«Санкт-Петербургский государственный электротехнический университет

«ЛЭТИ» им. В.И.Ульянова (Ленина)»

(СПбГЭТУ “ЛЭТИ”)

Направление подготовки: 09.03.01 – “Информатика и вычислительная техника”

Профиль: “Организация и программирование вычислительных и информационных систем”

Факультет компьютерных технологий и информатики

Кафедра вычислительной техники

К защите допустить:

Заведующий кафедрой

д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема:«Программное средство для создания 3D анимационных роликов»

Студент		_____ *подпись*	О. В. Евдокимов
Руководитель	д. т. н., профессор (Уч. степень, уч. звание)	_____ *подпись*	А. И. Водяхо
Консультанты	к. э. н., доцент (Уч. степень, уч. звание)	_____ *подпись*	М. А. Косухина
	к. т. н., доцент, с. н. с. (Уч. степень, уч. звание)	_____ *подпись*	И. С. Зуев

Санкт-Петербург
2023 г.

«Санкт-Петербургский государственный электротехнический университет

«ЛЭТИ» им. В.И.Ульянова (Ленина)»
(СПбГЭТУ “ЛЭТИ”)

Направление 09.03.01 – “Информатика и вычислительная техника”
Профиль “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2023 г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент Евдокимов Олег Витальевич Группа № **9306**

1. Тема Программное средство для создания 3D анимационных роликов
(утверждена приказом № _____ от _____)
Место выполнения ВКР: Санкт-Петербургский государственный электротехнический университет им. В.И.Ульянова(Ленина)

2. Объект и предмет исследования

Процесс создания 3D анимационных роликов. Программное средство для упрощения и ускорения процесса создания 3D анимационных роликов.

3. Цель

Разработка программного средства для создания 3D анимационных роликов на основе библиотек готовых анимаций, таких как Mixamo, которое позволяет легко и быстро подбирать, настраивать и комбинировать анимации для разных персонажей и сцен.

4. Исходные данные

Исходными данными для разработки являются русскоязычные и англоязычные статьи и видео в сети Интернет, документация к средствам разработки и библиотекам, форумы разработчиков.

5. Технические требования

5.1. Позволять импортировать 3D модели персонажей в разных форматах (FBX, OBJ, DAE и т.д.).

5.2. Позволять создавать и редактировать сцены с персонажами.

5.3. Позволять импортировать анимации из библиотеки готовых анимаций для конкретных персонажей.

5.4. Позволять комбинировать разные анимации в последовательности или параллельно с помощью временной шкалы.

5.5. Наличие простого и интуитивного графического интерфейса.

6. Содержание

6.1. Изучение теоретических основы создания 3D анимационных роликов и существующих программных средств для этого.

6.2. Разработка архитектуры и интерфейса программного средства с учетом требований к функциональности и качеству.

6.3. Реализация программного средства с использованием современных технологий и инструментов разработки.

7. Дополнительные разделы

Разработка и стандартизации программных средств

8. Результаты

Рабочее программное средство позволяющее загружать модели, добавлять анимации для моделей, выставлять анимации на временной шкале и воспроизводить полученную последовательность анимаций. Отчетные материалы: пояснительная записка, реферат, аннотация, презентация.

Дата выдачи задания	Дата представления ВКР к защите
«___»_____2023 г.	«___»_____2023 г.

Студент _____О. В. Евдокимов

Руководитель

д. т. н., профессор _____А. И. Водяхо

«Санкт-Петербургский государственный электротехнический университет

«ЛЭТИ» им. В.И.Ульянова (Ленина)»
(СПбГЭТУ “ЛЭТИ”)

Направление 09.03.01 – “Информатика и вычислительная техника”
Профиль “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2023 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема **Программное средство для создания 3D анимационных роликов**

Студент Евдокимов Олег Витальевич Группа № **9306**

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	05.04.2023–09.04.2023
2	Проектирование и разработка архитектуры	12.04.2023–25.04.2023
3	Реализация программных модулей	20.04.2023–02.05.2023
4	Написание пояснительной записки	04.05.2023–21.05.2023
5	Предварительное рассмотрение работы	22.05.2023–26.05.2023
6	Представление работы к защите	8.06.2023

Студент _____ О. В. Евдокимов
Руководитель
д. т. н., профессор _____ А. И. Водяхо

РЕФЕРАТ

Пояснительная записка содержит: ___стр., ___ рис., ___ист., ___ прил.

3D анимация — это искусство создания движущихся трехмерных изображений, которое требует большого объема данных, высокой производительности и творческого подхода. Для того чтобы сделать 3D анимацию, нужно уметь работать с различными инструментами и технологиями, такими как трехмерное моделирование, текстурирование, скелетная анимация, кинематика и многое другое. Однако существуют способы упростить и ускорить этот процесс с помощью готовых библиотек анимаций для разных персонажей и сцен. Одна из таких библиотек – Mixamo, которая предлагает множество бесплатных анимаций для людей, животных, фантастических существ и т.д. Mixamo позволяет легко и быстро выбирать и настраивать анимации для любых персонажей через веб-интерфейс. Но Mixamo не подходит для создания полноценных 3D анимационных роликов, так как она не дает возможности создавать сцены с несколькими персонажами и контролировать порядок воспроизведения анимаций. Данную проблему и призвано решить программное средство разрабатываемое в ходе выполнения выпускной квалификационной работы.

ABSTRACT

3D animation is the art of creating moving three-dimensional images, which requires a lot of data, high performance and creativity. In order to make a 3D animation, you need to know how to work with various tools and technologies, such as 3D modeling, texturing, skeleton animation, kinematics, and more. However, there are ways to simplify and speed up this process by using ready made animation libraries for different characters and scenes. One such library is Mixamo, which offers many free animations for people, animals, fantasy creatures, etc. Mixamo makes it quick and easy to select and customize animations for any characters through a web interface. But Mixamo is not suitable for creating full-fledged 3D animations, because it does not allow you to create scenes with multiple characters and control the order in which the animations are played. This problem is designed to solve a software tool developed in the course of the graduate qualification work.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями:

Аниматор — человек создающий анимации, анимационные ролики.

Графический API — это интерфейс программирования приложений, который позволяет взаимодействовать с графическим процессором (GPU).

Шейдер или шейдерная программа — это компьютерная программа, которая исполняется на графическом процессоре.

Ключевой кадр анимации — это точка во времени, в которой задается значение какого-то свойства, например, позиции ориентации или масштаба.

Текстура — массив, буффер позволяющий хранить какую-либо информацию. Чаще всего используется для хранения данных изображения.

Меш (сетка вершин) — это сетка из вершин, ребер и граней, которая определяет форму трехмерного объекта.

Вершины меша — это точки в трехмерном пространстве, которые имеют определенные свойства, например, позицию, цвет и т. п..

Вершинный буфер — это структура данных, которая хранит информацию о вершинах геометрических объектов в компьютерной графике.

Индексный буфер — это структура данных, которая хранит индексы вершин, определяющие порядок их соединения в полигоны.

Скелетная сетка (скелетный меш) — это сетка, связанная со скелетом или каркасом, что позволяет анимировать сетку путем трансформации костей. Скелетная сетка состоит из нескольких компонентов: сетка (меш), скелет, скин.

Скелет — это иерархическая структура костей и суставов, определяющая скелет объекта.

Скин — это процесс прикрепления вершин сетки (меша) к суставам скелета с определенными весами, которые определяют, насколько сильно каждый сустав влияет на вершину.

Скелетная анимация — это способ анимирования трехмерных моделей с помощью костной структуры, которая определяет движение и деформацию модели.

ECS (Entity Component System) — это архитектурный паттерн, используемый в разработке игр и графических приложений.

ВВЕДЕНИЕ

3D-анимация — это мощная и популярная форма визуального повествования, которая позволяет создать реалистичный и захватывающий опыт для различных аудиторий и целей. Однако создание 3D-анимационных роликов — задача не из легких, поскольку требует много времени, навыков и ресурсов.

Для создания 3D-анимационного ролика необходимо разработать и смоделировать 3D-персонажей, создать и отредактировать сцены с ними, задать ключевые кадры анимации для каждого объекта сцены, вплоть до суставов скелетных сеток. Существует множество программных инструментов для различных этапов этого процесса, таких как Blender [1], Unity [2] и т.д., но они часто имеют крутые кривые обучения, высокую стоимость или проблемы совместимости.

Цель — восполнить пробел в литературе и инструментарии аниматоров путем разработки программного инструмента, объединяющего различные этапы создания 3D-анимационных клипов в одном удобном для пользователя интерфейсе.

Этот инструмент позволит пользователям легко и быстро выбирать, настраивать и комбинировать анимацию для различных персонажей из библиотек готовых анимаций, таких, как Mixamo. Это позволит сократить время, затраты и навыки, необходимые для создания 3D-анимационных роликов, а также повысить разнообразие.

Объектом данного исследования является процесс создания анимационных 3D-клипов. Предметом данного исследования является программное средство, которое упрощает и ускоряет процесс создания 3D анимационных клипов на основе библиотек готовых анимаций.

Для достижения поставленной цели требуется определить функциональные требования потенциального пользователя и в соответствии с ними разработать концепцию разрабатываемого программного средства. Затем

определиться с технологиями и библиотеками для реализации программного средства. После чего спроектировать и разработать архитектуру программного средства. И наконец реализовать программное средство в коде.

В первом разделе проводится анализ и выбор существующих технологий и библиотек, которые будут использованы при разработке программного средства.

Во втором разделе рассматривается архитектура программного средства в общем и решения принятые при разработке различных составных модулей в частности.

В третьем разделе рассматриваются вопросы планирования работы над проектом, стандартизации и оценки стоимости проекта.

1 Методы и технологии разработки 3D анимационных роликов

Разработка программного средства для создания 3D анимационных роликов — это задача, требующая глубокого понимания методов и технологий, используемых в компьютерной графике и анимации. В данном разделе мы рассмотрим основные зависимости, которые нам требуется определить до начала разработки, начиная с выбора языка программирования и графического API.

Мы рассмотрим принципы представления объектов в сцене с помощью ECS подхода. Также мы изучим вопросы импорта моделей, и используемую для этого библиотеку Assimp [3]. Рассмотрим вопросы создания пользовательского интерфейса и связанную с этим библиотеку Dear ImGUI [4]. Наконец мы перейдем к изучению основ компьютерной трехмерной анимации, обсудим ключевые кадры и скелетную анимацию.

1.1 Выбор языка программирования

Для разработки программного средства для создания 3D анимационных роликов необходимо выбрать подходящий язык программирования, который обладает необходимыми характеристиками и возможностями. Существует множество языков программирования, которые могут быть использованы для разработки приложений для работы с трехмерной графикой.

Исходя из множества факторов, в качестве языка программирования для разработки программного средства для создания 3D анимационных роликов был выбран C++. C++ — это один из самых популярных и мощных языков программирования, который широко используется для работы с трехмерной графикой и анимацией. C++ обладает следующими преимуществами:

- C++ обеспечивает высокую производительность кода за счет компиляции в машинный код и низкоуровневого доступа к ресурсам компьютера.

- C++ поддерживает объектно-ориентированное программирование, которое позволяет абстрагировать данные и поведение объектов.
- C++ совместим с большинством платформ и операционных систем. C++ также поддерживает различные графические API и библиотеки, которые могут быть использованы для работы с трехмерной графикой и анимацией.

Таким образом, C++ является оптимальным выбором для разработки программного средства для создания 3D анимационных роликов.

1.2 Выбор графического API

Существует множество графических API, которые отличаются по функциональности, уровню абстракции, совместимости и производительности. Выбор графического API зависит от целей и требований программного средства.

Среди наиболее известных и распространенных графических API можно выделить следующие [5]:

- Direct3D — это часть DirectX, набора API для разработки игр и мультимедийных приложений на платформе Windows. Direct3D предоставляет низкоуровневый доступ к графическому процессору. Direct3D имеет высокую производительность и широкую совместимость с разными видеокартами и драйверами. Однако поддерживается только на платформах с операционной системой от Microsoft, что нас не устраивает.
- OpenGL — это кроссплатформенный API для работы с 3D графикой. OpenGL поддерживается большинством операционных систем. OpenGL также предоставляет относительно высокий уровень доступа к графическому процессору. OpenGL имеет хорошую производительность и гибкость в использовании. Однако считается достаточно устаревшим на сегодняшний день.
- Vulkan — это новый кроссплатформенный низкоуровневый API для

работы с 3D графикой. Vulkan предназначен для оптимизации использования ресурсов графического процессора и уменьшения накладных расходов при разработке графических приложений. Vulkan имеет высокую производительность и потенциал для развития.

Для разработки программного средства для создания 3D анимационных роликов я выбрал Vulkan как наиболее подходящий графический API по следующим причинам [6]:

- Vulkan предоставляет низкоуровневый доступ к GPU, что позволяет полностью контролировать работу графического процессора и избежать лишних затрат на синхронизацию, проверку ошибок и преобразование данных.
- Vulkan поддерживает передовые технологии для создания реалистичной и интерактивной 3D графики, такие как трассировка лучей и т. п.
- Vulkan является кроссплатформенным API и может работать на множестве различных устройств, включая ПК, мобильные телефоны и консоли.

В качестве библиотеки для работы с математикой будем использовать широко известную GLM [7], согласованную с языком программирования шейдеров — GLSL.

1.3 Принципы представления объектов в сцене с помощью ECS подхода.

ECS (Entity Component System) — это архитектурный паттерн, используемый в разработке игр и графических приложений, который позволяет эффективно организовать и обрабатывать данные, связанные с игровыми объектами. ECS основан на трех основных объектах: Entity (сущность), Component (компонент) и System (Система) [8].

Entity — это некоторая сущность, которая существует в сцене, например, персонаж, предмет. Entity не имеет никакой логики, поведения или дан-

ных, а лишь представляет собой уникальный идентификатор.

Component — это объект, который хранит некоторую информацию об Entity, например, положение в пространстве, здоровье, скорость или анимации. Component может быть добавлен или удален из Entity в любой момент.

System — это объект, который выполняет какие-то действия над Entity и их Component. System обычно работает с группой Entity, которые имеют определенный набор Component.

ECS подход имеет ряд преимуществ перед остальными подходами, например:

- ECS позволяет писать модульный, расширяемый код, так как каждый компонент и система отвечает за свою область ответственности и не зависит от других.
- ECS улучшает производительность игры, так как данные организованы по типам компонентов, хранятся линейно и легко кэшируются в памяти.

Для реализации ECS подхода в C++ мы будем использовать библиотеку entt. Это легковесная и быстрая библиотека, которая предоставляет удобный интерфейс для работы с ECS [9]. С помощью entt мы можем создавать и удалять Entity, добавлять и удалять Component, получать доступ к Component по Entity или по типу Component.

1.4 Импорт моделей

Для того чтобы отобразить 3D модели в нашем программном средстве, нам необходимо импортировать их из различных форматов файлов, таких как OBJ, FBX, STL и других. Для этого мы будем использовать библиотеку Assimp, которая позволяет загружать различные 3D форматы в общий внутренний формат.

Assimp (Open Asset Import Library) — это библиотека для загрузки различных форматов 3D-файлов в общий, внутренний формат [3]. Она поддерживает более 40 форматов для импорта и растущее количество форматов для

экспорта. Написанная на C++, она доступна под свободной лицензией BSD.

Assimp имеет ряд преимуществ, которые делают ее подходящей для нашей задачи:

- Она поддерживает множество популярных и широко используемых форматов 3D-файлов сцен, таких, как FBX, OBJ, DAE (COLLADA), glTF и т.д.. Это позволит пользователю загружать модели и анимации из различных источников и инструментов.
- Она предоставляет единый и понятный формат данных для представления объектов в сцене. Что упрощает процесс обработки и загрузки данных моделей в наше приложение.
- Она предлагает различные флаги постобработки при импорте, включая часто необходимые операции, такие как вычисление нормалей и косательных. Это помогает нам оптимизировать и улучшить качество загруженных данных.
- Она имеет активное сообщество, которое может помочь при возникновении проблем или вопросов.

Для импорта изображений будем использовать stb_image из библиотеки stb [10].

1.5 Создание графического пользовательского интерфейса

В этом разделе мы рассмотрим выбор в качестве библиотеки для интерфейса пользователя Dear ImGUI совместно с GLFW.

Dear ImGUI — это библиотека для создания графических пользовательских интерфейсов (GUI) на C++, которая имеет минимальные зависимости и высокую производительность [4]. Она использует подход немедленного режима (immediate mode), в отличие от традиционного подхода отложенного режима (retained mode).

В подходе отложенного режима, GUI создается заранее и хранится в памяти в виде дерева виджетов [11]. При каждом изменении состояния GUI, необходимо обновить дерево виджетов и перерисовать его на экране. Это

может приводить к избыточным вычислениям, сложности управления состоянием и проблемам с синхронизацией состояний.

В подходе немедленного режима, GUI создается на лету при каждом вызове функции отрисовки. Нет необходимости хранить дерево виджетов или обновлять его при изменении состояния. Вместо этого, состояние хранится в пользовательских переменных, а логика GUI интегрируется с логикой приложения. Это упрощает код и делает его более гибким и модульным.

Однако, подход немедленного режима требует от библиотеки GUI быть очень быстрой и эффективной, чтобы не снижать производительность приложения. Dear ImGUI решает эту проблему, оптимизируя процесс генерации вершинных буферов и индексных буферов для отрисовки виджетов. Она также поддерживает различные графические API, такие как рассмотренные ранее DirectX , OpenGL, Vulkan и другие.

Для того, чтобы использовать Dear ImGUI в нашем программном средстве, нам также нужна библиотека для работы с окнами и вводом с клавиатуры и мыши. Для этого мы выбрали GLFW, которая является легковесной и переносимой библиотекой для создания и управления окнами с поддержкой OpenGL и Vulkan [12].

GLFW хорошо интегрируется с Dear ImGUI, так как обе библиотеки имеют минимальные зависимости и простой API. Для того, чтобы связать их вместе, нам нужно использовать специальный модуль imgui_impl_glfw.cpp. Этот модуль содержит функции для инициализации и обновления Dear ImGUI с помощью данных из GLFW.

1.6 Технологические основы и методы компьютерной анимации.

Компьютерная трехмерная анимация — это процесс создания движения и формы трехмерных объектов с помощью компьютерных программ [13]. Для этого необходимо определить положение и ориентацию объектов в разные моменты времени, а также способ интерполяции между ними. Существует несколько методов компьютерной трехмерной анимации, но в данной

работе мы сосредоточимся на двух из них: анимации по ключевым кадрам и скелетной анимации.

Анимация по ключевым кадрам (keyframe animation) — это метод, при котором аниматор задает положение и ориентацию объекта в определенные моменты времени, называемые ключевыми кадрами (keyframes). Компьютерная программа затем интерполирует положение, ориентацию и масштаб объекта в промежуточные моменты времени, между ключевыми кадрами, создавая плавное движение. Для интерполяции могут использоваться различные способы, такие как линейная, кубическая или кривая Безье, в данной работе используется только линейная и сферическая интерполяции. Анимация по ключевым кадрам позволяет контролировать движение объекта с высокой точностью, но требует большого количества работы от аниматора. На рисунке 1.1 приведен пример создания ключевых кадров в программе Blender.

