Санкт-Петербургский государственный университет

Цырендашиев Сультим Баиржапович

Выпускная квалификационная работа

Разработка библиотеки для портирования приложений с трехмерной графикой, использующих конвейер фиксированной функциональности, на метод трассировки путей

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2017 «Программная инженерия»*

Научный руководитель:
к. ф.-м. н., доц., Д.В. Луцив

Консультант:
генеральный директор ООО «Системы компьютерного зрения» А.А. Пименов

Рецензент:
программист ООО «Леста» Е.М. Щавелев

Санкт-Петербург
2021

Saint Petersburg State University

Sultim Tsyrendashiev

Bachelor's Thesis

Development of the library for porting fixed function 3D graphics applications to path tracing

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2017 «Software Engineering»*

Scientific supervisor:
Ph.D., docent D.V. Luciv

Consultant:
CEO of «Computer Vision Systems» A.A. Pimenov

Reviewer:
programmer at «Lesta Studio» E.M. Shchavelev

Saint Petersburg
2021

Оглавление

Введение

Одной из главных проблем компьютерной графики является отрисовка трехмерных сцен, то есть набора трехмерных объектов, их свойств, источников света и камеры, с точки зрения которой и производится отрисовка сцены. При этом в интерактивных приложениях необходимо поддерживать высокую частоту смены кадров: чем плавнее анимированное изображение, тем легче пользователю его воспринимать.

С начала 90-х годов XX века самым распространённым методом решения этой проблемы является растеризация — преобразование трехмерных объектов, представленных в виде набора треугольников, в пиксели на экране (рис. 1). Этот метод наиболее популярен из-за высокой производительности и гибкости. Тем не менее, растеризация имеет недостатки: для создания реалистичного освещения необходимо использовать большое количество различных техник для каждой отдельной ситуации в трехмерной сцене.

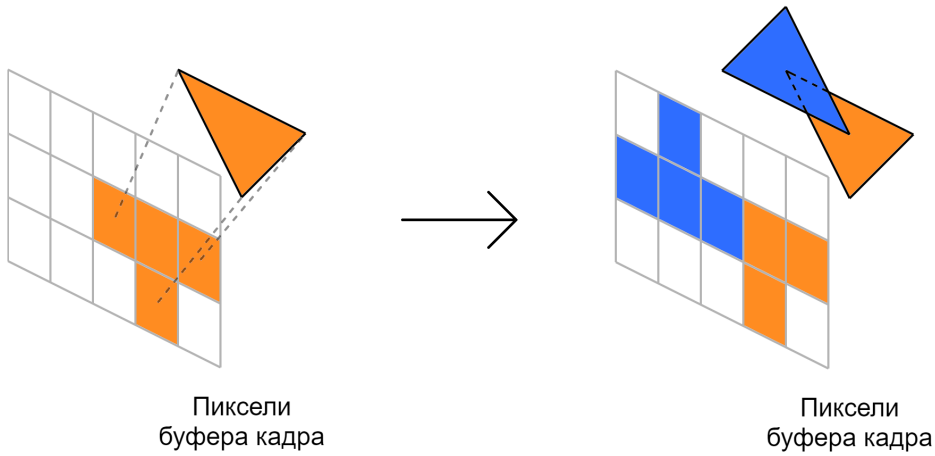


Рис. 1: Основной процесс растеризации – проецирование трехмерных объектов из специального пространства на пиксельную сетку буфера кадра.

Графическим конвейером называют последовательность операций над трехмерными объектами и их свойствами, которые необходимы для отрисовки трехмерных сцен. Различают два вида графического конвейера для растеризации: фиксированной функциональности и шейдерный. Они отличаются тем, что шейдерный контейнер даёт разработчи-

кам возможность переделывать/вставлять отдельные части с помощью программ, запускаемых на видеокарте — шейдеров. Конвейер фиксированной функциональности позволяет настраивать только определенные параметры, например, вершины треугольников, которые обрабатываются только одним заданным способом, а в шейдерном конвейере для этого необходимо реализовать программу, что позволяет задавать практически любую логику для обработки вершин на графическом процессоре: скелетная анимация, системы частиц и др.

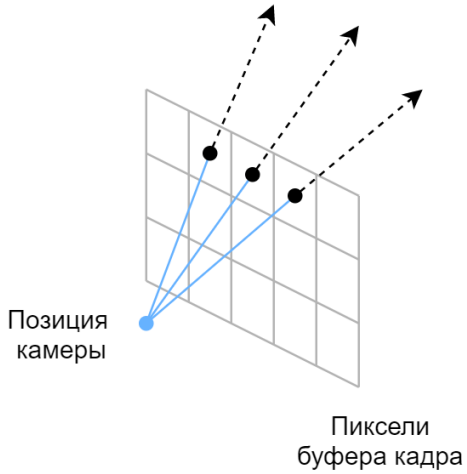


Рис. 2: При трассировке производится обход пикселей буфера кадра и запуск лучей в сцену из них.

Наряду с растеризацией, имеется метод трассировки лучей, который позволяет рассматривать расчёт освещения в трехмерной сцене более обобщенно, чем при растеризации. Трассировка лучей отличается от растеризации тем, что вместо последовательной отрисовки каждого отдельного объекта, в сцену испускаются лучи от камеры, и в зависимости от того, в какой объект попал луч, окрашивается пиксель (рис. 2). При этом разработчики самостоятельно задают логику взаимодействия лучей и объектов — следует ли лучу отразиться, преломиться и т.д. Взаимодействуя с разными поверхностями, луч строит некоторый путь, и если в каждой точке пересечения луча и сцены вычислять уравнение отрисовки (rendering equation) [9], то такой метод называется трассировкой путей. Он позволяет создавать реалистичное освещение, тени и полутени, корректные отражения и др. Однако каждый запуск отдель-

ного луча сопровождается высокой нагрузкой на видеокарту. Ситуация усугубляется тем, что на один пиксель необходимо запустить хотя бы несколько лучей, а на экранах пользователей таких пикселей — миллионы.

Из-за низкой производительности метод отрисовки лучей долгое время использовался только в системах, где нет жестких ограничений на время отрисовки одного кадра. Но с дальнейшим значительным увеличением мощностей пользовательских видеокарт и появлением графических процессоров NVIDIA Turing (2018 год) [30] данный метод стал доступен и для приложений с высокой частотой смены кадров.

Значительное различие между двумя концепциями — растеризацией и трассировкой путей, — а также высокая нагрузка на видеокарту во время расчета пересечений лучей, являются главными причинами сложности портирования приложений с одного метода на другой. Поэтому необходимо решение, которое позволяло бы автоматизировать перенос приложения с трехмерной графикой с растеризационного метода на трассировку путей. Главная мотивация такого портирования заключается в отрисовке более реалистичного и динамичного освещения, так как большинство целевых приложений используют такую технику, как карта теней, главным недостатком которой является статичность, то есть при изменении состояния сцены освещение в ней не меняется, например, если объект сдвинулся, то его тень всё ещё будет в прежнем положении.

Создание такого решения для произвольных приложений трудоёмко, поэтому данная работа направлена на приложения, использующие фиксированный графический конвейер. Это связано с тем, что в такого типа конвейере, разработчикам даётся меньше свободы во взаимодействии с графическим процессором (в связи с архитектурой графических ускорителей в конце 90-х гг.), поэтому и в создаваемой библиотеке необходимо рассмотреть и реализовать значительно меньшее количество различных случаев, по сравнению с шейдерным конвейером.

1. Постановка задачи

Цель данной работы — разработать библиотеку, позволяющую использовать трассировку путей в приложениях, использующих растеризацию с конвейером фиксированной функциональности для отрисовки трехмерной графики; а также портировать одно такое приложение, а именно, видеоигру “Serious Sam: The First Encounter” (2001) [25]. Основным требованием является сохранение высокой частоты смены кадров.

Для достижения этой цели были сформулированы следующие задачи.

- Проектирование интерфейса и архитектуры библиотеки, позволяющие приложениям с конвейером фиксированной функциональности загружать данные с минимальной нагрузкой.
- Реализация системы обработки данных для отрисовки: трехмерная геометрия, материалы, источники света.
- Реализация отрисовки трехмерных сцен с освещением, рассчитанным с помощью трассировки путей; а также отрисовки с помощью растеризации для геометрии, не участвующей в освещении.
- Апробация библиотеки посредством интеграции в целевое приложение.

2. Обзор

2.1. Предметная область

Концепция метода трассировки лучей достаточно простая в сравнении с другими способами отрисовки: на трёхмерной сцене расположены объекты, источники света и камера, из которой испускаются лучи, которые взаимодействуют с объектами, таким образом, окрашивается необходимый пиксель на экране. Такие техники, как теневые карты (англ. shadow maps), предрасчитанные карты освещенности (англ. baked lightmaps) и отражений (англ. baked reflection cubemaps), отражения, берущие информацию только из экранного пространства (англ. screen-space reflections), и другие, нацелены на симуляцию движения света в пространстве, и они были созданы для обхода ограничений, которые накладывает метод растеризации, но при этом все эти приёмы так или иначе имеют недостатки и визуальные артефакты. При трассировке лучей же, такие методы симуляции света неуместны, так как есть возможность симулировать упрощенные физические законы, по которым распространяется свет (видимое электромагнитное излучение). Например, в реальном мире свет распространяется от источников света и в конце попадает в оптический прибор, но по принципу обратимости хода лучей света этот процесс может быть устремлен в обратную сторону: лучи отправляются из оптического прибора.

В данной работе не будут приниматься во внимание гибридные методы отрисовки, то есть использующие одновременно и трассировку лучей, и растеризацию для расчёта освещения, так как целью является именно трассировка путей. Вместе с тем, библиотека предназначена для приложений с высокой частотой смены кадров, следовательно, алгоритмы, библиотеки, фреймворки созданные для автономной отрисовки кадров (англ. offline rendering), в которых время на отрисовку не ограничено, также не подлежат рассмотрению.

2.1.1. Уравнение отрисовки

Радиометрия – это методы для измерения электромагнитного излучения, в том числе и видимого. Величины, использующиеся в радиометрии:

- *Поток излучения* Φ (англ. *radiant flux*) (Вт) – это энергия, переносимая излучением через поверхность за единицу времени;
- *Облученность* E (англ. *irradiance*) ($\frac{\text{Вт}}{\text{м}^2}$) – поток излучения относительно площади поверхности;
- *Энергетическая яркость* L (англ. *radiance*) ($\frac{\text{Вт}}{\text{м}^2\text{ср}}$) – поток излучения относительно площади и телесного угла, то есть относительно отдельно взятого луча.

Оптические приборы (к примеру, камеры, глаза) измеряют *энергетическую яркость* (англ. *radiance*). Следовательно, для создания реалистичного освещения следует рассчитать энергетическую яркость на позиции каждого пересечения луча, выпущенного из камеры, со сценой.

Для оценки энергетической яркости в 1986 году было предложено уравнение отрисовки [9]:

$$L_o(p, v) = L_e(p, v) + \int_{l \in \Omega} f(l, v) L_i(p, l) |n \cdot l| \, dl, \tag{1}$$

где $L_o(p, v)$ – исходящая энергетическая яркость от поверхности в точке p и направлением v от точки p до камеры, $L_e(p, v)$ – энергетическая яркость, выпущенная самой поверхностью, n – вектор-нормаль в p, Ω – полусфера направлений[1] над точкой p, $|n \cdot l|$ – модуль скалярного произведения векторов n и l.

$L_i(p, l)$ — входящая[2] энергетическая яркость на точку p от направления $-l$. L_i можно рассчитывать [18] как:

$$L_i(p, l) = L_o(r(p, l), -l), \tag{2}$$

[1] вектора $l \in \Omega$ направлены из точки p

[2] по соглашению, L_i принимает принимает вектор l, направленный из точки p, поэтому слово "входящий" может несколько запутать

где $r(p, l)$ – функция возвращающая точку пересечения луча, испущенного из точки p в направлении l, с какой-либо поверхностью на сцене. Таким образом, чтобы рассчитать входящую энергетическую яркость в одной точке, необходимо найти исходящую в другой, для которой, в свою очередь, нужна входящая и так далее (рис. 3).

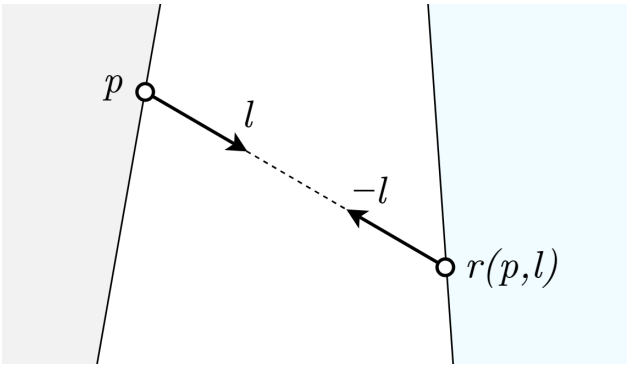


Рис. 3: Для расчёта входящей энергетической яркости $L_i(p, l)$ можно использовать исходящую энергетическую яркость L_o из другой точки, а именно $r(p, l)$.

Исходящая энергетическая яркость непрозрачных поверхностей чаще всего происходит не от самой поверхности, а извне и всего лишь отражается от неё. Рассмотрим случай, при котором такое отражение направлено в камеру, т.е. с направлением v. Тогда функция $f(l, v)$, описывающая отношение этой отраженной энергетической яркости к облученности входящего от направления l (т.е. потоку излучения на некоторой малой площади поверхности), называется *двулучевой функцией отражательной способности* (англ. *bidirectional reflectance distribution function*).

