Санкт-Петербургский государственный университет

Кафедра системного программирования

Цырендашиев Сультим Баиржапович

Разработка библиотеки визуализации трехмерной графики для симулятора дронов

Курсовая работа

Научный руководитель:
ст. преп. Пименов А. А.

Санкт-Петербург
2020

Оглавление

Введение

На текущий момент всё большую популярность набирают дроны (мультикоптеры, мультироторы) – гражданские беспилотные летательные аппараты. Для их управления используется специальное программное обеспечение, которое применяется для построения маршрута, облёта препятствий и т.д., но, как и любой другой программный продукт, его необходимо тестировать. Испытание в условиях реального мира может оказаться дорогостоящим, так как любая ошибка может повлечь поломку летательного аппарата, нанесение урона людям, их имуществу. Поэтому тестирование сначала проводят в специальном ПО – симуляторах, в которых имитируется реальность, то есть происходит отрисовка мира, расчет физики и т.п. для вычисления входных симулируемых значений датчиков; и, таким образом, разработка ПО управления дронами с минимальными изменениями может быть затем перенесена из виртуальной среды в реальную.

В течение некоторого времени кафедра Системного Программирования проводит исследования в области автономной навигации мультироторов и усилиями коллектива студентов и преподавателей сформировала некоторую кодовую базу взяв за основу библиотеку CoreCVS [22] – которая предоставляет примитивы компьютерного зрения и вычислительной математики, а также включает в себя симулятор дронов. С визуальной точки зрения, графика в нём относительно простая, поэтому и требуется её усовершенствование, а также повышение переносимости и удобства отображения. Но так как графика не является целью этого проекта, то для этого необходима библиотека, которая предоставляет легкую в освоении, но в то же время гибкую функциональность.

Поэтому разрабатываемое в рамках данной курсовой работы реше-

ние может быть применимо в проектах, где графика является только инструментом, изучение которого должно быть непродолжительным, например, в физических симуляциях твердых тел, системах инженерной графики и т.д.

1. Постановка задачи

Цель курсовой работы – разработка графической библиотеки с высокой конфигурируемостью и простотой использования в стандартной реализации.

Так как над проектом работает несколько человек, то было принято решение разделить работу и принять целями данной работы следующее:

- Обзор существующих решений;
- Реализация части обертки над графическим API;
- Реализация высокоуровневой системы визуализации трехмерной графики;
- Интеграция в подсистему отображения инженерной графики библиотеки CoreCVS.

2. Обзор

2.1. Глоссарий

- *Сцена (англ. scene)* – набор объектов с параметрами.
- *Визуализация трехмерной графики (отрисовка, рендеринг) (англ. rendering)* – создание и презентация изображения на основании информации о сцене.
- *Полигональная сетка (англ. polygon mesh)* – набор вершин, рёбер и граней, который представляет собой некоторый трехмерный объект. Гранями обычно являются треугольники.
- *Текстура (англ. texture)* – изображение, подготовленное для наложения на полигональную сетку.
- *Пробник (англ. sampler)* – специальный объект, определяющий способ взятия значений из текстуры.
- *Шейдерная программа (англ. shader)* – пользовательская программа, которая исполняется на видеокарте.
- *Буфер параметров шейдера (англ. uniform buffer)* – буфер, содержащий параметры, передаваемые в шейдерную программу и доступные в ней только для чтения.
- *Материал (англ. material)* – набор различных параметров и текстур, которые подаются на вход некоторому шейдеру.
- *Трёхмерная модель (англ. 3D model)* – набор полигональных сеток, текстур, анимаций и т.д.

- *Буфер команд (англ. command buffer)* – специальный буфер, в который записываются команды рендеринга. После его заполнения и готовности, он отправляется на видеокарту для исполнения.

- *Подпроход визуализации (англ. render subpass)* – фаза отрисовки, которая считывает и записывает в отведённые буферы изображения, которые хранятся в проходе визуализации.

- *Проход визуализации (англ. render pass)* – набор буферов изображений, подпроходов, а также зависимостей этих подпроходов.

- *Графический трубопровод (англ. graphics pipeline)* – последовательность этапов, необходимых для визуализации графики.

- *Буфер кадра (англ. framebuffer)* – набор буферов изображений, в которые происходит отрисовка графики. Он предоставляет буферы, необходимые в проходе визуализации.

- *Цепь изображений для презентации (англ. swap chain)* – набор изображений, презентация которых происходит поочередно.

- *Фреймворк для разработки игр (англ. game engine)* – программное обеспечение для работы компьютерных игр. Обычно оно включает в себя отрисовку графики, симуляцию физики, работу со звуком, сетью и др.

