ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук

Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему “Прототип фреймворка для рендеринга изображений”

Выполнил:

студент группы БПМИ205 И.К Листратов

И.О. Фамилия

Подпись  Дата 05.18.2022

Принял:

руководитель проекта Кускаров Тагир Фаридович

Имя, Отчество, Фамилия

Сервис Видеопоиск, группа Робота Видео, разработчик программного обеспечения

Должность

ООО "Яндекс.Технологии"

Место работы

Дата 19.05.2022 9 

Оценка (по 10-тибалльной шкале) Подпись

Москва 2022

Реферат

Целью проекта является написание фреймворка для рендеринга изображений, который одновременно сравнительно прост в использовании и по производительности сравним с низкоуровневыми графическими API.

Для решения этой задачи был выбран метод, основанный на архитектуре Render Graph[i]. Такой подход позволяет пользователю фреймворка описывать свои намерения посредством высокоуровневой модели, которая силами фреймворка оптимизируется и конвертируется в низкоуровневое описание через графический API.

В текущей версии, реализованы все необходимые объекты в упомянутой архитектуре, фреймворк уже может использоваться для построения графических конвейеров. Где возможно, дорогостоящие вызовы API были сгруппированы для повышения производительности, уменьшено кол-во аллокаций видеопамяти.

Ключевые слова

Rendering, VulkanAPI, Render Graph, фреймворк

Содержание

Введение

В современной жизни мы каждый день наблюдаем изображения, сгенерированные компьютером. Рендерингом называют процесс генерации изображения компьютером по модели(под моделью подразумевается информация о геометрических данных изображаемых объектов, информация об освещении, положении наблюдателя и т.д). Рендерером называют программу, осуществляющую рендеринг.

В контексте этой работы в основном будет рассматриваться 2D/3D рендеринг в реальном времени. В таком виде задача состоит в том, чтобы по внутреннему представлению сцены отрисовывать изображение со скоростью > 20 кадров в секунду, получая при этом как можно более качественное изображение.

Для отрисовки в реальном времени практически всегда используются GPU - устройства, предназначенного для однотипной обработки большого объема данных. GPU - отдельное от CPU устройство со значительно отличающейся архитектурой и областью решаемых задач[ii]. Для написания программ, исполняемых на них, используются отдельные (хотя во многом похожие на C/C++) языки программирования (Основные примеры: *GLSL*,[iii] *HLSL*[iv]). Программы на этих языках часто называют шейдерами.

Помимо средств для программирования самих GPU нужны средства для того, чтобы посылать туда команды из процессора. Это, конечно, можно делать через драйвер, но это крайне низкоуровневый и трудозатратный способ. К тому же страдает переносимость написанной программы. Для исправления этой проблемы существуют графические API, служащие слоем абстракции между программой и драйвером видеокарты. Наиболее известными API на данный момент являются:

- **OpenGL** (overview[v]) Мультиплатформенный, высокоуровневый API. Многие задачи решаются за программиста, что положительно сказывается на удобстве использования. Ценой этому становится непредсказуемое потребление ресурсов процессора, значительный overhead и неполное задействие возможностей GPU.
- **DirectX 11** Аналог от Microsoft, доступный только на windows.
- **Vulkan** (overview[vi]) Относительно недавно появившийся API, значительно отличающийся от своих предшественников. Основной упор ведется на передачу контроля в руки программиста. Многие вещи, которые происходили в OpenGL/DirectX автоматически, тут придется делать вручную. Так же по умолчанию тут отсутствует какая-либо проверка действий программиста на корректность. Значительный рост производительности при правильном использовании дается ценой простоты использования.

Ввиду серьезных требований к производительности и желания максимально эффективно использовать доступные возможности GPU теперь имеет смысл создавать рендереры, использующие Vulkan.

Как правило, графический конвейер может быть описан лишь списком задач, ресурсами и описанием того, как задачи используют эти ресурсы в процессе выполнения. Ввиду заложенных принципов, выражение этого описание через Vulkan занимает существенно больше кода и значительно сильнее подвержено ошибкам. (Для наглядности можно сравнить два туториала оба из которых описывают процесс отрисовки треугольника в окне: Vulkan[vii] OpenGl[viii]) Возникает потребность написать промежуточный фреймворк, который по высокоуровневому описанию процесса будет должным образом инициализировать и использовать API чтобы генерировать изображение по описанному процессу в реальном времени.

