МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА СИСТЕМЫ ОСВЕЩЕНИЯ В ГРАФИЧЕСКОМ ДВИЖКЕ НА ОСНОВЕ АЛГОРИТМА ТРАССИРОВКИ ПУТИ С ИСПОЛЬЗОВАНИЕМ API VULKAN

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Цоколо Александра Александровича

Научный руководитель
зав. каф., к.ф.-м.н., доцент _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2026

Постановка задачи

За последние годы требования к качеству изображения в интерактивной графике заметно выросли. От современных приложений ожидают реалистичных теней, отражений, непрямого освещения. Растеризационные методы позволяют получать изображение быстро, однако многие световые эффекты в них приходится приближать отдельными эвристическими техниками. Поэтому для получения более физически правдоподобного освещения всё чаще используется трассировка лучей и трассировка путей, особенно с учётом аппаратной поддержки ray tracing в современных GPU.

Целью настоящей работы является разработка графического движка для визуализации трёхмерных сцен методом трассировки путей с использованием графического API Vulkan и аппаратного ускорения трассировки лучей.

Задачи:

1. Изучить теоретические основы алгоритма трассировки пути.
2. Спроектировать модульную архитектуру движка.
3. Реализовать загрузку моделей в форматах OBJ и glTF.
4. Реализовать алгоритм трассировки путей на базе Vulkan Ray Tracing.
5. Реализовать накопление сэмплов между кадрами для снижения шума при неподвижной камере.

Архитектура приложения

Архитектура приложения разделена на несколько уровней. Уровень Application отвечает за точку входа, создание окна, сцены и renderer. Платформенный слой скрывает работу с окном и вводом через GLFW. Core содержит общие средства, не связанные напрямую с графическим API: логирование, математические типы, обёртки над handle и контейнеры для хранения ресурсов. Уровень Assets отвечает за загрузку и хранение мешей, материалов и текстур. Scene содержит сущности и компоненты. Renderer Frontend переводит сцену в компактные данные кадра. Vulkan backend создаёт GPU-ресурсы и выполняет трассировку.

Такое разделение выбрано для того, чтобы сцена не зависела от Vulkan. Компоненты не содержат `VkBuffer`, `VkImage`, `VkDescriptorSet` и другие низкоуровневые объекты. Они описывают только логическое состояние: где находится объект, какая модель ему назначена, какая камера активна, какие источники

света присутствуют в сцене. Все детали размещения данных в памяти GPU находятся внутри backend. Это упрощает расширение движка и позволяет отдельно изменять сценовую систему и реализацию renderer.

Система ресурсов и загрузка моделей

Система ресурсов построена вокруг `AssetManager`. Он хранит загруженные меши, материалы и текстуры, а доступ к ним выполняется не через прямые указатели, а через дескрипторы `MeshHandle`, `MaterialHandle` и `TextureHandle`. Такой handle содержит индекс ресурса и поколение ячейки. При удалении или замене ресурса поколение изменяется, поэтому старую ссылку можно распознать как недействительную. Это важно для движка, где один и тот же меш или материал может использоваться несколькими объектами сцены.

Для хранения ресурсов используется `SlotMap`. Этот контейнер позволяет добавлять и удалять элементы без немедленного нарушения всех внешних ссылок. При добавлении ресурс помещается в свободную ячейку, а при удалении ячейка помечается свободной и увеличивает своё поколение. Проверка handle сравнивает сохранённое поколение с текущим. Благодаря этому `AssetManager` может безопасно возвращать доступ к ресурсам и обнаруживать обращения к уже устаревшим объектам.

`MeshAsset` хранит данные геометрии в формате, удобном для дальнейшей загрузки на GPU. В него входят массив вершин, массив индексов, список подмешей и ограничивающий объём. Вершина содержит позицию, нормаль, текстурные координаты и tangent. Подмеш задаёт диапазон индексов и ссылку на материал. Такая структура нужна, потому что один файл модели может содержать несколько частей с разными материалами. При рендеринге эти части всё ещё принадлежат одному мешу, но используют разные записи материала.

`MaterialAsset` описывает параметры поверхности. В нём хранятся базовый цвет, roughness, metallic, эмиссивный цвет, ссылки на albedo texture, normal texture, emissive texture и дополнительные параметры прозрачности. Отдельно учитываются transmission, index of refraction и признак тонкой прозрачной поверхности. Такой материал затем преобразуется в GPU-структуру, где числовые параметры и индексы текстур доступны shader.

