _windowExtent.width = w;
    _windowExtent.height = h;
    create_swapchain(_windowExtent.width, _windowExtent.height);
    resize_requested = false;
}

```

```

void BlockEngine::init_commands() {
    VkCommandPoolCreateInfo commandPoolInfo =
        command_pool_create_info(_graphicsQueueFamily,
VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT);
    for (int i = 0; i < FRAME_OVERLAP; i++) {
        VK_CHECK(vkCreateCommandPool(_device, &commandPoolInfo,
nullptr, &_frames[i]._commandPool));
        VkCommandBufferAllocateInfo cmdAllocInfo =
command_buffer_allocate_info(_frames[i]._commandPool, 1);
        VK_CHECK(vkAllocateCommandBuffers(_device, &cmdAllocInfo,
&_frames[i]._mainCommandBuffer));
        _mainDeletionQueue.push_function([=]() {
vkDestroyCommandPool(_device, _frames[i]._commandPool, nullptr); });
    }
    VK_CHECK(vkCreateCommandPool(_device, &commandPoolInfo,
nullptr, &_immCommandPool));
    VkCommandBufferAllocateInfo cmdAllocInfo =
command_buffer_allocate_info(_immCommandPool, 1);
    VK_CHECK(vkAllocateCommandBuffers(_device, &cmdAllocInfo,
&_immCommandBuffer));
    _mainDeletionQueue.push_function([=]() {
vkDestroyCommandPool(_device, _immCommandPool, nullptr); });
}

void BlockEngine::init_sync_structures() {
    VkFenceCreateInfo fenceCreateInfo =
fence_create_info(VK_FENCE_CREATE_SIGNALED_BIT);
    VK_CHECK(vkCreateFence(_device, &fenceCreateInfo, nullptr,
&_immFence));
    _mainDeletionQueue.push_function([=]() {
vkDestroyFence(_device, _immFence, nullptr); });
    for (int i = 0; i < FRAME_OVERLAP; i++) {
        VK_CHECK(vkCreateFence(_device, &fenceCreateInfo, nullptr,
&_frames[i]._renderFence));
        VkSemaphoreCreateInfo semaphoreCreateInfo =
semaphore_create_info();
        VK_CHECK(vkCreateSemaphore(_device, &semaphoreCreateInfo,
nullptr, &_frames[i]._swapchainSemaphore));
        VK_CHECK(vkCreateSemaphore(_device, &semaphoreCreateInfo,
nullptr, &_frames[i]._renderSemaphore));
        _mainDeletionQueue.push_function([=]() {
            vkDestroyFence(_device, _frames[i]._renderFence,
nullptr);
        });
    }
}

```

```

        vkDestroySemaphore(_device,
        _frames[i]._swapchainSemaphore, nullptr);
        vkDestroySemaphore(_device,
        _frames[i]._renderSemaphore, nullptr);
    });
}

}

void BlockEngine::init_renderables() {}

void BlockEngine::init_imgui() {
    VkDescriptorPoolSize pool_sizes[] = {
        {VK_DESCRIPTOR_TYPE_SAMPLER, 1000},
        {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1000},
        {VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 1000},
        {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 1000},
        {VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 1000},
        {VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, 1000},
        {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1000},
        {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1000},
        {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 1000},
        {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, 1000},
        {VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, 1000}
    };

    VkDescriptorPoolCreateInfo pool_info = {};
    pool_info.sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    pool_info.flags =
VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT;
    pool_info.maxSets = 1000;
    pool_info.poolSizeCount =
(uint32_t)std::size(pool_sizes);
    pool_info.pPoolSizes = pool_sizes;

    VkDescriptorPool imguiPool;
    VK_CHECK(vkCreateDescriptorPool(_device, &pool_info, nullptr,
&imguiPool));
    ImGui::CreateContext();
    ImGui_ImplSDL2_InitForVulkan(_window);
    ImGui_ImplVulkan_InitInfo init_info = {};
    init_info.Instance = _instance;
    init_info.PhysicalDevice = _chosenGPU;
    init_info.Device = _device;

```

```

        init_info.Queue = _graphicsQueue;
        init_info.DescriptorPool = imguiPool;
        init_info.MinImageCount = 3;
        init_info.ImageCount = 3;
        init_info.UseDynamicRendering = true;
        init_info.PipelineRenderingCreateInfo = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_RENDERING_CREATE_INFO};
        init_info.PipelineRenderingCreateInfo.colorAttachmentCount =
1;
        init_info.PipelineRenderingCreateInfo.pColorAttachmentFormats =
&_swapchainImageFormat;
        init_info.MSAASamples =
VK_SAMPLE_COUNT_1_BIT;

        ImGui_ImplVulkan_Init(&init_info);
        ImGui_ImplVulkan_CreateFontsTexture();
        _mainDeletionQueue.push_function([=]() {
            ImGui_ImplVulkan_Shutdown();
            vkDestroyDescriptorPool(_device, imguiPool, nullptr);
        });
    }

void BlockEngine::init_pipelines() {
    init_background_pipelines();
    metalRoughMaterial.build_pipelines(this);
}

void BlockEngine::init_descriptors() {
    std::vector<DescriptorAllocator::PoolSizeRatio> sizes = {
        {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 3},
        {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 3},
        {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 3},
    };
    globalDescriptorAllocator.initPool(_device, 10, sizes);
    _mainDeletionQueue.push_function(
        [&]() { vkDestroyDescriptorPool(_device,
globalDescriptorAllocator.pool, nullptr); });
    {
        DescriptorLayoutBuilder builder;
        builder.addBinding(0, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE);
        _drawImageDescriptorLayout = builder.build(_device,
VK_SHADER_STAGE_COMPUTE_BIT);
    }
    {

```

```

        DescriptorLayoutBuilder builder;
        builder.addBinding(0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER);
        builder.addBinding(1,
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER);
        VkDescriptorSetLayoutBindingFlagsCreateInfo bindFlags = {
            .sType =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO,
            .pNext = nullptr};
        std::array<VkDescriptorBindingFlags, 2> flagArray{0,
VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT |

            VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT};
        builder.bindings[1].descriptorCount = 64;
        bindFlags.bindingCount = 2;
        bindFlags.pBindingFlags = flagArray.data();
        _gpuSceneDataDescriptorLayout =
            builder.build(_device, VK_SHADER_STAGE_VERTEX_BIT |
VK_SHADER_STAGE_FRAGMENT_BIT, &bindFlags);
    }
    _mainDeletionQueue.push_function([&]() {
        vkDestroyDescriptorSetLayout(_device,
_drawImageDescriptorLayout, nullptr);
        vkDestroyDescriptorSetLayout(_device,
_gpuSceneDataDescriptorLayout, nullptr);
    });
    _drawImageDescriptors =
globalDescriptorAllocator.allocate(_device,
_drawImageDescriptorLayout);
    {
        DescriptorWriter writer;
        writer.writeImage(0, _drawImage.imageView, VK_NULL_HANDLE,
VK_IMAGE_LAYOUT_GENERAL,
            VK_DESCRIPTOR_TYPE_STORAGE_IMAGE);
        writer.update_set(_device, _drawImageDescriptors);
    }
    for (int i = 0; i < FRAME_OVERLAP; i++) {
        std::vector<DescriptorAllocatorGrowable::PoolSizeRatio>
frame_sizes = {
            {VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 3},
            {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 3},
            {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 3},
            {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 4},
        };
        _frames[i]._frameDescriptors =

```