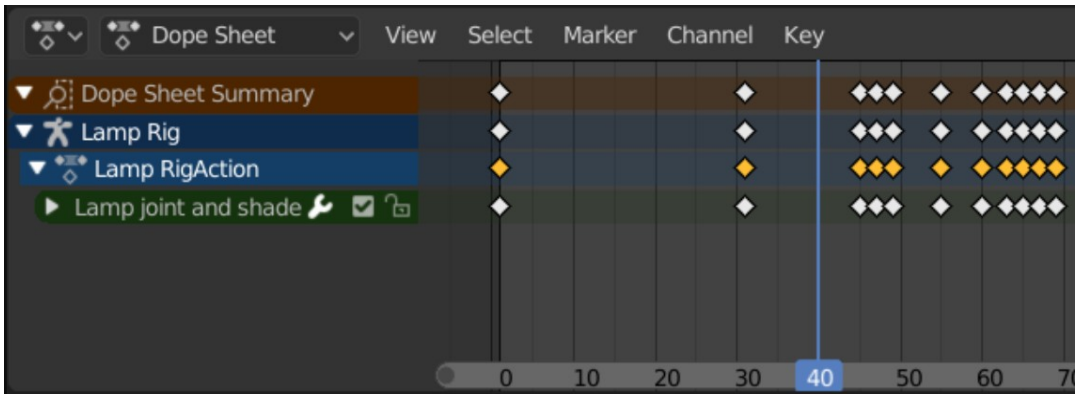
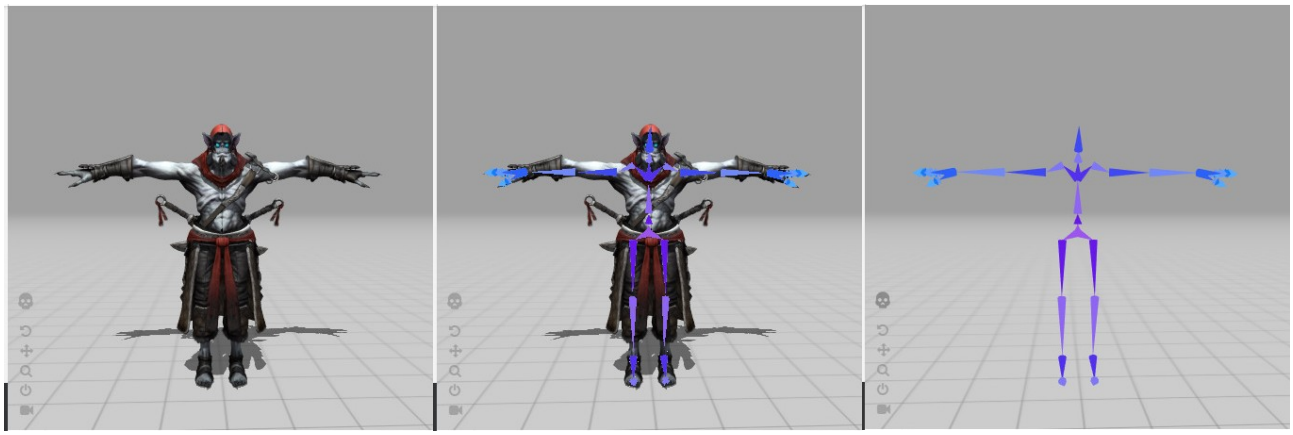


Рисунок 1.1 — Пример ключевых кадров в Blender

Скелетная анимация (skeletal animation) — это метод, при котором аниматор задает движение не самого объекта, а его скелета, состоящего из костей (bones) и суставов (joints). Каждая кость имеет свое положение и ориентацию в пространстве, а также связана с одной или несколькими вершинами (vertices) объекта. При изменении положения или ориентации кости вершины, связанные с ней в соответствии с весам, также меняют свое положение и ориентацию. Кости составляют иерархическую структуру, что, например, позволяет применить преобразование плеча также и к пальцам.

Таким образом, движение скелета определяет движение объекта. Скелетная анимация позволяет создавать более реалистичное и сложное движение объектов, таких как животные или люди, но требует предварительного создания скелета и привязки вершин к костям. На рисунке 1.2 приведены а) модель без скелета, б) скелет модели, в) скелет наложенный поверх модели.



а) б) в)

Рисунок 1.2 — Пример модели и её скелета

2 Разработка и проектирование программного средства

При разработке данного приложения для настольных компьютеров использовался язык C++ с использованием объектно ориентированного подхода, исходный код программы состоит из множества классов и объектов разного уровня абстракции. В данном разделе мы рассмотрим наибольшую часть объектов начиная с низкоуровневых объектов являющихся необходимыми при использовании Vulkan API и заканчивая наиболее высокоуровневыми объектами пользовательского интерфейса.

Спецификация и исходный код программы приведены в приложении А. Все пользовательские классы объявлены внутри пространства имен dmbrn, что позволяет отличать их от сторонних.

2.1 Обертки вокруг основных объектов VulkanAPI

Для начала работы с VulkanAPI требуется создать и настроить множество объектов, которые будут использоваться остальными частями программного комплекса, например, объект логического устройства для выделения памяти на GPU.

В нашем случае мы считаем, что такие объекты могут существовать только в единственном экземпляре в течении всего времени выполнения программы. Соответственно для определения таких объектов можно использовать шаблон одиночка (Singleton). Что означает, что конструктор объектов объявлен приватным и только один класс (Singletons) имеет к нему доступ.

Однако некоторые объекты являются зависимыми друг от друга, из-за чего появляется необходимость для урегулирования порядка инициализации и разрушения объектов. Для этого все такие объекты были объявлены под одной структурой Singletons, с удаленным конструктором, как static inline.

В таблице 2.1 приведены переменные-члены класса Singletons с коротким описанием их назначения. Также на рисунке 2.1, в виде диаграммы классов, продемонстрированы зависимости между классами.

Таблица 2.1 — Переменные-члены класса Singletons

Имя	Тип	Назначение
window	dmbrn::GLFWwindowWrapper	Класс предоставляет методы для создания, доступа, манипулирования и уничтожения окна. Он также обрабатывает события ввода и окна с помощью обратных вызовов.
context	vk::raii::Context	Это класс, который инкапсулирует инициализацию и деинициализацию библиотеки Vulkan
instance	dmbrn::Instance	Это класс, который оборачивает объект vk::raii::Instance, представляющий собой экземпляр приложения Vulkan.
surface	dmbrn::Surface	Это класс, который оборачивает объект vk::raii::SurfaceKHR, представляющий собой абстрактную поверхность, которая может представлять пользователю отрисованные изображения.
physical_device	dmbrn::PhysicalDevice	Это класс, который оборачивает объект vk::raii::PhysicalDevice, представляющий физическое устройство (например, GPU), поддерживающее Vulkan.
device	dmbrn::LogicalDevice	Это класс, который оборачивает объект vk::raii::Device, представляющий логическое устройство (например, абстракцию GPU), которое может быть использовано для взаимодействия с Vulkan.

Продолжение таблицы 2.1

Имя	Тип	Назначение
graphics_queue	vk::raii::Queue	Это класс, который оборачивает объект vk::Queue, представляющий собой упорядоченный список команд, передаваемых устройству для выполнения. Очередь graphics_queue получается от устройства с помощью индекса graphicsFamily, который указывает, что оно поддерживает графические операции.
present_queue	vk::raii::Queue	Получен от устройства с помощью индекса presentFamily, который указывает, что оно поддерживает представление изображений на поверхность.
command_pool	dmbrn::CommandPool	Это класс, который оборачивает объект vk::raii::CommandPool, представляющий собой пул памяти, из которого могут быть выделены буферы команд.
descriptor_pool	dmbrn::DescriptorPool	Это класс, который оборачивает объект vk::raii::DescriptorPool, представляющий собой пул памяти, из которого могут быть выделены множества дескрипторов.

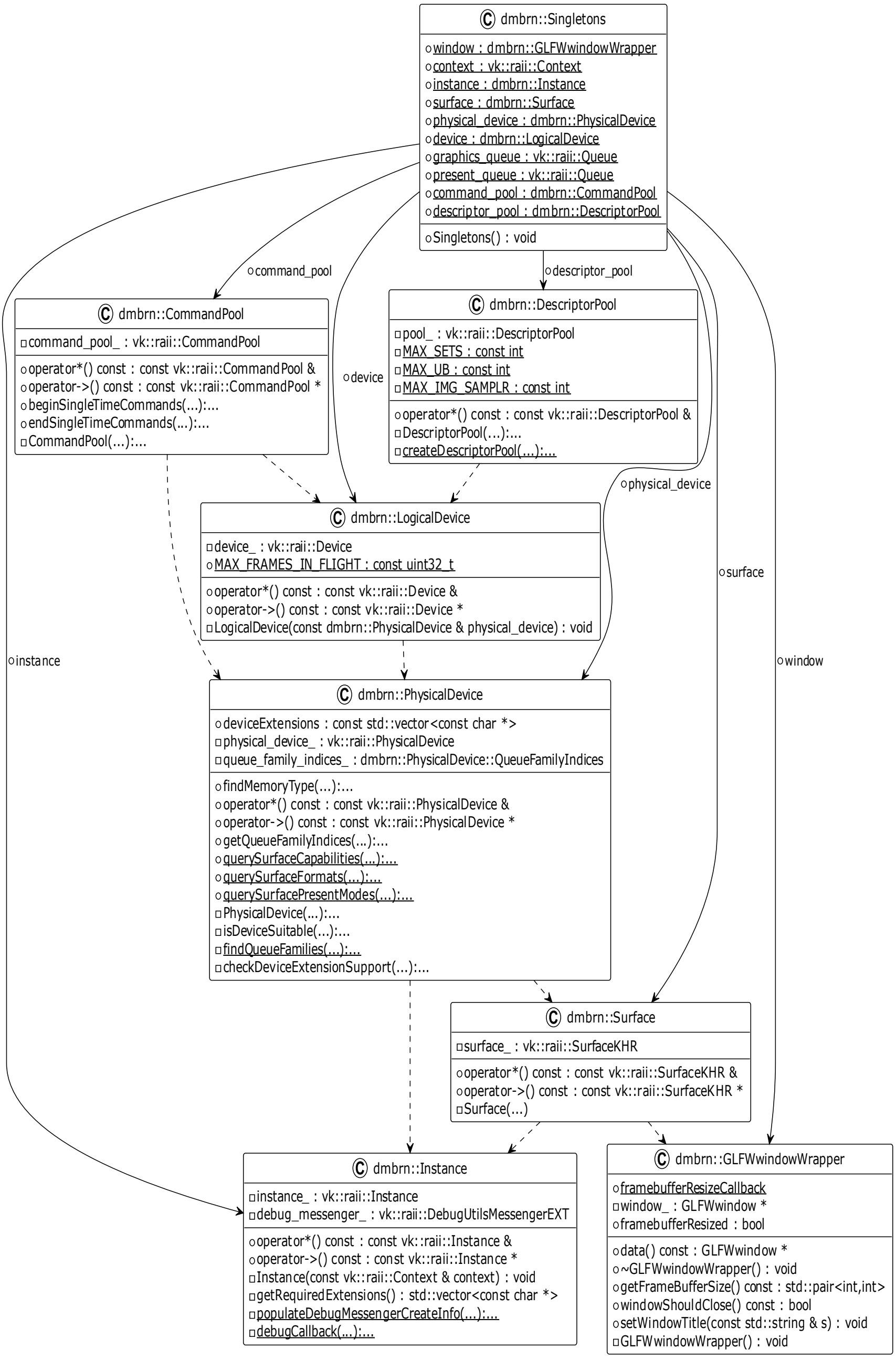


Рисунок 2.1 — Диаграмма классов одиночек

2.2 Описание классов ресурсов

В данном разделе мы рассмотрим структуру классов созданных для управления ресурсами, которые используются при отрисовке трехмерных моделей. Это такие объекты как материалы, текстуры, сети вершин и скелетные сети. Данные классы инкапсулируют объекты Vulkan такие, как буферы и множества дескрипторов (descriptor set) являющимися представлением данных для GPU. Для более простого управления, контроля времени жизни и модификации таких объектов.

2.2.1 Диффузный материал и его зависимости

Материал в трехмерной графике — это способ описания внешнего вида и свойств поверхности 3D-объекта. Материал определяет, как объект отражает или пропускает свет, какой цвет и текстуру он имеет, как он реагирует на тени и блики.

Для задания этих параметров также используются специальные изображения, называемые картами (текстурами). Карта — это файл с информацией о цвете, яркости или высоте каждого пикселя на поверхности объекта. Карты могут быть разных типов в зависимости от того, какой параметр материала они определяют.

В данной работе мы ограничимся только одним параметром — цвет (основной тон поверхности объекта) и картой диффузного цвета (diffuse map) — определяет основной цвет поверхности объекта.

Для исключения дублирования материалов и оптимизации использования памяти. Материалы хранятся в хэш таблице, ключем к которой является объект специального класса MaterialRegistryHandle.

Объект класса MaterialRegistryHandle содержит сырые данные текстуры и цвета и поддерживает требуемые для хэш таблицы операторы:

- Хеширования, который берет исключающее или от хэша изображения и вектора цвета.
- Оператор сравнения на равенство.

Описание переменных-членов класса DiffusionMaterial приведено в таблице 2.2. Описание основных функций-членов класса DiffusionMaterial приведено в таблице 2.3. На рисунке 2.2 приведена диаграмма рассмотренных классов.

Таблица 2.2 — Переменные-члены класса DiffusionMaterial

Имя	Тип	Назначение
diffuse	dmbrn::Texture	Инкапсуляция объектов Vulkan связанных с описанием диффузной текстуры
base_color	dmbrn::DiffusionUniformBuffer	Инкапсуляция объектов Vulkan связанных с описанием основного цвета
descriptor_sets_	dmbrn::DiffusionDescriptorSets	Инкапсуляция объектов Vulkan связанных с описанием множеств ресурсов. Множества ресурсов будут рассмотрены позже.
material_registry	std::unordered_map<MaterialRegistryHandle, DiffusionMaterial, MaterialRegistryHandle::hash>	Регистр материалов используемых в приложении.

Таблица 2.3 — Функции-члены класса DiffusionMaterial

Имя	Назначение
bindMaterialData	Привязывает данные материала для переданного кадра к буферу команд.
GetMaterialPtr	Эта функция проверяет, существует ли уже материал с таким же MaterialRegistryHandle в реестре материалов, и возвращает указатель на этот материал, если он найден. В противном случае она создает новый объект материала и добавляет его в реестр, после чего возвращает указатель на него.

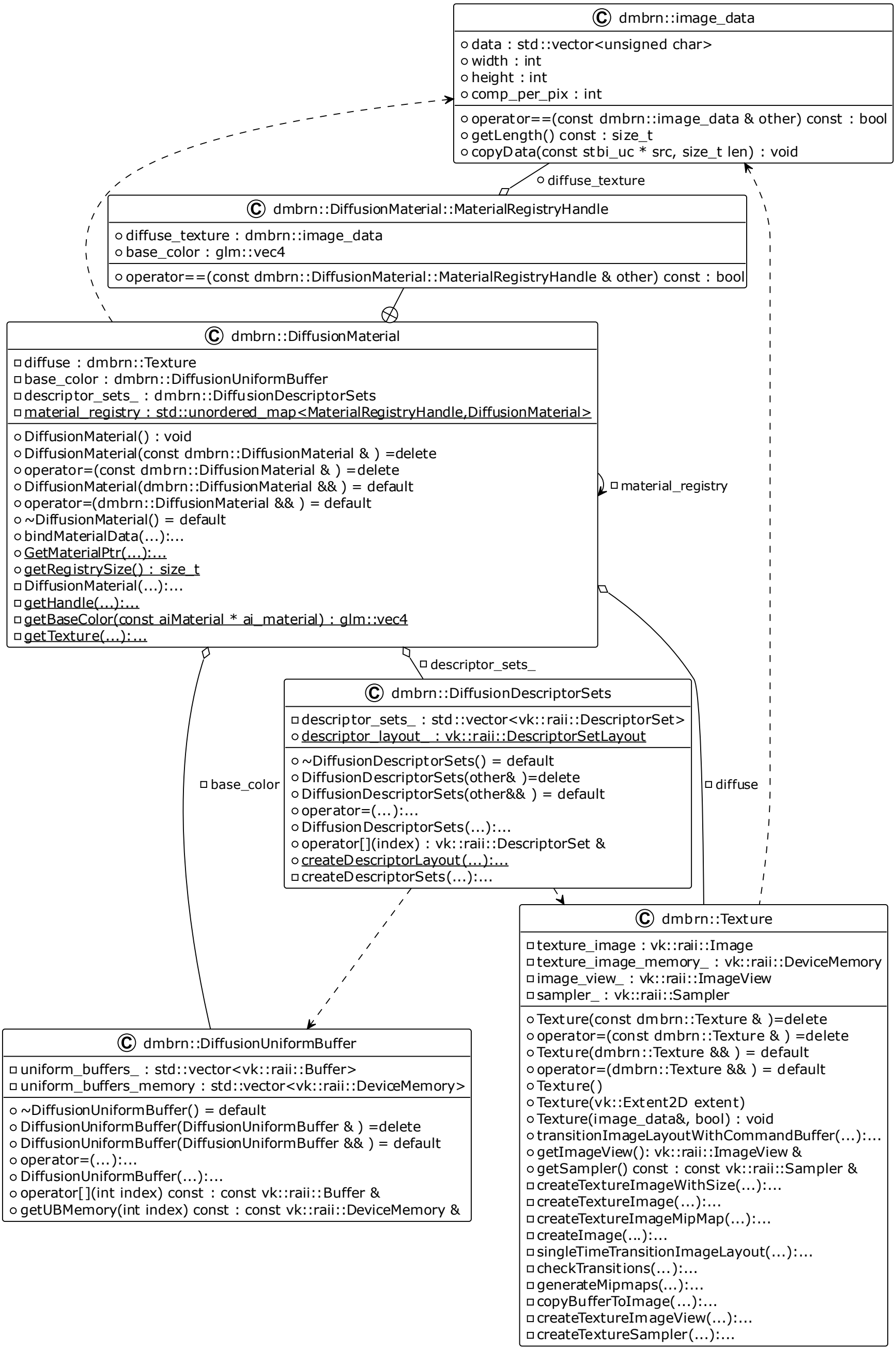


Рисунок 2.2 — Диаграмма классов связанных с DiffusionMaterial

2.2.2 Статический меш и его зависимости

Вершины могут также содержать дополнительную информацию, такую как цвет, нормаль, текстурные координаты и другие атрибуты, которые влияют на внешний вид и поведение меша. Вершины соединяются ребрами, которые образуют грани меша. Грани могут быть треугольными, четырехугольными или состоять из большего числа вершин.

Для использования данных меша на GPU их требуется загрузить в память графического процессора для это определен шаблонный класс HostLocalBuffer, инкапсулирующий объекты буфера и памяти Vulkan.

Для предотвращения дублирования информации о мешах. Было принято решение разбить меш на две части: Mesh и MeshRenderData. MeshRenderData содержит уникальную информацию нужную для отрисовки сетки и хранится в соответствующем реестре. А Mesh содержит указатели на MeshRenderData и материал, таким образом достигается возможность иметь множество объектов с одинаковой сеткой, но разными материалами.

Для определения реестра используется хэш таблица ключем к которой является массив вершин сети. Таким образом можно утверждать, что меши с одинаковым набором позиций вершин (вплоть до порядка) будут ссылаться на один и тот же элемент реестра.

В нашем случае вершины меша содержат: позицию, нормаль и тектурные координаты.

На рисунке 2.3 приведена диаграмма классов, на которой можно увидеть описанные зависимости и содержания классов.

Наиболее важными методами являются:

- bind — выполняющий привязку данных меша к конвейеру;
- drawIndexed — выполняющий команду буфера команд по отрисовке индексированных данных.