В совершенстве, во всех точках пересечения лучей камеры и сцены необходимо полностью решить уравнение отрисовки, то есть энергетическая яркость, входящая в некоторую точку, является энергетической яркостью, которая была испущена[3] поверхностью, на которой находится эта точка, или отражена[4] той или иной поверхностью. Для вычисле-

[3]первое слагаемое в уравнении отрисовки

[4]второе слагаемое в уравнении отрисовки

ния интеграла будет использоваться метод Монте-Карло:

$$L_o(p, v) = L_e(p, v) + \frac{1}{N} \sum_{i=0}^{N} f(l_i, v) L_o(r(p, l_i), -l_i)|n \cdot l_i|. \quad (3)$$

При этом трассировка лучей относительно дорогостоящая операция и из-за достаточно жёстких ограничений на время вычисления освещения объём выборки для метода Монте-Карло N особенно мал.

Стоит отметить, что использование выборки по значимости трехмерного направления l_0 на полусфере существенно, так как, например, если l_0 будет близко к горизонту, то скалярное произведение (или, что то же самое, косинус угла между l_0 и n) будет близко к нулю, а следовательно, рассчитанное значение $f(l_0, v)L_o(r(p, l_0), -l_0)$ будет иметь малый вклад в расчёт L_o. Поэтому также необходимо рассматривать различные случаи свойств поверхностей и в зависимости от них выбирать распределение для l_0.

2.1.2. Устранение шума

Так или иначе, малый размер выборки влечёт за собой огромное количество шума, поэтому необходимо также реализовать алгоритм устранения шума. Обычно при отрисовке на трехмерные объекты наносится множество различных изображений (текстур), придающих объектам высокую детализированность, но чтобы предотвратить размытие этих деталей, перед применением алгоритмов устранения шума необходимо отрисовать их в отдельный канал G-буфера [22]. Такой буфер представляет собой набор буферов с информацией о пикселях, которые впоследствии будут использованы при расчёте освещения и конструировании финального изображения, которое уже будет показано на экране. G-буфер содержит следующую информацию в каждом пикселе: цвет поверхности (на которую попал луч выпущенный из данного пикселя) без освещения (albedo), смещение вектора нормали (normal map), насколько шероховатая поверхность (roughness), насколько она является металлической (metallicity), изучает ли поверхность свет (emission), позиция

в пространстве, расстояние до этой позиции, направление взгляда и др. G-буфер не содержит шума. Таким образом можно производить устранение шума на изображении, содержащим исключительно освещение, но при этом пользуясь отдельными каналами G-буфера для управления фильтрами шумоподавления, но не модифицируя сам G-буфер, и более того, для улучшения качества всё освещение можно также разделить на несколько каналов (прямое, непрямое освещение, глянцевость) и обрабатывать их по отдельности [18]. После преобразований уже к итоговому изображению необходимо перепременить albedo-канал G-буфера.

Для устранения шума необходимо рассматривать алгоритмы, предназначенные специально для *малых объёмов выборки*, нацеленные на *высокую частоту кадров*, при этом быть наиболее *близкими*[5] *к истине*[6] по сравнению с другими, поэтому совместно с консультантом были выбраны следующие алгоритмы для анализа:

- SVGF (2017) [26],
- ASVGF (2018) [23],
- BMFR (2019) [2].

Такие алгоритмы как EAW (2010) [3], использующий функции определения границ и фильтры размытия, но без информации с предыдущих кадров; EDAGI (2017) [28], предназначенный для расчёта только непрямого освещения, и др. уже были сравнены в статьях об SVGF [26] и BMFR [2] и не удовлетворяют заданным критериям. При этом алгоритмы, использующие нейронные сети, например, NTASD (2020) [14], также не будут рассмотрены из-за необходимости обучения таких моделей.

Алгоритм ASVGF [23] (Adaptive spatiotemporal variance-guided filtering) является расширением алгоритма SVGF [26], идея последнего заключается в накоплении вычисленного освещения с прошлых кадров и применении фильтра размытия к участкам с большим количеством шума,

[5]например, используя такую метрику, как среднеквадратичная ошибка (RMSE, Root Mean Square Error)

[6]то есть та же сцена, отрисованная с большим объёмом выборки (например, `4096`) в методе Монте-Карло и неограниченным временем

но не к участкам без или с малым его количеством, и для этого используется оценка дисперсии энергетической яркости в каждом пикселе по сравнению с предыдущим кадром. При размытии шума, границы, например, чётких теней, объектов и других деталей должны быть сохранены, поэтому для этого используется Edge-avoiding À-Trous фильтрация [4], которой подаётся информация о глубине и векторах нормалей из G-буфера, а также об энергетической яркости. Одним из недостатков SVGF является ghosting – появление артефактов из-за использования данных из предыдущих кадров при устранении шума, например, если объект быстро движется слева направо, то слева от объекта всё ещё будет видно некоторые очертания этого объекта, или если источник света резко исчез, то свет от него по-прежнему будет оставаться на некоторое время. Алгоритм ASVGF решает данную проблему, отбрасывая старую информацию, если в участке 3×3 пикселя изменения энергетической яркости относительно времени были слишком велики. Для измерения изменений используется разница энергетической яркости одной и той же точки определенной поверхности между текущим и предыдущим кадром, но так как точки поверхности под одним и тем же пикселем между кадрами могут не совпадать, то необходимо перепроецировать пиксели с предыдущего кадра на текущий; при этом также нужно использовать одни и те же псевдослучайные числа, чтобы старая информация не отбрасывалась, основываясь на изменении случайный чисел. Таким образом, между кадрами есть одинаковые точки поверхности, в которых заново вычисляется функция отрисовки, и разница этих значений, нормализованная относительно некоторой абсолютной энергетической яркости, и используется для отбрасывания неактуальной информации с предыдущих кадров.

BMFR [2] (Blockwise Multi-Order Feature Regression) также оптимизирован для работы с объёмов выборки равным единице, но использует методы регрессионного анализа для устранения шума. Нахождение изображения без шума ставится как задача минимизации суммы квад-

ратов для блока пикселей Ω_i, расположенного около пикселя i:

$$\hat{\alpha} = \underset{\alpha \in \mathbb{R}^M}{\operatorname{argmin}} \sum_{p \in \Omega_i} \left(Z(p) - \sum_{m=1}^{M} \alpha_m T_m(p) \right)^2, \tag{4}$$

где $Z(p)$ – значение в пикселе p изображения с шумом, M – количество признаков, использующихся для минимизации, T – кортеж M определённых каналов G-буфера в некоторой степени. Тогда оценку $\hat{Y}(p)$ значения в пикселе p без шума можно вычислить как:

$$\hat{Y}(p) = \sum_{m=1}^{M} \hat{\alpha}_m T_m(p). \tag{5}$$

Используя QR-разложение, находится решение задачи наименьших квадратов, далее рассчитывается изображение с устраненным шумом, и применяются накопленные данные с предыдущих кадров для увеличения эффективности выборки.

Было также рассмотрено уже готовое решение для устранения шума: NVIDIA Real-Time Denoiser [13] (NRD), который был создан для трассировки путей с малым размером выборки (0.5 и 1 экземплярами на пиксель). NRD имеет большую производительность по сравнению с SVGF, принимает на вход обычные каналы G-буфера и имеет поддержку графического API Vulkan. Но на данный момент (18.04.2021) NRD всё ещё находится в разработке и доступ предоставляется ограниченному количеству заявителей[7]. Получить доступ не удалось.

Для устранения шума был выбран алгоритм ASVGF, так как, несмотря на то, что BMFR в 1.8 раза быстрее [2] SVGF и в общем случае оперирует только на позициях точек поверхностей и их нормалей, он имеет такие недостатки, как появление артефактов в виде блоков на изображении и чрезмерное размытие. Тем не менее, архитектура библиотеки обязана позволять добавление дополнительных алгоритмов устранения шума, так как значительная часть таких алгоритмов принимают на вход G-буфер без шума и результат трассировки путей с одним образ-

[7] https://developer.nvidia.com/nvidia-rt-denoiser-early-access-program

цом на пиксель [19].

2.2. Существующие решения

Рассматриваться будут только те решения, которые используют *исключительно* трассировку путей для расчёта освещения в трехмерных сценах, то есть решения с гибридными методами, использующими для расчёта освещения и растеризацию, и трассировку путей или лучей, не учитываются. Но при этом растеризация может использоваться, например, для генерации данных для G-буфера или отрисовки пользовательского интерфейса, то есть для операций, не связанных с освещением. Решение должно предоставлять высокую частоту смены кадров: по меньшей мере 30 кадров в секунду в разрешении 1280×720 на NVIDIA RTX 2060 (младшей модели серии видеокарт NVIDIA RTX 20), а также использовать алгоритмы устранения шума, так как на текущий момент данный вид отрисовки невозможен без значительного шума на изображении. Разрабатываемая библиотека будет иметь открытый исходный код с лицензией MIT, поэтому рассматриваемые решения тоже должны быть с открытым исходным кодом.

Одним из первых приложений, использовавших трассировку путей для отрисовки (и удовлетворяющую вышеперечисленным требованиям), является Q2VKPT [16] (2019), добавляющий трассировку путей в видеоигру Quake II[8] (1997) и демонстрирующий работу алгоритма ASVGF. Позднее проект был переработан в Quake II RTX [15] (2019), который также использует ASVGF, но при этом устранение шума происходит раздельно: отдельно для прямого, непрямого и зеркального освещений; доработано непрямое освещение с помощью дополнительных карт шероховатостей, металличности и т.д.; улучшены отражения и преломления за счёт Checkerboarded split-frame rendering [31] и др. Несмотря на это, Q2VKPT и Quake II RTX не используют отдельную библиотеку, а утилизируют различные особенности Quake II, например, специфичные для данного приложения структуры трехмерных сцен,

[8] https://en.wikipedia.org/wiki/Quake_II

структуры моделей на сцене, поэтому использование системы отрисовки из Q2VKPT и Quake II RTX в других проектах проблематично.

Библиотеки RTX Direct Illumination [11] (RTXDI) и RTX Global Illumi-nation [12] (RTXGI) на данный момент (18.04.2020) находятся на стадии разработки и доступ даётся ограниченному количеству заявителей, поэтому детали реализации неизвестны. Тем не менее, цели разрабатываемой библиотеки и RTXDI/RTXGI различаются, так как первый более сосредоточен на создании решения, предназначенного специально для приложений с конвейером фиксированной функциональности.

2.3. Используемые технологии

Для взаимодействия с графическим процессором будет использоваться графический API Vulkan, так как он предоставляет необходимые расширения для работы трассировки лучей, но и в отличие от графического API DirectX 12, который также поддерживает трассировку, Vulkan является кросс-платформенным.

Для написания шейдеров используется язык программирования GLSL, который впоследствии компилируется с помощью glslang[9] в промежуточный язык SPIR-V.

Многие приложения, использующие графический конвейер фиксированной функциональности, были в разработке в середине-конце 90-х годов и были написаны на языках программирования Си и C++, поэтому интерфейс библиотеки должен быть совместим с языком Си, при этом сама же библиотека будет реализована на языке C++.

[9] https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/

3. Архитектура

Главным сценарием использования библиотеки RTGL1[10] является загрузка данных о трехмерной сцене и получение готового изображения в графическом окне, предоставленном библиотеке. При этом предполагается, что время отрисовки должно быть минимальным: приблизительно $\frac{1}{60}$ секунды.

3.1. Интерфейс библиотеки

Интерфейс библиотеки представлен в виде единственного заголовочного файла, удовлетворяющего стандарту C99 [7] языка программирования Си, так как большинство приложений, использующих конвейер фиксированной функциональности, реализованы на языках Си и C++.

Для использования библиотеки необходимо создать и инициализировать её экземпляр, в результате получив дескриптор `RgInstance`, который должен быть указан при вызове других функций библиотеки. При инициализации, кроме различных параметров, также передаются платформозависимые данные об графическом окне, в которое и будет производиться отрисовка. Например, на операционной системе Windows, это дескрипторы `HINSTANCE` и `HWND`.

Основная информация о сцене – это трехмерные объекты, находящиеся в ней. В контексте создаваемой библиотеки, *трехмерный объект или геометрия* – это набор треугольников, у вершин которых есть такие атрибуты как позиция, вектор нормали и текстурные координаты, при этом сама геометрия имеет положение, вращение и масштаб на сцене, а также материалы.

Материалом же называются свойства отрисовки геометрии, которые представлены в виде карт (изображений), которые накладываются на поверхность геометрии:

- карта альбедо (цвет поверхности без освещения),

[10]RTGL1 – название создаваемой библиотеки; явл. сокращением от Ray Traced OpenGL1, т.к. графический API OpenGL до версии 2.0 предоставлял исключительно конвейер фиксированной функциональности.

- карта прозрачности (в каких частях поверхность прозрачна/полупрозрачна),
- карта нормалей (смещение векторов нормалей поверхности),
- карта металличности (насколько поверхность является металлом),
- карта шероховатостей (насколько гладкая или шероховатая),
- карта излучения (выделяет ли поверхность свет).

Таким образом, пользователю необходимо передать библиотеке данные о геометрии и их материалах, а также источниках света.

3.1.1. Геометрия

В библиотеке были выделены два основных вида геометрии по способу отрисовки.

- *Трассированная* – такая геометрия отрисовывается с помощью трассировки лучей, а следовательно, участвует в расчёте освещения. Но одним из ограничений является отсутствие полупрозрачности у такой геометрии, то есть поверхность может быть либо непрозраной, либо полностью прозрачной.
- *Растеризационная* же геометрия отрисовывается методом растеризации и не участвует в освещении. Основными причинами её использования могут быть отрисовка полупрозрачных объектов или пользовательского интерфейса.