`TextureAsset` содержит декодированные пиксели изображения, ширину, высоту и число каналов. Цветовые изображения рассматриваются как sRGB-данные, а служебные карты, например normal map или metallic-roughness, ис-

пользуются как линейные данные. Загрузка изображений выполняется через stb_image. После декодирования текстура сохраняется в `AssetManager` и может быть связана с материалом.

Для загрузки моделей поддерживаются форматы OBJ и glTF. OBJ используется как простой формат для геометрии и материалов MTL. Загрузка выполняется через tinyobjloader. Из файла извлекаются позиции вершин, нормали, текстурные координаты, индексы, группы геометрии и материалы. Если у модели отсутствуют нормали, они могут быть восстановлены по треугольникам. Полученные данные приводятся к внутреннему формату `Vertex` и сохраняются в `MeshAsset`.

При обработке OBJ учитываются материалы из MTL-файла. Для каждой группы геометрии создаётся подмеш, которому назначается соответствующий `MaterialHandle`. Диффузный цвет переносится в albedo color, a diffuse texture загружается как albedo texture. Если в MTL задан чёрный Kd, но при этом есть текстура, базовый цвет материала устанавливается белым, чтобы текстура не была полностью затемнена при умножении на цвет материала. Это решает практическую проблему, встречающуюся у некоторых OBJ-моделей.

Формат glTF используется как более современный источник PBR-данных. Загрузка выполняется через cgltf. Из файла читаются узлы сцены, меши, примитивы, материалы и текстуры. В отличие от OBJ, glTF лучше соответствует используемой модели материалов, потому что содержит base color, metallic-roughness, normal map, occlusion и emissive texture. При загрузке такие параметры напрямую переносятся в `MaterialAsset`.

При загрузке glTF данные модели приводятся к внутреннему представлению `MeshAsset`. Геометрия разбивается на подмеши, для каждого подмеша сохраняются диапазон индексов и ссылка на материал. Из файла извлекаются позиции вершин, нормали, текстурные координаты, индексы, параметры материалов и связанные с ними текстуры. Благодаря этому дальнейшая часть renderer работает с единым представлением мешей и материалов независимо от исходного формата модели.

Сцена и подготовка данных кадра

Сцена реализована на основе Entity-Component-System. Сущность является идентификатором, а содержательное состояние задаётся компонентами. Такой подход удобен для renderer, потому что объект может иметь только нуж-

ный набор свойств.

TransformComponent хранит локальную позицию, поворот и масштаб объекта. На его основе вычисляется матрица модели. MeshRendererComponent связывает сущность с MeshHandle и указывает, какая модель должна быть отображена. CameraComponent содержит параметры камеры, необходимые для построения первичных лучей. LightComponent описывает аналитические источники света, используемые при оценке прямого освещения.

Scene отвечает только за хранение сущностей и компонентов. Она не выполняет отрисовку и не создаёт GPU-ресурсы. Это делает сцену независимой от backend и позволяет использовать её как высокоуровневое описание кадра. Перед рендерингом данные сцены извлекаются отдельным слоем RendererFrontend. Он обходит сущности, находит активную камеру, собирает видимые mesh instances и источники света.

Результатом работы RendererFrontend является FrameData. В него помещаются параметры камеры, список экземпляров мешей, список источников света и ссылки на ресурсы. Данные имеют плоскую структуру и удобны для передачи в backend. VulkanRenderer не работает с ECS напрямую. Он получает уже подготовленный набор объектов и может сосредоточиться на обновлении GPU-буферов, TLAS, descriptor set и запуске трассировки.

Передача ресурсов на GPU

Для передачи геометрии на GPU используются специальные кэши. Они связывают CPU-ресурсы AssetManager с объектами Vulkan. Если MeshAsset уже был загружен на GPU, повторная загрузка не выполняется. Это предотвращает дублирование буферов и упрощает использование одного меша несколькими экземплярами сцены.

Для каждого меша создаются vertex buffer и index buffer. Они размещаются в памяти устройства, потому что closest-hit shader должен быстро читать вершины при обработке пересечений. Загрузка выполняется через staging buffer. Сначала данные записываются в память, доступную CPU, затем командой копирования переносятся в device-local buffer. После загрузки временный staging buffer удаляется.

Для GPU-буферов геометрии используется Buffer Device Address. После создания vertex и index buffers renderer получает их адреса в памяти устройства. Эти адреса записываются в GpuInstanceData. Благодаря этому shader может

читать геометрию через buffer reference, не создавая отдельный descriptor для каждого меша. Такой подход особенно удобен для ray tracing, где closest-hit shader должен быстро перейти от найденного экземпляра к нужным вершинам.