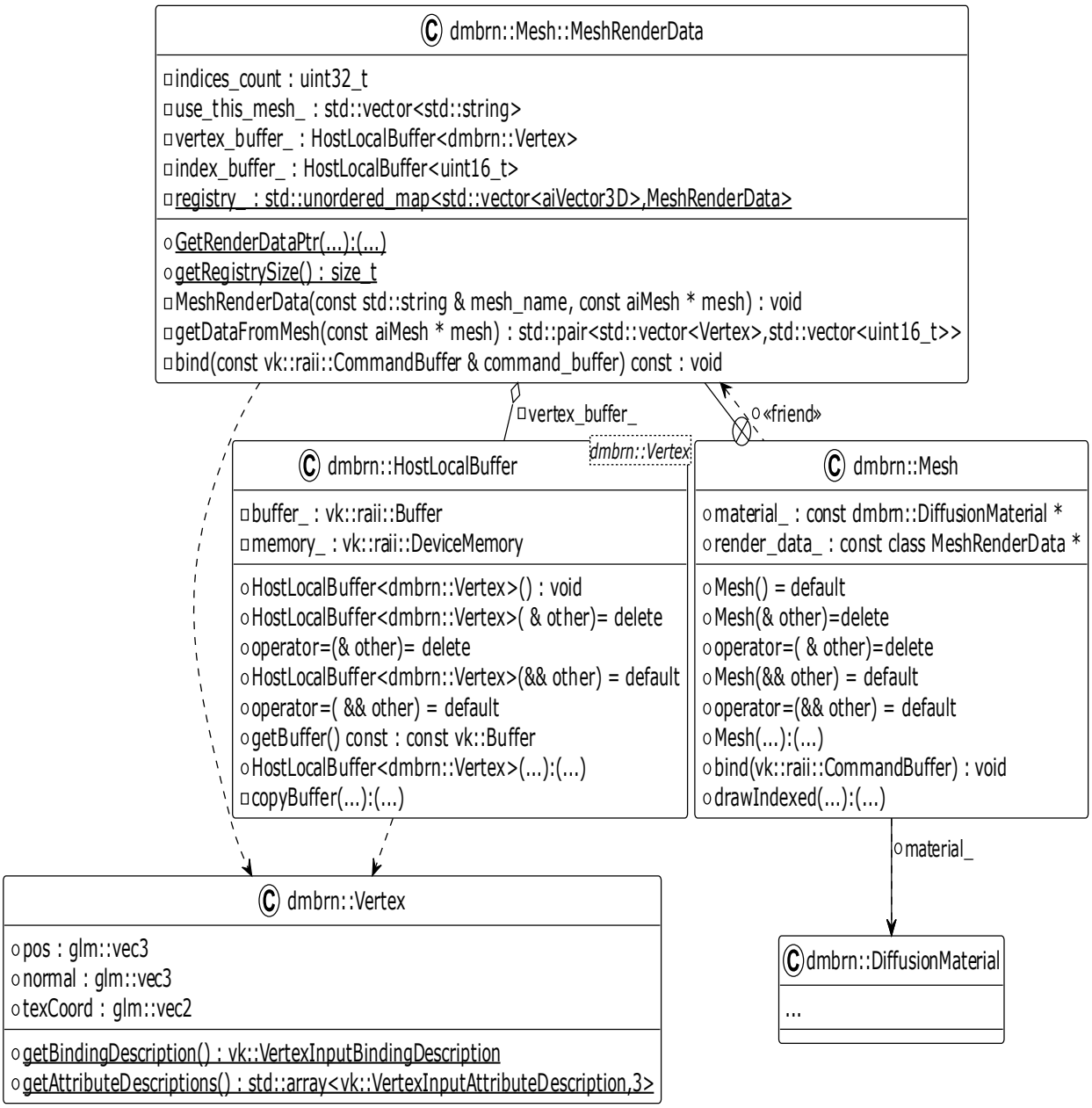


Рисунок 2.3 — Диаграмма классов связанных с классом Mesh

2.2.3 Скелетный меш и его зависимости

В данном разделе мы рассмотрим только объекты связанные с скелетной сеткой (мешем) и скином. Скелет будет рассматриваться в следующий разделах.

Как и в случае со статическими мешами для представления данных на GPU используется HostLocalBuffer. С шаблонным параметром BoneVertex, который рассмотрим позже.

Как и со статическими мешами для предотвращения дублирования данных был введен реестр и разделение данных меша на SkeletalMesh и SkeletalMeshRenderData.

Объект вершины скелетной сетки содержит:

- Позицию, нормаль, текстурные координаты, как и для статического.
- Массив из индексов в другом массиве, содержащем трансформации каждой кости имеющей влияние на данную вершину.
- Массив весов определяющих силу с которой кости (их трансформации) из прошлого массива влияют на данную вершину.

Последние два параметра имеют постоянный размер задающийся константой max_count_of_bones_per_vrtx, на данный момент равной четырем.

На рисунке 2.4 изображена диаграмма классов, на которой можно увидеть описанные отношения.

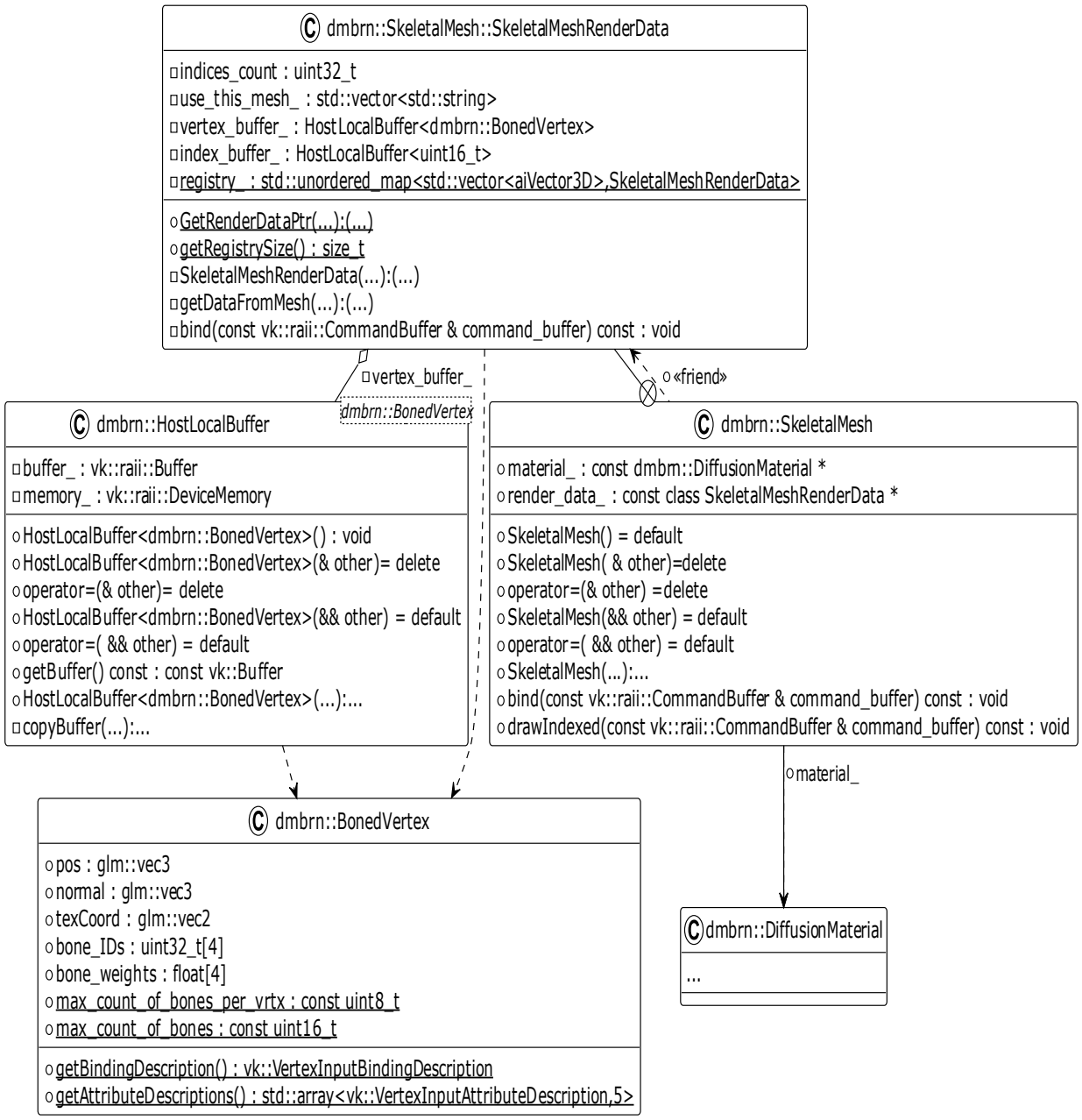


Рисунок 2.4 — Диаграмма классов связанных с классом SkeletalMesh

2.3 Описание классов подсистемы отрисовки

Для того чтобы использовать ранее описанные ресурсы и получить их отображение на экране требуется описать способ их отображения, также дополнительные данные о положении этих объектов в пространстве (на сцене) и информацию о положении и свойствах камеры.

Способ отображения объектов задается с помощью классов связанных с классом ShaderEffect.

Дополнительными данными о положении объектов является:

- Для статических мешей — матрица трансформации модели меша. Такие матрицы для каждого меша хранять в отдельном буфере на GPU подконтрольном классу PerStaticModelData.
- Для скелетных мешей — матрицы трансформации для каждой кости скелета, упомянутые ранее. Массивы таких матриц также содержаться в отдельном буфере на GPU подконтрольном классу PerSkeletonData.

Положение и свойства камеры задаются в объекте ViewportCamera, создаваемом для каждого окна просмотра сцены.

Объекты описанные ранее также являются одиночками, поэтому объединены под одним классом Renderer, для контроля времени жизни и удобства. На рисунке 2.5 приведена диаграмма классов описывающая данные отношения.

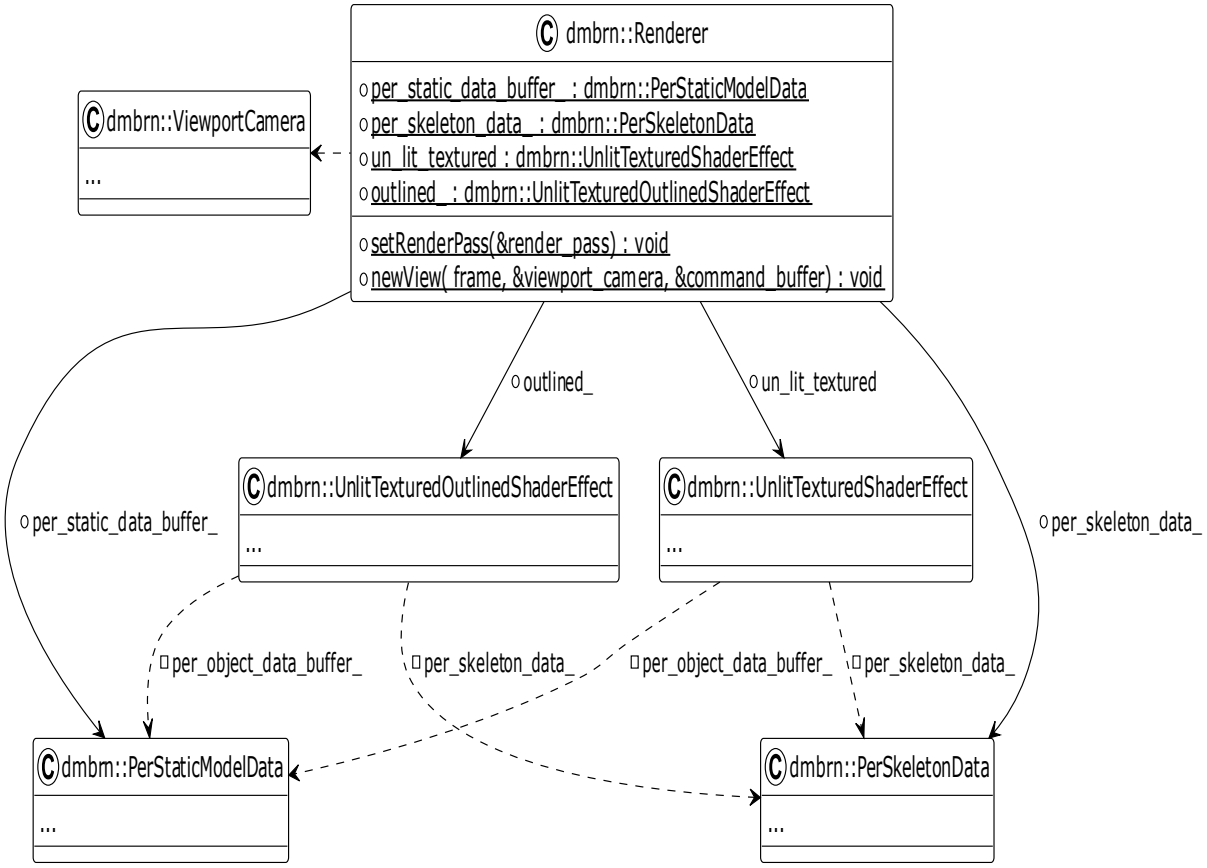


Рисунок 2.5 — Диаграмма классов подсистемы отрисовки

Для представления данных этих классов на GPU используются однородные буферы (uniform buffer). Для упрощения работы был создан шаблонный класс UniformBuffer, абстрагирующий процесс создания буфера и привязки памяти. Основной метод данного класса — mapMemory, выполняющий отображение памяти буфера из GPU на RAM.

Кроме передачи самих данных на GPU нам требуется описать множества дескрипторов (descriptor set) и размещение (layout) этих ресурсов, для доступа к ним из шейдеров.

«Множество дескрипторов – это набор ресурсов, которые привязаны к конвейеру как группа. Одновременно к конвейеру можно привязать несколько таких множеств. У каждого множества есть размещение (layout), определяющее порядок и типы ресурсов во множестве. Размещение множества дескрипторов представляется при помощи объекта, и множества дескрипторов создаются с участием этого объекта». [14]

Для этих каждый каждый объект, связанный с отрисовкой имеет также объекты классов vkDescriptorSet и vkDescriptorSetLayout.

2.3.1 Классы описывающие положение в пространстве

Как уже говорилось в нашей программе два класса описывающих непосредственное положение объекта на сцене и один описывающий положение камеры на сцене влияющее на положение всех объектов.

Первые два объекта имеют схожее описание и назначение поэтому опишем их вместе. Данные объекты являются объектными пулами, выделяющими при запросе registerObject место для соответствующего типа объекта в динамическом буфере на GPU (UniformBufferDynamic) и возвращают индекс начала выделенного объекта. Кроме того позволяют с помощью методов map и unMap отобразить память этого GPU буфера на оперативную память для модификации значений. Также с помощью функции bindDataFor привязать значение объекта с заданным сдвигом к графическому конвейеру.

Следующим классом который мы рассмотрим, является класс ViewportCamera. Определяет положение камеры переменной TransformComponent transform_, и свойства камеры, на данный момент это только соотношение сторон, определяющее матрицу проекции камеры, переменной CameraComponent camera_comp. Для представления управления данными о камере на GPU используется шаблонный класс CameraRenderData. На рисунке 2.6 приведена диаграмма классов, на которой можно увидеть описанные отношения.

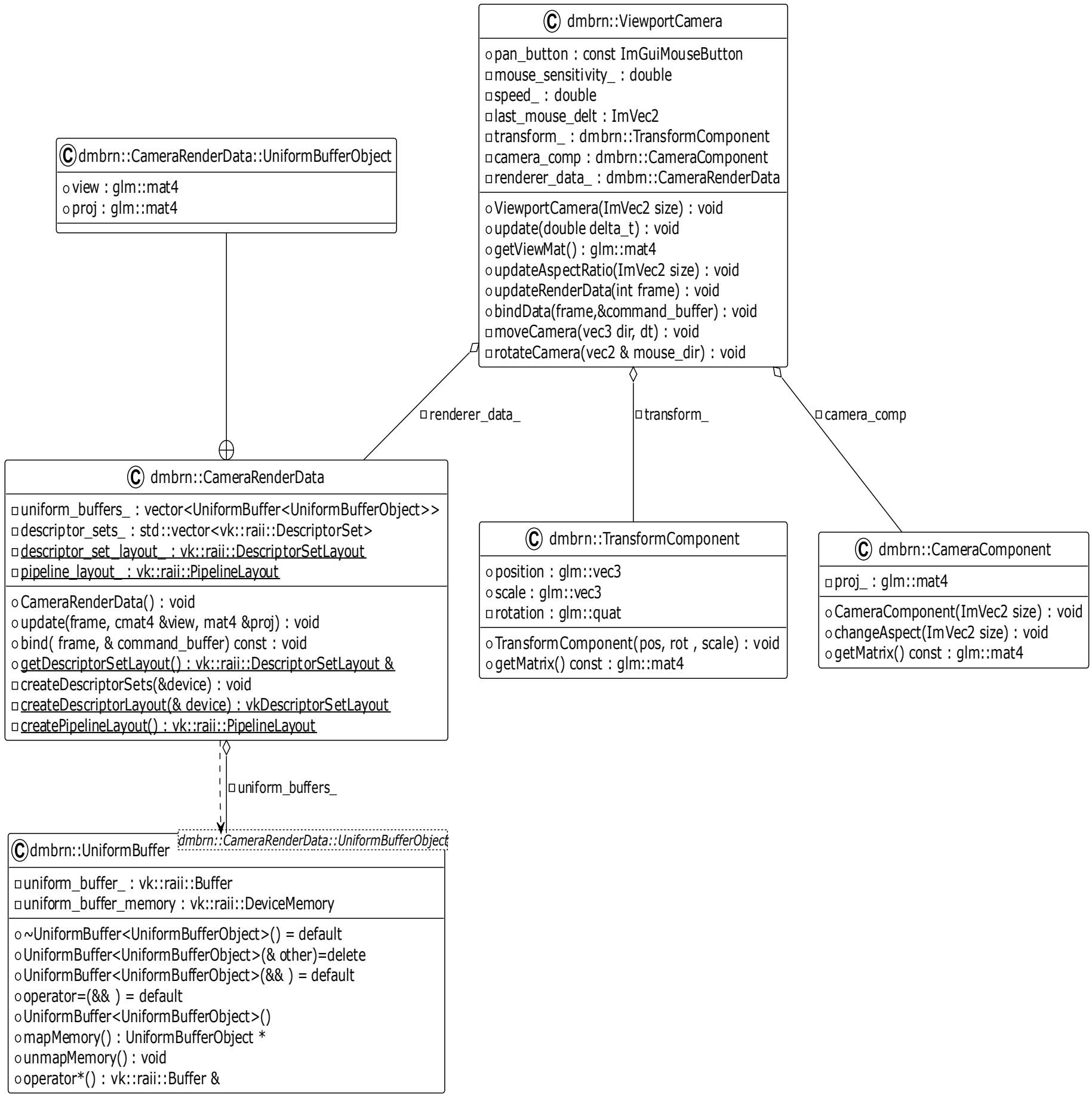


Рисунок 2.6 — Диаграмма классов описывающих положение в пространстве

2.3.2 Классы описывающие способ отрисовки

Ранее упомянутый класс ShaderEffect является абстрактным базовым классом для двух других:

- dmbrn::UnlitTexturedShaderEffect — класс отвечающий за стандартное отображение текстурированной модели с простым затенением.
- dmbrn::UnlitTexturedOutlinedShaderEffect — класс отвечающий за отображение текстурированной модели с контуром заданного цвета.

Класс ShaderEffect определяет наличие двух множеств отрисовки:

- для статических мешей — static_render_queue;
- для скелетных мешей — skeletal_render_queue.

Сами очереди это объекты типа std::set, элементы множества упорядочены по определенному пользователем оператору сравнения, задача которого сначала отсортировать объекты по указателю на меш, затем по указателю на материал, затем по сдвигу в буфер положения объекта. Таким образом проходя линейно по множеству мы будем идти сначала по объектам с одинаковыми мешами, внутри этой группы с одинаковыми материалами, и внутри этой группы проходить по всем положениям. Что позволяет нам сэкономить на вызове операций привязки данных к графическому конвейеру в дальнейшем.

Также определяет метод с двумя перегрузками по аргументу для добавления объектов в соответствующую очередь — addToRenderQueue.