```

DescriptorAllocatorGrowable{};
    _frames[i]._frameDescriptors.init(_device, 1000,
frame_sizes);
    _mainDeletionQueue.push_function([&, i]() {
_frames[i]._frameDescriptors.destroyPool(_device); });
    }
}

void GLTFMetallic_Roughness::build_pipelines(BlockEngine* engine) {
    VkShaderModule meshFragShader;
    if (!vkutil::load_shader_module("fragment.spv", engine->_device, &meshFragShader)) {
        BK_ERROR("Error when building the triangle fragment shader module");
    }
    VkShaderModule meshVertexShader;
    if (!vkutil::load_shader_module("vertex.spv", engine->_device, &meshVertexShader)) {
        BK_ERROR("Error when building the triangle vertex shader module");
    }
    VkPushConstantRange matrixRange{};
    matrixRange.offset = 0;
    matrixRange.size = sizeof(GPUDrawPushConstants);
    matrixRange.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    DescriptorLayoutBuilder layoutBuilder;
    layoutBuilder.addBinding(0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER);
    materialLayout = layoutBuilder.build(engine->_device,
VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT);
    VkDescriptorSetLayout layouts[] = {engine->_gpuSceneDataDescriptorLayout, materialLayout};
    VkPipelineLayoutCreateInfo mesh_layout_info =
pipeline_layout_create_info();
    mesh_layout_info.setLayoutCount = 2;
    mesh_layout_info.pSetLayouts = layouts;
    mesh_layout_info.pPushConstantRanges = &matrixRange;
    mesh_layout_info.pushConstantRangeCount = 1;
    VkPipelineLayout newLayout;
    VK_CHECK(vkCreatePipelineLayout(engine->_device,
&mesh_layout_info, nullptr, &newLayout));
    opaquePipeline.layout = newLayout;
    transparentPipeline.layout = newLayout;
    PipelineBuilder pipelineBuilder;
    pipelineBuilder.set_shaders(meshVertexShader, meshFragShader);

```

```

        pipelineBuilder.set_input_topology(VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST);
        pipelineBuilder.set_polygon_mode(VK_POLYGON_MODE_FILL);
        pipelineBuilder.set_cull_mode(VK_CULL_MODE_NONE,
VK_FRONT_FACE_CLOCKWISE);
        pipelineBuilder.set_multisampling_none();
        pipelineBuilder.disable_blending();
        pipelineBuilder.enable_depthtest(true,
VK_COMPARE_OP_GREATER_OR_EQUAL);
        pipelineBuilder.set_color_attachment_format(engine->_drawImage.imageFormat);
        pipelineBuilder.set_depth_format(engine->_depthImage.imageFormat);
        pipelineBuilder.pipelineLayout = newLayout;
        opaquePipeline.pipeline =
pipelineBuilder.build_pipeline(engine->_device);
        pipelineBuilder.enable_blending_additive();
        pipelineBuilder.enable_depthtest(false,
VK_COMPARE_OP_GREATER_OR_EQUAL);
        transparentPipeline.pipeline =
pipelineBuilder.build_pipeline(engine->_device);
        vkDestroyShaderModule(engine->_device, meshFragShader,
nullptr);
        vkDestroyShaderModule(engine->_device, meshVertexShader,
nullptr);
    }

```

```

void GLTFMetallic_Roughness::clear_resources(VkDevice device) {}

```

```

MaterialInstance GLTFMetallic_Roughness::write_material(VkDevice
device, MaterialPass pass,

```

```

        const MaterialResources& resources,

        DescriptorAllocatorGrowable& descriptorAllocator) {
    MaterialInstance matData;
    matData.passType = pass;
    if (pass == MaterialPass::Transparent) {
        matData.pipeline = &transparentPipeline;
    } else {
        matData.pipeline = &opaquePipeline;
    }
    matData.materialSet = descriptorAllocator.allocate(device,
materialLayout);

```

```

        writer.clear();
        writer.writeBuffer(0, resources.dataBuffer,
sizeof(MaterialConstants), resources.dataBufferOffset,
                                VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER);
        writer.update_set(device, matData.materialSet);
        return matData;
    }
    void MeshNode::Draw(const glm::mat4& topMatrix, DrawContext& ctx) {
        refreshTransform(glm::mat4{1.f});
        glm::mat4 nodeMatrix = topMatrix * worldTransform;
        for (auto& s : mesh->surfaces) {
            RenderObject def;
            def.indexCount          = s.count;
            def.firstIndex          = s.startIndex;
            def.indexBuffer          = mesh-
>meshBuffers.indexBuffer.buffer;
            def.material             = &s.material->data;
            def.bounds               = s.bounds;
            def.transform            = nodeMatrix;
            def.vertexBufferAddress = mesh-
>meshBuffers.vertexBufferAddress;
            if (s.material->data.passType ==
MaterialPass::Transparent) {
                ctx.TransparentSurfaces.push_back(def);
            } else {
                ctx.OpaqueSurfaces.push_back(def);
            }
        }
        Node::Draw(topMatrix, ctx);
    }

TextureID TextureCache::AddTexture(const VkImageView& image,
VkSampler sampler) {
    for (unsigned int i = 0; i < Cache.size(); i++) {
        if (Cache[i].imageView == image && Cache[i].sampler ==
sampler) {
            return TextureID{i};
        }
    }
    uint32_t idx = Cache.size();
    Cache.push_back(VkDescriptorImageInfo{.sampler      = sampler,
                                           .imageView    = image,
                                           .imageLayout  =

```

```
= VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL});  
    return TextureID{idx};  
}
```

## Файл images.cpp

```
#include <vulkan/vulkan.h>

void transitionImage(VkCommandBuffer cmd, VkImage image,
VkImageLayout currentLayout, VkImageLayout newLayout);
void copyImageToImage(VkCommandBuffer cmd, VkImage source, VkImage
destination, VkExtent2D srcSize,
                        VkExtent2D dstSize);
void generateMipmaps(VkCommandBuffer cmd, VkImage image, VkExtent2D
imageSize);

#include <images.h>
#include <initializers.h>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#include <initializers.h>

void transitionImage(VkCommandBuffer cmd, VkImage image,
VkImageLayout currentLayout, VkImageLayout newLayout) {
    VkImageMemoryBarrier2 imageBarrier{.sType =
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2};
    imageBarrier.pNext          = nullptr;
    imageBarrier.srcStageMask   =
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT;
    imageBarrier.srcAccessMask = VK_ACCESS_2_MEMORY_WRITE_BIT;
    imageBarrier.dstStageMask   =
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT;
    imageBarrier.dstAccessMask = VK_ACCESS_2_MEMORY_WRITE_BIT |
VK_ACCESS_2_MEMORY_READ_BIT;
    imageBarrier.oldLayout      = currentLayout;
    imageBarrier.newLayout      = newLayout;
    VkImageAspectFlags aspectMask =
        (newLayout == VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL) ?
VK_IMAGE_ASPECT_DEPTH_BIT : VK_IMAGE_ASPECT_COLOR_BIT;
    imageBarrier.subresourceRange =
image_subresource_range(aspectMask);
    imageBarrier.image            = image;
    VkDependencyInfo depInfo{};
    depInfo.sType                =
VK_STRUCTURE_TYPE_DEPENDENCY_INFO;
    depInfo.pNext                = nullptr;
```

```

        depInfo.imageMemoryBarrierCount = 1;
        depInfo.pImageMemoryBarriers    = &imageBarrier;
        vkCmdPipelineBarrier2(cmd, &depInfo);
    }

void copyImageToImage(VkCommandBuffer cmd, VkImage source, VkImage
destination, VkExtent2D srcSize,
                    VkExtent2D dstSize) {
    VkImageBlit2 blitRegion{.sType =
VK_STRUCTURE_TYPE_IMAGE_BLIT_2, .pNext = nullptr};
    blitRegion.srcOffsets[1].x      = srcSize.width;
    blitRegion.srcOffsets[1].y      = srcSize.height;
    blitRegion.srcOffsets[1].z      = 1;
    blitRegion.dstOffsets[1].x      = dstSize.width;
    blitRegion.dstOffsets[1].y      = dstSize.height;
    blitRegion.dstOffsets[1].z      = 1;
    blitRegion.srcSubresource.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;
    blitRegion.srcSubresource.baseArrayLayer = 0;
    blitRegion.srcSubresource.layerCount    = 1;
    blitRegion.srcSubresource.mipLevel      = 0;
    blitRegion.dstSubresource.aspectMask    =
VK_IMAGE_ASPECT_COLOR_BIT;
    blitRegion.dstSubresource.baseArrayLayer = 0;
    blitRegion.dstSubresource.layerCount    = 1;
    blitRegion.dstSubresource.mipLevel      = 0;
    VkBlitImageInfo2 blitInfo{.sType =
VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2, .pNext = nullptr};
    blitInfo.dstImage      = destination;
    blitInfo.dstImageLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    blitInfo.srcImage      = source;
    blitInfo.srcImageLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    blitInfo.filter        = VK_FILTER_LINEAR;
    blitInfo.regionCount   = 1;
    blitInfo.pRegions      = &blitRegion;
    vkCmdBlitImage2(cmd, &blitInfo);
}

void generateMipmaps(VkCommandBuffer cmd, VkImage image, VkExtent2D
imageSize) {
    int mipLevels =
int(std::floor(std::log2(std::max(imageSize.width,
imageSize.height)))) + 1;
    for (int mip = 0; mip < mipLevels; mip++) {

```