Материалы упаковываются в storage buffer с массивом `GpuMaterial`. В каждой записи хранятся базовый цвет, roughness, metallic, emissive color, параметры прозрачности и индексы текстур. Подмеши упаковываются в отдельную таблицу `GpuSubmeshData`. Она задаёт диапазон индексов, ссылку на материал и служебные параметры. Closest-hit shader по geometry index выбирает нужный подмеш и затем получает материал.

Текстуры загружаются в `VkImage`. Для каждой текстуры создаются image view и sampler. Все текстуры помещаются в bindless descriptor array. В материале хранится индекс текстуры, а shader использует этот индекс для выборки. Если у материала нет текстуры, используется fallback-текстура. Такая схема позволяет работать со множеством материалов без пересоздания descriptor set для каждого объекта.

Vulkan backend и ускоряющие структуры

Vulkan backend выполняет инициализацию графического API и всех объектов, необходимых для кадра. На этапе запуска создаётся Vulkan instance, выбирается физическое устройство, проверяется поддержка расширений ray tracing, создаётся логическое устройство и получаются очереди. Дополнительно загружаются функции расширений, связанные с acceleration structures, ray tracing pipeline и Buffer Device Address.

Для вывода изображения создаётся swapchain. Он содержит набор изображений, которые затем представляются на экран. Для каждого кадра создаётся frame context. В него входят командный буфер, fence, семафоры и вспомогательные данные синхронизации. Такая структура позволяет не перезаписывать ресурсы, которые ещё используются GPU, и корректно связывать получение изображения, выполнение команд и представление результата.

Path tracer не пишет результат сразу в swapchain image. Для этого используется отдельное accumulation image формата rgba32f. В нём хранится накопленный цвет пикселей. Ray generation shader читает предыдущее значение, добавляет новый сэмпл и записывает обновлённое среднее. После трассировки результат можно перенести или вывести на swapchain image.

Работа с буферами и изображениями вынесена в отдельные классы.

`Buffer` инкапсулирует `VkBuffer`, выделенную память и GPU-адрес. `Image` хранит `VkImage`, image view, формат и текущее назначение. `ResourceUploader` отвечает за загрузку данных через staging buffers. Такое разделение уменьшает количество повторяющегося Vulkan-кода в renderer и делает логику загрузки ресурсов более понятной.

Для аппаратной трассировки геометрия включается в acceleration structures. В реализации используется двухуровневая схема BLAS и TLAS. BLAS строится для меша и содержит его треугольную геометрию. TLAS строится для текущей сцены и содержит экземпляры объектов, каждый из которых ссылается на соответствующий BLAS.

При построении BLAS используются vertex и index buffers, уже находящиеся на GPU. Для Vulkan задаются формат вершин, адрес vertex buffer, адрес index buffer, число треугольников и параметры геометрии. После запроса размеров создаются buffer для самой acceleration structure и scratch buffer. Затем в командный буфер записывается команда построения BLAS. После её выполнения структура может использоваться при трассировке.

TLAS создаётся на основе списка экземпляров сцены. Каждый экземпляр содержит ссылку на BLAS, матрицу преобразования, маску видимости и instance custom index. Этот индекс затем доступен в closest-hit shader через `gl_InstanceCustomIndexEXT`. По нему shader обращается к массиву `GpuInstanceData` и получает адреса буферов, смещение подмешей и матрицы объекта.

При обновлении сцены renderer формирует массив `VkAccelerationStructureInstanceKHR`. В него записываются трансформации объектов и адреса BLAS. Если геометрия не изменилась, BLAS переиспользуется. При изменении положения объектов достаточно обновить TLAS. Это соответствует структуре сцены, где один и тот же mesh может использоваться в разных местах с разными transform.

Одновременно с подготовкой TLAS формируется список эмиссивных треугольников. Renderer проходит по подмешам и материалам, находит поверхности с ненулевой эмиссией и записывает ссылки на соответствующие треугольники. Для каждого элемента сохраняются индекс экземпляра, индекс подмеша, индекс примитива и накопленная вероятность выбора. Вес зависит от площади треугольника и яркости материала. Этот список нужен для прямого сэмплиро-

вания светящейся геометрии.

Ray tracing pipeline и Shader Binding Table

Ray tracing pipeline собирается из нескольких shader stages. В реализации используются ray generation shader, обычный miss shader, shadow miss shader и closest-hit shader. Any-hit и intersection shaders не применяются, потому что сцена представлена треугольниками, а пересечение с ними выполняется аппаратно. Closest-hit shader только восстанавливает данные поверхности и заполняет payload.