2.2. Обзор существующих решений

На текущий момент существует множество библиотек, фреймворков, API для визуализации трехмерной графики, но в контексте данной курсовой работы будут рассматриваться проекты, предложенные науч-

ным руководителем: Google Filament [21], OGRE3D [16], а также ПО для визуализации, используемое в симуляторах AirSim [1] и CARLA [4].

Основными критериями для обзора существующих решений является кросс-платформенность, возможность использования произвольных шейдеров, структур вершин и т.д., а также простой в использовании интерфейс, то есть затраченное на его изучение время должно быть минимальным. Поэтому такие API для графики, как OpenGL [13], DirectX [6], Vulkan [14] и др. в этом разделе рассматриваться не будут, так как эти API являются низкоуровневыми[1], так что их изучение может занять много времени, а это критично, если целью разрабатываемой программы не является сама графика.

Одной из библиотек для рендеринга трехмерных сцен в реальном времени, целью которой является простота использования, является Google Filament [21]. Несмотря на то, что рассматриваемая библиотека кросс-платформенная и поддерживает множество различных графических API, в ней используется свой формат полигональных сеток – filamesh, вершины в которых определяются только позицией, касательной, цветом и двумя текстурными координатами, поэтому в данном контексте она не подходит, так как вершины в полигональных сетках могут содержать произвольную информацию. То же самое можно сказать о материалах, видов которых всего пять [7].

Другой библиотекой с похожей целью является OGRE3D [16], которая также является кросс-платформенной и поддерживает различные API, но при этом поддерживает пользовательские структуры вершин в полигональных сетках и пользовательские шейдерные программы. Тем не менее, Ogre3D имеет множество зависимостей, в особенности

[1]например, необходимо оптимальное управление памятью [17], синхронизация работы между процессором и видеокартой [2]

– Cg [5] – язык шейдерных программ, разработанный NVidia, который уже несколько лет не поддерживается и его использование не рекомендуется.

Стоит заметить, что для симуляции обычно используют фреймворки для разработки игр, например, симуляторы AirSim [1] и CARLA [4] используют Unreal Engine [19] и Unity Engine [18]. В них уже есть отрисовка графики, симуляция физики, система скриптов, пользовательский интерфейс и т.д. С другой стороны, это ограничивает пользователя, так как, например, использовать собственные алгоритмы симуляции физики может быть невозможно из-за закрытого исходного кода[2].

Таким образом, несмотря на то, что существует множество различных библиотек для отрисовки трехмерной графики, необходима специальная библиотека с простым в использовании программным интерфейсом и достаточной гибкостью.

2.3. Используемые технологии

Основным языком программирования выбран C++, поскольку этот язык используется в симуляторе беспилотников на базе CoreCVS.

API для графики предоставляют абстракцию над графическими устройствами, достаточную для визуализации трехмерной графики. Для реализации библиотеки будет использоваться Vulkan API [14], так как это является обязательным условием научного руководителя, при этом Vulkan API предоставляет более низкоуровневый доступ, чем OpenGL [13], например, в OpenGL есть глобальное состояние, нет поддержки буферов команд и др. [3][2]. Также Vulkan является кросс-платформенным[3], что нельзя сказать о DirectX [6], который поддерживается только в ОС

[2]например, Unity Engine

[3]на MacOS и iOS – с помощью официальной библиотеки MoltenVK [11]

Windows, и Metal [15] — только MacOS и iOS. Более того, Vulkan API является одним из новейших графических API на данный момент[4].

Для инициализации графического трубопровода в Vulkan необходима информация о ресурсах в шейдере – их расположение, тип и т.д., но так как шейдеры определяются пользователем и Vulkan не предоставляет достаточной информации, то понадобится рефлексия для SPIR-V шейдеров [10][9]. Есть несколько библиотек с соответствующей функциональностью, одними из таких являются SPIRV-Cross [12] и SPIRV-Reflect [20]. В работе будет использоваться библиотека SPIRV-Cross, так как это официальный продукт от Khronos Group[5].

Для управления памятью видеокарты используется библиотека Vulkan Memory Allocator [8], позволяющая оптимально использовать память при использовании Vulkan API.