```

        VkExtent2D halfSize = imageSize;
        halfSize.width /= 2;
        halfSize.height /= 2;
        VkImageMemoryBarrier2 imageBarrier{.sType =
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2, .pNext = nullptr};
        imageBarrier.srcStageMask =
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT;
        imageBarrier.srcAccessMask =
VK_ACCESS_2_MEMORY_WRITE_BIT;
        imageBarrier.dstStageMask =
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT;
        imageBarrier.dstAccessMask =
VK_ACCESS_2_MEMORY_WRITE_BIT | VK_ACCESS_2_MEMORY_READ_BIT;
        imageBarrier.oldLayout =
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
        imageBarrier.newLayout =
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
        VkImageAspectFlags aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;
        imageBarrier.subresourceRange =
image_subresource_range(aspectMask);
        imageBarrier.subresourceRange.levelCount = 1;
        imageBarrier.subresourceRange.baseMipLevel = mip;
        imageBarrier.image = image;
        VkDependencyInfo depInfo{.sType =
VK_STRUCTURE_TYPE_DEPENDENCY_INFO, .pNext = nullptr};
        depInfo.imageMemoryBarrierCount = 1;
        depInfo.pImageMemoryBarriers = &imageBarrier;
        vkCmdPipelineBarrier2(cmd, &depInfo);
        if (mip < mipLevels - 1) {
            VkImageBlit2 blitRegion{.sType =
VK_STRUCTURE_TYPE_IMAGE_BLIT_2, .pNext = nullptr};
            blitRegion.srcOffsets[1].x =
imageSize.width;
            blitRegion.srcOffsets[1].y =
imageSize.height;
            blitRegion.srcOffsets[1].z = 1;
            blitRegion.dstOffsets[1].x =
halfSize.width;
            blitRegion.dstOffsets[1].y =
halfSize.height;
            blitRegion.dstOffsets[1].z = 1;
            blitRegion.srcSubresource.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;

```

```

        blitRegion.srcSubresource.baseArrayLayer = 0;
        blitRegion.srcSubresource.layerCount     = 1;
        blitRegion.srcSubresource.mipLevel       = mip;
        blitRegion.dstSubresource.aspectMask     =
VK_IMAGE_ASPECT_COLOR_BIT;
        blitRegion.dstSubresource.baseArrayLayer = 0;
        blitRegion.dstSubresource.layerCount     = 1;
        blitRegion.dstSubresource.mipLevel       = mip + 1;
        VkBlitImageInfo2 blitInfo{.sType =
VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2, .pNext = nullptr};
        blitInfo.dstImage           = image;
        blitInfo.dstImageLayout     =
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
        blitInfo.srcImage           = image;
        blitInfo.srcImageLayout     =
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
        blitInfo.filter              = VK_FILTER_LINEAR;
        blitInfo.regionCount         = 1;
        blitInfo.pRegions            = &blitRegion;
        vkCmdBlitImage2(cmd, &blitInfo);
        imageSize = halfSize;
    }
}
    transitionImage(cmd, image,
VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
}

```

## Файл initializers.h

```
#pragma once

#include <types.h>

VkCommandPoolCreateInfo      command_pool_create_info(uint32_t
queueFamilyIndex, VkCommandPoolCreateFlags flags = 0);
VkCommandBufferAllocateInfo
command_buffer_allocate_info(VkCommandPool pool, uint32_t count = 1);
VkCommandBufferBeginInfo
command_buffer_begin_info(VkCommandBufferUsageFlags flags = 0);
VkCommandBufferSubmitInfo
command_buffer_submit_info(VkCommandBuffer cmd);
VkFenceCreateInfo           fence_create_info(VkFenceCreateFlags
flags = 0);
VkSemaphoreCreateInfo
semaphore_create_info(VkSemaphoreCreateFlags flags = 0);
VkSubmitInfo2
submit_info(VkCommandBufferSubmitInfo* cmd, VkSemaphoreSubmitInfo*
signalSemaphoreInfo,

VkSemaphoreSubmitInfo* waitSemaphoreInfo);
VkPresentInfoKHR           present_info();
VkRenderingAttachmentInfo   attachment_info(VkImageView view,
VkClearColor* clear, VkImageLayout layout);
VkRenderingAttachmentInfo   depth_attachment_info(VkImageView
view, VkImageLayout layout);
VkRenderingInfo             rendering_info(VkExtent2D
renderExtent, VkRenderingAttachmentInfo* colorAttachment,

VkRenderingAttachmentInfo* depthAttachment);
VkImageSubresourceRange
image_subresource_range(VkImageAspectFlags aspectMask);
VkSemaphoreSubmitInfo
semaphore_submit_info(VkPipelineStageFlags2 stageMask, VkSemaphore
semaphore);
VkDescriptorSetLayoutBinding
descriptorset_layout_binding(VkDescriptorType type,
VkShaderStageFlags stageFlags,

uint32_t binding);
VkDescriptorSetLayoutCreateInfo
descriptorset_layout_create_info(VkDescriptorSetLayoutBinding*
```

```

bindings,

                                uint32_t                                bindingCount);
VkWriteDescriptorSet
write_descriptor_image(VkDescriptorType type, VkDescriptorSet dstSet,

                        VkDescriptorImageInfo* imageInfo, uint32_t binding);
VkWriteDescriptorSet
write_descriptor_buffer(VkDescriptorType type, VkDescriptorSet
dstSet,

                        VkDescriptorBufferInfo* bufferInfo, uint32_t binding);
VkDescriptorBufferInfo
buffer_info(VkBuffer buffer,
VkDeviceSize offset, VkDeviceSize range);
VkImageCreateInfo
image_create_info(VkFormat format,
VkImageUsageFlags usageFlags, VkExtent3D extent);
VkImageViewCreateInfo
imageview_create_info(VkFormat
format, VkImage image, VkImageAspectFlags aspectFlags);
VkPipelineLayoutCreateInfo
pipeline_layout_create_info();
VkPipelineShaderStageCreateInfo
pipeline_shader_stage_create_info(VkShaderStageFlagBits stage,

                                VkShaderModule                shaderModule,

                                const char*                    entry = "main");

```

## Файл initializers.cpp

```
#include <initializers.h>

VkCommandPoolCreateInfo command_pool_create_info(uint32_t
queueFamilyIndex, VkCommandPoolCreateFlags flags) {
    VkCommandPoolCreateInfo info = {};
    info.sType
        =
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    info.pNext
        = nullptr;
    info.queueFamilyIndex
        = queueFamilyIndex;
    info.flags
        = flags;
    return info;
}

VkCommandBufferAllocateInfo
command_buffer_allocate_info(VkCommandPool pool, uint32_t count) {
    VkCommandBufferAllocateInfo info = {};
    info.sType
        =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    info.pNext
        = nullptr;
    info.commandPool
        = pool;
    info.commandBufferCount
        = count;
    info.level
        =
VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    return info;
}

VkCommandBufferBeginInfo
command_buffer_begin_info(VkCommandBufferUsageFlags flags) {
    VkCommandBufferBeginInfo info = {};
    info.sType
        =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    info.pNext
        = nullptr;
    info.pInheritanceInfo
        = nullptr;
    info.flags
        = flags;
    return info;
}

VkFenceCreateInfo fence_create_info(VkFenceCreateFlags flags) {
    VkFenceCreateInfo info = {};
    info.sType
        = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    info.pNext
        = nullptr;
    info.flags
        = flags;
```

```

        return info;
    }

VkSemaphoreCreateInfo semaphore_create_info(VkSemaphoreCreateFlags
flags) {
    VkSemaphoreCreateInfo info = {};
    info.sType =
VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    info.pNext = nullptr;
    info.flags = flags;
    return info;
}

VkSemaphoreSubmitInfo semaphore_submit_info(VkPipelineStageFlags2
stageMask, VkSemaphore semaphore) {
    VkSemaphoreSubmitInfo submitInfo{};
    submitInfo.sType =
VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO;
    submitInfo.pNext = nullptr;
    submitInfo.semaphore = semaphore;
    submitInfo.stageMask = stageMask;
    submitInfo.deviceIndex = 0;
    submitInfo.value = 1;
    return submitInfo;
}