Сравнительный анализ

Альтернативами создаваемой программе могут послужить:

- Полноценный игровой движок(на пример Unity/Unreal Engine)
  - Не нужно вникать в низкоуровневые детали процесса.
  - Сопряженные процессы по типу добавления текстур/3D моделей и использование их в рендерере осуществляется автоматически.
  - Труднодоступность более низкоуровневых интерфейсов и как следствие более узкий спектр решаемых задач.
  - Тяжеловесность. Современные движки крайне универсальны и содержат в себе десятки различных подсистем. Это может создать ощутимый overhead как в runtime'е так и при разработке.
- Использовать Vulkan API напрямую
  - Широкий спектр решаемых задач.
  - Низкий overhead, высокий потенциал в производительности.
  - Низкоуровневость. Использование API трудозатратно, даже простые задачи по типу инициализации API могут занимать очень много кода (У меня код связанны с инициализацией Vulkan занимает порядка 500 строк C++ кода без учета заголовочных файлов).
- Использовать более высокоуровневый графический API(на пример OpenGL)
  - Средний вариант между двумя предыдущими пунктами.

Функциональные требования

Программа представляет из себя библиотеку и заголовочный файл с публичными интерфейсами. Пользователь может описывать графический конвейер используя следующие объекты:

- *RenderPass* - атомарная задача в процессе рендеринга. Это может быть запуск шейдера, инициализация ресурсов, перемещение данных и т.д. В фреймворке представлен как базовый класс, содержащий функционал для передачи в RenderGraph информации об используемых Resource. Ожидается, что пользователь будет создавать классы наследники этого класса, в которых будет задавать нужное ему поведение.

- *Resource* - объект, использующийся как ввод/вывод для *RenderPass*. Это может быть временная текстура, информация о геометрии сцены, массив с позициями источников освещения и т.д. Представлен классами Buffer и Image.
- *RenderGraph* - объект, хранящий в себе информацию о всех ресурсах, RenderPass'ах и последовательностью их исполнения и реализующий логику описания создания и процесса исполнения графического конвейера.

Нефункциональные требования

Требования к надежности.

В отладочной сборке

1. Некорректное использование VulkanAPI логируется описание ошибки, после чего программа аварийно завершается.
2. Нарушение внутренних инвариантов классов, ошибка во время выполнения, некорректное использование фреймвокра приводит к аварийному завершению программы. Место возникновения ошибки и прочие детали логируются.

В релизной сборке

- В пункте 2 вместо аварийного завершения программы выбрасывается исключение. Объект, бросивший исключение, вообще говоря, может остаться в некорректном состоянии. Исключения используются лишь для более безопасного завершения программы. Их отлавливание на стороне пользователя, вообще говоря, не ожидается.

Реализация

В работе фреймворка можно выделить следующие этапы:

- Инициализация фреймворка
- Описание графического конвейера
- Инициализация графического конвейера
- Исполнение графического конвейера

Инициализация фреймворка

Производится автоматически силами фреймворка. Логика, отвечающая за инициализацию, находится в *src/base*. Для инициализации нужно позвать base::Base::Init() передав конфиг для класса Base и для класса Context. В конфиге для Base перечисляются InstanceExtensions/Layers - список опциональных модулей VulkanAPI, предоставляющих дополнительный функционал/валидирующих вызовы к VulkanAPI. В конфиге для Context - аналогичный список, но уже для дополнительного функционала, требующего поддержки на стороне видеокарты. После предоставления конфига начинается инициализация.

- Создается VulkanAPI объект VkInstance. Он потребуется при использовании базовых вызовов к API, не отправляющих команды на конкретное GPU устройство и не управляющих ресурсами на нем. На этом этапе используется конфиг для Base
- Создается DebugMessenger - объект API, присылающий сообщения с ошибками и прочей информацией из встроенного в Vulkan валидатора.
- У операционной системы запрашивается окно. Для окна создается API объект VkSurface, являющий собой некоторую поверхность, на которую можно показывать изображение.
- Создается объект фреймворка Context, инициализирующий и содержащий API объекты для взаимодействия с конкретными GPU устройствами. На этом этапе используется конфиг для Context. Инициализация объекта проходит так:
    - Смотрится список доступных в системы GPU устройств. (Обычно это 1 – 2 устройства, выделенное и встроенное в процессор). Среди них выбирается наиболее производительное, предоставляющее весь запрошенный функционал (список расширений из конфига для класса Context). Эта логика целиком расположена в классе DevicePicker.
    - Для выбранного DevicePicker'ом устройства создаются API объекты VkDevice и VkQueue, необходимые для отправки команд и для создания различных ресурсов в видеопамяти этого устройства.
- Создается объект фреймворка Swapchain, который внутри себя инициализирует API объект VkSwapchain. Этот объект занимается предоставлением изображений, куда записывается результат работы графического конвейера, и последующим показом этих изображений на поверхности VkSurface.