[4]по дате выхода первой версии

[5]LunarG (компания-разработчик Vulkan API) является ассоциированным членом Khronos Group

3. Реализация

3.1. Обертка над Vulkan API

Задачей обёртки является абстракция над низкоуровневым графическим API, поэтому при необходимости возможно использовать другие программные интерфейсы, отличные от Vulkan API. Функциональность обёртки включает следующие пункты, при этом курсивом выделены функции, реализованные в ходе данной курсовой работы[6]:

- *Регистрация структур вершин;*
- *Создание, обновление, удаление вершинных, индексных буферов,* а также буферов параметров шейдера;
- Регистрация структур буферов параметров шейдера;
- *Регистрация текстур*;
- *Создание, удаление пробников для текстур;*
- Загрузка шейдерных программ;
- Создание, удаление графических трубопроводов;
- Создание, удаление буферов кадра;
- Отправка командного буфера на исполнение и замена изображения в цепи изображений для презентации;
- *Работа с командными буферами:*
  - *начало и конец записи в командный буфер;*

[6]остальная часть функциональности реализована Е. Орачевым

– *установка буфера кадра;*

– *установка графического трубопровода;*

– *установка структуры буфера параметров шейдера;*

– *установка вершинного и индексного буферов;*

– *рисование без и с помощью индексного буфера.*

Таким образом, в общем случае для отрисовки трёхмерной графики пользователю необходимо:

1. Зарегистрировать и создать необходимые ресурсы (различные буферы, текстуры и т.п.);
2. Начать запись в командный буфер;
3. Установить буфер кадра;
4. Для каждого объекта на сцене:
   (a) установить графический трубопровод и структуру буфера параметров шейдера, использующегося в материале этого объекта;
   (b) установить вершинный и индексный буферы для полигональной сетки этого объекта;
   (c) вызвать команду для рисования с помощью индексного буфера;
5. Закончить запись командного буфера;
6. Отправить командный буфер на исполнение и заменить изображение в цепи изображений для презентации.

При этом шаги 2 – 6 могут быть помещены в цикл для создания поочередно показывающихся изображений, то есть производится отрисовка сцены в реальном времени.

Далее будут рассмотрены только те функции обёртки, обоснование которых необходимо.

3.1.1. Вершинные и индексные буферы

Буферы вершин и индексов используются при описании полигональных сеток. Vulkan API предоставляет интерфейс для создания, изменения и удаления таких буферов.

Так как полигональные сетки могут быть статическими, либо динамическими – атрибуты вершин таких сеток меняются[7], но количество самих вершин всегда остаётся одним и тем же, то их тип указывается при создании этих буферов посредством библиотеки.

Если указанный тип – динамический, то возможны доступ и запись в этот буфер со стороны процессора, то есть указываются следующие флаги при выделении памяти на видеокарте с помощью Vulkan API:

- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` – для чтения данных буфера из памяти видеокарты с помощью функции `vkMapMemory`;
- `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` – для автоматической загрузки в память видеокарты.

А при статическом происходит единовременная запись в память видеокарты, которая осуществляется в несколько этапов:

1. Создаётся временный буфер, который доступен со стороны процессора, выделяется память на видеокарте, в неё записываются

[7] имеется в виду то, что данные о вершинах меняются в памяти видеокарты

данные вершин или индексов;

2. Создаётся и выделяется память для окончательного буфера, в которую копируются данные из временного;

3. Удаляется временный буфер и освобождается его память.

Таким образом, окончательный буфер располагается в памяти видеокарты и не имеет флага `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`, поэтому доступ к нему более оптимальный[8].

3.1.2. Структура вершин

Вершины в разных полигональных сетках могут иметь различную структуру[9], например, у одной – это (вектор позиции), у другой – (вектор позиции, цвет, вектор нормали, вектор касательной). Поэтому обёртка включает в себя специальные дескрипторы для описания структуры буферов вершин – `VertexBufferLayoutDesc` и `VertexAttributeDesc` (рис. 1).

`VertexBufferLayoutDesc` содержит список `VertexAttributeDesc`, размер одного элемента в байтах и как часто будет указана информация – для каждой вершины или для всего буфера. Этот дескриптор необходим для указания того, как будет интерпретироваться каждый элемент из данных из вершинного буфера. Каждый элемент состоит из атрибутов.

`VertexAttributeDesc` – это описание конкретного атрибута вершины. В нём указываются тип атрибута; индекс атрибута (`location`), указанный в шейдерной программе, которая будет использована для отрисовки вершинного буфера; и смещение в байтах от начала элемента.