VkCommandBufferSubmitInfo command_buffer_submit_info(VkCommandBuffer
cmd) {
    VkCommandBufferSubmitInfo info{};
    info.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO;
    info.pNext = nullptr;
    info.commandBuffer = cmd;
    info.deviceMask = 0;
    return info;
}

VkSubmitInfo2 submit_info(VkCommandBufferSubmitInfo* cmd,
VkSemaphoreSubmitInfo* signalSemaphoreInfo,
                                VkSemaphoreSubmitInfo*
waitSemaphoreInfo) {
    VkSubmitInfo2 info = {};
    info.sType =
VK_STRUCTURE_TYPE_SUBMIT_INFO_2;

```

```

        info.pNext = nullptr;
        info.waitSemaphoreInfoCount = waitSemaphoreInfo == nullptr ?
0 : 1;
        info.pWaitSemaphoreInfos = waitSemaphoreInfo;
        info.signalSemaphoreInfoCount = signalSemaphoreInfo == nullptr
? 0 : 1;
        info.pSignalSemaphoreInfos = signalSemaphoreInfo;
        info.commandBufferInfoCount = 1;
        info.pCommandBufferInfos = cmd;
        return info;
}

```

```

VkPresentInfoKHR present_info() {
    VkPresentInfoKHR info = {};
    info.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
    info.pNext = 0;
    info.swapchainCount = 0;
    info.pSwapchains = nullptr;
    info.pWaitSemaphores = nullptr;
    info.waitSemaphoreCount = 0;
    info.pImageIndices = nullptr;
    return info;
}

```

```

VkRenderingAttachmentInfo attachment_info(VkImageView view,
VkClearColor* clear, VkImageLayout layout) {
    VkRenderingAttachmentInfo colorAttachment{};
    colorAttachment.sType =
VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO;
    colorAttachment.pNext = nullptr;
    colorAttachment.imageView = view;
    colorAttachment.imageLayout = layout;
    colorAttachment.loadOp = clear ?
VK_ATTACHMENT_LOAD_OP_CLEAR : VK_ATTACHMENT_LOAD_OP_LOAD;
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    if (clear) {
        colorAttachment.clearValue = *clear;
    }
    return colorAttachment;
}

```

```

VkRenderingAttachmentInfo depth_attachment_info(VkImageView view,
VkImageLayout layout) {
    VkRenderingAttachmentInfo depthAttachment{};

```

```

        depthAttachment.sType =
VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO;
        depthAttachment.pNext = nullptr;
        depthAttachment.imageView = view;
        depthAttachment.imageLayout = layout;
        depthAttachment.loadOp =
VK_ATTACHMENT_LOAD_OP_CLEAR;
        depthAttachment.storeOp =
VK_ATTACHMENT_STORE_OP_STORE;
        depthAttachment.clearValue.depthStencil.depth = 0.f;
        return depthAttachment;
    }

VkRenderingInfo rendering_info(VkExtent2D renderExtent,
VkRenderingAttachmentInfo* colorAttachment,
                                VkRenderingAttachmentInfo*
depthAttachment) {
    VkRenderingInfo renderInfo{};
    renderInfo.sType = VK_STRUCTURE_TYPE_RENDERING_INFO;
    renderInfo.pNext = nullptr;
    renderInfo.renderArea = VkRect2D{
        VkOffset2D{0, 0},
        renderExtent
    };
    renderInfo.layerCount = 1;
    renderInfo.colorAttachmentCount = 1;
    renderInfo.pColorAttachments = colorAttachment;
    renderInfo.pDepthAttachment = depthAttachment;
    renderInfo.pStencilAttachment = nullptr;
    return renderInfo;
}

VkImageSubresourceRange image_subresource_range(VkImageAspectFlags
aspectMask) {
    VkImageSubresourceRange subImage{};
    subImage.aspectMask = aspectMask;
    subImage.baseMipLevel = 0;
    subImage.levelCount = VK_REMAINING_MIP_LEVELS;
    subImage.baseArrayLayer = 0;
    subImage.layerCount = VK_REMAINING_ARRAY_LAYERS;
    return subImage;
}

VkDescriptorSetLayoutBinding

```

```

descriptorset_layout_binding(VkDescriptorType type,
VkShaderStageFlags stageFlags,

    uint32_t binding) {
    VkDescriptorSetLayoutBinding setbind = {};
    setbind.binding                    = binding;
    setbind.descriptorCount           = 1;
    setbind.descriptorType            = type;
    setbind.pImmutableSamplers        = nullptr;
    setbind.stageFlags                = stageFlags;
    return setbind;
}

VkDescriptorSetLayoutCreateInfo
descriptorset_layout_create_info(VkDescriptorSetLayoutBinding*
bindings,

    uint32_t bindingCount) {
    VkDescriptorSetLayoutCreateInfo info = {};
    info.sType                        =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    info.pNext                        = nullptr;
    info.pBindings                    = bindings;
    info.bindingCount                 = bindingCount;
    info.flags                        = 0;
    return info;
}

VkWriteDescriptorSet write_descriptor_image(VkDescriptorType type,
VkDescriptorSet dstSet,

    VkDescriptorImageInfo* imageInfo, uint32_t binding) {
    VkWriteDescriptorSet write = {};
    write.sType                  =
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    write.pNext                  = nullptr;
    write.dstBinding              = binding;
    write.dstSet                  = dstSet;
    write.descriptorCount         = 1;
    write.descriptorType          = type;
    write.pImageInfo              = imageInfo;
    return write;
}

```

```
VkWriteDescriptorSet write_descriptor_buffer(VkDescriptorType type,
VkDescriptorSet dstSet,
```

```
VkDescriptorBufferInfo* bufferInfo, uint32_t binding) {
    VkWriteDescriptorSet write = {};
    write.sType =
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    write.pNext = nullptr;
    write.dstBinding = binding;
    write.dstSet = dstSet;
    write.descriptorCount = 1;
    write.descriptorType = type;
    write.pBufferInfo = bufferInfo;
    return write;
}
```

```
VkDescriptorBufferInfo buffer_info(VkBuffer buffer, VkDeviceSize
offset, VkDeviceSize range) {
    VkDescriptorBufferInfo binfo{};
    binfo.buffer = buffer;
    binfo.offset = offset;
    binfo.range = range;
    return binfo;
}
```

```
VkImageCreateInfo image_create_info(VkFormat format,
VkImageUsageFlags usageFlags, VkExtent3D extent) {
    VkImageCreateInfo info = {};
    info.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    info.pNext = nullptr;
    info.imageType = VK_IMAGE_TYPE_2D;
    info.format = format;
    info.extent = extent;
    info.mipLevels = 1;
    info.arrayLayers = 1;
    info.samples = VK_SAMPLE_COUNT_1_BIT;
    info.tiling = VK_IMAGE_TILING_OPTIMAL;
    info.usage = usageFlags;
    return info;
}
```

```
VkImageViewCreateInfo imageview_create_info(VkFormat format, VkImage
image, VkImageAspectFlags aspectFlags) {
    VkImageViewCreateInfo info = {};
```

```

        info.sType =
VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        info.pNext = nullptr;
        info.viewType = VK_IMAGE_VIEW_TYPE_2D;
        info.image = image;
        info.format = format;
        info.subresourceRange.baseMipLevel = 0;
        info.subresourceRange.levelCount = 1;
        info.subresourceRange.baseArrayLayer = 0;
        info.subresourceRange.layerCount = 1;
        info.subresourceRange.aspectMask = aspectFlags;
        return info;
    }

VkPipelineLayoutCreateInfo pipeline_layout_create_info() {
    VkPipelineLayoutCreateInfo info{};
    info.sType =
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    info.pNext = nullptr;
    info.flags = 0;
    info.setLayoutCount = 0;
    info.pSetLayouts = nullptr;
    info.pushConstantRangeCount = 0;
    info.pPushConstantRanges = nullptr;
    return info;
}

VkPipelineShaderStageCreateInfo
pipeline_shader_stage_create_info(VkShaderStageFlagBits stage,