После описанных шагов фреймворк готов к использованию.

Описание графического конвейера

На этом этапе пользователь с помощью фреймворка описывает процесс рендеринга. Процесс описания строится следующим образом:

- Создается пустой объект фреймворка RenderGraph. В последствии это и станет классом, являющим собой графический конвейер.
- В ResourceManager, находящийся внутри RenderGraph, добавляются записи о всех ресурсах(картинках/буферах), нужных описываемому конвейеру. В examples/raytracer это происходит тут: https://github.com/Ilistratov/RL/blob/master/examples/raytracer.cpp#L199
- В RenderGraph добавляются Pass'ы в той последовательности, в которой они будут исполняться. Pass'ми здесь называются определенные пользователем наследники класса render_graph/Pass. Наследоваться от Pass'а и определять свое поведение нужно следующим образом:

- В конструкторе наследника нужно при помощи методов AddBuffer/Image родительского класса объявить все ресурсы, к которым Pass хочет иметь доступ.
- В переопределенном OnResourcesInitialized можно поместить логику инициализации Pass'а, требующую наличия инициализированных запрошенных ресурсов. На пример тут в инициализированный буфер записывается информация, которую потом надо будет скопировать в видеопамять. https://github.com/Ilistratov/RL/blob/master/examples/raytracer.cpp#L124
- В переопределенном ReserveDescriptorSets можно запросить из находящегося внутри RenderGraph DescriptorPool'а дескрипторы. Эти объекты нужны для передачи информации о ресурсах в шейдеры, исполняемые на видеокарте. (Если pass не запускает шейдеры и манипулирует ресурсами лишь с помощью вызовов к VulkanAPI, то этот метод можно проигнорировать) Пример: https://github.com/Ilistratov/RL/blob/master/examples/raytracer.cpp#L177
- В переопределенном методе OnRecord наследник должен записывать желаемые команды в переданный ему VkCommandBuffer.

После того, как все нужные ресурсы объявлены, а желаемые Pass'ы добавлены, можно звать метод Init у RenderGraph, после чего начнется процесс инициализации графического конвейера.

Инициализация графического конвейера

На этом этапе RenderGraph собирает из Pass'ов информацию об использовании ресурсов, инициализирует их и прочие объекты, необходимые в процессе работы.

- При создании RenderGraph внутри создается объект фреймворка CommandPool, выдающий буферы для записи команд и их последующей отправки на исполнение на gpu.
- В момент, когда в RenderGraph добавлятся очередной Pass, объявленные используемыми в Pass'е ресурсы соотностятся с ресурсами, объявленными в ResourceManager, информация об использовании передается от Pass'а к ResourceManager. В этот же момент Pass может запросить дескрипторы из находящегося в RenderGraph пула.
- Создаются буферы/изображения, добавленые в RenderGraph через ResourceManager, при создании используется информация об использовании, предоставленная Pass'ами. Созданные буферы/изображения запрашивают память у DeviceMemoryAllocator'а, находящегося внутри ResourceManager'а.
- DeviceMemoryAllocator разово выделяет нужное кол-во памяти нужных типов для всех поступивших запросов (На видеокарте может иметься несколько видов памяти, отличающихся доступностью и эффективностью операций над ними, у

каждого ресурса может быть свой набор типов памяти, где он может быть размещен.), выделенная память закрепляется за ресурсами.
- Находящийся в RenderGraph DescriptorPool создает все запрошенные дескрипторы.
- Ресурсы, требующие инициализации своего содержимого в видеопамяти, записывают необходимые для этого команды, после чего те отправляются на исполнение.
- По окончанию исполнения отправленных на предыдущем этапе команд, у всех Pass'ов вызывается метод OnResourceInitialized.

По завершению работы метода RenderGraph::Init графический конвейер считается инициализированным, можно запускать процесс рендеринга вызовом RenderGraph::RenderFrame(). Обычно это делается внутри цикла.