[8] см. главу «10.2. Device Memory» спецификации Vulkan API [14]

[9] англ. layout

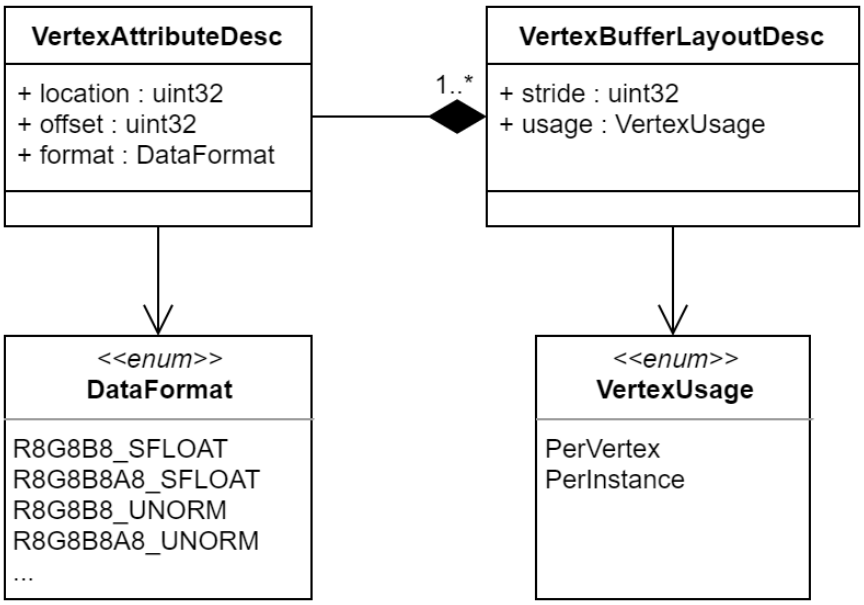

Рис. 1: Объявления `VertexBufferLayoutDesc` и `VertexAttributeDesc`

.

Например, для описания структуры вершин, состоящих из трехмерного вектора позиции и четырехмерного вектора цвета, необходимо указать следующие данные:

- `stride = 3 * sizeof(float) + 4 * sizeof(float);`
- `usage = PerInstance;`
- Атрибуты:
  - `location = 0,`
    `offset = 0,`
    `format = R8G8B8_SFLOAT;`
  - `location = 1,`
    `offset = 3 * sizeof(float),`
    `format = R8G8B8A8_SFLOAT.`

3.1.3. Управление памятью

На начальном этапе реализации обертки для выделения памяти на видеокарте и создания различных буферов были использованы функции Vulkan API без промежуточных этапов, в частности, каждому буферу сопоставлялся отдельный дескриптор аллокации памяти видеокарты (`VkDeviceMemory`). Но так как Vulkan имеет жёсткие ограничения на количество одновременно существующих таких дескрипторов, максимальное число которых указано в поле `maxMemoryAllocationCount` структуры `VkPhysicalDeviceLimits`, специфичной для каждой видеокарты; то было принято решение о интеграции библиотеки Vulkan Memory Allocator (VMA), поскольку она решает проблему ограниченного количества дескрипторов. При запросе памяти VMA возвращает дескриптор аллокации `VkDeviceMemory` и смещение, таким образом, VMA оптимально использует количество дескрипторов, выделяя память большого размера (один дескриптор) и уже в ней распределяя смещения для запросов.

3.1.4. Командные буферы

Командные буферы необходимы для записи в них команд, которые будут исполнены на видеокарте. Такими командами являются команды отрисовки полигональных сеток (`vkCmdDraw`), установки буферов индексов (`vkCmdBindIndexBuffer`), вершин (`vkCmdBindVertexBuffers`), начала прохода визуализации (`vkCmdBeginRenderPass`) и другие. Vulkan API предоставляет несколько видов командных буферов: многоразовые и для разового использования. Так как сцены с большой вероятностью меняются каждый кадр, например, произойдёт сдвиг камеры, поворот объекта, то для каждого кадра создаётся новый одноразовый команд-

ный буфер, при этом их аллокация происходит из специального пула, предназначенного для командных буферов с коротким временем жизни.

3.2. Создание стандартной реализации

Стандартная реализация предоставляет интерфейс `IRenderEngine`, использующий высокоуровневые примитивы, в которые инкапсулирована большая часть обёртки (рис. 2).