                                   VkShaderModule shaderModule, const char*
entry) {
    VkPipelineShaderStageCreateInfo info{};
    info.sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    info.pNext = nullptr;
    info.stage = stage;
    info.module = shaderModule;
    info.pName = entry;
    return info;
}

```

## Файл loader.h

```
#pragma once

#include <types.h>

#include "descriptors.h"
#include <filesystem>
#include <unordered_map>

#include <Jolt/Jolt.h>

#include <Jolt/Core/Factory.h>
#include <Jolt/Core/JobSystemThreadPool.h>
#include <Jolt/Core/TempAllocator.h>
#include <Jolt/Physics/Body/BodyActivationListener.h>
#include <Jolt/Physics/Body/BodyCreationSettings.h>
#include <Jolt/Physics/Collision/Shape/BoxShape.h>
#include <Jolt/Physics/Collision/Shape/SphereShape.h>
#include <Jolt/Physics/PhysicsSettings.h>
#include <Jolt/Physics/PhysicsSystem.h>

class BlockEngine;
struct TRS {
    glm::vec3 translation;
    glm::vec3 rotation;
    glm::vec3 scale;
};

struct Bounds {
    glm::vec3 origin;
    float sphereRadius;
    glm::vec3 extents;
};

struct GLTFMaterial {
    MaterialInstance data;
};

struct GeoSurface {
    uint32_t startIndex;
    uint32_t count;
    Bounds bounds;
    std::shared_ptr<GLTFMaterial> material;
```

```

};

struct MeshAsset {
    std::string name;

    std::vector<GeoSurface> surfaces;
    GPUMeshBuffers          meshBuffers;
};

struct LoadedGLTF: public IRenderable {
    std::unordered_map<std::string, std::shared_ptr<MeshAsset>>
meshes;
    std::unordered_map<std::string, std::shared_ptr<Node>>
nodes;
    std::unordered_map<std::string, AllocatedImage>
images;
    std::unordered_map<std::string, std::shared_ptr<GLTFMaterial>>
materials;
    std::vector<std::shared_ptr<Node>>
topNodes;
    std::vector<VkSampler>
samplers;
    DescriptorAllocatorGrowable
descriptorPool;
    AllocatedBuffer
materialDataBuffer;
    BlockEngine*
creator;
    std::string
uuid;
    std::string
path;
    std::string
name;

    virtual void Draw(const glm::mat4& topMatrix, DrawContext&
ctx);
private:
    void clearAll();
};

std::optional<std::shared_ptr<LoadedGLTF>> loadGltf(BlockEngine*
engine, std::string_view filePath, TRS& trsOut);

```

## Файл loader.cpp

```
#include "stb_image.h"
#include <loader.h>

#include "logger.h"

#include "engine.h"
#include "initializers.h"
#include "types.h"
#include <glm/gtx/quaternion.hpp>

#include <fastgltf/glm_element_traits.hpp>
#include <fastgltf/parser.hpp>
#include <fastgltf/tools.hpp>
#include <fastgltf/util.hpp>

std::optional<AllocatedImage> load_image(BlockEngine* engine,
fastgltf::Asset& asset, fastgltf::Image& image) {
    AllocatedImage newImage{};

    int width, height, nrChannels;

    std::visit(fastgltf::visitor{
        [](auto& arg) {},
        [&](fastgltf::sources::URI& filePath) {
            assert(filePath.fileByteOffset == 0);
            assert(filePath.uri.isLocalPath());
            const std::string
path(filePath.uri.path().begin(), filePath.uri.path().end());
            unsigned char* data =
stbi_load(path.c_str(), &width, &height, &nrChannels, 4);
            if (data) {
                VkExtent3D imagesize;
                imagesize.width = width;
                imagesize.height = height;
                imagesize.depth = 1;
                newImage = engine-
>create_image(data, imagesize, VK_FORMAT_R8G8B8A8_UNORM,
VK_IMAGE_USAGE_SAMPLED_BIT, false);
                stbi_image_free(data);
            }
        },
```

```

[&](fastgltf::sources::Vector& vector) {
    unsigned char* data =
stbi_load_from_memory(
                                vector.bytes.data(),
static_cast<int>(vector.bytes.size()), &width, &height, &nrChannels,
4);

    if (data) {
        VkExtent3D imagesize;
        imagesize.width = width;
        imagesize.height = height;
        imagesize.depth = 1;
        newImage = engine-
>create_image(data, imagesize, VK_FORMAT_R8G8B8A8_UNORM,

        VK_IMAGE_USAGE_SAMPLED_BIT, false);

        stbi_image_free(data);
    }
},
[&](fastgltf::sources::BufferView& view) {
    auto& bufferView =
asset.bufferViews[view.bufferViewIndex];
    auto& buffer =
asset.buffers[bufferView.bufferIndex];

    std::visit(fastgltf::visitor{
        [](auto& arg) {},

[&](fastgltf::sources::Vector& vector) {

                                unsigned
char* data = stbi_load_from_memory(

vector.bytes.data() + bufferView.byteOffset,

static_cast<int>(bufferView.byteLength), &width, &height,
&nrChannels, 4);

                                if (data) {

VkExtent3D imagesize;

imagesize.width = width;

imagesize.height = height;

```

```

imagesize.depth = 1;

newImage = engine->create_image(data, imagesize,
VK_FORMAT_R8G8B8A8_UNORM,

VK_IMAGE_USAGE_SAMPLED_BIT,
false);

stbi_image_free(data);

    }
    }},
    buffer.data);

    },
    },
    image.data);

    if (newImage.image == VK_NULL_HANDLE) {
        return {};
    } else {
        return newImage;
    }
}

VkFilter extract_filter(fastgltf::Filter filter) {
    switch (filter) {
        case fastgltf::Filter::Nearest:
        case fastgltf::Filter::NearestMipMapNearest:
        case fastgltf::Filter::NearestMipMapLinear: return
VK_FILTER_NEAREST;
        case fastgltf::Filter::Linear:
        case fastgltf::Filter::LinearMipMapNearest:
        case fastgltf::Filter::LinearMipMapLinear:
        default: return
VK_FILTER_LINEAR;
    }
}

VkSamplerMipmapMode extract_mipmap_mode(fastgltf::Filter filter) {
    switch (filter) {
        case fastgltf::Filter::NearestMipMapNearest:
        case fastgltf::Filter::LinearMipMapNearest: return
VK_SAMPLER_MIPMAP_MODE_NEAREST;

```

```

        case fastgltf::Filter::NearestMipMapLinear:
        case fastgltf::Filter::LinearMipMapLinear:
        default:
            return
VK_SAMPLER_MIPMAP_MODE_LINEAR;
    }
}

std::optional<std::shared_ptr<LoadedGLTF>> loadGltf(BlockEngine*
engine, std::string_view filePath, TRS& trsOut) {
    BK_INFO("Loading GLTF: {}", filePath);
    std::shared_ptr<LoadedGLTF> scene =
std::make_shared<LoadedGLTF>();
    scene->creator = engine;
    LoadedGLTF& file = *scene.get();
    fastgltf::Parser parser{};
    constexpr auto gltfOptions =
fastgltf::Options::DontRequireValidAssetMember |
fastgltf::Options::AllowDouble |

fastgltf::Options::LoadGLBBuffers |
fastgltf::Options::LoadExternalBuffers |

fastgltf::Options::DecomposeNodeMatrices |
fastgltf::Options::GenerateMeshIndices;

    fastgltf::GltfDataBuffer data;
    data.loadFromFile(filePath);
    fastgltf::Asset gltf;
    std::filesystem::path path = filePath;

    auto type = fastgltf::determineGltfFileType(&data);
    if (type == fastgltf::GltfType::glTF) {
        auto load = parser.loadGLTF(&data, path.parent_path(),
gltfOptions);
        if (load) {
            gltf = std::move(load.get());
        } else {
            BK_ERROR("Failed to load glTF: {} ",
fastgltf::to_underlying(load.error()));
            return {};
        }
    } else if (type == fastgltf::GltfType::GLB) {
        auto load = parser.loadBinaryGLTF(&data,

```