Определяет чисто виртуальный метод draw выполняющий отрисовку статических и скелетных мешей из очереди.

На рисунке 2.8 приведена диаграмма классов, демонстрирующая описанные отношения.

Давайте подробнее рассмотрим устройство классов наследников класса ShaderEffect. Оба имеющихся класса содержат один или несколько объектов с окончанием GraphicsPipelineStatics. Такие объекты содержат объекты Vulkan требуемые для описания отрисовки такие, как:

- размещение конвейера (pipeline layout) — «… множество множеств, которые доступны конвейеру, сгруппировано в другой объект: размещение конвейера. Конвейеры создаются с учетом этого объекта». [14]
- графический конвейер (pipeline) — объект графического конвейера непосредственно определяющий отрисовку.

Давайте рассмотрим вопросы связанные с размещением конвейера. Как уже говорилось у нас имеется несколько ресурсов требующихся для отрисовки объекта это: данные камеры, данные материала, данные шейдерного эффекта. Каждые из этих данных группируются в свои множества дескрипторов с их размещениями, затем из них собирается размещение конвейера.

В своем блоге [15] NVIDIA рекомендует придерживаться принципа расположения ресурсов по частоте их привязки. Так с увеличением индекса привязки увеличивается и частота. Этот принцип также оправдан поскольку спецификация Vulkan [16] в пункте 14.2.1 вводит понятие совместимости размещений конвейеров и говорит о не беспокойстве ранее привязанных дескрипторов при привязки нового дескриптора с совместимым размещением конвейера.

В соответствии с ранее сказанным в данной работе было принято решение сначала размещать дескрипторы в следующем порядке:

1. данные камеры;
2. данные шейдерного эффекта;
3. данные материала;
4. данные статической или скелетной модели.

Описанные отношения продемонстрированы на рисунке 2.7. Приведены дескрипторы для OutlineGraphicsPipelineStatics и статической модели.

Теперь давайте обобщенно рассмотрим вопросы связанные с конвейером. Конвейер Vulkan состоит из множества стадий, которые можно различ-

ным образом настраивать. Наиболее интересными для нас являются стадии вершинного и фрагментного (пиксельного) шейдеров. Для создания конвейера нам нужно передать прочитанный из файла SPIR-V код, исходный код шейдеров на языке GLSL приведен в приложении Б.

2.4 Классы подсистемы ECS

Как уже говорилось ранее объекты сцены являются сущностями, которые могут иметь различный состав компонентов. А сами сущности только идентификаторы. Для удобства работы с библиотекой entt, был написан класс Enttity — обертка вокруг типа entt::entity являющийся, если посмотреть вглубь целым числом std::uint32_t. Соответственно данный объект, как и объект entt::entity не является полным владельцем, т. е. не управляет памятью и временем жизни сущности.

При разработке программы было принято решение утвердить следующий набор правил:

- не может существовать сущности (кроме entt::null) без имени (за это отвечает класс TagComponent);
- позиции на сцене (за это отвечает класс TransformComponent);
- каждая сущность находится в иерархическом отношении с другими (за это отвечает класс RelationshipComponent).
- Для соблюдения этого правила конструктор при создании сущности добавляет эти компоненты.

TagComponent содержит только std::string и все.

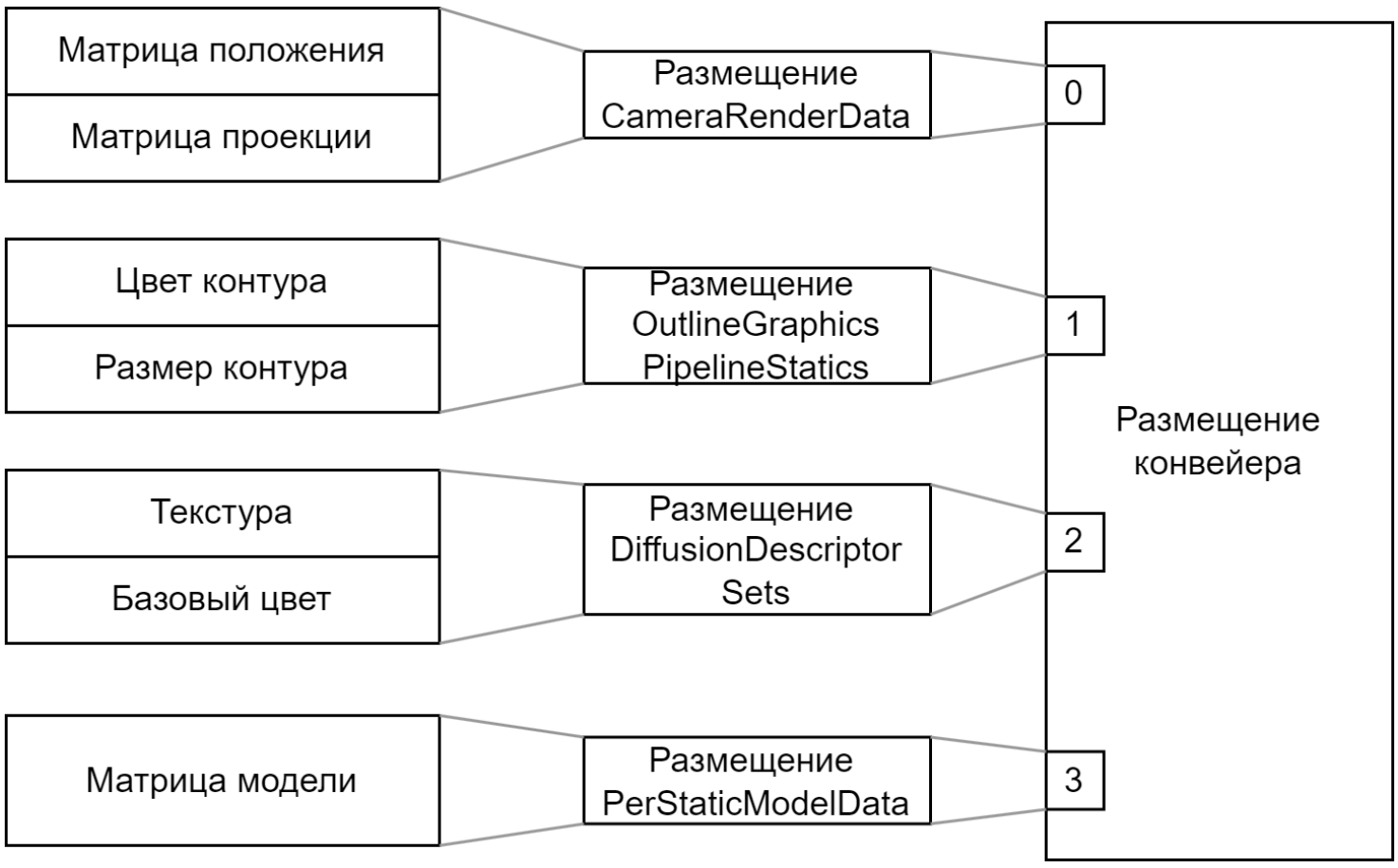


Рисунок 2.7 — Размещения конвейера

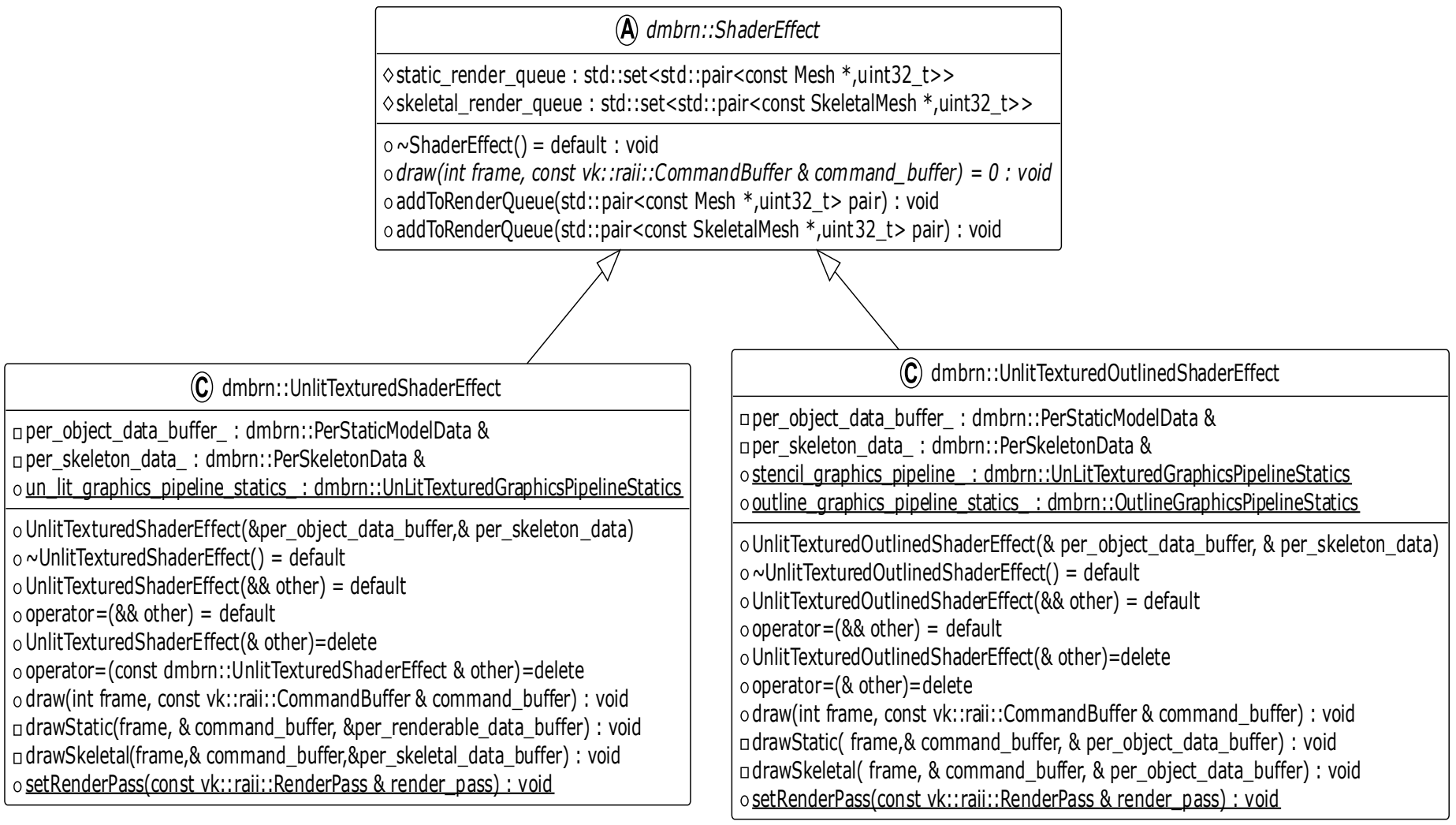


Рисунок 2.8 — Диаграмма классов описывающих способ отрисовки

TransformComponent содержит вектора позиции и масштаба, а также кватернион задающий ориентацию сущности. Кроме этого два дополнительных атрибута-массива dirty (признака загрязнения) и edited (признака изменения) для каждого кадра. Данные атрибуты нужны для процесса иерархического обновления позиций сущностей происходящего каждый кадр. Идея заключается в том, что при изменении положения какой-то сущности в месте изменения требуется вызвать функцию Enttity::markTransformAsEdited, которая помечает TransformComponent данной сущности как измененный (edited), затем помечает все сущности расположенные выше по иерархии как грязные (dirty) с помощью метода Enttity::markTransformAsDirty.

RelationshipComponent — компонент содержащий идентификатор предка (parent), первого своего потомка (first), следующего потомка своего предка (next), предыдущего потомка своего предка (prev). Таким образом получается организовать древовидное отношение между сущностями.

StaticModelComponent — компонент содержащий объект Mesh (говорящий с каким мешем и материалом нужно отрисовать данный объект), указатель на ShaderEffect (говорящий как отрисовать меш) и uint32_t индекс смещения в буфере PerStaticModelData, рассмотренный ране. Также булеву переменную need_GPU_state_update показывающую требуется ли обновлять данные о положении этого объекта в памяти GPU, поскольку обновлять их каждый кадр может быть дорого.

BoneComponent — компонент, наличие которого говорит, что данная сущность является суставом какого-то скелета. Компонент содержит булеву переменную need_gpu_state_update показывающую требуется ли обновлять данные о положении данного сустава в памяти GPU. Также содержит матрицу привязки сустава — определяет преобразование, необходимое для трансформации из пространства меша в локальное пространство данной кости. Также содержит bone_ind — индекс данного сустава внутри одного из объектов буфера PerSkeletonData.

SkeletonComponent — компонент, наличие которого говорит, что сущность является корнем скелета. Компонент содержит переменную in_GPU_mtxs_offset, которая представляет собой смещение объекта внутри буфера PerSkeletonData. Массива bone_enttities хранящего идентификаторы сущностей-суставов (имеющих компонент BoneComponent) которые составляют данный скелет.

AnimationComponent — компонент, наличие которого говорит, что сущность и все её поддерево может быть анимировано. Содержит булеву переменную признак записи — is_recording. Также множество (set) элементов анимационных клипов типа AnimationClip, который мы рассмотрим позже. Также методы для взаимодействия с анимационными клипами updateClipName — для смены имени и insert для вставки новых клипов.

AnimationClip — класс описывающий готовые анимационные клипы для объектов сцены, содержит следующие атрибуты:

- name — название анимации;
- min и max — минимальное и максимальное значение времени ключевого кадра;
- channels — хэш таблица с каналами анимации типа AnimationChannels для каждой сущности задействованной в анимации.

Главное, что также содержит AnimationClip это следующий метод: updateTransforms — метод для обновления положения сущностей задействованных в анимации в соответствии с локальным временем клипа.

AnimationChannels — класс хранящий ключевые кадры анимации для конкретной сущности разделенные на 3 типа (канала): позиции, ориентации, масштаба. Сами ключевые кадры хранятся в упорядоченном отображении (map), ключом в котором является время ключевого кадра, а значением — значение ключевого кадра в данный момент времени. Для интерполяции значений ключевых кадров для промежуточных значений позиции, ориентации

и масштаба существуют соответственно функции mixPositions, slerpRotation и mixScale.

Описанные отношения между классами AnimationComponent, AnimationClip, AnimationChannels и Enttity, можно увидеть на диаграмме классов представленной на рисунке 2.9.

2.5 Классы подсистемы сцены и импорта моделей

За представления сцены отвечает класс Scene. Как уже говорилось ранее сцена является контейнером для сущностей за это отвечает содержащийся объект entt::registry, выполняющий контроль времени жизни упомянутых ранее entt::entity. Кроме этого сцена содержит идентификатор корня сцены scene_root_ и объект animation_sequence_ класса AnimationSequence, который мы рассмотрим позже.

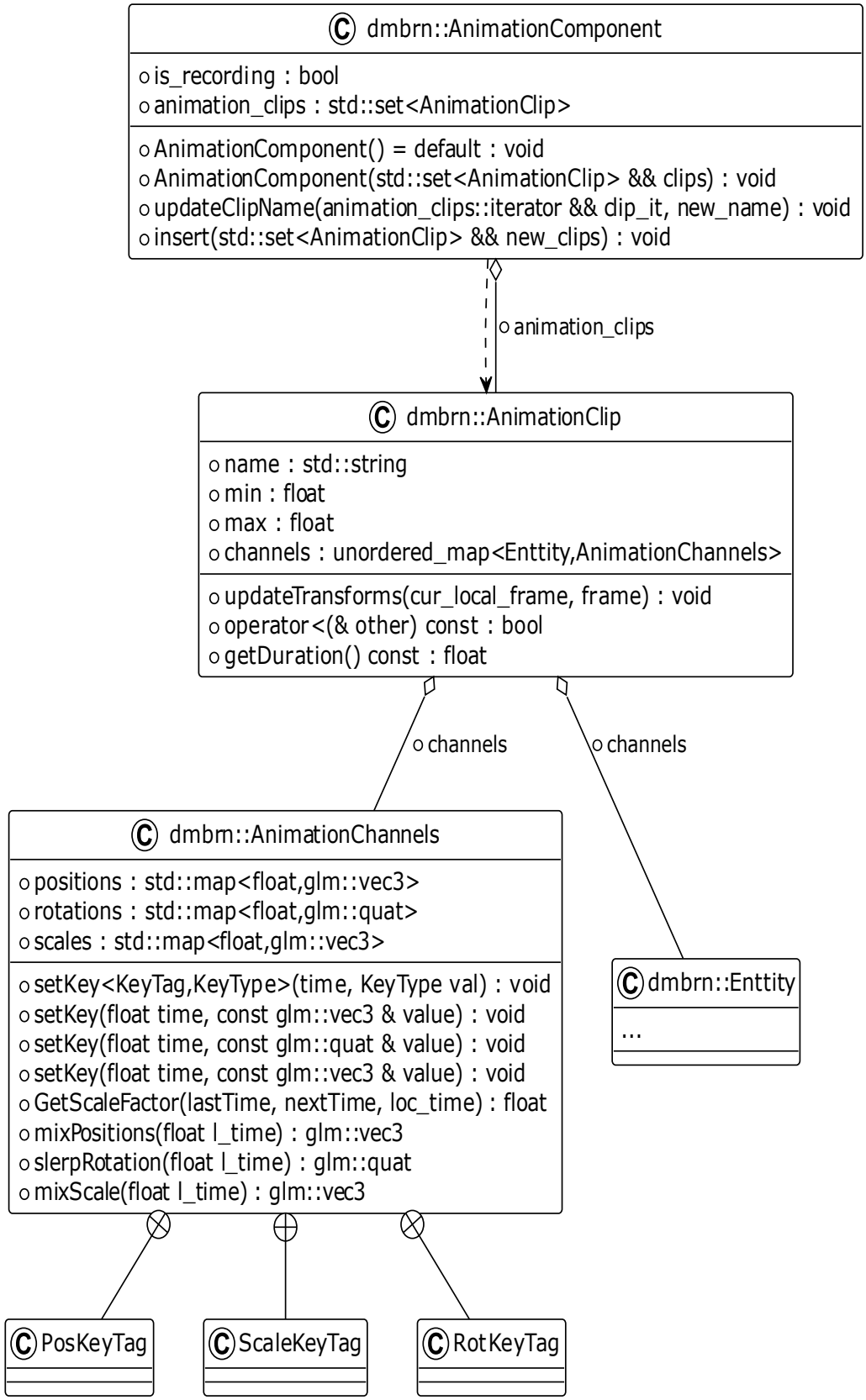


Рисунок 2.9 — Диаграмма классов связанных с AnimationComponent

Давайте коротко рассмотрим основные методы содержащиеся в данном классе:

- addNewEntityToRoot — добавляет новую сущность как дочернюю к scene_root_.
- addNewEntityAsChild — добавляет новую сущность как дочернюю к заданной.
- updateAnimations — проходит по всем анимированным сущностям, для которых существует последовательность в animation_sequence_. Находит клип соответствующий текущему глобальному времени воспроизведения анимаций, находит локальное время клипа и обновляет позиции сущностей при помощи метода AnimationClip::updateTransforms.
- updateGlobalTransforms — проходит начиная с корня сцены по всем грязным (dirty) компонентам трансформации, пока не будут найдены все измененные (edited). Когда измененный найден, проходит до листьев дерева, аккумулируя преобразования текущих сущностей, если на данном пути встречаются сущности связанные с памятью графического процессора (такие как StaticModelComponent, BoneComponent) помечает их как требующие обновления в памяти GPU.
- updatePerStaticModelData — отображает память буфера PerStaticModelData на ОЗУ, проходит по всем StaticModelComponent, если находится требующий обновления состояния в GPU, производит запись матрицы используя inGPU_transform_offset из StaticModelComponent.
- updatePerSkeletalData — отображает память буфера PerSkeletonData на ОЗУ, проходит по всем SkeletonComponent и каждому из их BoneComponent, если находится сустав требующий обновления состояния в GPU, производит запись матрицы используя

inGPU_transform_offset и bone_ind из BoneComponent.