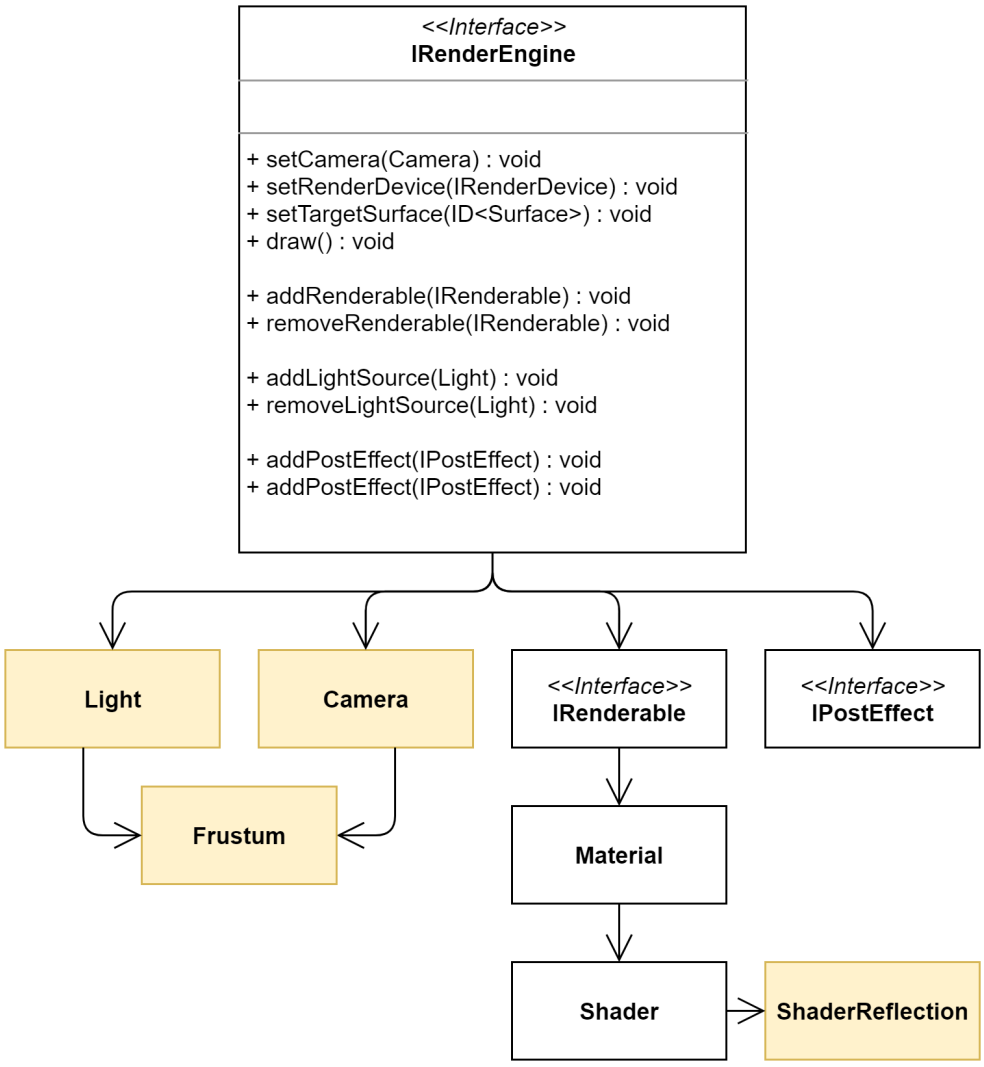

Рис. 2: Интерфейс `IRenderEngine` (жёлтым цветом выделены классы реализованные в ходе данной работы).

Далее будут рассмотрены необходимые детали реализации.

3.2.1. Рефлексия шейдеров

Так как шейдерные программы задаются пользователем, то необходимо иметь доступ к информации об их параметрах. Такими параметрами являются:

- входные и выходные данные;
- буферы параметров шейдера, которые могут иметь произвольную структуру, агрегируя другие, уже заданные, типы данных, т.е. являются аналогами `struct` в языке C;
- текстуры.

У каждого параметра существует свой собственный идентификатор, который необходим для предоставления данных в шейдерную программу для её дальнейшего исполнения. Для входных и выходных данных этот идентификатор называется `location`, а для буферов и текстур – `binding`. Для того, чтобы у пользователя не было необходимости указывать числовые идентификаторы, эти данные получаются с помощью рефлексии, при этом каждому идентификатору соответствует конкретное название, которое может быть использовано для индексации.

Поскольку буферы параметров шейдера представляют собой некоторую структуру, то необходимо рассмотреть каждое её поле: у всех членов буфера идентификатор является таким же, как и у всего буфера, так как нельзя установить только часть буфера; но при этом смещение и тип у каждого члена будет разным. Однако для избежания коллизий названия членов буфера составляются следующим образом: ''`<имя типа буфера>.<название члена>`''. Используется именно имя типа буфера, так как SPIRV-Cross предоставляет только это название для структур.

3.2.2. Источники света и тени

Объекты отбрасывают тени с помощью алгоритма теневых карт [10] и функций `IRenderDevice`, независимо от использованного графического API. Создаётся отдельный буфер кадра, в который происходит отрисовка объектов с точки зрения источника света, далее информация о глубине из этого буфера кадра используется для затенения при отрисовке с камеры. Для отсечения объектов, невидимых для источника света используется пирамида видимости камеры, на основе которой строится минимальный параллелепипед, вращение которого совпадает с вращением источника света.