Трассированная геометрия, в свою очередь, подразделяется на:

- статичную недвижимую: позиции вершин треугольников объекта не меняются от кадра к кадру и сам объект не передвигается по сцене;
- статичную движимую: позиции вершин так же не меняются, но сам объект может изменять свою позицию, вращение и масштаб на сцене;

Рис. 4: Различные виды геометрии на сцене: персонажи – динамическая геометрия, двери – статичная движимая, земля и здания – статичная недвижимая.

- динамическую: и позиции вершин, и состояние объекта на сцене могут меняться.

Пользователю необходимо отправляться динамическую геометрию в библиотеку каждый кадр, статичную же, и движимую, и недвижимую, только в том случае, когда сцена сменяется совершенно другой. При загрузке трассированной геометрии требуется указывать 64-битный уникальный индекс для возможности сопоставления экземпляров геометрии между текущим и предыдущим кадром, что особенно важно для постоянно обновляемой динамической геометрии. При необходимости сменить положение движимой статичной геометрии пользователю необходимо вызвать для этого отдельную функцию, которая при этом недоступна для других типов геометрии.

Трассированная геометрия может быть отрисована с использованием альфа-тестирования, то есть быть полностью непрозрачной или полностью прозрачной в определенных участках поверхности, что определяется картой прозрачности, в которой значение варьируется в пределах `[0, 1]`. Если это значение больше некоторого порогового (по умолчанию равно `0.5`), то считается, что в данной точке поверхность пол-

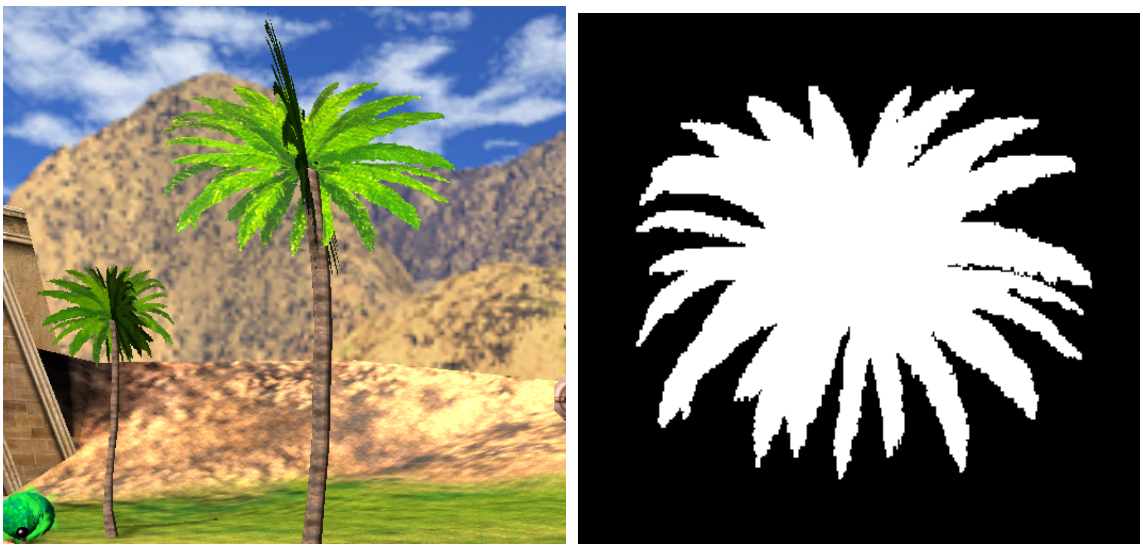


Рис. 5: На левом рисунке показана трассированная геометрия пальмы, у которой листва была отрисована с помощью альфа-тестирования. На рисунке справа показана карта прозрачности, она имеет значение `1.0` в местах с листвой, `0.0` в местах без неё.

ностью непрозрачна, иначе полностью прозрачна (рис. 5).

Растеризованная геометрия должна загружаться каждый кадр вне зависимости от того, является ли она статичной или динамической.

Так как растеризованная геометрия в первую очередь (в контексте библиотеки) предназначена для отрисовки полупрозрачности, то при её загрузке указываются параметры, необходимые для этого: включать ли смешивание (англ. blending), если да, то необходимо указать режимы смешивания. Также доступны для настройки тест глубины (англ. depth test) и запись в буфер глубины (англ. depth write).

При загрузке растеризованной геометрии, есть возможность указать альтернативные настройки камеры, отличающиеся от настроек общей сценовой камеры. Это необходимо, например, для отрисовки пользовательского интерфейса, который скорее всего требует камеру с ортографической проекцией.

Фон – это геометрия, которая отрисовывается позади всей сцены (рис. 6).Библиотека предоставляет несколько способов отрисовки фона:

- монотонный фон (используется лишь один цвет для всего фона);
- кубическая текстура;

- растеризация;
- трассировка лучей.

Для кубической текстуры необходимо указать 6 изображений, которые будут использованы для 6 сторон куба, пересечение луча с которым будет давать цвет фона в данном направлении.

Для использования растеризованного фона, необходимо загружать растеризованную геометрию со специальном флагом `RG_RASTERIZED_GEOMETRY_RENDER_TYPE_SKY`. Для трассированного фона необходимо аналогично указывать флаг `RG_GEOMETRY_VISIBILITY_TYPE_SKYBOX`.

Рис. 6: Небо отрисовано с помощью фоновой геометрии.

3.1.2. Материалы

Библиотека предлагает два способа загрузки карт (изображений) материалов:

- либо пользователь самостоятельно предоставляет указатель на память с данными карты;
- либо указывается путь к файлу изображения, и библиотека загружает этот файл с диска.

Материалы могут изменяться во времени, поэтому выделяется три вида материалов по частоте и типу обновления:

- статичные: не изменяются во времени;

- анимированные: загружаются в виде набора статичных материалов, и, при необходимости смены кадра анимированного материала, пользователь сообщает библиотеке индекс нового кадра;

- динамические: при необходимости обновления библиотеке предоставляются новые данные для карт, которые сменяют старые.

3.1.3. Источники света

Рис. 7: Тестовая сцена без источников света (слева), и с направленным источником света (справа).

Без явных источников света сцена освещается лишь фоном (например, небом) и поверхностями с картами излучения (рис. 7), что в идеальном случае и должно происходить, то есть, например, лампа накаливания должна быть представлена в виде трехмерной модели с очень высоким значением излучения. Но такой способ в условиях ограниченного количества лучей не эффективен по сравнению с явным указанием источников света, так как необходимо дополнительно искать нужное направление к поверхности с высоким излучением. Поэтому библиотека предоставляет следующие типы источников света.

- Направленный: задаётся направлением и угловым размером. Например, это солнце.

- Сферический: задаётся позицией на сцене, радиусом сферы и дистанцией, на которой прекращается влияние источника света. С помощью сферического источника можно аппроксимировать, например, свет от факелов.

3.2. Структура компонент библиотеки

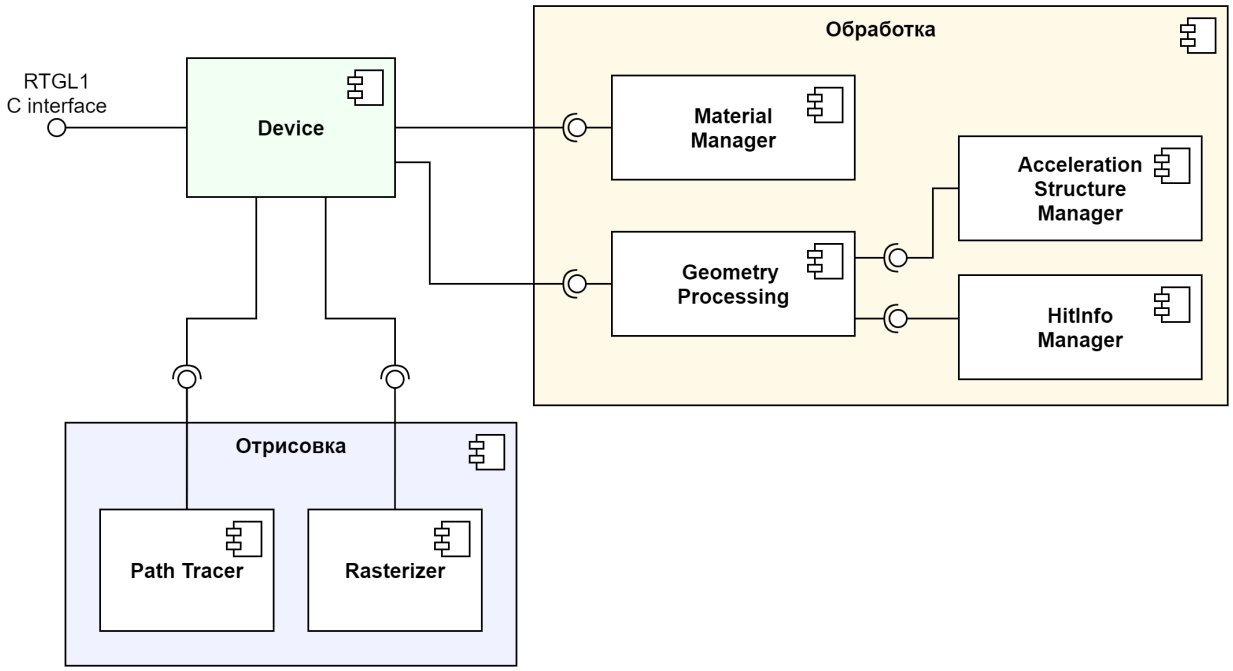


Рис. 8: Диаграмма компонентов внутренней реализации библиотеки.

Главный интерфейс библиотеки реализуется компонентом `Device`, который делегирует другим компонентам работу (рис. 8).

Компоненты по *обработке данных* для отрисовки:

- `Geometry Processing` обрабатывает треугольники на сцене для возможности трассировки лучей;
- `Material Manager` управляет материалами, которые определяют какие свойства должны иметь треугольники.

Компоненты по *отрисовке*:

- `Path Tracer` включает в себя трассировку путей для расчёта реалистичного освещения и устранение шума с полученного трассировкой изображения;
- `Rasterizer` выполняет отрисовку объектов классическим способом растеризации, которая в основном используется для отрисовки полупрозрачности.

В свою очередь, обработка геометрии (`Geometry Processing`) включает в себя:

- `Acceleration Structure Manager`: управление экземплярами специальных структур – *acceleration structure* (сокр. *AS*) –, которые содержат все треугольники сцены в специальном виде, позволяющем эффективно проверять пересечение лучей с ними на аппаратном уровне, то есть с помощью графического процессора;

- `HitInfo Manager`: при найденном пересечении луча и сцены, необходимо найти свойства поверхности пересечённого экземпляра геометрии (объединённого множества треугольников), поэтому требуется управление информацией о каждом таком экземпляре.

4. Реализация библиотеки

Работа библиотеки делится на две части: обработка данных и отрисовка. При этом стоит отметить, что для упрощения разработки были созданы программные инструменты, но которые не являются частью самой библиотеки.

4.1. Обработка данных для отрисовки

4.1.1. Управление экземплярами acceleration structure

Для эффективной трассировки лучей необходимо строить acceleration structure нижнего уровня (англ. bottom level acceleration structure, сокр. BLAS) (рис. 9), передав функции Vulkan, `vkCmdBuildAccelerationStructuresKHR`, указатели на видеопамять, хранящую информацию о вершинах экземпляров геометрии. Для запуска трассировки луча используется acceleration structure верхнего уровня (англ. top level acceleration structure, сокр. TLAS), который содержит в себе набор экземпляров BLAS.

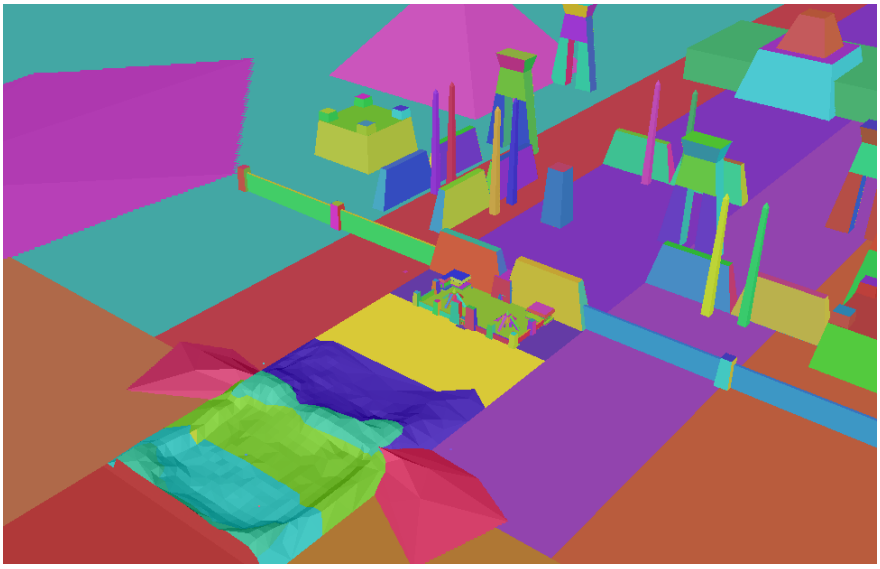


Рис. 9: Acceleration structure нижнего уровня статичной геометрии в графическом отладчике Nvidia Nsight Graphics.

Копирование вершинных данных с оперативной памяти в видеопамять производится через промежуточный буфер (англ. staging buffer),

чтение и запись в который доступны и центральному, и графическому процессорам, благодаря свойству `HOST_COHERENT`[11] памяти такого буфера. Но так как использование графическим процессором такого типа памяти неоптимально, из промежуточного буфера данные далее копируются в финальный буфер со свойством памяти `DEVICE_LOCAL`[12].