Исполнение графического конвейера

Во время исполнения, все Pass'ы записывают свои команды в буфер, предоставленный CommandPool'ом. Pass сначала записывает команды, определенные пользователем в методе OnRecord, затем генерирует команды для синхронизации.

Дело в том, что VulkanAPI гарантирует лишь что отправленные на исполнение команды начнут выполняться в том-же порядке, в котором были отправлены. Никаких гарантий ни касаемо порядка завершения, ни касаемо видимости операций предыдущих команд для следующих, не дается. Для этого требуется вручную вставлять команды синхронизации, гарантирующие в определенных местах видимость изменений более ранних команд для более поздних. Для этого когда пользователь внутри Pass'а объявляет используемые ресурсы, для каждого ресурса он должен показать, каким именно образом Pass к нему обращается. Эта информация затем используется чтобы в нужных местах после команд самого Pass'а вставить команды синхронизации и сделать изменения в ресурсах видимыми для следующего потребителя.

Как только все команды были записаны, они отправляются на исполнение. Буфер возвращается обратно в CommandPool. CommandPool самостоятельно отследит, когда все команды в буфере выполнятся и буфер можно будет переиспользовать. При необходимости, CommandPool будет выделять дополнительные буферы.

Вот так в итоге будет выглядить .obj, скачанный отсюда: https://free3d.com/3d-model/serpertine-city-32219.html , отрисованный с помощью examples/raytracer. (Рис. **1**, **2**)