3.3. Интеграция в CoreCVS

3.3.1. Qt

Для отображения в окно[11] необходимо получить платформозависимые дескрипторы, например, `HWND` и `HINSTANCE` в операционной системе Windows; и с помощью этих дескрипторов создать экземпляр `VkSurfaceKHR`, который уже используется в Vulkan API.

CoreCVS использует фреймворк Qt для отображения пользовательского интерфейса, при этом с версии Qt 5.10 в нём уже предусмотрено получение `VkSurfaceKHR` из `QWindow`. Для этого был создан класс `VkIbWindow` (рис. 3), в котором инкапсулировано:

- создание экземпляра `VulkanRenderDevice`, при его создании необходимо указать расширения в виде массива их названий, которые будут использоваться для инициализации `VkInstance`; при

[10] англ. shadow mapping

[11] часть оконного интерфейса (windowing system)

этом одним из них является ''VK_KHR_surface'', а другой зависит от оконного интерфейса, например, для системы окон ОС Windows – ''VK_KHR_win32_surface'', а для X Window System – ''VK_KHR_xcb_surface'';

- создание экземпляра QVulkanInstance с помощью VkInstance, получаемого из VulkanRenderDevice;
- после инициализации самого окна через функцию QWindow::show(), возможно получение VkSurfaceKHR из QWindow, последующая его инициализация и создание дескриптора для IRenderDevice.

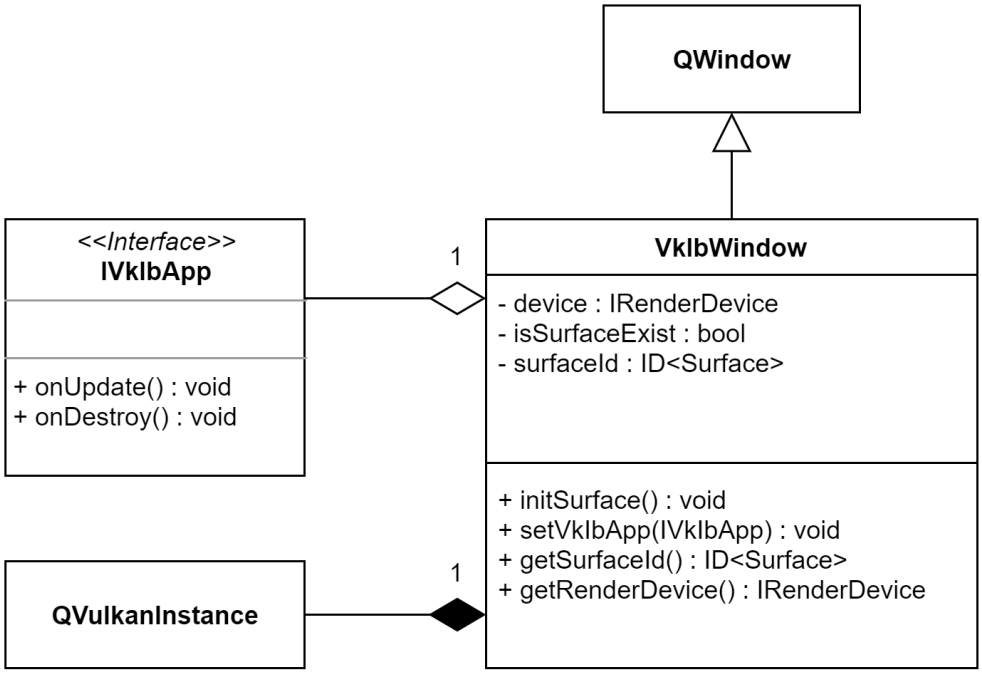

Рис. 3: Класс VkIbWindow, наследуемый от QWindow.

VkIbWindow также обрабатывает события, которые посылаются от Qt: при получении события QEvent::UpdateRequest запрашивается ещё одно такое же событие, которое будет распланировано и будет отправлено в следующем кадре. При этом, если экземпляр класса, наследованного от IVkIbApp, установлен, то будет вызвана функция onUpdate(),

в котором может содержаться логика для отрисовки.

3.3.2. Адаптер

CoreCVS использует классы `Mesh3D` и `Mesh3DDecorated` для хранения информации о полигональных сетках, при этом массивы для каждого атрибута вершины (вектора позиции, нормали, цвета и т.д.) хранятся отдельно, однако Vulkan API требует (а следовательно и `IRenderDevice`), чтобы буфер вершин хранил информацию не об отдельных массивах атрибутов, а в одном массиве, в котором каждый элемент содержит эти атрибуты.