Операция построения AS дорогостоящая, поэтому необходимо держать количество их вызовов минимальным в каждом кадре. Для этого было сделано разделение трассированной геометрии на статичную недвижимую, статичную движимую и динамическую, так как такие виды геометрии имеют различную частоту обновления, и каждый вид имеет свой собственный BLAS (рис. 10). Следовательно, перестройка для статичных происходит довольно редко, а для динамических каждый кадр.

Для динамической BLAS указывается флаг `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_KHR`, что сокращает время построения, но эффективность трассировки лучей в таком BLAS может снизиться. Перестройка же BLAS статичной движимой геометрии происходит при изменении состояния (положения, ориентации, масштаба) хотя бы одного экземпляра геометрии такого типа. При этом указывается флаг `VK_BUILD_ACCELERATION_STRUCTURE_ ALLOW_UPDATE_BIT_KHR`, чтобы также ускорить построение, но благодаря тому, что позиции вершин статичной движимой геометрии не меняются со временем.

4.1.2. Информация о пересечении луча

При нахождении пересечении луча со сценой доступна следующая информация:

1. `InstanceId` – индекс экземпляра в AS верхнего уровня;
2. `InstanceCustomIndexKHR` – пользовательский индекс, то есть индекс, который можно задать при создании экземпляра AS верхнего уровня;

[11] полное название: `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
[12] полное название: `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`

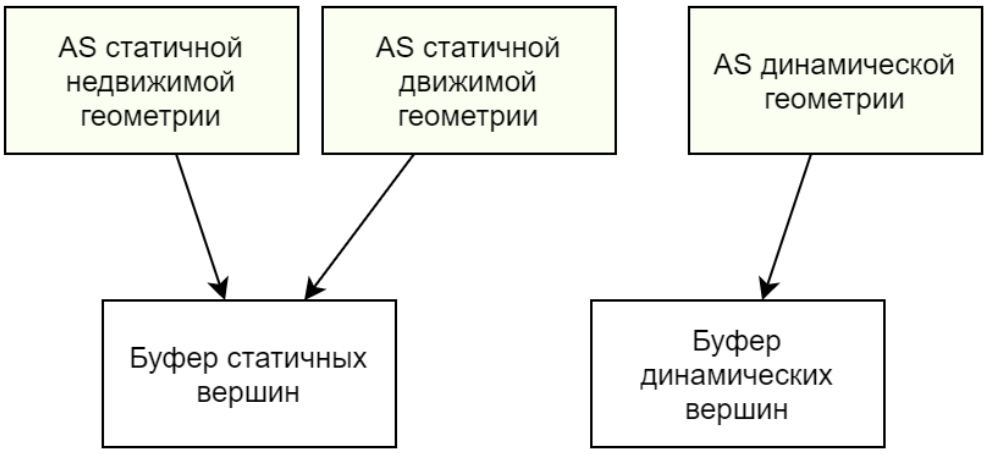


Рис. 10: Каждый тип имеет свой собственный BLAS. При этом вершины для статичных и динамических экземпляров геометрии хранятся в разных буферах так же из-за разной частоты обновления.

3. `RayGeometryIndexKHR` – индекс экземпляра геометрии в AS нижнего уровня;
4. `PrimitiveId` – индекс треугольника в экземпляре геометрии;
5. `RayTmaxKHR` – расстояние от начала луча к точке пересечения;
6. барицентрические координаты треугольника, с которым столкнулся луч.

При этом каждый луч может иметь данные полезной нагрузки (англ. payload), содержание которых задаётся разработчиком. Размер структуры полезной нагрузки необходимо держать минимальным для лучшей производительности. Поэтому в библиотеке, вместо обработки данных о треугольниках при непосредственном нахождении пересечения луча, они упаковываются в структуру полезной нагрузки (листинг 1).

Листинг 1: Определение структуры полезной нагрузки.

```
struct ShPayload
{
    float   baryCoords[2];
    uint    instIdAndIndex;
    uint    geomAndPrimIndex;
```

```
    float   clsHitDistance;
};
```

Таким образом, размер структуры равен 20 байт, благодаря упаковке `InstanceId` и `InstanceCustomIndexKHR` в один `uint`, `RayGeometryIndexKHR` и `PrimitiveId` в один `uint`. При этом пределы значений проверяются библиотекой для безопасной упаковки без потери данных. Была попытка упаковать барицентрические координаты в 4 байта, но возникли видимые артефакты из-за недостаточной точности.

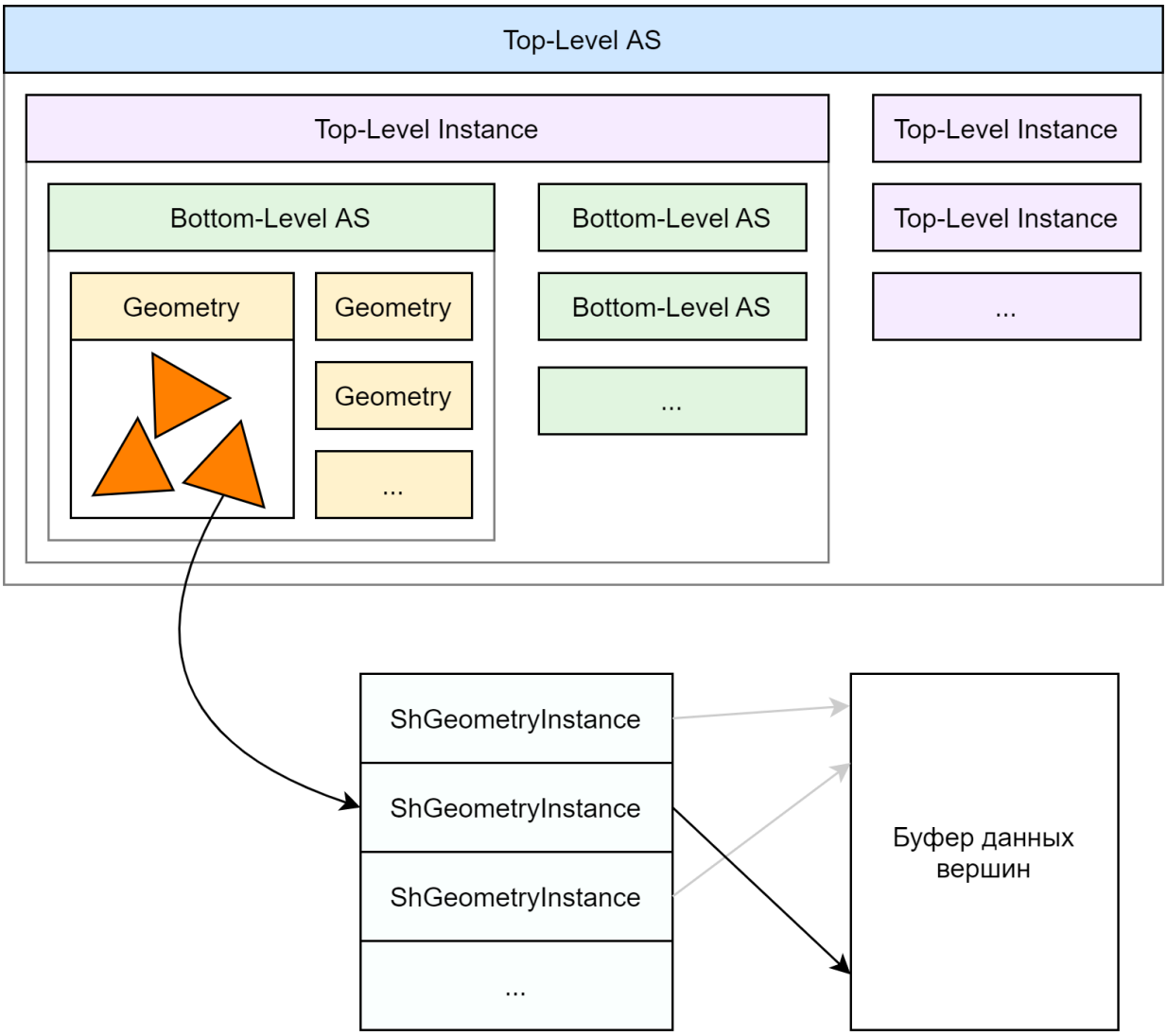


Рис. 11: Для доступа к данным о вершинах и экземпляре геометрии необходимо четыре индекса: индекс треугольника, индекс геометрии в BLAS, индекс экземпляра в TLAS и пользовательский индекс.

После получения данных о пересечении луча и сцены, необходимо

узнать информацию об экземпляре геометрии и атрибутах вершин треугольника для последующего расчёта освещения. Для этого библиотека хранит в видеопамяти массив структур `ShGeometryInstance` (рис. 11), каждая из которых содержит информацию о смещении в буфере вершин для данной геометрии, её материалах и матрицах перехода (англ. transformation matrix). По смещению в буфере вершин и индексу треугольника можно рассчитать индексы вершин треугольника, и используя их, получить атрибуты вершин: позиции, текстурные координаты, вектора нормалей.

Для расчёта освещения также необходимо иметь доступ к состоянию сцены в предыдущем кадре, при этом достаточно хранить только позиции вершин. А так как вершины статичной геометрии не меняются, то копировать данные о позициях необходимо только для динамической геометрии. Но матрицы перехода предыдущего кадра должны быть доступны и для динамической, и для статичной движимой геометрии.

При этом необходима возможность сопоставлять `ShGeometryInstance` текущего и предыдущего кадра, так как смещение одного и того же экземпляра геометрии в массиве `ShGeometryInstance` может различаться от кадра к кадру, ведь скорее всего пользователь загружает данные о геометрии в неопределенном порядке. Для этого библиотека принимает 64-битные уникальные идентификаторы при загрузке геометрии, которые используются для соотношения смещений с помощью хэш-таблицы, ключом в которой является уникальный идентификатор, а значением структура, содержащая предыдущую матрицу перехода, предыдущее смещение в массиве `ShGeometryInstance` и предыдущее смещение в буфере вершин.

4.1.3. Система материалов

Так как при трассировке лучей в сцене пересечение может произойти с любым из объектов, то необходимо иметь доступ ко всем материалам на сцене, то есть все изображения всех материалов должны быть доступны в шейдерах через дескрипторы. Для этого используется расширение `VK_EXT_descriptor_indexing` для Vulkan, которое позволяет

создавать массивы дескрипторов и выбирать нужный дескриптор по индексу в них. При этом карты материала скомпонованы в 3 отдельных изображения: альбедо и прозрачность; нормали и металличность; излучение и шероховатость. Следовательно, материал представляется в виде всего лишь трёх индексов, так как каждое изображение может быть индексировано в массиве дескрипторов.

Для управления видеопамятью под изображения используется библиотека Vulkan Memory Allocator (VMA) [27], так как количество отдельных аллокаций видеопамяти, существующих одновременно, по спецификации Vulkan ограничено 4096. А так как количество одновременно загруженных изображений может быть достаточно велико (порядка тысяч), то есть возможность выйти за предел при наивном алгоритме выделения памяти (по одной аллокации на изображение). VMA решает данную проблему, поддерживая сразу по множеству изображений в одной и той же аллокации.

В библиотеке, анимированные и динамические материалы используют изображения, меняющиеся со временем, в отличие от статических материалов. Поэтому их загрузка происходит специальным образом.

- Анимированные, в сущности, являются массивом статических материалов (в данном контексте, они называются кадрами анимации), один из которых является активным и используется в сцене. Пользователь библиотеки указывает, когда стоит сменить индекс текущего активного кадра, и в этом случае библиотека оповещает через шаблон "наблюдатель" все экземпляры геометрии, который используют данный материал, о том, что необходимо сменить активный кадр. При этом оповещается только статичная геометрия, так как динамическая геометрия обновляется постоянно, то есть у таких экземпляров всегда актуальная информация. Обновление анимированного материала у экземпляра геометрии происходит с помощью записи в соответствующую структуру `ShGeometryInstance`.

- У динамических материалов данные в видеопамяти задаются поль-

зователем библиотеки, поэтому необходимо постоянно держать в готовности промежуточные буферы со свойством памяти `HOST_COHERENT`, в которые будут копироваться данные пользователя, а впоследствии из этих буферов в видеопамять с `DEVICE_LOCAL` свойством через функцию `vkCmdCopyBufferToImage`.

4.2. Инструменты разработки

Для использования графического процессора необходимо отправлять информацию из оперативной памяти в видеопамять. При использовании структур, которые используются и в коде библиотеки (C++) на центральном процессоре, и в шейдерах (язык шейдеров GLSL) на графическом процессоре, необходимо поддерживать *консистентность определения структур*, так как выравнивание в них и размер типов могут различаться при использовании в C++ и GLSL.