```

path.parent_path(), gltfOptions);
    if (load) {
        gltf = std::move(load.get());
    } else {
        BK_ERROR("Failed to load glTF: {}");
        fastgltf::to_underlying(load.error());
        return {};
    }
} else {
    BK_ERROR("Failed to determine glTF container");
    return {};
}
std::vector<DescriptorAllocatorGrowable::PoolSizeRatio> sizes =
{
    {VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 3},
    {VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 3},
    {VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1}
};

file.descriptorPool.init(engine->_device,
gltf.materials.size(), sizes);

for (fastgltf::Sampler& sampler : gltf.samplers) {
    VkSamplerCreateInfo sampl = {.sType =
VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO, .pNext = nullptr};
    sampl.maxLod = VK_LOD_CLAMP_NONE;
    sampl.minLod = 0;
    sampl.magFilter =
extract_filter(sampler.magFilter.value_or(fastgltf::Filter::Nearest))
;
    sampl.minFilter =
extract_filter(sampler.minFilter.value_or(fastgltf::Filter::Nearest))
;
    sampl.mipmapMode =
extract_mipmap_mode(sampler.minFilter.value_or(fastgltf::Filter::Near
est));
    VkSampler newSampler;
    vkCreateSampler(engine->_device, &sampl, nullptr,
&newSampler);
    file.samplers.push_back(newSampler);
}

std::vector<std::shared_ptr<MeshAsset>> meshes;
std::vector<std::shared_ptr<Node>> nodes;

```

```

std::vector<AllocatedImage>          images;
std::vector<TextureID>              imageIDs;
std::vector<std::shared_ptr<GLTFMaterial>> materials;

for (fastgltf::Image& image : gltf.images) {
    std::optional<AllocatedImage> img = load_image(engine,
gltf, image);

    if (img.has_value()) {
        images.push_back(*img);
        file.images[image.name.c_str()] = *img;
    } else {
        images.push_back(engine->_errorCheckerboardImage);
        BK_ERROR("gltf failed to load texture {}",
image.name);
    }
}

file.materialDataBuffer =
    engine-
>create_buffer(sizeof(GLTFMetallic_Roughness::MaterialConstants) *
gltf.materials.size(),

VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, VMA_MEMORY_USAGE_CPU_TO_GPU);
int data_index = 0;
GLTFMetallic_Roughness::MaterialConstants*
sceneMaterialConstants =

(GLTFMetallic_Roughness::MaterialConstants*)file.materialDataBu
ffer.info.pMappedData;

for (fastgltf::Material& mat : gltf.materials) {
    std::shared_ptr<GLTFMaterial> newMat =
std::make_shared<GLTFMaterial>();
    materials.push_back(newMat);
    file.materials[mat.name.c_str()] = newMat;
    GLTFMetallic_Roughness::MaterialConstants constants;
    constants.colorFactors.x =
mat.pbrData.baseColorFactor[0];
    constants.colorFactors.y =
mat.pbrData.baseColorFactor[1];
    constants.colorFactors.z =
mat.pbrData.baseColorFactor[2];
    constants.colorFactors.w =

```

```

mat.pbrData.baseColorFactor[3];
    constants.metal_rough_factors.x =
mat.pbrData.metallicFactor;
    constants.metal_rough_factors.y =
mat.pbrData.roughnessFactor;
    MaterialPass passType          = MaterialPass::MainColor;
    if (mat.alphaMode == fastgltf::AlphaMode::Blend) {
        passType = MaterialPass::Transparent;
    }

    GLTFMetallic_Roughness::MaterialResources
materialResources;
    materialResources.colorImage      = engine->_whiteImage;
    materialResources.colorSampler    = engine-
>_defaultSamplerLinear;
    materialResources.metalRoughImage = engine->_whiteImage;
    materialResources.metalRoughSampler = engine-
>_defaultSamplerLinear;
    materialResources.dataBuffer      =
file.materialDataBuffer.buffer;
    materialResources.dataBufferOffset = data_index *
sizeof(GLTFMetallic_Roughness::MaterialConstants);

    if (mat.pbrData.baseColorTexture.has_value()) {
        size_t img      =
gltf.textures[mat.pbrData.baseColorTexture.value().textureIndex].imageIndex.value();
        size_t sampler =
gltf.textures[mat.pbrData.baseColorTexture.value().textureIndex].samplerIndex.value();
        materialResources.colorImage  = images[img];
        materialResources.colorSampler =
file.samplers[sampler];
    }

    constants.colorTexID =
        engine-
>texCache.AddTexture(materialResources.colorImage.imageView,
materialResources.colorSampler).Index;
    constants.metalRoughTexID =
        engine->texCache

        .AddTexture(materialResources.metalRoughImage.imageView,
materialResources.metalRoughSampler)

```

```

        .Index;
        sceneMaterialConstants[data_index] = constants;
        newMat->data = engine-
>metalRoughMaterial.write_material(engine->_device, passType,
materialResources,

        file.descriptorPool);
        data_index++;
    }

    std::vector<uint32_t> indices;
    std::vector<Vertex> vertices;

    for (fastgltf::Mesh& mesh : gltf.meshes) {
        std::shared_ptr<MeshAsset> newmesh =
std::make_shared<MeshAsset>();
        meshes.push_back(newmesh);
        file.meshes[mesh.name.c_str()] = newmesh;
        newmesh->name = mesh.name;
        indices.clear();
        vertices.clear();

        for (auto&& p : mesh.primitives) {
            GeoSurface newSurface;
            newSurface.startIndex = (uint32_t)indices.size();
            newSurface.count =
(uint32_t)gltf.accessors[p.indicesAccessor.value()].count;

            size_t initial_vtx = vertices.size();

            {
                fastgltf::Accessor& indexaccessor =
gltf.accessors[p.indicesAccessor.value()];
                indices.reserve(indices.size() +
indexaccessor.count);

                fastgltf::iterateAccessor<std::uint32_t>(
                    gltf, indexaccessor, [&](std::uint32_t
idx) { indices.push_back(idx + initial_vtx); });
            }

            {
                fastgltf::Accessor& posAccessor =
gltf.accessors[p.findAttribute("POSITION")->second];

```

```

        vertices.resize(vertices.size() +
posAccessor.count);

    fastgltf::iterateAccessorWithIndex<glm::vec3>(gltf,
posAccessor, [&](glm::vec3 v, size_t index) {
        Vertex newvtx;
        newvtx.position           = v;
        newvtx.normal             = {1, 0, 0};
        newvtx.color              =
glm::vec4{1.f};

        newvtx.uv_x              = 0;
        newvtx.uv_y              = 0;
        vertices[initial_vtx + index] = newvtx;
    });
}

auto normals = p.findAttribute("NORMAL");
if (normals != p.attributes.end()) {
    fastgltf::iterateAccessorWithIndex<glm::vec3>(
        gltf, gltf.accessors[(*normals).second],
        [&](glm::vec3 v, size_t index) {
vertices[initial_vtx + index].normal = v; });
}

auto uv = p.findAttribute("TEXCOORD_0");
if (uv != p.attributes.end()) {

    fastgltf::iterateAccessorWithIndex<glm::vec2>(gltf,
gltf.accessors[(*uv).second],

        [&](glm::vec2 v, size_t index) {

            vertices[initial_vtx + index].uv_x = v.x;

            vertices[initial_vtx + index].uv_y = v.y;

        });
}

auto colors = p.findAttribute("COLOR_0");
if (colors != p.attributes.end()) {
    fastgltf::iterateAccessorWithIndex<glm::vec4>(
        gltf, gltf.accessors[(*colors).second],

```

```

        [&](glm::vec4 v, size_t index) {
vertices[initial_vtx + index].color = v; });
    }

    if (p.materialIndex.has_value()) {
        newSurface.material =
materials[p.materialIndex.value()];
    } else {
        newSurface.material = materials[0];
    }

    glm::vec3 minpos = vertices[initial_vtx].position;
    glm::vec3 maxpos = vertices[initial_vtx].position;
    for (int i = initial_vtx; i < vertices.size(); i++) {
        minpos = glm::min(minpos,
vertices[i].position);
        maxpos = glm::max(maxpos,
vertices[i].position);
    }
    newSurface.bounds.origin          = (maxpos + minpos) /
2.f;
    newSurface.bounds.extents         = (maxpos - minpos) /
2.f;
    newSurface.bounds.sphereRadius =
glm::length(newSurface.bounds.extents);
    newmesh->surfaces.push_back(newSurface);
}

    newmesh->meshBuffers = engine->uploadMesh(indices,
vertices);
}

    for (fastgltf::Node&node : gltf.nodes) {
        std::shared_ptr<Node> newNode;
        if (node.meshIndex.has_value()) {
            newNode =
std::make_shared<MeshNode>();
            static_cast<MeshNode*>(newNode.get())->mesh =
meshes[*node.meshIndex];
        } else {
            newNode = std::make_shared<Node>();
        }
        nodes.push_back(newNode);
        newNode->name = node.name;
    }
}

```