- getModelsToDraw — получить список всех StaticModelComponent для дальнейшей отрисовки.
- getSkeletalModelsToDraw — получить список всех SkeletalModelComponent для дальнейшей отрисовки.

Внутри класса Scene содержится класс ModelImporter, данный класс содержит только статические методы и переменные, раскрывает для класса Scene только два метода: ImportModel и ImportAnimationTo, инкапсулируя вспомогательные функции.

ImportModel создает в корне сцене дочернюю сущность модели из файла с указанным путем. Причем можно указать импортировать ли кости, в таком случае к корню модели будет добавлен компонент SkeletonComponent, и импортировать ли анимации, тогда будет добавлен AnimationComponent.

ImportAnimationTo выполняет импорт анимаций из файла с указанным путем. И добавляет их в AnimationComponent::animation_clips указанной сущности, причем AnimationComponent уже должен существовать у сущности.

2.6 Классы графического интерфейса

Прежде чем рассмотреть конкретные элементы интерфейса давайте рассмотрим фундаментальные объекты на которых основывается отрисовка интерфейса на экран.

Первым из таких объектов является объект класса EditorRenderPass, данный объект является оберткой над объектом прохода рендера renderpass Vulkan, Команды рисования должны быть записаны в экземпляре прохода рендеринга. Каждый экземпляр передачи рендеринга определяет набор ресурсов изображения, называемых вложениями, которые используются во время рендеринга, их начальные и конечные типы компоновки (image layout) изображений для этих буферов. Так цветной буфер данного прохода графики в начале имеет неизвестную компоновку (ImageLayout::eUndefined), а в конце компоновку для показа на экран (ImageLayout::ePresentSrcKHR).

Следующий объект — это объект класса EditorSwapChain, этот объект управляет цепочкой подкачки для окна редактора. Он содержит вектор объектов EditorFrame каждый из которых представляет кадр в цепочке подкачки, более подробно мы рассмотрим его позже. Объект EditorSwapChain контролирует время жизни своих атрибутов. Он также обрабатывает изменение размера цепочки подкачки при изменении размера окна.

Давайте рассмотрим упомянутый класс EditorFrame, он представляет кадр, используемый в при отображении приложении. Он содержит несколько ресурсов Vulkan, связанных с рендерингом и синхронизацией. Давайте рассмотрим наиболее важные его атрибуты:

- command_buffer — буфер команд Vulkan. Он используется для записи команд рендеринга.
- image_available_semaphore и render_finished_semaphore: семафоры Vulkan Они используются для синхронизации внутри GPU во время рендеринга:
    - image_available_semaphore сигнализирует, когда изображение доступно для отрисовки на него;
    - render_finished_semaphore сигнализирует о завершении рендеринга и готовности изображения к отображению на экране;
- in_flight_fence — ограждение Vulkan. Оно используется для синхронизации между CPU и GPU, чтобы информировать CPU о выполнении всех команд в буфере данного кадра.

Последним объектом из таких является объект класса ImGuiRaii, данный объект нужен для инициализации и контроля времени жизни объектов ImGUI.

Главным оркестратором, контейнером для этих и последующих объектов является объект класса EditorUI. Перечисленные ранее и последующие объекты являются его атрибутами и контролируются им.

Для наглядной демонстрации описанных отношений на рисунке 2.10

представлена диаграмма классов, содержащая данные классы.

Теперь давайте определим какие именно элементы графического интерфейса и их функциональные возможности требуются для взаимодействия с программным средством:

- Окно дерева сцены, в котором отображается все поддерево начиная с дочерних элементов корня сцены, функциональные требования:
  - возможность по левой кнопки мыши выбрать сущность.
- Окно просмотра сцены (viewport), в котором мы можем видеть сцену отрисованную с позиции камеры этого окна, функциональные требования:
  - выбранная сущность, каким-то образом должна выделятся;
  - возможность изменение позиции камеры кнопками на клавиатуре;
  - возможность изменения ориентации камеры протаскиванием курсора мыши с зажатой клавишей.
- Окно инспектора, позволяющее увидеть и взаимодействовать с компонентами выбранной сущности.
- Окно секвенсора анимаций, отображающее на временной линии расположение анимационных клипов, для каждой анимированной сущности, функциональные требования:
  - возможность перетаскивания (drag’n’drop) новых анимационных клипов на панель из AnimationComponent;
  - возможность изменения времени начала установленных клипов.

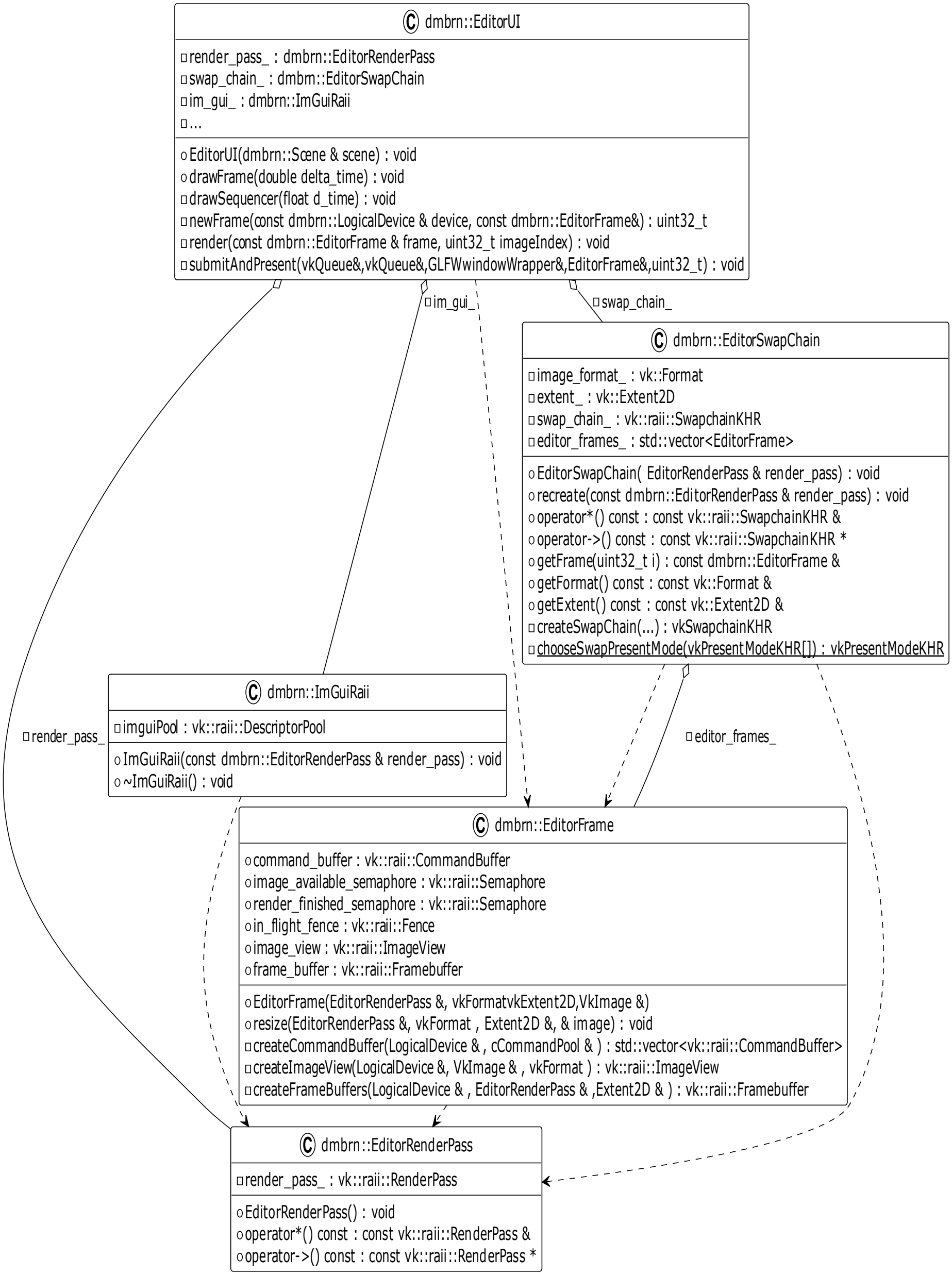


Рисунок 2.10 — Диаграмма фундаментальных классов графического интерфейса

Первым давайте рассмотрим окно дерева сцены и связанный с этим класс SceneTree. Данный класс содержит только два атрибута это scene_ ссылка на сцену и selected_ идентификатор выбранной сущности.

Методом отвечающим за отрисовку данного окна является newImGuiFrame. Отрисовка дерева выполняется при помощи функции ImGui::TreeNodeEx, идентификатором вершины дерева является идентификатор соответствующей сущности. При помощи функции ImGui::IsItemClicked осуществляется проверка на нажатие на вершину дерева с выбором сущности.

Кроме этого в данном окне имеется кнопка с названием "Add new from file", при нажатии на которою появляется модальное окно, в котором пользователь может ввести путь до модели и параметры импорта такие как: импортировать ли с костями или с анимациями. На рисунке 2.11 демонстрируется вид окна при заранее импортированных двух моделях, выбрана сущность с названием mixamorig_RightUpLeg, цвета инвертированы для экономии чернил при печати.

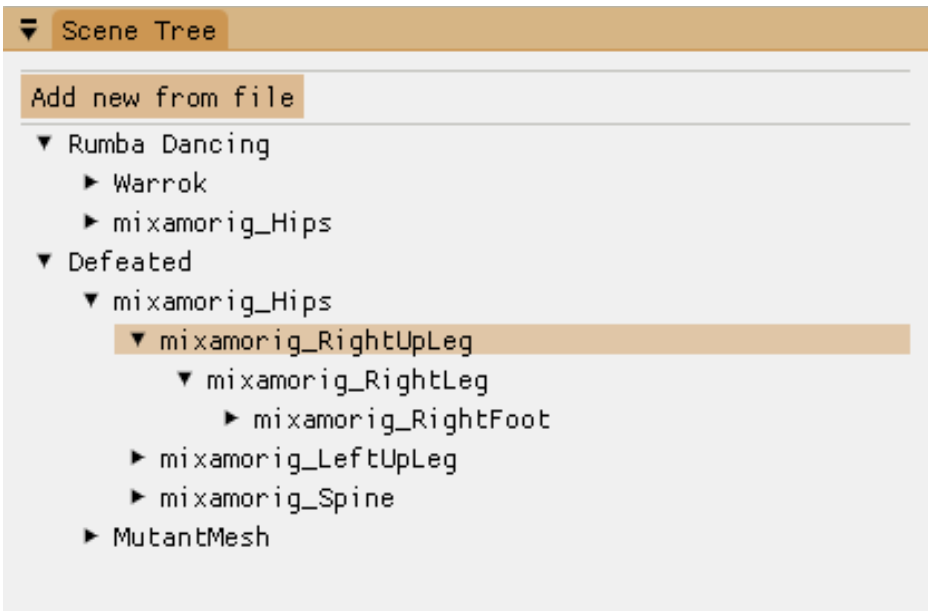


Рисунок 2.11 — Окно дерева сцены

Далее рассмотрим окно просмотра сцены и связанный с ним класс Viewport. По своей сути окно просмотра содержит только изображение опи-

сываемое классом Texture, на которое было отрисовано состояние сцены. Однако поскольку в нашем движке используется технология двойной буферизации, требуется ввести дополнительный класс отвечающий за смену изображений ViewportSwapChain.

Один «кадр», сменой которых управляет ViewportSwapChain, состоит из:

- color_buffers_ — цветового буфера за описание которого отвечает встреченный ранее класс dmbrn::Texture, однако в отличии от прошлых текстур созданная в этот раз имеет дополнительный флаг использования как цели для отрисовки (vk_ImageUsageFlagBits_ eColorAttachment);
- depth_buffer_ — буфера глубины, описывается новым классом dmbrn::DepthBuffer;
- imgui_images_ds — множеством дескрипторов полученных от ImGui с помощью метода ImGui_ImplVulkan_AddTexture с компоновкой изображений VK_IMAGE_LAYOUT_SHADER_READ_ONLY_ OPTIMAL, для каждого из цветовых буферов.

DepthBuffer главным образом отличается от Texture тем, что вместо RGB пикселей в карте глубины находятся float32 значения, также в случае, если включен буфер трафарета может содержать 8uint значение. И флаг использования как цель для глубинного и трафаретного буферов (vk::ImageUsageFlagBits::eDepthStencilAttachment).

Каждое окно просмотра выполняет отрисовку относительно положения своей камеры и выполняет её на свой цветовой буфер из ViewportSwapChain с размером соответствующим размеру окна просмотра. Для организации отрисовки на этот цветовой буфер нам снова требуется создать проход графики для каждого окна просмотра за это отвечает класс ViewportRenderPass. Причем в этот раз начальной и конечной компоновкой является ImageLayout::eShaderReadOnlyOptimal, чтобы ImGui мог использовать это

изображения в фрагментном шейдере при отрисовке UI.

Для наглядности описанные отношения предсавлены на рисунке 2.12 в виде диаграммы классов.

При отрисовке интерфейса ImGui данного окна, располагается только изображение ImGui::Image и далее происходит обработка и отрисовка инструмента гизмо с помощью класса ImGuizmo из библиотеки ImGuizmo [17].

Также давайте рассмотрим содержание метода отвечающего за отрисовку содержания окна — render. Данный метод сначала начинает проход рендера, затем проходит по всем статическим моделям, полученным от сцены и помещает их в очередь для отрисовки для лежащего в StaticModelComponent шейдера. Затем аналогичное с скелетными моделями. Затем вызывает отрисовку всех шейдеров и заканчивает проход рендеринга. Для большей наглядности диаграмма последовательности для данного метода представлена на рисунке 2.13.

Теперь давайте разберем окно инспектора. Если выбранная сущность существует, то мы проверяем наличие каждого компонента с помощью метода Enttity::tryGetComponent, если текущий компонент существует мы создаем вершину выпадающего дерева-списка с помощью метода ImGui::TreeNodeEx и далее вызывая нужные функции ImGui выводим информацию о компонентах.

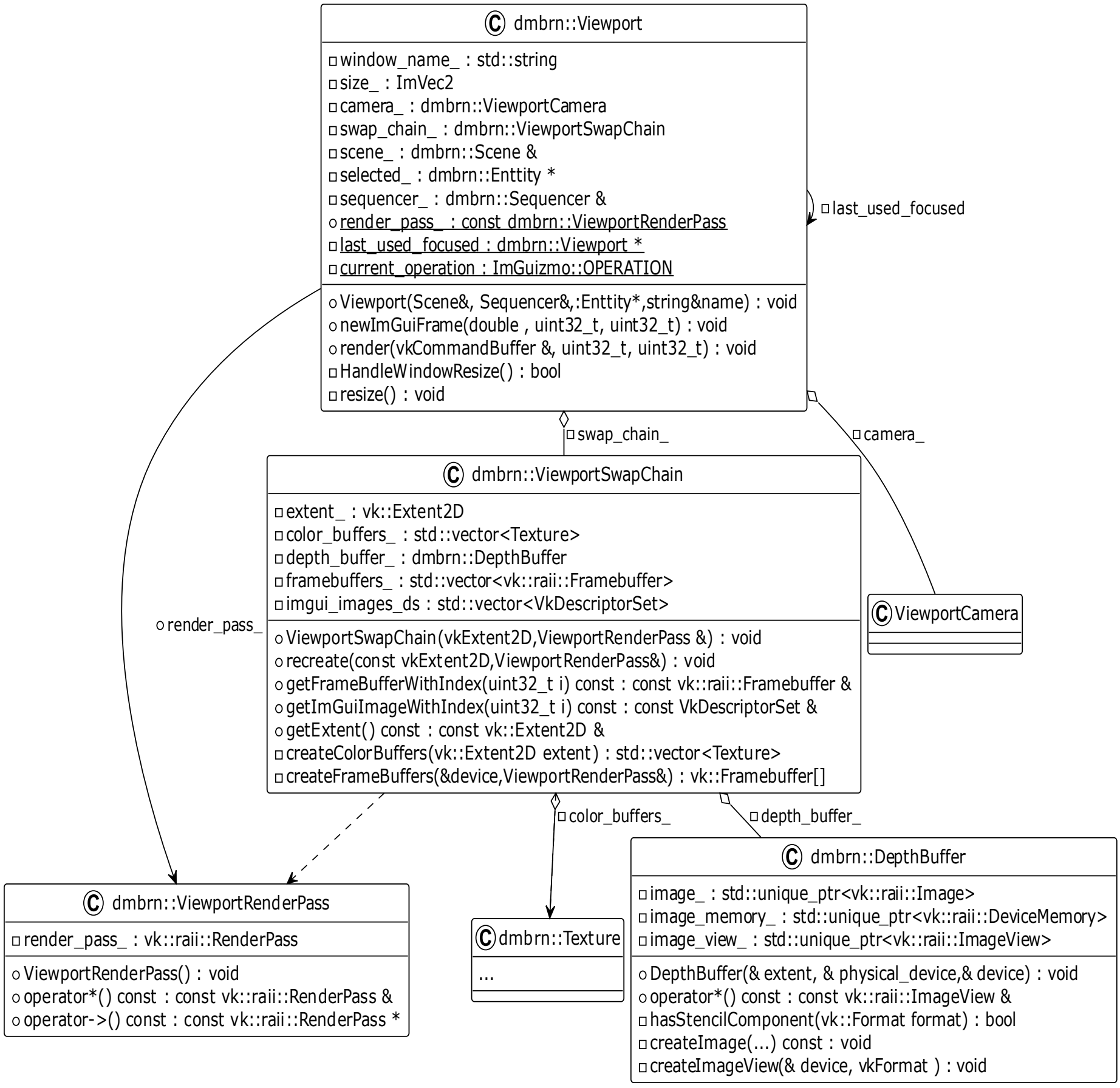


Рисунок 2.12 — Диаграмма классов, связанных с отрисовкой окна просмотра сцены

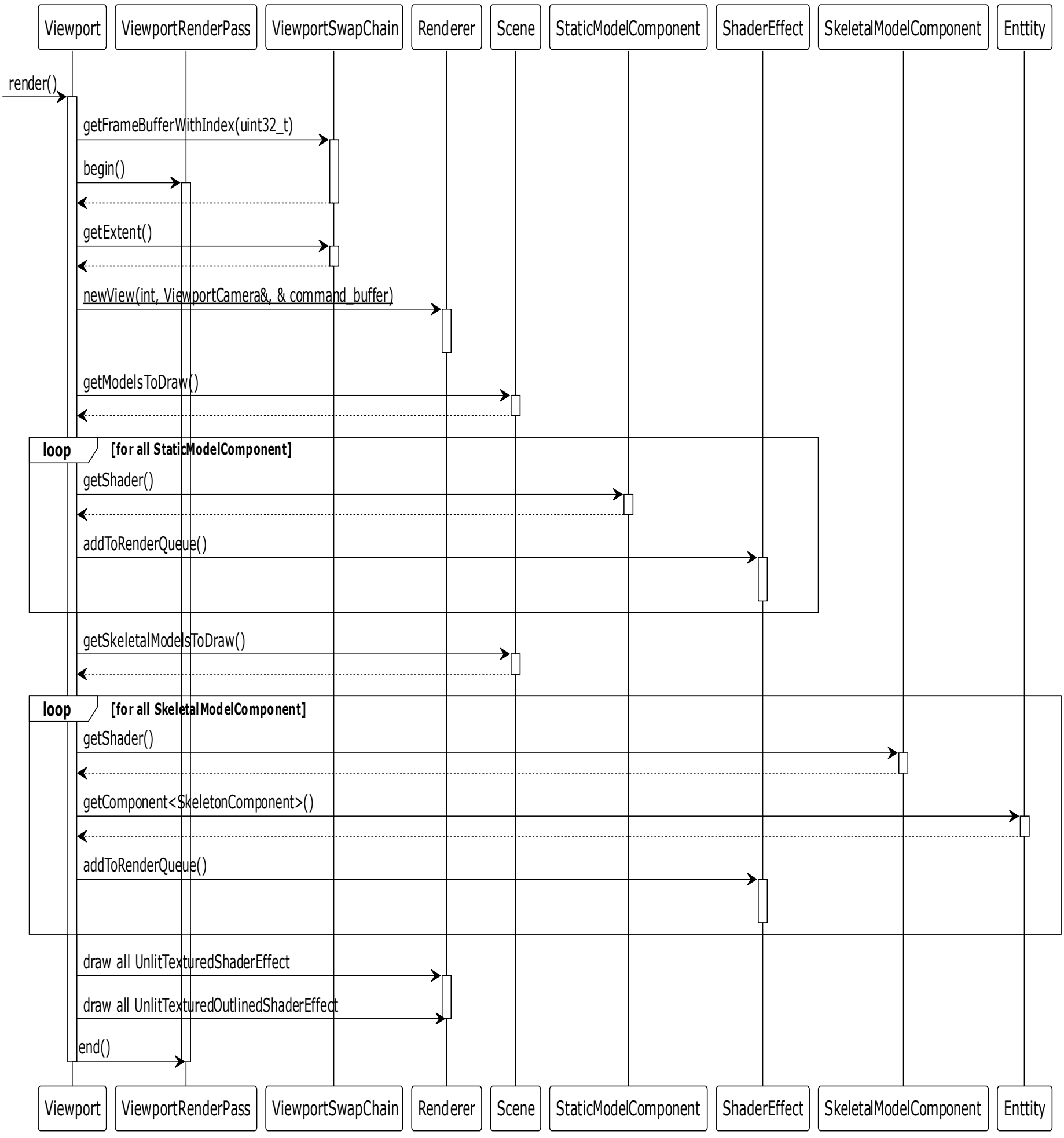


Рисунок 2.13 — Диаграмма последовательности функции отрисовки окна просмотра сцены

Наиболее интересным для рассмотрения является компонент анимаций (AnimationComponent), поскольку содержит источник перетаскивания (drag and drop). В ImGui таким источником может быть любой элемент интерфейса, в нашем случае им является ImGui::InputText, содержащий название анимации. Для того что бы объявить элемент перетаскивания нужно написать ImGui::BeginDragDropSource, если данная функция вернула истину, значит началось перетаскивание и нам требуется сформировать элемент для перетаскивания. В нашем случае это пара значений: идентификатор сущности с данным компонентом и указатель на данный анимационный клип, далее вызвав ImGui::SetDragDropPayload регистрируем его в ImGui. После чего груз (payload) может быть принят в DragDropTarget.