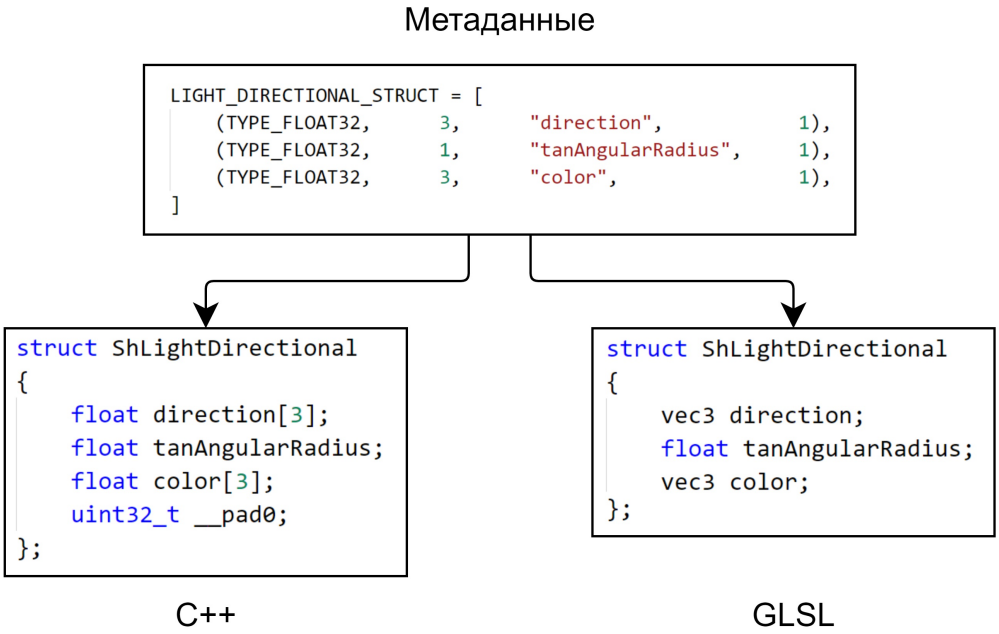


Рис. 12: Пример метаданных, подаваемых в `GenerateShaderCommon` и результаты генерации: структура на языках C++ и GLSL. Из-за выравнивания размер структуры на GLSL неявно на 4 байта больше, и, для эквивалентности, в C++ нужно явно указать дополнительное 4-байтное поле.

Для этого был реализована программа `GenerateShaderCommon` на

языке Python, которая по метаданным генерирует эквивалентные определения структур для C++ и GLSL (рис. 12), а также определения констант. И помещает сгенерированные данные в отдельные заголовочные файлы для каждого языка, которые уже могут быть включены в программы на соответствующих языках.

Большинство буферов, располагающихся в видеопамяти и используемых в библиотеке, являются `storage` буферами, которые используют базовое[13] выравнивание. Поэтому было реализовано автоматическое выравнивание только в такого рода структурах.

Буферы кадра – это изображения, в которые сохраняются данные во время отрисовки. Во время разработки их количество и свойства часто меняются, поэтому необходим инструмент, который бы генерировал определения для их инициализации с помощью Vulkan и для чтения и записи в них в шейдерах на языке GLSL. Поэтому `GenerateShaderCommon` также генерирует такие определения по указанным форматам изображения и опциональным флагам (размер буфера кадра, можно ли использовать буфер кадра при растеризации).

[13]англ. base alignment https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/chap15.html#interfaces-resources-layout

4.3. Отрисовка

После обработки всех данных кадра библиотека начинает непосредственную отрисовку. Далее приведён полный алгоритм отрисовки, в результате которого пользователь получает изображение в графическом окне.

1. Если тип фона – это растеризованная геометрия:
   (a) рисовать геометрию фона в кубическую текстуру;
   (b) рисовать геометрию фона в альбедо буфер кадра.
2. Запустить первичные лучи в сцену (рис. 13, луч 1).
3. Запустить лучи для прямого освещения (рис. 13, лучи 2).
4. Запустить лучи для непрямого освещения (рис. 13, лучи 3 и 4).
5. Устранить шум для освещения.
6. Скомбинировать финальное изображение.
7. Рисовать растеризованную геометрию в финальное изображение.
8. Скопировать финальное изображение в графическое окно.

Логика по отрисовке реализована в виде шейдеров для возможности исполнения на графическом процессоре. Центральный процессор же с помощью Vulkan API отправляет графическому процессору только команды запуска работы.

Для запуска трассировки лучей с аппаратной поддержкой необходимо вызвать функцию Vulkan API `vkCmdTraceRaysKHR`, которая запросит активацию шейдера генерации лучей (англ. ray generation shader) на графическом процессоре. С помощью таких шейдеров и производится запуск лучей для пунктов 2–4. Если луч пересекается с какой-либо геометрией, вызывается шейдер ближайшего столкновения (англ. closest hit shader). Шейдер любого столкновения (англ. any hit shader) вызывается во всех потенциальных точках столкновения (но если только

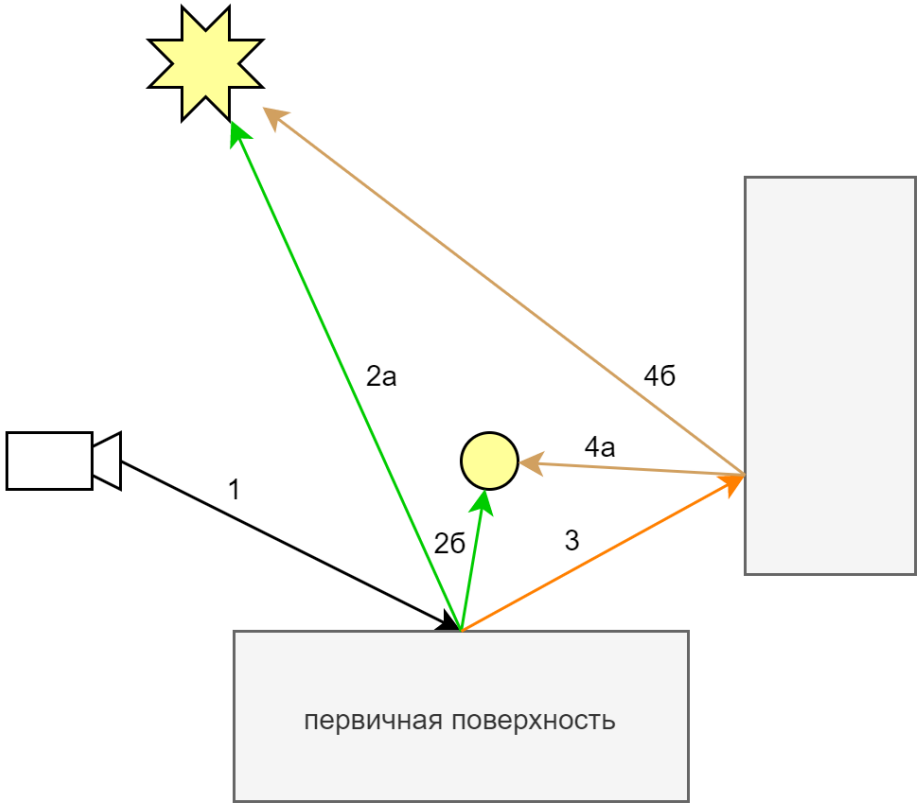


Рис. 13: Путь луча в сцене. 1 – первичный луч. 2 – проверка находится ли точка в тени от направленного (2а) и сферического (2б) источника света. 3 – лучи непрямого рассеянного и зеркального освещения. 4а и 4б – аналогичная проверка затенения в точке пересечения лучей 3 и сцены.

геометрия была помечена специальным флагом для этого). При отсутствии пересечения вызывается шейдер промаха (англ. miss shader).

4.3.1. Первичные лучи

Первичными лучами называются лучи, запускаемые непосредственно из камеры. При нахождении пересечения первичного луча и сцены в буферах кадра сохраняются данные об альбедо (цвете поверхности без освещения), направлении нормали поверхности в этой точке, насколько поверхность металлическая и шероховатая, глубина (расстояние от камеры до точки пересечения), вектор движения (как по сравнению с предыдущим кадром сместилась данная точка поверхности в пространстве экрана). Данная информация необходима для последующего расчёта освещения.

Если для геометрии необходимо альфа-тестирование, то для неё вызывается шейдер любого столкновения, в котором из материала данной геометрии извлекается значение прозрачности и, если оно меньше порогового, то столкновение игнорируется с помощью ключевого слова `ignoreIntersectionEXT`, поэтому шейдер ближайшего столкновения не вызывается.

Для предотвращения алиасинга изображений, наложенных на поверхности используется метод лучевых дифференциалов [8] (рис. 14). При этом в алгоритме предлагается трассировать два дополнительных луча, но так как эта операция довольно дорогостоящая, то вместо этого проверяется пересечение этих лучей с тем же самым треугольником, который был получен пересечением первичного луча. Результат может оказаться не совсем корректным для случаев с большой плотностью треугольников, но несмотря на это, достигается экономия запусков лучей в сцену. Для нахождения пересечения луча и треугольника используется алгоритм Моллера — Трумбора [10].

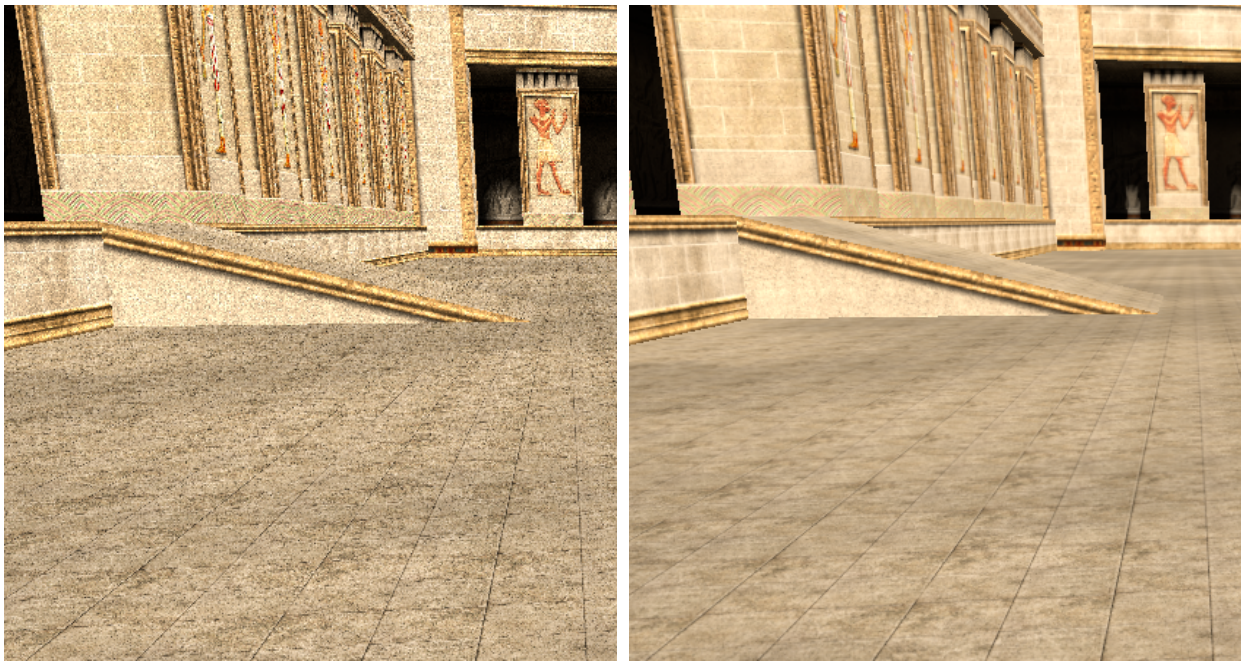
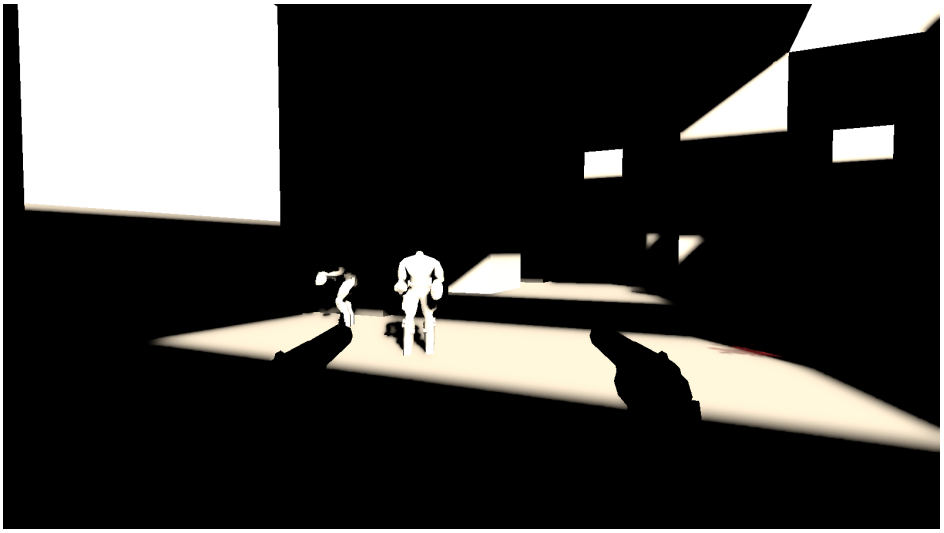


Рис. 14: Использование лучевых дифференциалов: слева – без, справа – с.

4.3.2. Расчёт освещения

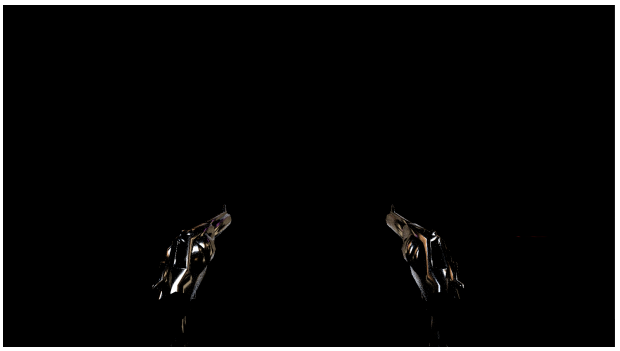
Расчёт освещения разделён на три канала.

- Прямое рассеянное (рис. 15a) – освещение от явных источников света, при этом все поверхности рассматриваются как абсолютно матовые.
- Непрямое рассеянное (рис. 15b) – освещение, вызванное от неявных источников света, такими как свет от фона (например, небо), свет, отраженный от стены, на которую падает солнечный свет, свет от карт излучения поверхностей и другие. Поверхности также предполагаются абсолютно матовыми.
- Зеркальное (рис. 15c) – это отражения на поверхностях, которые



(a) Прямое рассеянное

(b) Непрямое рассеянное



(c) Зеркальное

Рис. 15: Каналы освещения с устраненным шумом.