```

        file.nodes[node.name.c_str()];

        std::visit(fastgltf::visitor{[&](fastgltf::Node::TransformMatrix matrix) {}},

[&](fastgltf::Node::TRS transform) {
    glm::vec3
    tl(transform.translation[0], transform.translation[1],
        transform.translation[2]);
    glm::quat
    rot(transform.rotation[3], transform.rotation[0],
        transform.rotation[1], transform.rotation[2]);
    glm::vec3
    sc(transform.scale[0], transform.scale[1], transform.scale[2]);
    glm::mat4 tm
    = glm::translate(glm::mat4(1.f), tl);
    glm::mat4 sm
    = glm::scale(glm::mat4(1.f), sc);

    trsOut.translation = tl;

    trsOut.rotation    = glm::degrees(glm::eulerAngles(rot));
    trsOut.scale
    = sc;

    newNode->position = tl;
    newNode->rotation = glm::degrees(glm::eulerAngles(rot));
    newNode->scale
    = sc;

    }},
        node.transform);
    }
    for (int i = 0; i < gltf.nodes.size(); i++) {
        fastgltf::Node& node = gltf.nodes[i];
        std::shared_ptr<Node>& sceneNode = nodes[i];
        for (auto& c : node.children) {
            sceneNode->children.push_back(nodes[c]);
            nodes[c]->parent = sceneNode;
        }
    }
    for (auto& node : nodes) {

```

```

        if (node->parent.lock() == nullptr) {
            file.topNodes.push_back(node);
            node->refreshTransform(glm::mat4{1.f});
        }
    }
    return scene;
}

void Object::Draw(const glm::mat4& topMatrix, DrawContext& ctx) {
    if (!isVisible) {
        return;
    }

    glm::mat4 localTransform = glm::translate(glm::mat4(1.0f),
position) *

    glm::toMat4(glm::quat(glm::radians(rotation))) *
    glm::scale(glm::mat4(1.0f), scale);
    glm::mat4 worldTransform = topMatrix * localTransform;
    scene->Draw(worldTransform, ctx);
}

void LoadedGLTF::Draw(const glm::mat4& topMatrix, DrawContext& ctx) {
    for (auto& n : topNodes) {
        n->Draw(topMatrix, ctx);
    }
}

void LoadedGLTF::clearAll() {
    try {
        VkDevice dv = creator->_device;

        for (auto& [k, v] : meshes) {}

        for (auto& [k, v] : images) {
            if (v.image == creator-
>_errorCheckerboardImage.image) {
                continue;
            }
            creator->destroy_image(v);
        }

        for (auto& sampler : samplers) {
            vkDestroySampler(dv, sampler, nullptr);
        }
    }
}

```

```

    }

    auto materialBuffer    = materialDataBuffer;
    auto samplersToDestroy = samplers;

    descriptorPool.destroyPool(dv);

    creator->destroy_buffer(materialBuffer);
} catch (std::exception& e) {
    BK_ERROR("Error cleaning up GLTF: {}", e.what());
}
}

```

## Файл pipelines.h

```
#pragma once

#include <types.h>

class PipelineBuilder {
public:
    std::vector<VkPipelineShaderStageCreateInfo> shaderStages;

    VkPipelineInputAssemblyStateCreateInfo inputAssembly;
    VkPipelineRasterizationStateCreateInfo rasterizer;
    VkPipelineColorBlendAttachmentState colorBlendAttachment;
    VkPipelineMultisampleStateCreateInfo multisampling;
    VkPipelineLayout pipelineLayout;
    VkPipelineDepthStencilStateCreateInfo depthStencil;
    VkPipelineRenderingCreateInfo renderInfo;
    VkFormat colorAttachmentFormat;

    PipelineBuilder() {
        clear();
    }
    void clear();
    VkPipeline build_pipeline(VkDevice device);
    void set_shaders(VkShaderModule vertexShader, VkShaderModule
fragmentShader);
    void set_input_topology(VkPrimitiveTopology topology);
    void set_polygon_mode(VkPolygonMode mode);
    void set_cull_mode(VkCullModeFlags cullMode, VkFrontFace
frontFace);
    void set_multisampling_none();
    void disable_blending();
    void enable_blending_additive();
    void enable_blending_alphablend();
    void set_color_attachment_format(VkFormat format);
    void set_depth_format(VkFormat format);
    void disable_depthtest();
    void enable_depthtest(bool depthWriteEnable, VkCompareOp op);
};
bool load_shader_module(const char* filePath, VkDevice device,
VkShaderModule* outShaderModule);
```

## Файл pipelines.cpp

```
#include "initializers.h"
#include "logger.h"
#include <fstream>
#include <pipelines.h>

void PipelineBuilder::clear() {
    inputAssembly          = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO};
    rasterizer             = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO};
    colorBlendAttachment = {};
    multisampling          = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO};
    pipelineLayout         = {};
    depthStencil           = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO};
    renderInfo             = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_RENDERING_CREATE_INFO};
    shaderStages.clear();
}

VkPipeline PipelineBuilder::build_pipeline(VkDevice device) {
    VkPipelineViewportStateCreateInfo viewportState = {};
    viewportState.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
    viewportState.pNext =
nullptr;
    viewportState.viewportCount = 1;
    viewportState.scissorCount = 1;
    VkPipelineColorBlendStateCreateInfo colorBlending = {};
    colorBlending.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
    colorBlending.pNext =
nullptr;
    colorBlending.logicOpEnable =
VK_FALSE;
    colorBlending.logicOp =
VK_LOGIC_OP_COPY;
    colorBlending.attachmentCount = 1;
    colorBlending.pAttachments =
&colorBlendAttachment;
    VkPipelineVertexInputStateCreateInfo _vertexInputInfo = {
```

```

        .sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO};
    VkGraphicsPipelineCreateInfo pipelineInfo = {.sType =
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO};
    pipelineInfo.pNext = &renderInfo;
    pipelineInfo.stageCount =
(uint32_t) shaderStages.size();
    pipelineInfo.pStages =
shaderStages.data();
    pipelineInfo.pVertexInputState = &_vertexInputInfo;
    pipelineInfo.pInputAssemblyState = &inputAssembly;
    pipelineInfo.pViewportState = &viewportState;
    pipelineInfo.pRasterizationState = &rasterizer;
    pipelineInfo.pMultisampleState = &multisampling;
    pipelineInfo.pColorBlendState = &colorBlending;
    pipelineInfo.pDepthStencilState = &depthStencil;
    pipelineInfo.layout = pipelineLayout;

    VkDynamicState state[] =
{VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR};
    VkPipelineDynamicStateCreateInfo dynamicInfo = {.sType =
VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO};
    dynamicInfo.pDynamicStates = &state[0];
    dynamicInfo.dynamicStateCount = 2;
    pipelineInfo.pDynamicState = &dynamicInfo;
    VkPipeline newPipeline;
    if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1,
&pipelineInfo, nullptr, &newPipeline) != VK_SUCCESS) {
        BK_ERROR("failed to create pipeline");
        return VK_NULL_HANDLE;
    } else {
        return newPipeline;
    }
}

void PipelineBuilder::set_shaders(VkShaderModule vertexShader,
VkShaderModule fragmentShader) {
    shaderStages.clear();
    shaderStages.push_back(pipeline_shader_stage_create_info(VK_SHA
DER_STAGE_VERTEX_BIT, vertexShader));
    shaderStages.push_back(pipeline_shader_stage_create_info(VK_SHA
DER_STAGE_FRAGMENT_BIT, fragmentShader));
}

```