Pipeline создаётся через вспомогательный builder. В builder передаются descriptor set layout, push constant range, shader modules и hit groups. После этого создаются `VkPipelineLayout` и `VkPipeline`. Для каждой shader group Vulkan возвращает shader group handle. Эти записи нужны для формирования Shader Binding Table. SBT создаётся как GPU-буфер и делится на регионы raygen, miss и hit.

Регион raygen содержит запись ray generation shader. Регион miss содержит записи для обычного miss shader и shadow miss shader. Регион hit содержит hit group с closest-hit shader. Все записи укладываются с учётом выравнивания, которое сообщает устройство. При вызове `vkCmdTraceRaysKHR` renderer передаёт адреса и размеры этих регионов. Vulkan использует SBT для выбора нужного shader во время трассировки.

Descriptor set связывает shader с ресурсами кадра. В нём находятся TLAS, accumulation image, Camera UBO, buffer экземпляров, buffer источников света, buffer материалов, таблица подмешей, bindless texture array и список эмиссивных треугольников. Push constants передают номер кадра, флаг сброса накопления, максимальное число отскоков и количества источников.

Шейдеры трассировки пути

Ray generation shader является основной частью path tracer. Он запускается для каждого пикселя изображения. Shader получает координаты пикселя, размер изображения, параметры камеры и номер кадра. Для антиалиасинга внутри пикселя выбирается случайное смещение. После этого через обратные матрицы проекции и вида строится первичный луч из камеры.

Для одного пикселя за кадр может рассчитываться несколько сэмплов. Каждый сэмпл вызывает функцию tracePath. Внутри неё хранится накоплен-

ный световой вклад radiance и текущий вес пути throughput. Сначала radiance равен нулю, а throughput равен единице. На каждом отскоке shader вызывает `traceRayEXT` по TLAS и получает результат в payload.

Если луч не попадает в геометрию, miss shader записывает цвет окружения. Этот цвет добавляется к radiance с учётом текущего throughput, после чего путь завершается. Если пересечение найдено, closest-hit shader заполняет payload данными поверхности. В payload передаются позиция попадания, нормаль, UV, параметры материала, эмиссия, прозрачность и дополнительные значения, необходимые ray generation shader.

После возврата из `traceRayEXT` ray generation shader проверяет материал. Если поверхность эмиссивная, её вклад добавляется к radiance. Для непрозрачных материалов выполняется оценка прямого освещения, затем выбирается новое направление пути. Для прозрачных материалов используется отдельная ветка: shader выбирает отражение или прохождение луча с учётом параметров материала. После выбора направления throughput обновляется, начало луча немного смещается от поверхности, и цикл продолжается.

Closest-hit shader получает от Vulkan barycentric coordinates, primitive ID, geometry index и instance custom index. Через instance custom index он читает `GpuInstanceData`. В этой структуре находятся GPU-адреса vertex buffer и index buffer, смещение в таблице подмешей и матрицы преобразования. По primitive ID shader находит индексы треугольника, читает три вершины и интерполирует их атрибуты.

На основе барицентрических координат вычисляются позиция попадания, нормаль и текстурные координаты. Нормаль преобразуется в мировое пространство. Если у материала есть normal map, shader считывает её значение, переводит из диапазона текстуры в вектор и применяет через TBN-матрицу. Для построения TBN используется tangent из вершины или восстановленное направление на основе UV.

Материал считывается из массива `GpuMaterial`. Если задана albedo texture, shader выбирает цвет из bindless texture array. Если есть metallic-roughness texture, из неё берутся roughness и metallic. Если задана emissive texture, она умножается на эмиссивный цвет материала. Также учитываются normal map, occlusion и параметры прозрачности. После этого payload возвращается в ray generation shader.

Для промаха используется miss shader. Он задаёт цвет окружения для обычных лучей. Для теневых лучей используется отдельный shadow miss shader. Перед запуском shadow ray флаг видимости устанавливается в ноль. Если луч доходит до miss shader, значит препятствий на пути не было, и флаг visible устанавливается в единицу. Если луч попал в геометрию, closest-hit не вызывается из-за флага SkipClosestHitShader, а visible остаётся нулём.

Освещение, материалы и накопление

Прямое освещение рассчитывается в ray generation shader. Для каждого аналитического источника определяется направление к свету, расстояние и интенсивность. Затем запускается shadow ray. Если путь перекрыт геометрией, вклад источника не добавляется. Если источник видим, shader вычисляет значение BRDF в направлении источника и добавляет вклад к radiance. Для направленных источников используется направление света, для точечных и spot lights учитывается положение источника.