зависят от их степени шероховатости: значение 0.0 означает абсолютно зеркальное отражение.

Поведение именно этих типов освещения значительно различается в пространстве экрана от кадра к кадру в одних и тех же условиях сцены. Зеркальное освещение зависит от направления камеры, поэтому оно может существенно смениться при небольшом повороте камеры, в отличие от рассеянного. При этом непрямое рассеянное освещение разреженное, а прямое рассеянное – высококонтрастное. Такое разделение необходимо для получения более качественных результатов алгоритмов устранения шума [1].

4.3.3. Прямое освещение

После завершения нахождения точек пересечения первичных лучей со сценой, в них происходит расчёт прямого освещения, то есть как точка освещена явными источниками света.

Так как количество запускаемых лучей в сцене должно быть минимальным, то для прямого освещения используется только 2 луча на пиксель: первый – для проверки: находится ли точка в тени от направленного источника света, второй – от сферического.

На сцене может находится большое количество источников света, порядка сотен, поэтому выбор только двух из них является сложной проблемой, ведь при плохой выборке качество изображения может значительно ухудшиться из-за недостатка информации. Например, был выбран сферический источник света, находящийся в здании на другом конце сцены от камеры, и проверка такого луча, скорее всего, будет отрицательной, в итоге, бюджет лучей на пиксель был израсходован и никакой ценной информации получено не было.

В текущей реализации выбор источников света происходит с помощью равномерного распределения, что плохо сказывается на сценах с большим количеством источников: при движении камеры на конечном изображении заметны артефакты в виде чёрных пятен из-за отсутствия информации об освещённости.

После выбора двух источников, для каждого из них рассчитывается прямое рассеянное освещение с помощью модели отражения Ламберта [20] для абсолютно матовых поверхностей, и зеркальное отражение явного источника света с помощью модели микрограней Smith-GGX [21], используя на входе степень шероховатости поверхности в данном пикселе, её нормаль, направление первичного луча и направление к источнику света. Эти модели аппроксимируют значение двулучевой функции отражательной способности f и, таким образом, есть возможность рассчитать подынтегральную часть уравнения отрисовки (ур. 1) при l равным направлению к источнику света и L_i – цвету источника света.

Стоит также отметить, что l не всегда направлена ровно к позиции источника света, а добавляется небольшое смещение. Так как, например, солнце на небе является не бесконечно малой точкой, а диском с некоторым угловым размером, вследствие чего тени имеют нечёткие границы.

Получив значения рассеянного и зеркального освещения от явных источников света в каждом пикселе, они записываются в буферы кадра `UnfilteredDirect` и `UnfilteredSpecular`, соответственно.

4.3.4. Непрямое освещение

В тех же точках пересечения первичных лучей со сценой рассчитывается и непрямое освещение: непрямое рассеянное и зеркальное. Для этого выделяется по три луча на каждый из них: для нахождения точки, в которую попал луч отскока от первичной поверхности, и два луча для проверки затенения от двух источников света.

В отличие от прямого освещения, где направления l для уравнения отрисовки даны явно, в непрямом направление l может любым на полусфере, ориентированной вдоль нормали поверхности. Однако выбирать необходимо тщательно, так как, например, при выборе непригодного направления значение подынтегрального выражения в уравнении отрисовки становится близким к нулю, что не даёт никакой полезной информации. Такое может случиться, если значение двулучевой функции

отражательной способности f для такого направления близко к нулю, либо нормаль поверхности и выбранное направление почти перпендикулярны друг другу.

Во избежание таких случаев, для зеркальных отражений был использован алгоритм VNDF [6], который выдаёт нормаль микрограни поверхности. Отразив направление камеры относительно такой нормали, можно получить необходимое направление l, при котором подынтегральная часть будет иметь незначительный вклад с довольно малой вероятностью.

При расчёте непрямого рассеянного, для двулучевой функции отражательной способности f используется модель Ламберта, которая, в сущности, возвращает константу. То есть единственный источник обнуления подынтегрального выражения – это модуль скалярного произведения векторов n и l. Поэтому направление l берётся из косинус-взвешенной полусферы (агнл. cosine-weighted hemisphere) [5], ориентированной вдоль нормали n.

После выбора направления l, для непрямого рассеянного и зеркального освещения в сцену запускаются лучи (рис. 13, луч 3), и в точке пересечения рассчитывается прямое рассеянное освещение (рис. 13, луч 4а и 4б). В целях оптимизации глубина рекурсии таких запусков равна единице. Таким образом, оценивается значение L_i в силу уравнения 2. Следовательно, имея все множители, возможно вычислить подынтегральное выражение; обозначим его как L_s.

Значение L_s зеркального освещения в пикселе прибавляется к уже существующему значению в буфере кадра `UnfilteredSpecular`.

Для непрямого рассеянного, значение L_s в совокупности с нормалью поверхности n проецируются на ортонормированный базис сферических функций (англ. spherical harmonics) [29] [17] и коэффициенты полученной сферической функции сохраняются в буфер кадра `framebufUnfilteredIndirectSH_*`. Сферические функции в данном случае используются для эффективного представления сферы вокруг точки пересечения, которая впоследствии используется для расчёта облу-

чённости $E(n)$ относительно нормали n:

$$E(n) = \int_{w \in \Omega} L(w)|n \cdot w| \, dw, \tag{6}$$

где Ω – полусфера направлений, ориентированная вдоль n, $L(w)$ – значение энергетической яркости в направлении w.

Для библиотеки выбраны сферические функции со степенью $l = 1$, так как это позволяет использовать всего 4 коэффициента. Для $l = 1$ алгоритмы перевода сферической функции в облученность $E(n)$ и облученности $E(n)$ в сферическую функцию уже рассчитаны [17] и, в сущности, тривиальны.

Энергетическая яркость в библиотеке представлена вещественным трёхмерным вектором, компонентами в которой являются цвета: красный, зелёный, синий. Поэтому буфер кадра для непрямого рассеянного освещения должен хранить 12 коэффициентов: по 4 на каждую компоненту цвета. А так как один буфер кадра позволяет сохранять до 4 значений переменных типа `float`, то `framebufUnfilteredIndirectSH_*` в реализации представлен в виде трех буферов кадра.

В итоге, такое представление сферы вокруг точки пересечения позволяет эффективно рассчитывать облученность $E(n)$, а также эффективно аккумулировать непрямое рассеянное освещение.

4.3.5. Устранение шума

Из-за очень ограниченного количества лучей буферы кадра освещения после трассировки путей содержат шум, который препятствует восприятию изображения (рис. 16).

Выбранный алгоритм устранения шума A-SVGF делится на два этапа:

1. расчёт образцов для оценки изменений (gradient samples) [23];
2. аккумуляция и фильтрация буферов кадра освещения [26].

Для алгоритма необходимо хранить данные и текущего, и предыдущего кадра: для этого созданы дополнительные буферы кадра. Это

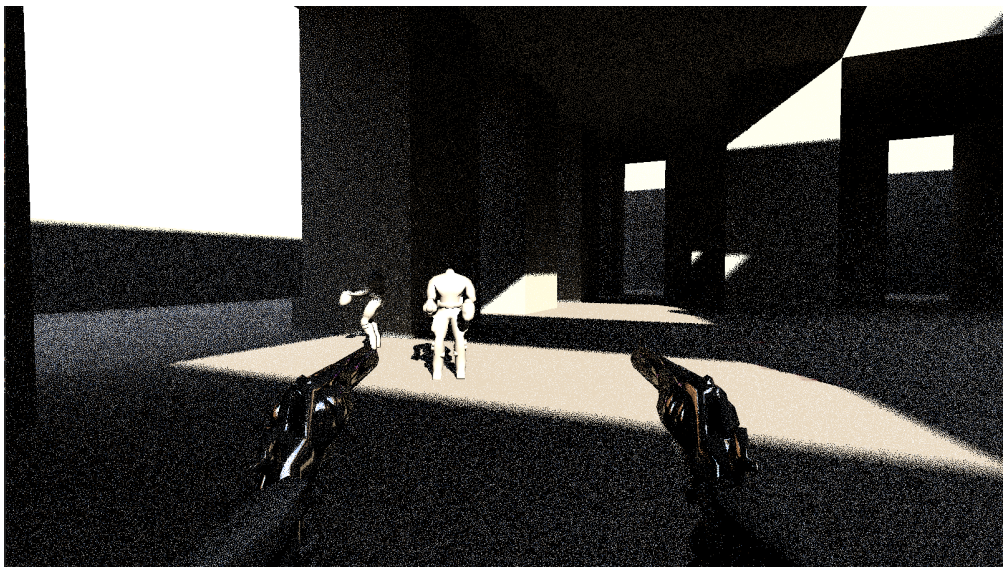


Рис. 16: Объединённые буферы кадра освещения без устранённого шума.

необходимо для повторное использования результатов предыдущего кадра, которые перепроецируются на текущий с помощью векторов движения. При этом проводятся различные проверки консистентности: является ли перепроецированный экземпляр предыдущего кадра совместимым с текущим.

Далее приведены основные отличия реализации в библиотеке от реализации, предлагаемой авторами.

- Устранение шума производится по-отдельности для трёх видов освещения, а не для всех совместно (рис. 17). Это увеличевает качество конечного изображения, так как алгоритм работает на более однородных данных.

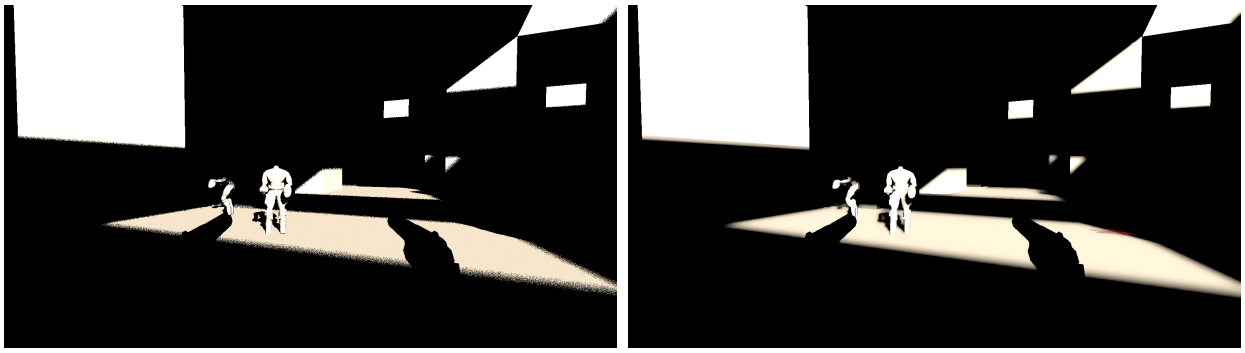
- Значительно уменьшен размер ядра фильтра A-trous [4] для повышения производительности.

- Существенно упрощён алгоритм фильтрации для образцов оценки изменений, опять же, для повышения производительности.

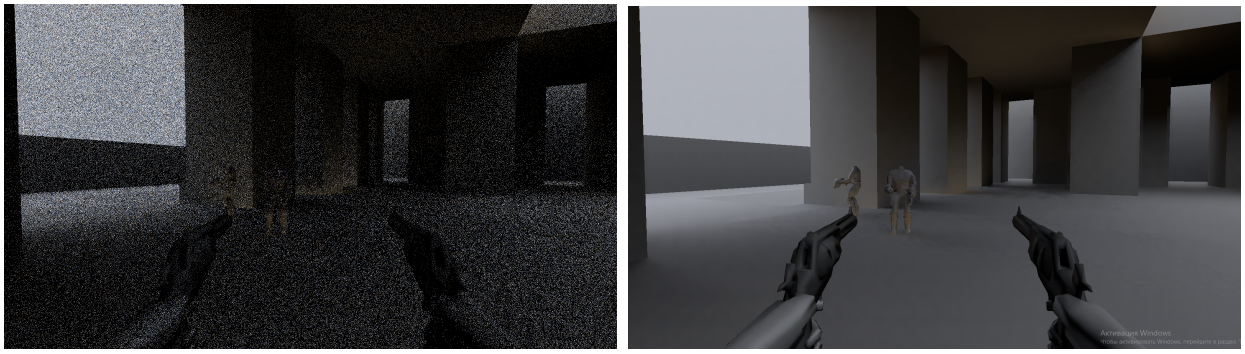
Для оптимального качества, необходимы возможность менять параметры и менять частей алгоритма во время исполнения для ускорения разработки. Для смены параметров используется буфер переменных,

заполняемый данными из структуры `RgDrawFrameInfo`, находящийся в главном интерфейсе библиотеки, то есть пользователи имеют доступ к указанным в этой структуре параметрам для более точной настройки.

Для смены же частей алгоритма для начала необходима перекомпиляция шейдеров, которая заново генерирует файлы с кодом на промежуточном языке SPIR-V. И, если шейдер используется в процессе отрисовки, то необходимо перезагрузить файл шейдера, и пересоздать конвейеры отрисовки (`VkPipeline`), использующие данный шейдер. Для этого используется шаблон "наблюдатель", который оповещает все клас-



(a) Прямое рассеянное



(b) Непрямое рассеянное

(c) Зеркальное

Рис. 17: Каналы освещения с (слева) и без (справа) устранения шума.

сы с конвейерами отрисовки, что необходимо пересоздать их.