На рисунке 2.14 демонстрируется вид окна при выбранной сущности, имеющей AnimationComponent и SkeletonComponent. Цвета инвертированы для экономии чернил при печати.

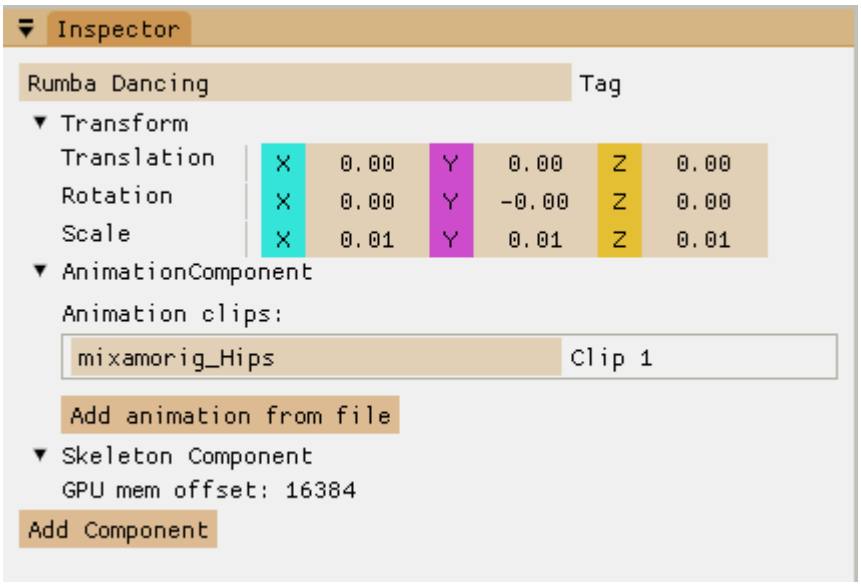


Рисунок 2.14 — Окно инспектора

Теперь давайте рассмотрим окно секвенсора и определяющий его класс Sequencer. Данный класс является сильно измененным и модифицированным ImSequencer из ImGuizmo [17]. Для его отрисовки используется возможности ImGui по созданию пользовательских виджетов при помощи списков отрисовки ImDrawList. Пользователь может наполнять списки различными

примитивами, например, прямоугольниками (AddRectFilled), текстом (AddText) и т. п.. При этом осуществляется абсолютное позиционирование с помощью координат экрана в пикселях. Для определения положения и размера окна используются функции ImGui::GetCursorScreenPos, ImGui::GetContentRegionAvail соответственно. Кроме этого неудобного фактора нужно учесть, что список отрисовки не имеет показателей глубины или слоев элементов и элементы отображаются в порядке их добавления в список.

Таким образом коротко отрисовку данного элемента как последовательность следующих действий:

- Отрисовка элементов управления с помощью стандартных функций ImGui. Элементы управления это поле ввода минимального, максимального и текущего кадра анимационного клипа, начала проигрывания или остановки.
- Обновление положений левого и правого концов полосы прокрутки\ масштаба (frameBarPixelOffsets), линейной интерполяцией от текущего до целевого.
- Определение первого видимого кадра firstFrame. Определение ширины кадра в пикселях.
- Отрисовка заднего фона.
- Определение передвижения пользователем текущего кадра при нажатии на верхнюю часть панели, обновление текущего кадра.
- Обновление текущего кадра, если проигрывается.
- Отрисовка верхней панели, линий цены деления и цифровых значений кадров.
- Отрисовка имен анимированных сущностей слева.
- Отрисовка вертикальных линий в зоне клипов.
- Отрисовка клипов каждой сущности в соответствующей позиции.
- Начало цели для сброса груза ImGui::BeginDragDropTarget.

- После того как мы приняли груз ImGui::AcceptDragDropPayload, мы можем проверить, навел ли только пользователь на цель IsPreview, или уже сбросил IsDelivery. В первом случае мы можем отрисовать для пользователя клип на временной линии. А уже после доставки добавить его к соответствующей сущности

- Отрисовать вертикальный прямоугольник текущего кадра.
- Отрисовать полосу прокрутки и обработать начало передвижения её самой или её концов.

На рисунке 2.15 демонстрируется вид окна секвенсора. Текущий кадр равен 60, Rumba Dancing имеет два клипа, Defeated один. Цвета инвертированы для экономии чернил при печати.

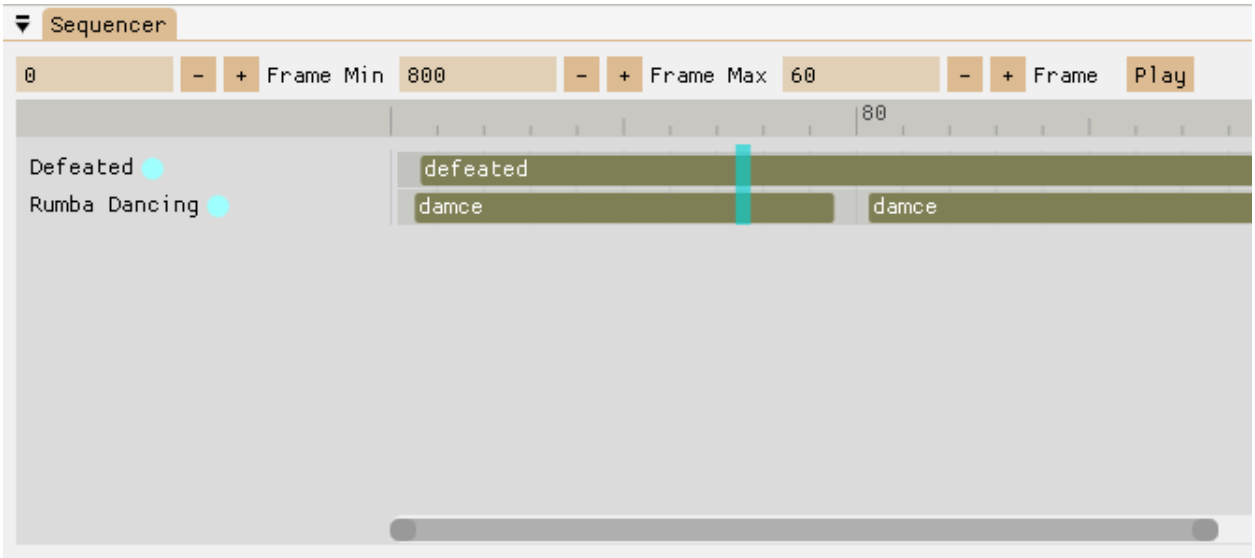


Рисунок 2.15 — Окно секвенсора анимаций

Последним давайте рассмотрим метод, собирающий все ранее описное воедино. Метод отрисовки всего интерфейса и обновления всех состояний сцены — EditorUI::drawFrame. На рисунке 2.16 приведена диаграмма последовательности данного метода. Дополнительно давайте коротко разберем последовательность действий выполняемые в данном методе:

- из EditorSwapChain берется следующий\текущий EditorFrame;
- выполняется ожидание барьера in_flight_fence на выполнение всех

прошлых команд отрисовки текущего кадра;

- выполняется захват изображения вызывающий сигнализирование семафора image_available_semaphore;
- вызов функций ImGui сигнализирующих о новом кадре;
- выполняется отрисовка каждого окна интерфейса;
- обновляются анимации, затем TransformComponent’ы объектов на сцене;
- обновляется данные на GPU;
- выполняется запись команд отрисовки окона просмотра сцены;
- выполняется запись команд отрисовки интерфейса ImGui;
- выполняется отправка буфера команд в очередь с семафором для ожидания image_available_semaphore и семафором для сигнализирования render_finished_semaphore;
- выполняется вывод на экран изображения с семафором ожидания render_finished_semaphore.

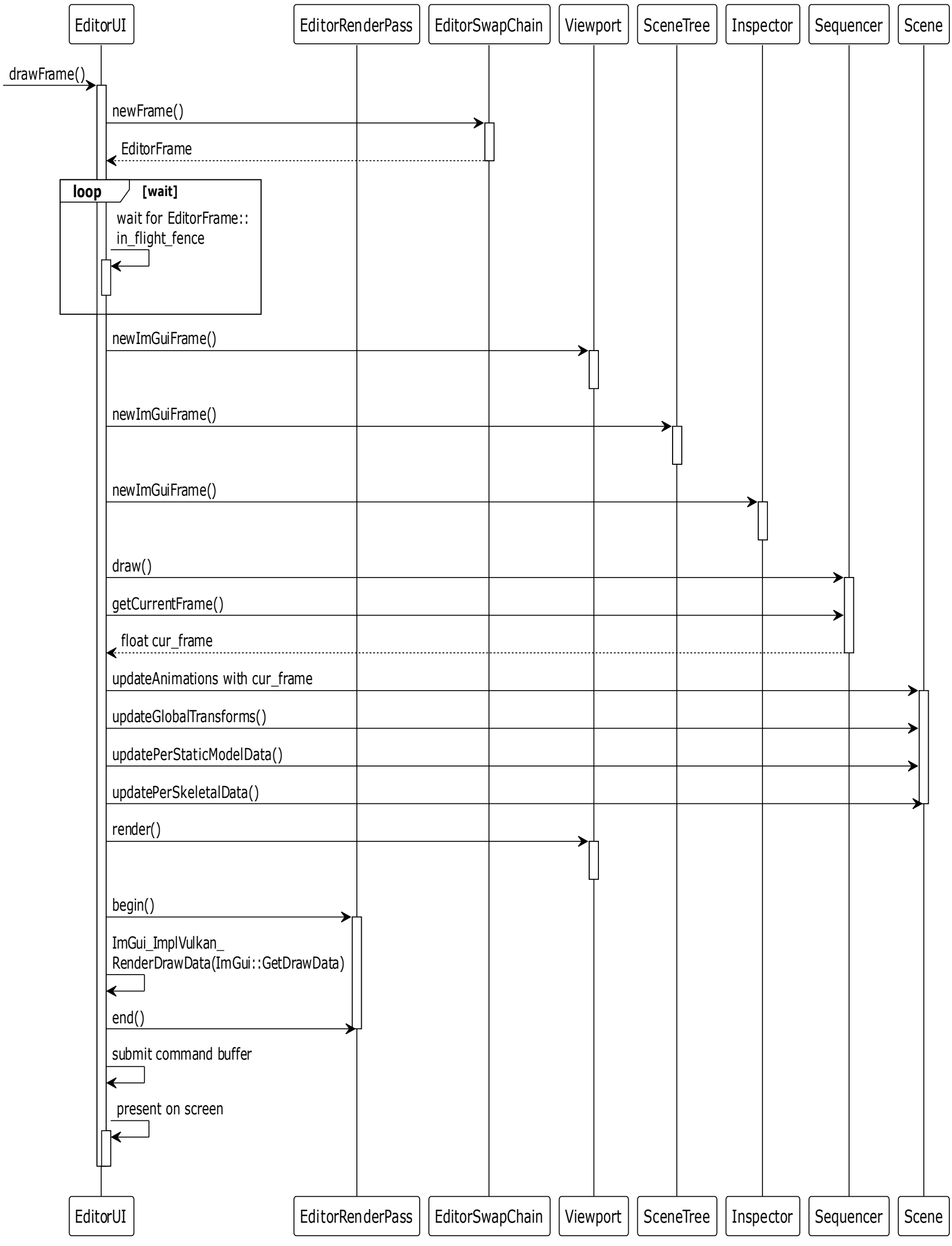


Рисунок 2.16 — Диаграмма последовательности метода отрисовки интерфейса

3 РАЗРАБОТКА И СТАНДАРТИЗАЦИЯ ПРОГРАММНЫХ СРЕДСТВ

В качестве дополнительного раздела был выбран раздел «Разработка и стандартизация программных средств», т.к. главная цель выпускной квалификационной работы заключается в разработке программного средства.

В этом разделе мы рассмотрим вопросы организации процесса проектирования программных средств (ПС) с использованием международных и отечественных методов, регулирующих основные этапы жизненного цикла ПС и определяющих требования к конечному продукту.

В данный раздел входят решения и результаты, полученные по следующим вопросам:

- планирование работ проекта, в рамках техно-рабочего проектирования с использованием диаграмм Ганта (ленточных диаграмм);
- определение кода разрабатываемого программного изделия в соответствии с общероссийскими классификаторами продукции;
- расчет затрат на выполнение и внедрение проекта, расчет цены проекта и цены предлагаемого программного продукта.

3.1 Планирование работ проекта

Планирование работ – это ключевая функция управления, необходимая для руководства процессами проектирования и внедрения ПС. Основные цели планирования работ заключаются в следующем:

- установление общего объема работ и порядка их реализации с учетом разных факторов, например, таких, как связи и зависимости между работами или время, требуемое для освобождения ресурсов;
- назначение исполнителей и соисполнителей для каждой работы;
- определение срока выполнения каждой работы и времени реализации всего проекта в целом.

Для формального описания набора запланированных работ существует несколько методов, основанных на визуализации процессов и позволяющих в

разной мере отслеживать выполнение работ и корректировать их организацию. Для целей планирования и управления проектами чаще всего применяют ленточные диаграммы (диаграммы Ганта), оперограммы и сетевые диаграммы (PERT — диаграммы).

Для визуализации набора запланированных работ мы будем использовать диаграмму Ганта, которая является столбчатой диаграммой с отрезками, соответствующими длительности работ. Кроме того, на диаграмме можно указать даты начала и конца каждой работы, а также ее исполнителя.

Прежде чем установить сроки работ и построить полную диаграмму, сформируем перечень работ, которые требуется сделать, для удобства расположим работы в соответствии с очередностью их выполнения:

1. формулирование технического задания;
2. поиск научной литературы по теме;
3. изучение научной литературы по теме;
4. выбор технологий и библиотек;
5. проектирование и разработка архитектуры программных модулей;
6. реализация программных модулей;
7. тестирование и отладка;
8. написание пояснительной записки и отзыва по работе;
9. подготовка иллюстрационных материалов;
10. проверка и подтверждение соответствия ГОСТ’ам;
11. проверка на плагиат.

Составим диаграмму Ганта со следующими показателями: длительность работы в днях, даты начала и завершения работы и перечень исполнителей. При составлении перечня исполнителей использовалась следующая кодировка имен: руководитель дипломного проектирования: А. И. Водяхо — АИВ, дипломник: О. В. Евдокимов — ОВЕ, консультант от кафедры ВТ И.С. Зуев — ИСЗ. Таким образом получили диаграмму, приведенную на рисунке 3.1.

Задача	Наименование	Дата начала	Дата завершения	Длительность (дни)	Исполнители
1	формулирование технического задания	04.04.23	04.04.23	1	ОВЕ, АИВ
2	поиск научной литературы по теме	05.04.23	09.04.23	5	ОВЕ
3	изучение научной литературы по теме	07.04.23	09.04.23	3	ОВЕ
4	выбор необходимых технологий и библиотек	10.04.23	11.04.23	2	ОВЕ
5	проектирование и разработка архитектуры программных модулей	12.04.23	25.04.23	14	ОВЕ
6	реализация программных модулей	20.04.23	02.05.23	13	ОВЕ
7	тестирование и отладка	03.05.23	03.05.23	1	ОВЕ, АИВ
8	написание пояснительной записки и отзыва по работе	04.05.23	21.05.23	18	ОВЕ
9	подготовка иллюстрационных материалов	22.05.23	23.05.23	2	ОВЕ
10	проверка и подтверждение соответствия ГОСТу	24.05.23	24.05.23	1	ОВЕ, ИСЗ
11	проверка на плагиат	25.05.23	25.05.23	1	ОВЕ

Рисунок 3.1 — Диаграмма Ганта

3.2 Определение кода разрабатываемого программного средства

Классификаторы продукции нужны для того, чтобы разработчики могли рассказать потенциальным покупателям о своих продуктах, а потенциальные покупатели могли легко найти среди многочисленных рыночных предложений тот продукт, который им подходит.

С момента образования нашего государства в сфере программного обеспечения были разработаны и впоследствии отменены несколько различных классификаторов. Мы остановимся на рассмотрении только двух из них:

- Первый общероссийский классификатор продукции ОКП (ОК 005-93) действовавший до 2017;
- Актуальный классификатор ОКПД 2 введенный в 2014 году.

Рассмотрение отмененного классификатора может быть полезно. Поскольку, программы, разработанные до 2017 года, сохраняют код, соответствующий данному классификатору. Поэтому для нахождения аналогов программного средства, созданного в рамках ВКР, полезно рассмотреть и уже отменный классификатор. Кроме того, он дает более точную классификацию программного средства, чем действующий.

Коды по обоим классификаторам состоят из цифр десятичной системы. Классификация является иерархической движение от старших разрядов к младшим уточняет классификацию.

Проведем определение кода нашего программного средства по действующему классификатору ОКПД 2, структура которого представлена в таблице 3.1.