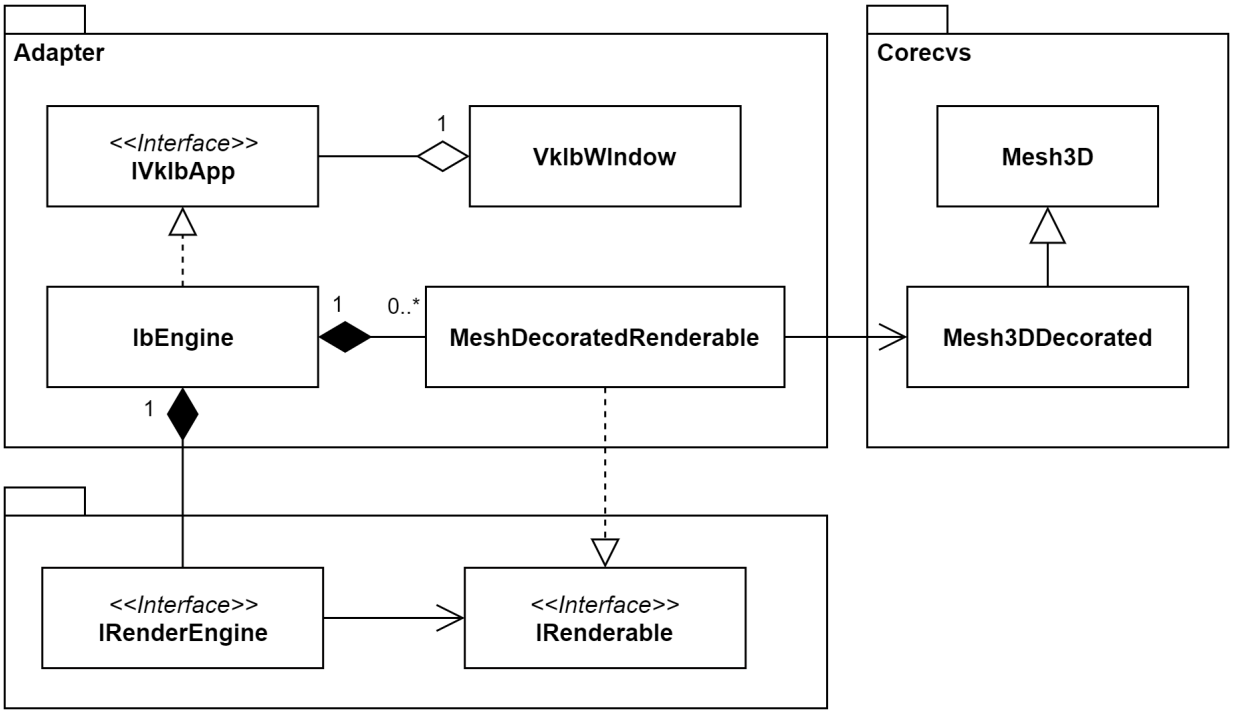

Рис. 4: Класс-адаптер `IbEngine` и класс `MeshDecoratedRenderable`, использующий `Mesh3DDecorated` в качестве источника информации для отрисовки.

Для их отображения через `IRenderEngine` был создан класс `MeshDecoratedRenderable` (рис. 4), который наследуется от интерфейса `IRenderable` и объединяет информацию из `Mesh3DDecorated` в вершинный и

индексный буферы для последующей отрисовки. Класс `IbEngine` является адаптером между графической библиотекой и CoreCVS. Он также является подклассом интерфейса `IVkIbApp` с целью получения доступа к реализации функции `onUpdate()`.

`Mesh3D` и `Mesh3DDecorated` также содержат информацию о точках и отрезках, которые должны быть отрисованы, поэтому при добавлении `MeshDecoratedRenderable` происходит отделение информации о полигональных сетках, об отрезках и о точках, так как каждый из них используют различные шейдерные программы, графические трубопроводы.

Заключение

В рамках курсовой работы были выполнены следующие задачи:

- Реализована обертка над Vulkan API:
  - работа с вершинными, индексными буферами;
  - регистрация структур вершин;
  - работа с командными буферами;
  - загрузка изображений, пробники;
- Интегрирована библиотека `Vulkan Memory Allocator`;
- Реализована стандартная реализация:
  - интегрирована библиотека для рефлексии шейдеров на языке SPIRV;
  - реализованы примитивы: камеры, источники света, отсечение по пирамиде видимости;
  - тени в реальном времени;
- Интеграция в CoreCVS:
  - реализован класс для Qt, инкапсулирующий логику созданной библиотеки;
  - реализован адаптер для отрисовки полигональных сеток, отрезков и точек, который принимает на вход классы CoreCVS.

Список литературы

[1] Airsim: High-fidelity visual and physical simulation for autonomous vehicles / Shital Shah, Debadeepta Dey, Chris Lovett, Ashish Kapoor // Field and service robotics / Springer. — 2018. — С. 621–635.