4.3.6. Полупрозрачность

Первая реализация для отрисовки полупрозрачных поверхностей полагалась на трассировку лучей. Использование шейдеров ближайшего столкновения излишне затратно: если на пути луча стоит множество полупрозрачных объектов, то при ближайшем столкновении необходимо запустить ещё один луч, который попадет в следующий объект и так далее. Во избежание запуска неопределенного количества однотипных лучей, использовались шейдеры любого столкновения, и тогда достаточно запустить только один луч. Но при таком подходе из-за того, что последовательность вызовов шейдеров любого столкновения не определена, конечное изображение становится нестабильным (рис. 18).

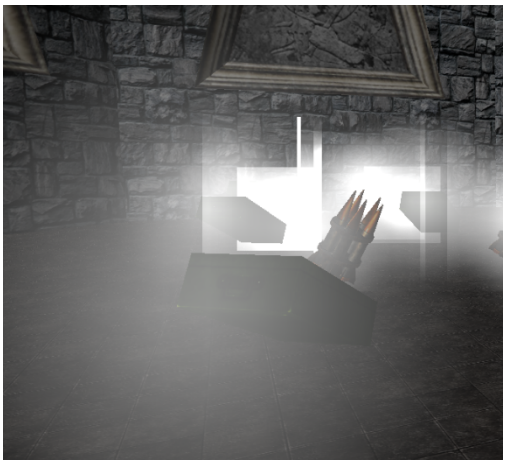


Рис. 18: Неудачная попытка отрисовки полупрозрачности с помощью шейдеров любого столкновения.

Поэтому вместо отрисовки с помощью трассировки лучей полупрозрачная геометрия отрисовывается растеризацией. Для этого, после комбинирования, финальный буфер кадра преобразовывается в изображение с типом `COLOR_ATTACHMENT`[14], чтобы в него можно было записывать в процессе растеризации.

Далее, трассированная геометрия должна загораживать растеризованную (чтобы объекты, рисующиеся вдалеке, заслонялись более близ-

[14]полное название: `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`

Рис. 19: Отрисовка полупрозрачной растеризованной геометрии (вспышки, зелёные и синие лазеры) вместе с трассированной геометрией. Стоит заметить, что одна из растеризованных вспышек загораживается трассированной геометрией благодаря копированию буфера глубины.

кими объектами), поэтому необходимо подать буфер глубины с типом `DEPTH_STENCIL_ATTACHMENT`[15] при начале отрисовке с помощью растеризации. Для этого можно повторно использовать буфер кадра глубины, сгенерированный при трассировке лучей.

Но возникает проблема: в буфер изображения с типом `DEPTH_STENCIL_ATTACHMENT` невозможно копировать данные напрямую (например, с помощью `vkCmdCopyBufferToImage`). Тем не менее, в буферы глубины возможно делать запись с помощью фрагментных (пиксельных) шейдеров в процессе растеризации. Поэтому создан фрагментный шейдер (листинг 2), который считывает данные из трассированного буфера глубины (`framebufDepth`) и записывает в буфер глубины данного пикселя (`gl_FragDepth`).

Листинг 2: Основная часть фрагметного шейдера для копирования в буфер глубины.

```
gl_FragDepth = texelFetch(framebufDepth, pix, 0).w;
```

Таким образом, при отрисовке геометрии при растеризации данные о глубине будут актуальными между трассированной и растеризованной геометрией. И, так как растеризация используется для полупрозрачных объектов, то возможно использовать встроенные функции смешивания, не реализовывая их заново самостоятельно (рис. 19).

[15] полное название: `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`

4.3.7. Фон

Если луч не попадает в геометрию сцены, то возникает ситуация промаха и необходимо получить цвет фона в направлении луча.

Вариант с трассированным фоном является самым низкопроизводительным, так как для этого строится дополнительная "фоновая" сцена и используется дорогостоящая операция трассировки лучей в этой фоновой сцене.

Поэтому в добавление был реализован более оптимальный вариант – растеризационный фон: библиотека сначала отрисовывает фоновую геометрию в альбедо буфер кадра и при трассировке первичных лучей в ситуации промаха альбедо буфер кадра оставляется неизменным, так как он уже содержит данные о фоне. Затем генерируется кубическая текстура, отрисовывая фоновую геометрию с 6 сторон. Вместо последовательной отрисовки каждой стороны по отдельности, используются расширения `VK_KHR_multiview` для Vulkan API и `GL_EXT_multiview` для GLSL, позволяющие отрисовывать сразу в несколько буферов кадра параллельно. Перед отрисовкой для каждой стороны рассчитываются матрицы вида, направленные в разные стороны и матрица проекции с углом обзора 90 градусов, чтобы границы сторон соединялись бесшовно. К этим матрицам впоследствии обращаются вершинные шейдеры, через переменную `gl_ViewIndex`, указывающую индекс отрисовываемого вида.

Такая кубическая текстура, сгенерированная "на лету", используется для непрямого освещения и отражений, так как информации, которая находится в буфере кадра недостаточно: она представляет только одну конкретную сторону, а необходимо всё окружение, что и позволяет кубическая текстура.

Более того, растеризованный метод позволяет отрисовывать в фон полупрозрачные объекты, такие как облака, то есть с таким вариантом фона возможно отрисовывать динамичные полупрозрачные объекты фона в отражениях и непрямом освещении.

5. Апробация библиотеки

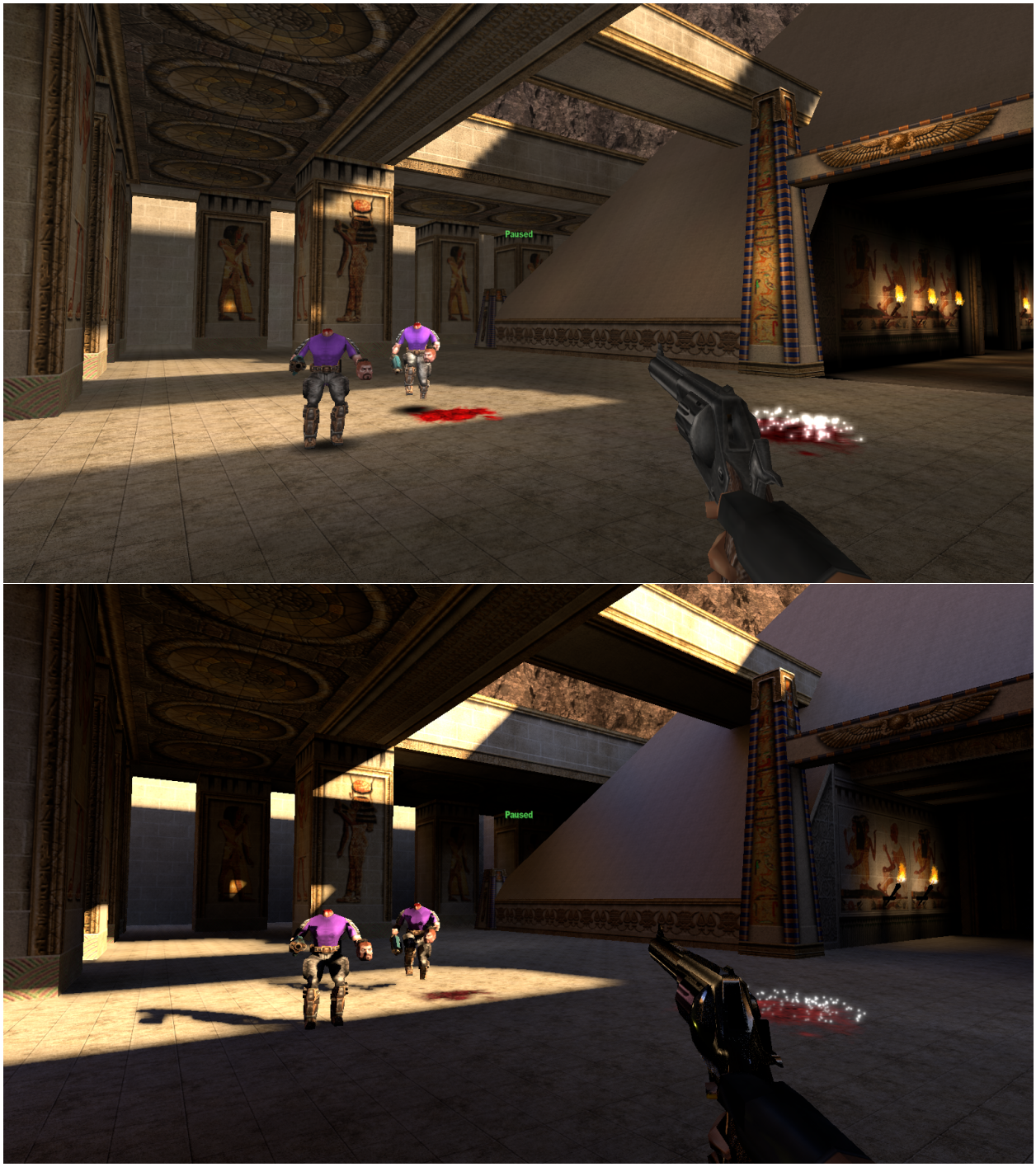


Рис. 20: Отрисованный кадр из видеоигры Serious Sam: The First Encounter. Сверху – исходный алгоритм отрисовки с помощью растеризации. Снизу – использование библиотеки для отрисовки.

В качестве приложения для демонстрации работы библиотеки выбрана видеоигра Serious Sam: The First Encounter [25] (2001) (рис. 20),

которая имеет открытый[16] исходный код в виде фреймворка Serious Engine [24], использует конвейер фиксированной функциональности для отрисовки трёхмерных сцен с высокой частотой кадров, то есть удовлетворяет условиям целевых приложений созданной библиотеки.

Для передачи информации между библиотекой и Serious Engine необходимо реализовать адаптер.

5.1. Геометрия

Для подачи библиотеке данных о сцене необходимо выделить статичную, динамическую и растеризационную геометрию.

В Serious Engine вся геометрия делится на 2 вида: браш (brush), представляющий собой геометрию, вершинные данные которой не меняется во времени, и модель (model), данные которой меняются от кадра к кадру. При этом они всегда прикрепляются к некоторой сущности (entity), которая имеет позицию, ориентацию и масштаб. Каждая сущность имеет либо один браш, либо одну модель. Сущности имеют различные флаги, один из них – `EPF_MOVABLE` – указывает, что прикреплённый браш является перемещаемым, то есть может изменять свою позицию и ориентацию на сцене. Следовательно, в терминах библиотеки модели – динамическая геометрия, а браши – это статичная, движимая или недвижимая – в зависимости от флага `EPF_MOVABLE`.

Модели также могут иметь некоторое множество других моделей, прикрепленных (attachments) к данной. При подаче библиотеке, все прикрепленные модели будут совмещены в один экземпляр динамической геометрии для уменьшения количества экземпляров для обработки библиотекой.

Браши и модели могут иметь различные материалы для своих треугольников, библиотека же требует, чтобы экземпляр геометрии имел один и тот же материал для всех своих треугольников. Поэтому перед отправкой браши и модели делятся по материалам на части, что вызывает проблему с уникальными идентификаторами геометрии: части

[16] под лицензией GNU GPL v2

имеют одно и то же значение – идентификатор сущности, к которой они прикреплены. Для решения каждой части присваивается уникальный 32-битный индекс, который впоследствии битовым сдвигом совмещается с 32-битным идентификатором сущности, образуя 64-битный уникальный идентификатор геометрии.

5.2. Логика отрисовки

Так как исходный алгоритм отрисовки в Serious Engine был разработан для метода растеризации, то необходимо переработать логику отрисовки и реализовать в виде адаптера между Serious Engine и библиотекой.

Во-первых, отправка статичной геометрии библиотеке должна происходить единожды, и только при смене сцены; при растеризации же такая геометрия отправляется каждый кадр. Во-вторых, оптимизации для растеризации, наподобие отбрасывания геометрии, не попавшей в пирамиду видимости камеры (frustum culling), необходимо отключить, так как при трассировке путей объекты позади камеры могут быть пересечены с лучом, например, при отражении.

Далее, при растеризации Serious Engine отрисовывает предпросчитанные карты теней поверх статичной геометрии и вычисляет повершинное освещение для динамической геометрии. Но так как при использовании библиотеки для расчета освещения применяется трассировка путей, то карты теней и повершинное освещение игнорируются.

Фон в Serious Engine представлен в виде трехмерной геометрии, поэтому для его отрисовки используется растеризованный фон.

Любая полупрозрачная геометрия отправляется библиотеке в виде растеризованной геометрии с соответствующими настройками смешивания.

Пользовательский интерфейс, показываемый с помощью Serious Engine, является растеризационной геометрией, так как он не участвует в обсчёте освещения с помощью трассировки путей.

Для взаимодействия с графическими API OpenGL 1.1 и DirectX 8, Serious Engine использует интерфейс в виде набора указателей на функ-

ции, которые с свою очередь вызывают функции соответствующего API. Объявления данных функций находятся в файле `Gfx_wrapper.h`. При использовании адаптера большинство этих функции имеют пустое тело, так как логика отрисовки использует для этого адаптер, которому нужна информация более высокого уровня, например, о типе геометрии.

В Serious Engine сущности на сцене используют ссылки на экземпляры класса `CTextureData`, который содержит логику по работе с изображениями. При этом этот класс напрямую использует множество вызовов функций из `Gfx_wrapper.h`, что значительно затрудняет повторное использование логики. Поэтому вне класса `CTextureData` были созданы копии его главных функций, и уже они были модифицированы для работы с адаптером. Копии функций вызываются из `CTextureData`, если включена отрисовка с помощью библиотеки, иначе используется исходная логика. Адаптер, принимая обработанные таким образом изображения, уже отправляет их в библиотеку.