Таблица 3.1 — Структура кода ОКПД2

Положение в коде (обозначено 0)	Название уровня
00.XX.XX.XXX	Класс
XX.0X.XX.XXX	Подкласс
XX.X0.XX.XXX	Группа
XX.XX.0X.XXX	Подгруппа
XX.XX.X0.XXX	Вид
XX.XX.XX.00X	Категория
XX.XX.XX.XX0	Подкатегория

По нему программное средство принадлежит к:

- классу 62 с расшифровкой: «продукты программные и услуги по разработке программного обеспечения; консультационные и аналогичные услуги в области информационных технологий»;
- подклассу 0 с расшифровкой: «продукты программные и услуги по разработке программного обеспечения; консультационные и аналогичные услуги в области информационных технологий»;
- группе 1 с расшифровкой: «продукты программные и услуги по разработке и тестированию программного обеспечения»;
- подгруппе 2 с расшифровкой: «оригиналы программного обеспечения»;
- виду 9 с расшифровкой: «оригиналы программного обеспечения прочие»;
- категории и подкатегории 000 с расшифровкой: «оригиналы программного обеспечения прочие».

Таким образом собирая все составляющие кода воедино получим: 62.01.29.000.

Теперь давайте проведем определение кода по классификатору ОКП, структура которого представлена на рисунке 3.2.

X X	X	X	X	X	КЧ	Наименование продукции
Класс продукции	Подкласс	Группа	Подгруппа	Вид продукции	Контрольное число	

Рисунок 3.2 — Структура кода ОКП

По классификатору ОКП программное средство принадлежит к:

- классу 50 с расшифровкой: «программные средства и информационные продукты вычислительной техники»;
- подклассу 2 с расшифровкой: «программные средства общего назначения»;
- группе 6 с расшифровкой: «программные средства инструментальные для систем мультимедиа»;
- подгруппе 1 с расшифровкой: «программные средства инструментальные для проектирования элементов мультимедиа, пояснение: в данную группировку входят пс для проектирования видеоизображения, звука, машинной графики и прочих элементов мультимедиа».

Таким образом собирая все составляющие кода воедино получим: 502610.

3.3 Определение затрат на выполнение и внедрение проекта

Для того чтобы оценить себестоимость и цену проекта в сфере разработки ПС на начальной стадии договора с потенциальным заказчиком (инвестором), мы будем использовать методику укрупненного расчета. Эта методика позволяет определить стоимость проекта на основе стоимости одного рабочего дня проектировщика и трудозатрат (трудоемкости) проекта в целом. При этом мы учитываем состав и степень загрузки участников команды проекта.

Условия, которые мы будем принимать при расчете затрат на проект и цены проекта:

- Основным разработчиком проекта (ВКР) является дипломник выполняющий проектирование в сроки: с 4 апреля 2023 по 25 мая 2023, что составляет 34 рабочих дня с полной загрузкой ($К_{загр.разр}=1,0$).
- В проекте принимают участие руководитель ВКР с коэффициентом загрузки $К_{загр.рук}=0,05$ и консультант по дополнительному разделу с коэффициентом загрузки $К_{загр.доп}=0,03$. Зарплата этих участников принимается в соответствии с их должностями в вузе. Процент накладных расходов для ЛЭТИ условно принимается 42%.

Далее нам нужно определить месячную зарплату. В качестве источника данных будем использовать карьерный сервис “Хабр Карьера” (https://career.habr.com/salaries). Так по данным этого сервиса средняя зарплата по всем ИТ-специализациям на основании 6754 анкет за 1-е полугодие 2023 года составляет 179 307 руб.

Для начинающего проектировщика, выполняющего ВКР, скорректируем полученное значение зарплаты до 60 000 руб.

В соответствии с приказом № 2206 от 04.10.2013 «О повышении уровня оплаты труда работников университета», норматив финансирования для штатных единиц профессорско-преподавательского состава составляет 25 000 (в месяц).

Перед началом расчетов определим необходимые постоянные:

- $Т_{ср}=20,58$ — среднемесячное число рабочих дней для 2023 года, при пятидневной рабочей недели;
- $Ф=0,302$ — процент (доля) страховых взносов, исчисляемых от фонда заработной платы, включает в себя взносы в ФНС и ФСС;
- $П=0,15$ — средняя прибыль исполнителя при выполнении проектов в сфере ИТ;
- $Н=0,5$ — процент накладных расходов, для проектных организаций в сфере информатики колеблется, как правило, от 40% до 80%;
- $НДС=0,2$ — ставка налога на добавленную стоимость.

Произведем расчет полных затрат в день на разработчика, представлены в таблице 3.2, и штатного сотрудника ЛЭТИ, представлены в таблице 3.3. В таблицах также приведены расчетные формулы.

Таблица 3.2 — Расчет ставки разработчика

Наименование статьи	ед.изм	Затраты, руб	Примечание
Величина среднемесячной начисленной заработной платы специалиста	руб./месяц	60 000,00	Оклад сотрудника $Z_{зп}$
Расчет ставки			
Тарифная ставка дневная ($Z_д$)	руб./день	2 915,45	$Z_д = Z_{зп} / T_{ср}$
Страховые взносы 30,2% от суммы зарплаты работников ($C_{сд}$)	руб./день	880,47	$C_{сд} = Z_д * Ф$
Оплата основных работников со страховыми взносами ($Z_{дс}$)	руб./день	3 795,92	$Z_{дс} = Z_д + C_д$
Накладные расходы ($C_{нр}$)	руб./день	1 457,73	$C_{нр} = Z_д * Н$
Себестоимость одного человек/дня ($C_{ч/д}$)	руб./день	5 253,64	$C_{ч/д} = Z_{дс} + C_{нр}$
Дневная прибыль ($C_{прд}$)	руб./день	788,05	$C_{прд} = C_{ч/д} * П$
Ставка специалиста без учета НДС ($C_{дсс}$)	руб./день	6 041,69	$C_{дсс} = C_{ч/д} + C_{прд}$
Дневная сумма НДС ($C_{ндс}$)	руб./день	1 208,34	$C_{ндс} = C_{дсс} * НДС$
Ставка специалиста в день с учётом НДС ($C_{полн.разр}$)	руб./день	7 250,03	$C_{полн.разр} = C_{дсс} + C_{ндс}$

Теперь мы можем определить цену проекта, используя следующую формулу:

$$C_{пр} = \sum_{i=1}^{n} C_{полні} T_i K_{загрі},$$

где:

- $C_{полні}$ — полная дневная стоимость работы i-го специалиста [руб./день];
- T_i — время участия i-го специалиста в работе над проектом [дней];
- $K_{загрі}$ — коэффициент загрузки i-го специалиста работами в проекте;
- n — число специалистов, занятых в проекте.

Применяя данную формулу к нашему случаю получим следующую расчетную формулу, где первое слагаемое затраты на разработчика, второе слагаемое на руководителя и третье слагаемое на двух консультантов:

$$C_{пр} = 7250{,}03 * 34 * 1{,}0 \ + \ 2886{,}73 * 2 * 0{,}05 \ + \ 2886{,}73 * 2 * 0{,}03 = 246962{,}89 \text{ руб}.$$

Таблица 3.3 — Расчет ставки сотрудника ЛЭТИ

Наименование статьи	ед.изм	Затраты, руб	Примечание
Величина среднемесячной начисленной заработной платы специалиста	руб./месяц	25 000,00	Оклад сотрудника $Z_{зп}$
Расчет ставки			
Тарифная ставка дневная ($Z_д$)	руб./день	1 214,77	$Z_д = Z_{зп} / T_{ср}$
Страховые взносы 30,2% от суммы зарплаты работников ($C_{сд}$)	руб./день	366,86	$C_{сд} = Z_д * Ф$
Оплата основных работников со страховыми взносами ($Z_{дс}$)	руб./день	1 581,63	$Z_{дс} = Z_д + C_д$
Накладные расходы ($C_{нр}$)	руб./день	510,20	$C_{нр} = Z_д * Н$
Себестоимость одного человек/дня ($C_{ч/д}$)	руб./день	2 091,84	$C_{ч/д} = Z_{дс} + C_{нр}$
Дневная прибыль ($C_{прд}$)	руб./день	313,78	$C_{прд} = C_{ч/д} * П$
Ставка специалиста без учета НДС ($C_{дсс}$)	руб./день	2 405,61	$C_{дсс} = C_{ч/д} + C_{прд}$
Дневная сумма НДС ($C_{ндс}$)	руб./день	481,12	$C_{ндс} = C_{дсс} * НДС$
Ставка специалиста в день с учётом НДС ($C_{полн.штат}$)	руб./день	2 886,73	$C_{полн.штат} = C_{дсс} + C_{ндс}$

Если новое (созданное в проекте) программное обеспечение выпускается на рынок как заказное решение и используется метод ценообразования на основе затрат, то его цену можно найти из следующего выражения:

$$C_{прогр} = C_{пр} + C_{изгот}(1 + П)(1 + НДС),$$

где:

- $C_{пр}$ — цена проекта, рассчитанная ранее;
- $C_{изгот}$ — затраты на копирование (изготовлнеие копий), примем данное значение $C_{изгот} = 200$ руб, как включающее в себя затраты на

хостинг сервера с раздачей установочных файлов программы и производство одного CD-диска с программой;

- $П = 0{,}5$ — прибыль, заложенная разработчиком в цену.

Тогда расчетная формула:

$$C_{прогр} = 246962{,}89 + 200 * 1{,}5 * 1{,}2 = 247322{,}89 \text{ руб}$$

Когда проект подразумевает продажу программы нескольким потребителям, то расходы на проект (с учетом прибыли продавца) следует поделить между покупателями, по следующей формуле:

$$C_{пргр.тир} = \frac{C_{прогр}}{N_{тир}},$$

где:

- $C_{пргр.тир}$ — цена программы при ее тиражировании;
- $N_{тир}$ — планируемый(гарантируемый) тираж реализации.

Примем $N_{тир} = 100$, тогда получим следующую расчетную формулу для цены программы:

$$C_{пргр.тир} = \frac{247322{,}89}{100} = 2473{,}22 \text{ руб}$$

ЗАКЛЮЧЕНИЕ

В результате работы было создано программное средство, позволяющее пользователю легко и быстро комбинировать анимации для различных персонажей из библиотек готовых анимаций, таких, как Mixamo. Программное средство имеет интуитивно понятный графический интерфейс, поддерживает импорт 3D-моделей и их анимаций в различных форматах, а также предоставляет возможность просмотра и редактирования последовательности воспроизведения анимаций для любого объекта на сцене.

Таким образом, цель работы была достигнута, а все поставленные задачи были решены. Разработанное программное средство является вкладом в развитие области 3D-анимации.

Возможными направлениями дальнейшего развития темы являются:

- Возможность сохранения сцены и созданных последовательностей анимаций в файл на диске.
- Возможность экспорта полученных анимационных роликов в различные видео форматы.
- Добавление функционала для создания собственных анимаций или редактирования существующих.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Главная страница Blender [Электронный ресурс] — Режим доступа: https//www.blender.org/. (Дата обращения: 12.04.2023).

2. Главная страница Unity [Электронный ресурс] — Режим доступа: https://unity.com/ru. (Дата обращения: 12.04.2023).

3. Assimp GitHub [Электронный ресурс] — Режим доступа: https://github.com/assimp/assimp. (Дата обращения: 12.04.2023).

4. Dear ImGui GitHub [Электронный ресурс] — Режим доступа: https://github.com/ocornut/imgui. (Дата обращения: 12.04.2023).

5. Графические API высокого и низкого уровня: различия и принцип работы [Электронный ресурс] — Режим доступа: https://itigic.com/ru/high-and-low-level-graphical-apis-differences/. (Дата обращения: 4.05.2023).

6. Графический API Vulkan представлен, и ваши графические процессоры NVIDIA к нему готовы [Электронный ресурс] — Режим доступа: https://www.nvidia.com/ru-ru/drivers/vulkan-graphics-api-blog/. (Дата обращения: 04.05.2023).

7. GLM github [Электронный ресурс] — Режим доступа: https://github.com/g-truc/glm. (Дата обращения: 12.04.2023).

8. Что такое ECS и с чем его компилят [Электронный ресурс] — Режим доступа: https://dtf.ru/gamedev/954579-chto-takoe-ecs-i-s-chem-ego-kompilyat. (Дата обращения: 04.05.2023).

9. GitHub Entt [Электронный ресурс] — Режим доступа: https://github.com/skypjack/entt. (Дата обращения: 12.04.2023).

10. stb github [Электронный ресурс] — Режим доступа: https://github.com/nothings/stb. (Дата обращения: 12.04.2023).

11. IMGUI wiki на GitHub [Электронный ресурс] — Режим доступа: https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm. (Дата обращения: 04.05.2023).

12. GLFW GitHub [Электронный ресурс] — Режим доступа:

https://github.com/glfw/glfw. (Дата обращения: 04.05.2023).

13. Jason Gregory . Game engine architecture. 2018.

14. Селлерс, Г. . Vulkan. Руководство разработчика. 2017.

15. Vulkan Shader Resource Binding [Электронный ресурс] — Режим доступа: https://developer.nvidia.com/vulkan-shader-resource-binding. (Дата обращения: 12.04.2023).

16. Vulkan® 1.3.250 - A Specification [Электронный ресурс] — Режим доступа: https://registry.khronos.org/vulkan/specs/1.3/html/. (Дата обращения: 12.04.2023).

17. ImGuizmo GitHub [Электронный ресурс] — Режим доступа: https://github.com/CedricGuillemet/ImGuizmo. (Дата обращения: 12.04.2023).

ПРИЛОЖЕНИЕ А
Спецификация и исходный код программы

Спецификация программного продукта приведена в таблице А.1.

Таблица А.1 — Спецификация программного продукта

Идентификатор модуля	Назначение модуля
Singletons	Контейнер для одиночек
Instance	Обертка для vkInstance и отладочных слоев валидации
Surface	Обертка для SurfaceKHR
PhysicalDevice	Обертка для vkPhysicalDevice
LogicalDevice	Обертка для vkDevice
CommandPool	Обертка для vkCommandPool
DescriptorPool	Обертка для vkDescriptorPool
MaterialRegistryHandle	Ключ в реестре материалов
DiffusionMaterial	Описывает свойства поверхности объекта
Texture	Управляет данными текстурных изображений на GPU
DiffusionUniformBuffer	Управляет простыми данными на GPU диффузного материала
DiffusionDescriptorSets	Описывает и хранит множества дескрипторов диффузного материала
image_data	Представляет данные изображения на CPU
MeshRenderData	Управляет данными рендеринга сетки на GPU
Mesh	Просто комбинация материала и данных рендеринга сетки
HostLocalBuffer	Представляет локальный буфер GPU
Vertex	Описывает одну вершину статической сетки
SkeletalMeshRenderData	Управляет данными рендеринга скелетной сетки на GPU
SkeletalMesh	Просто комбинация материала и данных рендеринга скелетной сетки
BonedVertex	Описывает одну вершину скелетной сетки
Renderer	Управляет и хранит объекты, необходимые для рендеринга
ShaderEffect	Описывает общий интерфейс для дальнейших шейдерных эффектов
UnlitTexturedShaderEffect	Описывает эффект шейдера для рисования объекта как неосвещенной текстуры

Продолжение таблицы А.1

Идентификатор модуля	Назначение модуля
UnLitTexturedGraphicsPipelineStatics	Управляет объектами, необходимыми для рисования с UnlitTexturedShaderEffect
UnLitTexturedGraphicsPipeline	Помогает в создании графического конвейера для UnlitTexturedShaderEffect
UnLitTexturedRenderData	Управляет данными рендеринга UnlitTexturedShaderEffect на GPU
UnlitTexturedOutlinedShaderEffect	Описывает эффект шейдера для рисования объекта в виде очерченной неосветленной текстуры
OutlineGraphicsPipelineStatics	Управляет объектами, необходимыми для рисования с UnlitTexturedOutlinedShaderEffect
OutlineGraphicsPipeline	Помогает в создании графического конвейера для рисования контура объекта
OutlineShaderEffectRenderData	Управляет данными контурного рендеринга на GPU
PerStaticModelData	Управляет едиными данными GPU для каждой статической модели в сцене
PerSkeletonData	Управляет едиными данными GPU для каждой скелетной модели в сцене
ViewportCamera	Представляет камеру окна просмотра сцены
CameraRenderData	Управляет данными GPU камеры окна просмотра сцены
UniformBuffer	Представляет собой CPU видимый и когерентный буфер данных на GPU
Enttity	Является оберткой для entt::entity идентификатора для удобства
TagComponent	Имя сущности
TransformComponent	Представляет собой преобразование сущности
RelationshipComponent	Описывает иерархические отношения между сущностями
BoneComponent	Описывает сущность, которая является костью некоторого скелета
SkeletonComponent	Сущность, имеющая такой компонент, является корнем скелета
StaticModelComponent	Описывает сущность с визуальным представлением в виде статической модели

Продолжение таблицы А.1

Идентификатор модуля	Назначение модуля
SkeletalModelComponent	Описывает сущность с визуальным представлением в виде скелетной модели
CameraComponent	Описывает свойства камеры
AnimationComponent	Присутствие означает, что эта сущность и ее поддерево могут быть анимированы
AnimationClip	Хранит каналы анимации для всех сущностей, участвующих в этом клипе
AnimationChannels	Хранит значения ключевых кадров для одной сущности
Scene	Является контейнером и менеджером всех сущностей
ModelImporter	Выполняет импорт данных модели из файла
AnimationSequence	Хранит и управляет последовательностью анимационных клипов для каждого анимированного объекта в сцене
EditorUI	Является основным оркестратором и контейнером для всех элементов пользовательского интерфейса
EditorFrame	Представляет собой кадр пользовательского интерфейса редактора с данными синхронизации
EditorRenderPass	Представляет собой проход рендеринга пользовательского интерфейса редактора
EditorSwapChain	Представляет собой список подкачки пользовательского интерфейса редактора
ImGuiRaii	Обертка RAII для объектов ImGui
SceneTree	Представляет окно пользовательского интерфейса с деревом сцены
Viewport	Представляет окно пользовательского интерфейса просмотра сцены
ViewportRenderPass	Представляет собой проход рендеринга окна просмотра сцены
ViewportSwapChain	Представляет собой список подкачки окна просмотра сцены
DepthBuffer	Представляет собой буфер глубины окна просмотра сцены
Inspector	Представляет собой окно пользовательского интерфейса инспектора
Sequencer	Представляет собой окно пользовательского интерфейса секвенсора анимации

Ввиду ограниченности объема ВКР ниже распечатываются только основные модули: EditorUI::newFrame, EditorUI::render, EditorUI::submitAndPresent, EditorUI::drawFrame, Viewport::render, Scene::updateAnimations,