[2] Bishop L, Kubisch C, Schott M. High-performance low-overhead rendering with OpenGL and Vulkan // Proc. of the Game Developers Conference (GDC). — 2016.

[3] Blackert Axel. Evaluation of multi-threading in vulkan. — 2016.

[4] CARLA: An Open Urban Driving Simulator / Alexey Dosovitskiy, German Ros, Felipe Codevilla et al. // Conference on Robot Learning. — 2017. — P. 1–16.

[5] Cg Toolkit | NVIDIA Developer // NVIDIA Developer. — 2019. — Режим доступа: `https://developer.nvidia.com/cg-toolkit` (дата обращения: 14.12.2019).

[6] Direct3D 12 programming guide - Win32 apps // Technical documentation, API, and code examples | Microsoft Docs. — 2019. — Режим доступа: `https://docs.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide` (дата обращения: 08.12.2019).

[7] Filament Materials Guide // GitHub – The world's leading software development platform. — 2019. — Режим доступа: `https://google.github.io/filament/Materials.html#materialmodels` (дата обращения: 10.12.2019).

[8] GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator: Easy to integrate Vulkan memory allocation library // GitHub – The world's leading software development platform. — Режим доступа: https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator (дата обращения: 02.04.2020).

[9] Kessenich John, Ouriel Boaz, Krisch Raun. Khronos SPIR-V Registry - The Khronos Group Inc // The Khronos Group Inc. — 2019. — Режим доступа: https://www.khronos.org/registry/spir-v/ (дата обращения: 08.12.2019).

[10] Kessenich John, Ouriel Boaz, Krisch Raun. SPIR-V Specification // The Khronos Group Inc. — 2019. — Режим доступа: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html (дата обращения: 08.12.2019).

[11] Khronos Group Inc. KhronosGroup/MoltenVK: MoltenVK is an implementation of the high-performance, industry-standard Vulkan graphics and compute API, that runs on Apple's Metal graphics framework, bringing Vulkan to iOS and macOS. // GitHub – The world's leading software development platform. — 2019. — Режим доступа: https://github.com/KhronosGroup/MoltenVK (дата обращения: 08.12.2019).

[12] Khronos Group Inc. KhronosGroup/SPIRV-Cross: SPIRV-Cross is a practical tool and library for performing reflection on SPIR-V and disassembling SPIR-V back to high level languages. // GitHub – The world's leading software development platform. — 2019. — Режим доступа: https://github.com/KhronosGroup/SPIRV-Cross (дата обращения: 08.12.2019).

[13] Khronos Group Inc. The OpenGL® Graphics System: A Specification // The Khronos Group Inc. — 2019. — Режим доступа: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (дата обращения: 08.12.2019).

[14] Khronos Vulkan Working Group and others. Vulkan® 1.1.130 - A Specification (with all registered Vulkan extensions) // The Khronos Group Inc. — 2019. — Режим доступа: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html (дата обращения: 08.12.2019).

[15] Metal Shading Language Specification // Apple Developer. — 2019. — Режим доступа: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf (дата обращения: 08.12.2019).

[16] OGRE - Open Source 3D Graphics Engine | Home of a marvelous rendering engine // Ogre3D. — 2019. — Режим доступа: https://www.ogre3d.org/ (дата обращения: 14.12.2019).

[17] Tovey Steven. Vulkan Memory Management // Proc. of the Vulkanised. — 2018.

[18] Unity Real-Time Development Platform | 3D, 2D VR & AR Visualizations. — 2019. — Режим доступа: https://www.unity.com/ (дата обращения: 08.12.2019).

[19] What is Unreal Engine 4 // Unreal Engine. — 2019. — Режим доступа: https://www.unrealengine.com/en-US/ (дата обращения: 08.12.2019).

[20] chaoticbob/SPIRV-Reflect: SPIRV-Reflect is a lightweight library that provides a C/C++ reflection API for SPIR-V shader bytecode in Vulkan applications. // GitHub – The world's leading software development platform. — Режим доступа: `https://github.com/chaoticbob/SPIRV-Reflect` (дата обращения: 10.12.2019).

[21] google/filament: Filament is a real-time physically based rendering engine for Android, iOS, Windows, Linux, macOS and WASM/WebGL // GitHub – The world's leading software development platform. — 2019. — Режим доступа: `https://github.com/google/filament` (дата обращения: 08.12.2019).

[22] Пименов Александр. PimenovAlexander/corecvs: Computer Vision primitives library // GitHub – The world's leading software development platform. — 2019. — Режим доступа: `https://github.com/PimenovAlexander/corecvs` (дата обращения: 08.12.2019).