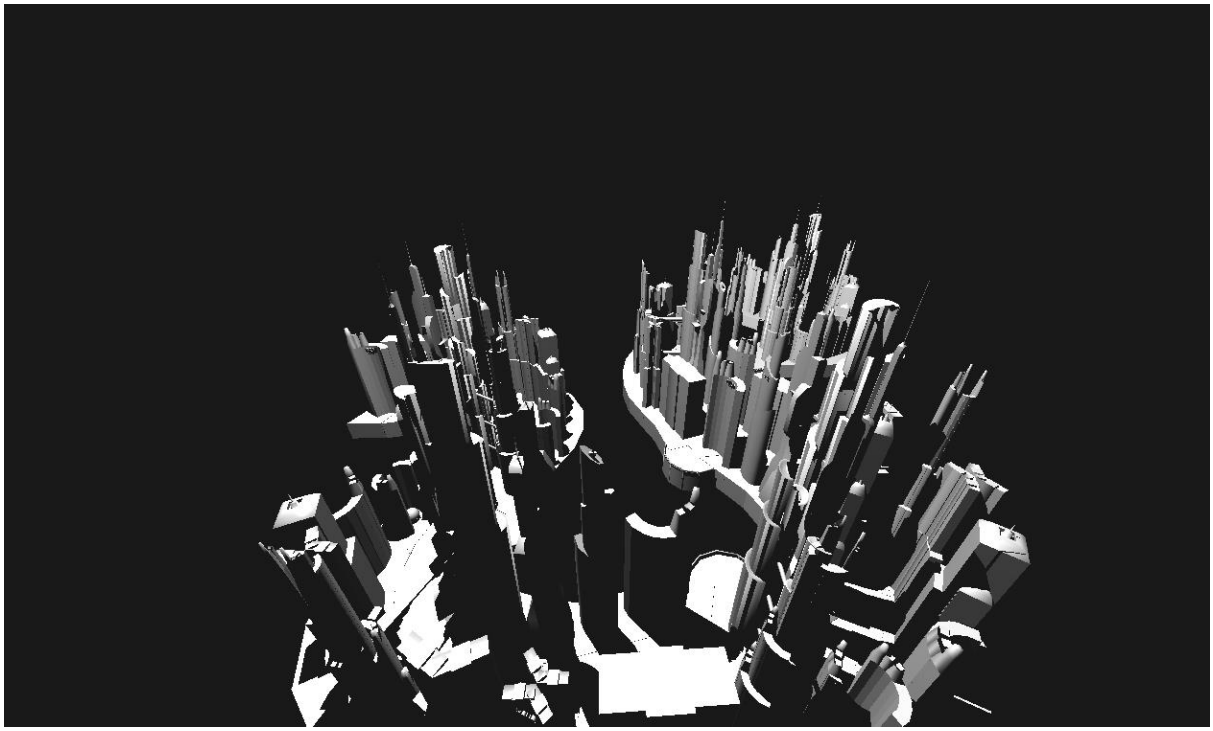

Рис. 1

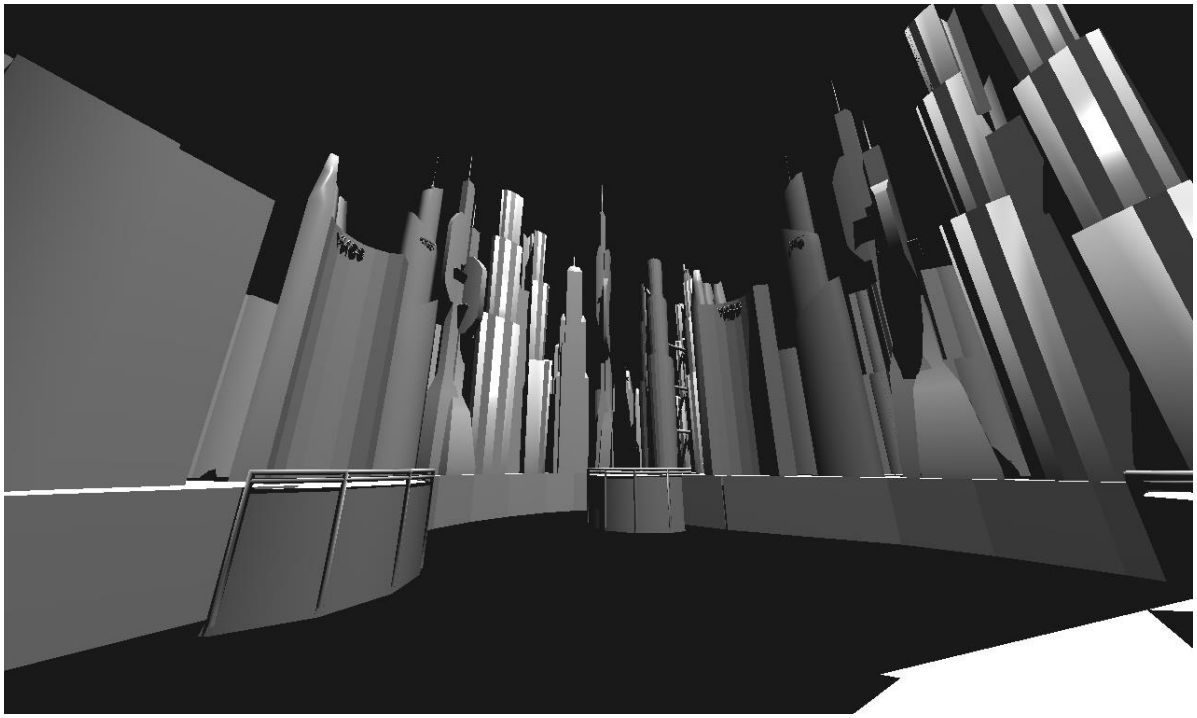

Рис. 2

Сборка/Запуск

Клонировать [репозиторий] с программой

Предварительно требуется установить

- VulkanSDK
- GLFW
- Header only tiny_obj_loader.h
- Header only stb_image.h
- GLM

.Lib от GLFW должны оказаться в src/third_party/lib.

.h от GLFW в src/third_party/include/GLFW

Вся папку с .h от GLM в src/third_party/include

Header only библиотеки тоже в src/third_party/include

Далее можно собирать проект CMake'ом без особых трудностей. По умолчанию собирается библиотека + рендерер из examples/raytracer. Чтобы пользоваться библиотекой, достаточно слинковать таргет с rl_lib и подключить заголовочный файл rl.h.

Заключение

Перечисленные объекты архитектуры render graph реализованы, фреймворк уже можно использовать. В дальнейшем планируется автоматизировать установку сторонних зависимостей при сборке, расширить возможности фреймворка(опциональное создание окна, 'склейка' ресурсов для экономии видеопамяти и т.д).

Список источников

[i] https://apoorvaj.io/render-graphs-1/
https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in

[ii] https://core.vmware.com/resource/exploring-gpu-architecture

[iii] https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

[iv] https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl

[v] https://www.khronos.org/opengl/#:~:text=OpenGL%C2%AE%20is%20the%20most,as%20well%20as%20network%2Dtransparent.

[vi] https://vulkan-tutorial.com/Overview

[vii] https://vulkan-tutorial.com/Overview

[viii] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-1-opening-a-window/