Scene::updateGlobalTransforms, Scene::dirtyTraverseTree, Scene::editedTraverse-Tree, UnlitTexturedShaderEffect::draw, UnlitTexturedShaderEffect::drawStatic.

```
/**
 * \brief get an image index for frame and be sure that all synchronization
 * is done
 * \param device vulkan logical device for fence manipulation
 * \param frame editor ui frame data
 * \return index of acquired image
 */
uint32_t EditorUI::newFrame(const LogicalDevice& device, const EditorFrame&
  frame)
{
  // wait while all previous work for this frame wasn't done
  device->waitForFences(*frame.in_flight_fence, true, UINT64_MAX);

  // acquire image and signal semaphore when we can start render to it
  const auto result = swap_chain_->acquireNextImage(UINT64_MAX,
  *frame.image_available_semaphore);

  device->resetFences(*frame.in_flight_fence);

  frame.command_buffer.reset();

  // ImGui new frame
  ImGui_ImplVulkan_NewFrame();
  ImGui_ImplGlfw_NewFrame();
  ImGui::NewFrame();
  ImGuizmo::BeginFrame();

  return result.second;
}
/**
 * \brief record render commands to frame command buffer
 * \param frame editor frame for command buffer access
 * \param imageIndex swap chain image index of frame buffer
 */
void EditorUI::render(const EditorFrame& frame, uint32_t imageIndex)
{
  const vk::raii::CommandBuffer& command_buffer = frame.command_buffer;

  command_buffer.begin({vk::CommandBufferUsageFlags()});

  // record commands of viewports
  viewport_.render( command_buffer, current_frame_, imageIndex);
  viewport2_.render( command_buffer, current_frame_, imageIndex);

  // begin imgui render pass
  vk::ClearValue clearValue;
  clearValue.color = vk::ClearColorValue(std::array<float, 4>({0.5f, 0.5f,
   0.5f, 1.0f}));
  command_buffer.beginRenderPass({
   **render_pass_,
   *swap_chain_.getFrame(imageIndex).frame_buffer,
    {{0, 0}, swap_chain_.getExtent()},
    1, &clearValue
    }, vk::SubpassContents::eInline);
  // record imgui commands
  ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), *command_buffer);

  command_buffer.endRenderPass();
```

```
  command_buffer.end();
}

/**
 * \brief submit command buffer to queue and call present
 * \param present present queue
 * \param graphics graphics queue
 * \param window window wrapper to handle window resize
 * \param frame editor frame data
 * \param imageIndex swap chain image index for presenting
 */
void EditorUI::submitAndPresent(vk::raii::Queue& present,
      vk::raii::Queue& graphics, GLFWwindowWrapper& window,
      const EditorFrame& frame, uint32_t imageIndex)
{
  const vk::Semaphore waitSemaphores[] = {*frame.image_available_semaphore};
  const vk::PipelineStageFlags waitStages[] =
    {vk::PipelineStageFlagBits::eColorAttachmentOutput};
  const vk::Semaphore signalSemaphores[] =
    {*frame.render_finished_semaphore};

  const vk::SubmitInfo submitInfo
  {
    waitSemaphores,
    waitStages,
    *frame.command_buffer,
    signalSemaphores
  };

  graphics.submit(submitInfo, *frame.in_flight_fence);

  try
  {
    const vk::PresentInfoKHR presentInfo
    {
      signalSemaphores,
      **swap_chain_,
      imageIndex
    };
    present.presentKHR(presentInfo);
  }
  catch (vk::OutOfDateKHRError e)
  {
    window.framebufferResized = false;
    swap_chain_.recreate(render_pass_);
  }
}

/**
 * \brief draw UI, update scene both CPU and GPU states, record and submit
 * command buffers
 * \param delta_time time of previous frame in ms
 */
void EditorUI::drawFrame(double delta_time)
{
  // get current Editor Frame from swap chain
  const EditorFrame& frame = swap_chain_.getFrame(current_frame_);

  // get an image index of it and be sure that all synchronization is done
  const uint32_t imageIndex = newFrame(Singletons::device, frame);

  // begin drawing all the UI
```

```
  beginDockSpace();

  showAppMainMenuBar();
  ImGui::ShowDemoWindow();

  // draw all windows
  viewport_.newImGuiFrame(delta_time, current_frame_, imageIndex);
  viewport2_.newImGuiFrame(delta_time, current_frame_, imageIndex);
  scene_tree_.newImGuiFrame();
  inspector_.newImGuiFrame(current_frame_);
  drawStatsWindow();
  drawSequencer(static_cast<float>(delta_time));

  // end drawing all the UI
  endDockSpace();

  // update transforms according to current animation states
  scene_.updateAnimations(sequencer_.getCurrentFrame(), current_frame_);
  // hierarchically update transforms
  scene_.updateGlobalTransforms(current_frame_);
  // update GPU data of static models
  scene_.updatePerStaticModelData(current_frame_);
  // update GPU data of skeletal modes
  scene_.updatePerSkeletalData(current_frame_);

  // record render commands to frame command buffer
  render(frame, imageIndex);
  // submit command buffer to queue and call present
  submitAndPresent(Singletons::present_queue, Singletons::graphics_queue,
   Singletons::window, frame, imageIndex);
  current_frame_=(current_frame_ + 1)%
   Singletons::device.MAX_FRAMES_IN_FLIGHT;
}
/**
 * \brief record commands drawing scene to viewport
 * \param command_buffer command buffer to record comands
 * \param current_frame current frame index to bind proper descriptor sets
 * \param imageIndex swap chain image index to access frame buffer
 */
void Viewport::render(const vk::raii::CommandBuffer& command_buffer,
            uint32_t current_frame,
            uint32_t imageIndex)
{
  // set color of background
  const std::array<vk::ClearValue, 2> clear_values
  {
    vk::ClearValue{vk::ClearColorValue{std::array<float, 4>
       {0.3f, 0.3f, 0.3f, 1.0f}}},
    vk::ClearValue{vk::ClearDepthStencilValue{1.0f, 0}}
  };

  // begin viewport render pass with proper frame buffer
  const vk::RenderPassBeginInfo renderPassInfo
  {
    **render_pass_,
    *swap_chain_.getFrameBufferWithIndex(imageIndex),
    vk::Rect2D{vk::Offset2D{0, 0}, swap_chain_.getExtent()},
    clear_values
  };
  command_buffer.beginRenderPass(renderPassInfo,
     vk::SubpassContents::eInline);
```

```
  // set dynamic viewport
  const vk::Viewport viewport
  {
    0.0f, 0.0f,
    static_cast<float>(swap_chain_.getExtent().width),
    static_cast<float>(swap_chain_.getExtent().height),
    0.0f, 1.0f
  };
  command_buffer.setViewport(0, viewport);

  // set dynamic scissors
  const vk::Rect2D scissor
  {
    vk::Offset2D{0, 0},
    swap_chain_.getExtent()
  };
  command_buffer.setScissor(0, scissor);

  // update and bind new view
  Renderer::newView(current_frame, camera_, command_buffer);

  // add static models to corresponding shader queue
  auto static_view = scene_.getModelsToDraw();
  for (auto entity : static_view)
  {
    StaticModelComponent& model = static_view.get<StaticModelComponent>
      (entity);
    model.getShader()→addToRenderQueue(
      {&model.mesh, model.inGPU_transform_offset});
  }

  // add skeletal models to corresponding shader queue
  auto skeletal_group = scene_.getSkeletalModelsToDraw();
  for (auto entity : skeletal_group)
  {
    SkeletalModelComponent& skeletal_model = skeletal_group.get<
      SkeletalModelComponent>(entity);
    skeletal_model.getShader()->addToRenderQueue({
      &skeletal_model.mesh,
      skeletal_model.skeleton_ent.
        getComponent<SkeletonComponent>().in_GPU_mtxs_offset
    });
  }

  // draw all shader effects
  Renderer::un_lit_textured.draw(current_frame, command_buffer);
  Renderer::outlined_.draw(current_frame, command_buffer);

  command_buffer.endRenderPass();
}

/**
 * \brief update transforms according to current animation states
 * \param anim_frame current global animation frame
 * \param frame index of inflight frame
 */
void Scene::updateAnimations(float anim_frame, uint32_t frame)
{
  auto view = registry_.view<AnimationComponent>();

  // iterate all animated entities
  for (auto ent : view)
```

```
  {
    // if it has some animations in sequence
    if (!animation_sequence_.entries_[Enttity{registry_, ent}].empty())
    {
      // get clip which start time is grater or equal to global time
      auto clip_it = animation_sequence_.entries_
         [Enttity{registry_, ent}].lower_bound(anim_frame);

      // if it is past-the-end or
      // (is not the first in sequence and not equal)
      if (clip_it == animation_sequence_.entries_[
             Enttity{registry_, ent}].end() ||
        clip_it != animation_sequence_.entries_[
           Enttity{registry_, ent}].begin()&&clip_it->first!=anim_frame)
        // move back to get less or equal
        --clip_it;

      // here clip_it have less or equal

      // calculate clip local time
      const float local_time = glm::clamp(clip_it->second.min +
           anim_frame - clip_it->first,
           clip_it->second.min,
           clip_it->second.max);

      // actually updating transform with local time
      clip_it->second.updateTransforms(local_time, frame);
    }
  }
}

/**
 * \brief hierarchically update transforms
 * \param frame index of current in flight frame
 */
void Scene::updateGlobalTransforms(uint32_t frame)
{
  // begin traversing tree starting from scene root
  dirtyTraverseTree(scene_root_, frame);
}

/**
 * \brief traversing all dirty paths in tree to find edited entities
 * \param ent current entity
 * \param frame index of current in flight frame
 */
void Scene::dirtyTraverseTree(Enttity ent, uint32_t frame)
{
  // transform component of this entity
  TransformComponent& this_trans = ent.getComponent<TransformComponent>();

  if (this_trans.isEditedForFrame(frame))
    // if is edited traverse tree up until leaves to
    // update global trans mtx's
  {
    // unedit and clear
    this_trans.edited[frame] = false;
    this_trans.dirty[frame] = false;

    // get relationship component of entity
    const RelationshipComponent& ent_rc = ent.
        getComponent<RelationshipComponent>();
```

```
    glm::mat4 parent_trans = glm::mat4(1.0f);

    // if this is not a root, parent could be null only for scene root
    if (ent_rc.parent)
    {
      parent_trans = ent_rc.parent.getComponent<TransformComponent>()
         .globalTransformMatrix;
      //ent_rc.parent.getComponent<TransformComponent>().getMatrix();
    }

    // traverse tree up until leaves to update global trans mtx's
    editedTraverseTree(ent, parent_trans, frame);
  }
  else if (this_trans.isDirtyForFrame(frame))
    // if is dirty traverse tree while edited not found
  {
    // clear transform of this
    this_trans.dirty[frame] = false;

    // get relationship component of this entity
    const RelationshipComponent& cur_comp = ent.
        getComponent<RelationshipComponent>();
    // get first child
    Enttity cur_child = cur_comp.first;

    // recursively call dirtyTraverseTree for all children
    while (cur_child)
    {
      dirtyTraverseTree(cur_child, frame);
      cur_child = cur_child.getComponent<RelationshipComponent>().next;
    }
  }
}

/**
 * \brief traverse tree accumulating transformation matrix of each entity
 * \param ent current entity
 * \param parent_trans_mtx parent transformation matrix
 * \param frame index of current in flight frame
 */
void Scene::editedTraverseTree(Enttity ent, glm::mat4 parent_trans_mtx,
   uint32_t frame)
{
  // transformation of this node is mul of parent and this
  TransformComponent& ent_tc = ent.getComponent<TransformComponent>();
  const glm::mat4 this_matrix = parent_trans_mtx * ent_tc.getMatrix();

  // unedit and clear
  ent_tc.edited[frame] = false;
  ent_tc.dirty[frame] = false;

  // memorize new global transformation matrix
  ent_tc.globalTransformMatrix = this_matrix;

  // if this node have model its model matrix GPU state should be updated too
  if (StaticModelComponent* static_model_component = ent.
        tryGetComponent<StaticModelComponent>())
  {
    static_model_component->need_GPU_state_update = true;
  }

  // if this node is bone its transform matrix GPU state
```

```
  // should be updated too
  if (BoneComponent* bone = ent.tryGetComponent<BoneComponent>())
  {
    bone->need_gpu_state_update = true;
  }

  // further traverse tree up util the leaves
  const RelationshipComponent& ent_rc = ent.
     getComponent<RelationshipComponent>();
  Enttity cur_child = ent_rc.first;
  while (cur_child)
  {
    editedTraverseTree(cur_child, this_matrix, frame);
    cur_child = cur_child.getComponent<RelationshipComponent>().next;
  }
}

/**
 * \brief record commands to draw all objects in render queues
 * \param frame current frame index to bind proper descriptor sets
 * \param command_buffer command buffer record commands to
 */
void UnlitTexturedShaderEffect::draw(int frame, const
    vk::raii::CommandBuffer& command_buffer) override
{
  // draw all in static queue
  drawStatic(frame, command_buffer, per_object_data_buffer_);
  // draw all in skeletal queue
  drawSkeletal(frame, command_buffer, per_skeleton_data_);
}

/**
 * \brief draw all in static queue
 * \param frame current frame index to bind proper descriptor sets
 * \param command_buffer command buffer record commands to
 * \param per_renderable_data_buffer ref to Per Static Model Data to bind
 */
void UnlitTexturedShaderEffect::drawStatic(int frame, const
   vk::raii::CommandBuffer& command_buffer,
   const PerStaticModelData& per_renderable_data_buffer)
{
  // ptr of previously binded mesh and material
  const Mesh::MeshRenderData* prev_mesh=nullptr;
  const DiffusionMaterial* prev_mat = nullptr;

  // bind shader effect data
  un_lit_graphics_pipeline_statics_.bindStaticPipeline(command_buffer);
  un_lit_graphics_pipeline_statics_.bindStaticShaderData(frame,
    command_buffer);

  // linearly iterate all objects in queue
  for(auto& [mesh, offset]: static_render_queue){

    // if mesh doesn't change no need to rebind it
    if(mesh->render_data_!=prev_mesh)
    {
      mesh->bind(command_buffer);
      prev_mesh = mesh->render_data_;
    }


    // if material doesn't change no need to rebind it
```

```
    if(mesh->material_!=prev_mat)
    {
      mesh->material_->bindMaterialData(frame, command_buffer,
        *un_lit_graphics_pipeline_statics_.static_pipeline_layout_);
      prev_mat=mesh->material_;
    }

    // bind per object data with given offset
    per_renderable_data_buffer.bindDataFor(frame, command_buffer,
     *un_lit_graphics_pipeline_statics_.static_pipeline_layout_, offset);

    // issue draw command
    mesh->drawIndexed(command_buffer);
  }

  // clear queue for next frame
  static_render_queue.clear();
}
```

ПРИЛОЖЕНИЕ Б
Исходный код шейдерных программ

При создании графических конвейеров шейдерных эффектов требуется указать два типа шейдерных программ вершинные и фрагментные. Шейдерные программы пишутся на языке GLSL после чего интерпретируются в SPIR-V. Для двух описанных в работе типов графических объектов: статических моделей и скелетных требуется два разных вершинных шейдера давайте их рассмотрим.

Вершинный шейдер статической модели:

```
#version 450
// viewport camera render data
layout(binding = 0) uniform UniformBufferObject {
    mat4 view;
    mat4 proj;
} ubo;

// per static model data
layout(set = 3, binding=0) uniform DynamicUBO
{
    mat4 model;
}dubo;

// model position of vertex
layout(location = 0) in vec3 inPosition;
// normal of vertex
layout(location = 1) in vec3 inNormal;
// UV texture coordinate
layout(location = 2) in vec2 inTexCoord;

// out texture coord for fragment shader
layout(location = 0) out vec2 fragTexCoord;
// out transformed normal
layout(location = 1) out vec3 outNormal;

void main() {
    // apply camera tra trasformations and model matrix
    gl_Position = ubo.proj * ubo.view * dubo.model * vec4(inPosition, 1.0);
    // pass texture coords
    fragTexCoord = inTexCoord;
    // transform normal with inverse transpose matrix
    // http://www.lighthouse3d.com/tutorials/
    // glsl-12-tutorial/the-normal-matrix/
    outNormal = normalize(mat3(transpose(inverse(dubo.model))) *inNormal);
}
```

Вершинный шейдер скелетной модели:

```
#version 450
// viewport camera render data
layout(binding = 0) uniform UniformBufferObject {
    mat4 view;
    mat4 proj;
} ubo;

// skeletal constants MUST BE IN SYNC with same in dmbrn::BonedVertex
const int MAX_BONES = 256;
const int MAX_BONE_INFLUENCE = 4;

// per skeletal model data
layout(set = 3, binding=0) uniform DynamicSkelUBO
{
    mat4[256] finalBonesMatrices;
}skel_dubo;


// model position of vertex
layout(location = 0) in vec3 inPosition;
// normal of vertex
layout(location = 1) in vec3 inNormal;
// UV texture coordinate
layout(location = 2) in vec2 inTexCoord;
// bone indexs of bones influence this from finalBonesMatrices
layout(location = 3) in uvec4 inBoneIDs;
// weights of influences bones
layout(location = 4) in vec4 inBoneWeights;

// out texture coord for fragment shader
layout(location = 0) out vec2 fragTexCoord;
// out transformed normal
layout(location = 1) out vec3 outNormal;

void main()
{
    // accumalate all influences bone transformaton with weights
    mat4 boneTransform = skel_dubo.finalBonesMatrices[inBoneIDs[0]] *
                         inBoneWeights[0];
    boneTransform     += skel_dubo.finalBonesMatrices[inBoneIDs[1]] *
                         inBoneWeights[1];
    boneTransform     += skel_dubo.finalBonesMatrices[inBoneIDs[2]] *
                         inBoneWeights[2];
    boneTransform     += skel_dubo.finalBonesMatrices[inBoneIDs[3]] *
                          inBoneWeights[3];

    // apply camera trasformations and model matrix
    gl_Position = ubo.proj * ubo.view  * boneTransform *
                  vec4(inPosition, 1.0);
    // pass texture coords to fragment shader
    fragTexCoord = inTexCoord;
    // transform normal with inverse transpose matrix
    // http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/
    // the-normal-matrix/
    outNormal = normalize(mat3(transpose(inverse(boneTransform)))*inNormal);
}
```

При обычной отрисовке как статической так и скелетной моделей используется один и тот же фрагментый шейдер код которого приведен ниже:

```
#version 450
// simple white light simulation with light from up to down
const vec3 light_dir = vec3(0,0,-1);
const vec4 lightColor = vec4(1.0,1.0,1.0,1.0);

// shader effect data
layout(set=1,binding=0) uniform UnLitTexturedUBO
{
    // gamma corrention
    float gamma;
}ult;

// model diffuse texture
layout(set=2, binding = 0) uniform sampler2D texSampler;

// material simple properties
layout(set=2, binding = 1) uniform Properties
{
    vec4 base_color;
}properties;

// texture coord from vertex shader
layout(location = 0) in vec2 fragTexCoord;
// transformed vertex normal
layout(location = 1) in vec3 inNormal;

// output color for this pixel
layout(location = 0) out vec4 outColor;

void main()
{
    // phong shading https://en.wikipedia.org/wiki/Phong_shading
    // full ambient strength
    const float ambientStrength = 1.0;
    vec4 ambient = ambientStrength * lightColor;

    // calculate diffuse reflection as dot between
    // nomal and direction *towards* light
    float diff = 1.0*max(dot(inNormal,-light_dir),0.0);
    vec4 diffuse = diff * lightColor;

    // collect all with average balance between ambient and diffuse
    outColor = (ambient + diffuse)*0.5*properties.base_color *
    texture(texSampler, fragTexCoord);

    // add gamma correction to color
    const float gamma = 2.2;
    outColor.rgb = pow(outColor.rgb, vec3(1.0/gamma));
}
```

При отрисовке контура объекта используется похожие на ранее описаные шейдеры, за исключением того, что они выполняют дополнительное масштабирование объекта. Для демонстрации этого подхода распечатаем только код вершинного шейдера для статической модели:

```
#version 450
// construt sacle matrix with given scale
#define scaleMat(scale)
mat4(vec4(scale,0,0,0),vec4(0,scale,0,0),vec4(0,0,scale,0),vec4(0,0,0,1))

// viewport camera render data
layout(binding = 0) uniform UniformBufferObject {
    mat4 view;
    mat4 proj;
} ubo;

// outline shader effect data
layout(set = 1, binding=0) uniform OutlineData{
    vec3 color;
    float scale;
} outline;

// per static object data
layout(set = 3, binding=0) uniform DynamicUBO
{
    mat4 model;
}dubo;

// model position of vertex
layout(location = 0) in vec3 inPosition;

void main()
{
    // apply camera transform than model than transfrom for outline
    // this order gives better results IMHO, but outline can be putted
    //  in different places
    gl_Position =  ubo.proj * ubo.view  * dubo.model *
                   scaleMat(outline.scale) * vec4(inPosition, 1.0);
}
```

Фрагментный шейдер для контура просто выводит в качестве цвета пикселя указанный цвет контура.