Отдельно реализована работа с эмиссивной геометрией. Вместо того чтобы ждать случайного попадания bounce ray в маленький светящийся треугольник, shader выбирает такой треугольник из заранее подготовленного списка. Выбор выполняется по CDF. После этого внутри треугольника выбирается случайная точка, строится shadow ray и проверяется видимость. Если точка видима, её вклад добавляется как прямое освещение.

После оценки прямого освещения выбирается направление следующего отскока. Для непрозрачного материала используется смесь диффузной и зеркальной стратегии. Диффузное направление выбирается по косинусной полусфере относительно нормали. Зеркальное направление строится через выбор микронормали и отражение луча. После выбора направления вычисляются BRDF и плотность вероятности, а коэффициент переноса пути(throughput) обновляется для продолжения пути.

Для прозрачных материалов используется другая ветка. Shader определяет, будет ли луч отражён или пройдёт через поверхность. При прохождении может использоваться преломление с учётом IOR или прямое прохождение для тонких поверхностей. Такая ветка позволяет отображать стеклоподобные объекты и не смешивать прозрачность с обычной непрозрачной моделью материала.

Чтобы путь не продолжался бесконечно, используется ограничение `maxBounces`. Дополнительно после нескольких отскоков применяется Russian

roulette. Вероятность продолжения зависит от текущего `throughput`. Если путь завершается, дальнейшие отскоки не считаются. Если путь продолжается, `throughput` корректируется так, чтобы не вносить систематическое затемнение результата.

Изображение, полученное path tracing, содержит шум при малом числе сэмплов. Поэтому renderer выполняет накопление между кадрами. Внутри одного кадра shader может усреднить несколько сэмплов пикселя. Затем полученное значение смешивается с предыдущим цветом из accumulation image. Если накопление сброшено, значение записывается как первый сэмпл. Если накопление продолжается, новое среднее вычисляется с учётом `frameIndex`.

Накопление сбрасывается при изменении камеры, размера окна или параметров, влияющих на построение лучей. Старые сэмплы соответствуют другому положению камеры или другой сетке пикселей, поэтому их нельзя использовать. При неподвижной камере `frameIndex` увеличивается, и изображение постепенно очищается от шума. Такой способ хорошо подходит для интерактивного просмотра сцены, когда пользователь остановил камеру и ожидает улучшения качества.

При работе с непрозрачными материалами renderer использует параметры albedo, roughness и metallic. Текстуры позволяют задавать цвет и свойства поверхности на уровне отдельных участков модели. Normal maps изменяют локальную нормаль без изменения самой геометрии. Это позволяет получать более детализированный вид поверхности при сохранении исходного числа треугольников.

При использовании эмиссивных материалов геометрия становится источником света. Такие поверхности участвуют не только как видимые объекты, но и как элементы освещения. Для их обработки используется список эмиссивных треугольников и shadow rays. Это позволяет получать вклад от светящихся объектов даже тогда, когда случайный path ray редко попадает в них напрямую.

Аналитические источники света используются для сцен, где нужен управляемый источник освещения. Они передаются в shader через отдельный buffer. Renderer поддерживает обработку таких источников в общем цикле path tracing. Для каждого источника выполняется проверка видимости, после чего его вклад добавляется к текущей точке поверхности.

Прозрачные объекты обрабатываются через параметры transmission и IOR.

При попадании в такую поверхность shader выбирает дальнейшее направление луча отдельно от обычной непрозрачной ветки. Это позволяет отображать прохождение луча через стеклоподобные материалы и учитывать отражение на границе среды. Для практического renderer это важно, потому что прозрачные объекты часто вызывают ошибки, если обрабатывать их как обычный diffuse или metallic material.

В реализации также учитывается порядок обновления GPU-ресурсов. Перед запуском трассировки backend убеждается, что необходимые меши, материалы и текстуры загружены в GPU-кэши. Затем обновляются buffers с материалами, источниками, подмешами и instance data. После этого строится или обновляется TLAS, обновляется descriptor set и только затем записывается команда `vkCmdTraceRaysKHR`. Такой порядок нужен, чтобы shader видел согласованные данные текущего кадра.

Главный цикл приложения связывает все подсистемы. Пользовательский ввод изменяет камеру, `Scene` хранит состояние объектов, `RendererFrontend` формирует `FrameData`, `VulkanRenderer` обновляет ресурсы и запускает трассировку. После выполнения команд GPU результат оказывается в accumulation image и выводится на экран через swapchain. В следующем кадре процесс повторяется, при этом накопление либо продолжается, либо сбрасывается.