`CTextureData` предоставляет три вида изображений: "неменяющаяся", "анимированная" и "эффектная". Они загружаются в библиотеку как статичный, анимированный и динамический материал, соответственно. Видеоигра Serious Sam: The First Encounter, разрабатывавшаяся в конце 90-х, не имеет карт нормалей, металличности, шероховатостей и излучения, вследствие чего детальность получаемого трассировкой путей изображения ниже, чем если бы данные карты присутствовали.

На рис. 21 показана диаграмма последовательности обработки сцены Serious Engine, и функция `ProcessWorld` вызывается каждый кадр. На диаграмме рассмотрен случай, при котором статичная геометрия уже была отправлена в библиотеку, поэтому достаточно обновить данные о расположении статичных движимых объектов (`rgUpdateGeometry Transform`), обработать динамическую геометрию, то есть обновить их вершины, загрузить в библиотеку материалы (`rgUploadStaticMaterial`[17]),

[17]для краткости указаны только статичные материалы, в действительности могут быть и анимированные, и динамические

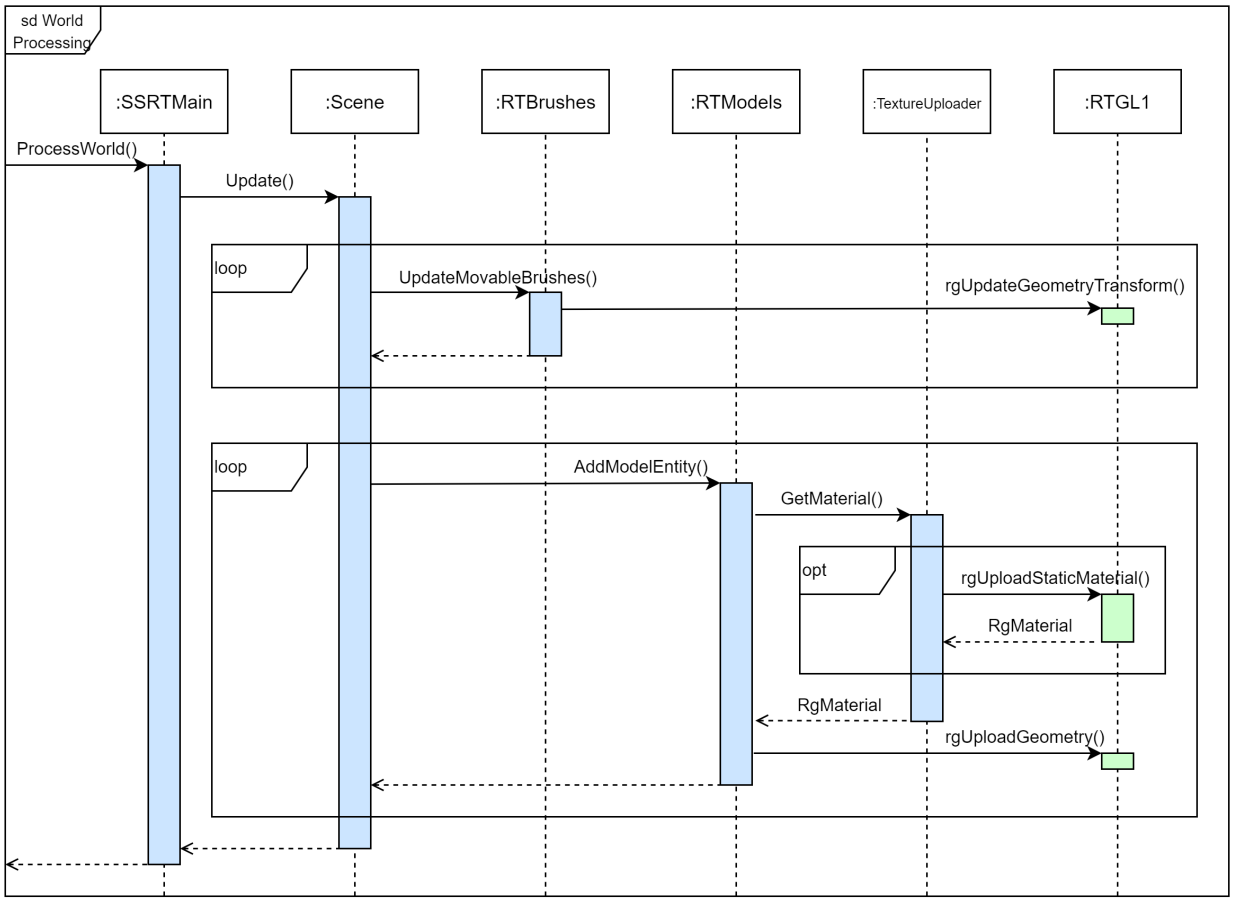


Рис. 21: Диаграмма последовательности обработки сцены Serious Engine.

если они всё ещё не были загружены для данной геометрии, и отправить новые данные в библиотеку (`rgUploadGemetry`).

После обработки, в конце кадра, вызывается функция библиотеки `rgDrawFrame`, которая и осуществляет непосредственную отрисовку сцены в графическое окно приложения.

5.3. Портирование на x64

Расширение Vulkan для поддержки трассировки лучей требует запуска приложения в 64-битном режиме, поэтому необходимо портировать Serious Engine на x64.

За основу для портирования на x64 был взят проект[18], адаптирующий исходный код Serious Engine для операционных систем Linux и

[18] https://github.com/rcgordon/Serious-Engine

MacOS, распространяемый под лицензией GNU GPL v2. Основные изменения касаются перевода кода с языка ассемблера на C++, так как для компиляции Serious Engine используется MSVC, который не поддерживает[19] встроенный ассемблер для x64; а также с заменой некоторых функций WinAPI.

В некоторых классах (например, `BSPEdge`, `BSPPolygon`) были использованы 32-битные типы данных для хранения указателей – они были заменены 64-битными; но при чтении данных с диска производилась проверка размера с помощью `sizeof` от этих классов, поэтому вместо этого используется размер типов классов с учётом разницы размеров указателей в 4 байта. Serious Engine предоставляет собственный интерпретатор команд `CShell`, в котором все целые значения хранятся в виде 32-битных чисел. Одна из переменных – `pwoCurrentWorld` – является указателем на текущий экземпляр класса `CWorld`, хранящий информацию о сцене. Для решения проблемы в класс `CShell` была добавлена функциональность для хранения 64-битных значений с помощью двух 32-битных переменных.

[19] https://docs.microsoft.com/ru-ru/cpp/assembler/inline/inline-assembler?view=msvc-160

Заключение

В ходе данной работы были получены следующие результаты.

- Спроектированы интерфейс и архитектура библиотеки для отрисовки трехмерной графики с помощью трассировки лучей и выводом полученного изображения в графическое окно. Библиотека предназначена для приложений с конвейером фиксированной функциональности. Реализация произведена на языке C++ и графическом API Vulkan.

- Реализована система для обработки геометрии, материалов и источников света в библиотеке.

- Реализована система отрисовки в библиотеке, позволяющая рассчитывать прямое и непрямое освещение посредством трассировки путей с минимальным количеством лучей, а также реализован алгоритм A-SVGF устранения шума для освещения. Реализована отрисовка полупрозрачных объектов с помощью растеризации.

- Проведена апробация с помощью реализации адаптера между библиотекой и целевым приложением, позволяющего использовать трассировку путей в видеоигре “Serious Sam: The First Encounter”.

Список литературы

[1] Alexey Panteleev Christoph Schied. Real-Time Path Tracing and Denoising in 'Quake 2' (Presented by NVIDIA). — 2019. — Game Developers Conference. Access mode: https://www.gdcvault.com/play/1026185/.

[2] Blockwise multi-order feature regression for real-time path-tracing reconstruction / Matias Koskela, Kalle Immonen, Markku Mäkitalo и др. // ACM Transactions on Graphics (TOG). — 2019. — Т. 38, № 5. — С. 1–14.

[3] Edge-avoiding À-Trous wavelet transform for fast global illumination filtering / Holger Dammertz, Daniel Sewtz, Johannes Hanika, Hendrik PA Lensch // Proceedings of the Conference on High Performance Graphics / Citeseer. — 2010. — С. 67–75.

[4] Edge-avoiding À-Trous wavelet transform for fast global illumination filtering / Holger Dammertz, Daniel Sewtz, Johannes Hanika, Hendrik PA Lensch // Proceedings of the Conference on High Performance Graphics / Citeseer. — 2010. — С. 67–75.

[5] Haines Eric, Akenine-Möller Tomas. Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs. — Springer, 2019. — С. 239–241.

[6] Heitz Eric. Sampling the GGX distribution of visible normals // Journal of Computer Graphics Techniques Vol. — 2018. — Vol. 7, no. 4.

[7] ISO/IEC JTC1/SC22/WG14 - C: Approved standards. — Режим доступа: http://www.open-std.org/jtc1/sc22/wg14/www/standards (дата обращения: 18.04.2021).

[8] Igehy Homan. Tracing ray differentials // Proceedings of the 26th annual conference on Computer graphics and interactive techniques. — 1999. — P. 179–186.

[9] Kajiya James T. The rendering equation // Proceedings of the 13th annual conference on Computer graphics and interactive techniques. — 1986. — С. 143–150.

[10] Möller Tomas, Trumbore Ben. Fast, minimum storage ray-triangle intersection // Journal of graphics tools. — 1997. — Vol. 2, no. 1. — P. 21–28.

[11] NVIDIA RTX DI | NVIDIA Developer. — 2021. — Режим доступа: https://developer.nvidia.com/rtxdi (дата обращения: 18.04.2021).

[12] NVIDIA RTX Global Illuminaiton | NVIDIA Developer. — 2021. — Режим доступа: https://developer.nvidia.com/rtxgi (дата обращения: 18.04.2021).

[13] NVIDIA Real-time Denoiser | NVIDIA Developer. — 2021. — Режим доступа: https://developer.nvidia.com/nvidia-rt-denoiser (дата обращения: 18.04.2021).

[14] Neural Temporal Adaptive Sampling and Denoising / J Hasselgren, J Munkberg, M Salvi и др. // Computer Graphics Forum / Wiley Online Library. — Т. 39. — 2020. — С. 147–155.

[15] Q2RTX: NVIDIA's implementation of RTX ray-tracing in Quake II. — https://github.com/NVIDIA/Q2RTX.

[16] Q2VKPT. — 2019. — Режим доступа: http://brechpunkt.de/q2vkpt/ (дата обращения: 18.04.2021).

[17] Ramamoorthi Ravi, Hanrahan Pat. An efficient representation for irradiance environment maps // Proceedings of the 28th annual conference on Computer graphics and interactive techniques. — 2001. — P. 497–500.

[18] Real-time rendering / Tomas Akenine-Möller, Eric Haines, Naty Hoffman и др. — 4 изд. — Boca Raton : Taylor & Francis, CRC Press, 2018. — ISBN: 9781138627000.

[19] Real-time rendering, глава 26 "Real-Time Ray Tracing" / Tomas Akenine-Möller, Eric Haines, Naty Hoffman и др. — 4 изд. — Boca Raton : Taylor & Francis, CRC Press, 2018. — ISBN: 9781138627000.

[20] Real-time rendering, глава 9.3 "Physically Based Shading. The BRDF" / Tomas Akenine-Möller, Eric Haines, Naty Hoffman и др. — 4 изд. — Boca Raton : Taylor & Francis, CRC Press, 2018. — ISBN: 9781138627000.

[21] Real-time rendering, глава 9.8 "BRDF Models for Surface Reflection" / Tomas Akenine-Möller, Eric Haines, Naty Hoffman и др. — 4 изд. — Boca Raton : Taylor & Francis, CRC Press, 2018. — ISBN: 9781138627000.

[22] Saito Takafumi, Takahashi Tokiichiro. Comprehensible rendering of 3-D shapes // Proceedings of the 17th annual conference on Computer graphics and interactive techniques. — 1990. — С. 197–206.

[23] Schied Christoph, Peters Christoph, Dachsbacher Carsten. Gradient estimation for real-time adaptive temporal filtering // Proceedings of the ACM on Computer Graphics and Interactive Techniques. — 2018. — Т. 1, № 2. — С. 1–16.

[24] Serious-Engine: An open source version of a game engine developed by Croteam for the classic Serious Sam games. — https://github.com/Croteam-official/Serious-Engine.

[25] Serious Sam Classic: The First Encounter - Croteam - Croteam. — 2001. — Режим доступа: http://www.croteam.com/serious-sam-classic-first-encounter/ (дата обращения: 18.04.2021).

[26] Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination / Christoph Schied, Anton Kaplanyan, Chris Wyman и др. // Proceedings of High Performance Graphics. — 2017. — С. 1–12.

[27] VulkanMemoryAllocator. — https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator.

[28] An efficient denoising algorithm for global illumination. / Michael Mara, Morgan McGuire, Benedikt Bitterli, Wojciech Jarosz // High Performance Graphics. — 2017. — T. 10. — C. 3105762–3105774.

[29] A global illumination solution for general reflectance distributions / Françis X Sillion, James R Arvo, Stephen H Westin, Donald P Greenberg // Proceedings of the 18th annual conference on Computer graphics and interactive techniques. — 1991. — P. 187–196.

[30] Видеокарты с архитектурой Turing | NVIDIA. — 2018. — Режим доступа: https://www.nvidia.com/ru-ru/geforce/turing/ (дата обращения: 18.04.2021).

[31] Пантелеев Алексей. Rendering Perfect Reflections and Refractions in Path-Traced Games // NVIDIA Developer Blog | Technical content: For developers, by developers on NVIDIA Developer Blog. — 2020. — Режим доступа: https://developer.nvidia.com/blog/rendering-perfect-reflections-and-refractions-in-path-traced-ga (дата обращения: 18.04.2021).