```

void PipelineBuilder::set_input_topology(VkPrimitiveTopology
topology) {
    inputAssembly.topology          = topology;
    inputAssembly.primitiveRestartEnable = VK_FALSE;
}

void PipelineBuilder::set_polygon_mode(VkPolygonMode mode) {
    rasterizer.polygonMode = mode;
    rasterizer.lineWidth   = 1.f;
}

void PipelineBuilder::set_cull_mode(VkCullModeFlags cullMode,
VkFrontFace frontFace) {
    rasterizer.cullMode = cullMode;
    rasterizer.frontFace = frontFace;
}

void PipelineBuilder::set_multisampling_none() {
    multisampling.sampleShadingEnable = VK_FALSE;
    multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
    multisampling.minSampleShading    = 1.0f;
    multisampling.pSampleMask          = nullptr;
    multisampling.alphaToCoverageEnable = VK_FALSE;
    multisampling.alphaToOneEnable     = VK_FALSE;
}

void PipelineBuilder::disable_blending() {
    colorBlendAttachment.colorWriteMask =
        VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
    colorBlendAttachment.blendEnable = VK_FALSE;
}

void PipelineBuilder::enable_blending_additive() {
    colorBlendAttachment.colorWriteMask =
        VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
    colorBlendAttachment.blendEnable = VK_TRUE;
    colorBlendAttachment.srcColorBlendFactor =
VK_BLEND_FACTOR_SRC_ALPHA;
    colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE;
    colorBlendAttachment.colorBlendOp       = VK_BLEND_OP_ADD;
    colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
    colorBlendAttachment.dstAlphaBlendFactor =

```

```

VK_BLEND_FACTOR_ZERO;
    colorBlendAttachment.alphaBlendOp          = VK_BLEND_OP_ADD;
}

void PipelineBuilder::enable_blending_alphablend() {
    colorBlendAttachment.colorWriteMask =
        VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
    colorBlendAttachment.blendEnable          = VK_TRUE;
    colorBlendAttachment.srcColorBlendFactor =
VK_BLEND_FACTOR_SRC_ALPHA;
    colorBlendAttachment.dstColorBlendFactor =
VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    colorBlendAttachment.colorBlendOp          = VK_BLEND_OP_ADD;
    colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
    colorBlendAttachment.dstAlphaBlendFactor =
VK_BLEND_FACTOR_ZERO;
    colorBlendAttachment.alphaBlendOp          = VK_BLEND_OP_ADD;
}

void PipelineBuilder::set_color_attachment_format(VkFormat format) {
    colorAttachmentformat          = format;
    renderInfo.colorAttachmentCount = 1;
    renderInfo.pColorAttachmentFormats = &colorAttachmentformat;
}

void PipelineBuilder::set_depth_format(VkFormat format) {
    renderInfo.depthAttachmentFormat = format;
}

void PipelineBuilder::disable_depthtest() {
    depthStencil.depthTestEnable          = VK_FALSE;
    depthStencil.depthWriteEnable          = VK_FALSE;
    depthStencil.depthCompareOp            = VK_COMPARE_OP_NEVER;
    depthStencil.depthBoundsTestEnable     = VK_FALSE;
    depthStencil.stencilTestEnable         = VK_FALSE;
    depthStencil.front                      = {};
    depthStencil.back                      = {};
    depthStencil.minDepthBounds             = 0.f;
    depthStencil.maxDepthBounds             = 1.f;
}

void PipelineBuilder::enable_depthtest(bool depthWriteEnable,
VkCompareOp op) {

```

```

    depthStencil.depthTestEnable      = VK_TRUE;
    depthStencil.depthWriteEnable     = depthWriteEnable;
    depthStencil.depthCompareOp       = op;
    depthStencil.depthBoundsTestEnable = VK_FALSE;
    depthStencil.stencilTestEnable     = VK_FALSE;
    depthStencil.front                 = {};
    depthStencil.back                  = {};
    depthStencil.minDepthBounds        = 0.f;
    depthStencil.maxDepthBounds        = 1.f;
}

bool load_shader_module(const char* filePath, VkDevice device,
VkShaderModule* outShaderModule) {
    std::ifstream file(filePath, std::ios::ate | std::ios::binary);
    if (!file.is_open()) {
        return false;
    }

    size_t          fileSize = (size_t)file.tellg();
    std::vector<uint32_t> buffer(fileSize / sizeof(uint32_t));
    file.seekg(0);
    file.read((char*)buffer.data(), fileSize);
    file.close();
    VkShaderModuleCreateInfo createInfo = {};
    createInfo.sType                 =
VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.pNext                 = nullptr;
    createInfo.codeSize              = buffer.size() *
sizeof(uint32_t);
    createInfo.pCode                 = buffer.data();
    VkShaderModule shaderModule;
    if (vkCreateShaderModule(device, &createInfo, nullptr,
&shaderModule) != VK_SUCCESS) {
        return false;
    }
    *outShaderModule = shaderModule;
    return true;
}

```

## Файл types.h

```
#pragma once

#include "logger.h"
#include <array>
#include <deque>
#include <functional>
#include <memory>
#include <optional>
#include <span>
#include <string>
#include <vector>

#include <vulkan/vulkan.h>

#include <vk_mem_alloc.h>
#include <vulkan/vk_enum_string_helper.h>

#include <glm/gtx/quaternion.hpp>
#include <glm/gtx/transform.hpp>
#include <glm/mat4x4.hpp>
#include <glm/vec4.hpp>

struct AllocatedImage {
    VkImage      image;
    VkImageView  imageView;
    VmaAllocation allocation;
    VkExtent3D  imageExtent;
    VkFormat     imageFormat;
};

struct AllocatedBuffer {
    VkBuffer      buffer;
    VmaAllocation allocation;
    VmaAllocationInfo info;
};

struct GPUGLTFMaterial {
    glm::vec4 colorFactors;
    glm::vec4 metal_rough_factors;
    glm::vec4 extra[14];
};
```

```

static_assert(sizeof(GPUGLTFMaterial) == 256);

struct GPUSceneData {
    glm::mat4 view;
    glm::mat4 proj;
    glm::mat4 viewproj;
    glm::vec4 ambientColor;
    glm::vec4 sunlightDirection;
    glm::vec4 sunlightColor;
};

enum class MaterialPass : uint8_t {
    MainColor,
    Transparent,
    Other
};

struct MaterialPipeline {
    VkPipeline pipeline;
    VkPipelineLayout layout;
};

struct MaterialInstance {
    MaterialPipeline* pipeline;
    VkDescriptorSet materialSet;
    MaterialPass passType;
};

struct Vertex {
    glm::vec3 position;
    float uv_x;
    glm::vec3 normal;
    float uv_y;
    glm::vec4 color;
};

struct GPUMeshBuffers {
    AllocatedBuffer indexBuffer;
    AllocatedBuffer vertexBuffer;
    VkDeviceAddress vertexBufferAddress;
};

struct GPUDrawPushConstants {
    glm::mat4 worldMatrix;
    VkDeviceAddress vertexBuffer;
};

```

```

};

struct DrawContext;

class IRenderable {
    virtual void Draw(const glm::mat4& topMatrix, DrawContext& ctx)
= 0;
};

struct Node: public IRenderable {
    std::weak_ptr<Node> parent;
    std::vector<std::shared_ptr<Node>> children;
    glm::vec3 position;
    glm::vec3 scale;
    glm::vec3 rotation;
    glm::mat4 worldTransform;
    std::string name;

    void refreshTransform(const glm::mat4& parentMatrix) {
        glm::mat4 localTransform = glm::translate(glm::mat4(1.0f),
position) *

glm::toMat4(glm::quat(glm::radians(rotation))) *
glm::scale(glm::mat4(1.0f), scale);
        worldTransform = parentMatrix * localTransform;
        for (auto c : children) {
            c->refreshTransform(worldTransform);
        }
    }

    virtual void Draw(const glm::mat4& topMatrix, DrawContext& ctx)
{
    for (auto& c : children) {
        c->Draw(topMatrix, ctx);
    }
}
};

#define VK_CHECK(x)
\
do {
\
    VkResult err = x;
\

```

```

        if (err) {
\
            BK_ERROR("Detected Vulkan error: {}",
string_VkResult(err)); \
            abort();
\
        }
\
    } while (0)

```

## Файл main.cpp

```
#include "logger.h"
#include <engine.h>

int main(int argc, char* argv[]) {
    try {
        BlockEngine engine;
        engine.init();
        engine.run();
        engine.cleanup();
    } catch (const std::exception& ex) {
        BK_CRITICAL(ex.what());
        return -1;
    }
    return 0;
}
```