Федеральное государственное учреждение
«Федеральный исследовательский центр
Институт прикладной математики им. М.В. Келдыша
Российской академии наук»

На правах рукописи

Санжаров Вадим Владимирович

РАЗРАБОТКА РАСШИРЯЕМОЙ СИСТЕМЫ ФОТОРЕАЛИСТИЧНОГО РЕНДЕРИНГА НА GPU

Специальность 2.3.5 –
«математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
кандидат физико-математических наук
Фролов Владимир Александрович

Москва 2023

Оглавление

Введение

Актуальность работы

Фотореалистичный рендеринг подразумевает возможность синтеза изображений с высоким уровнем реализма, которые визуально практически неотличимы от фотографий.

Области применения программных систем фотореалистичной визуализации включают в себя промышленный дизайн, архитектурную визуализацию (рис. 1) и светодизайн, создание кино- и анимационных фильмов, создание различных видеопоследовательностей (рекламных, демонстрационных, обучающих и др.), компьютерные игры и тренажерные комплексы, генерацию наборов данных для задач машинного обучения. Требуемая функциональность рендер-системы в перечисленных областях варьируется как между отдельными областями, так и между различными проектами внутри одной области. Поэтому одной из наиболее критичных характеристик современных фотореалистичных рендер-систем является расширяемость, т.е. возможность оперативного добавления новой функциональности для адаптации рендер-системы к разным практическим задачам. К новой функциональности также относятся и новые *математические модели*.

При этом, практически невозможно заранее предусмотреть и реализовать в рендер-системе всё, что когда-либо потребуется её пользователям в будущем, – например, определённый тип геометрического примитива или модели материала с новыми свойствами, которые не поддерживались ранее. Поэтому расширяемая рендер-система должна позволять осуществлять добавление новой функциональности в том числе и конечным пользователям.

Другой важной характеристикой является скорость расчета, которая является варьируемым, но критичным параметром, поскольку чем быстрее расчёт – тем более сложные сцены и модели 3D художник (или любой другой пользователь) может использовать в своём проекте. В настоящее время реализация рендер-систем

на GPU является одним из наиболее действенных и практичных способов ускорения фотореалистичного синтеза изображений.

Рис. 1 Пример дизайна интерьера, выполненный с помощью разработанной системы.

Серьезным препятствием использования GPU для фотореалистичного рендеринга является высокая трудоемкость добавления новой функциональности и интеграции GPU рендер-систем с пользовательскими CPU приложениями. Это связано, в том числе, с тем, что в этом случае рендер-система и пользовательское приложение работают в разных адресных пространствах. Последнее приводит к тому, что невозможно напрямую передавать или вызывать функции одного приложения из другого.

Другая проблема связана со сложностью и ограничениями в процессе разработки на GPU, что приводит к высокой стоимости и низких возможностей по расширяемости у GPU-решений, что, в свою очередь, не позволяет конечным пользователям легко добавлять необходимую им специфическую функциональность – новые материалы, процедурные текстуры и геометрию и т.д.

При этом в индустриальных применениях фотореалистичного рендеринга (в частности, таких как кинопроизводство, создание видеороликов различного

назначения, архитектурная визуализация и др.), часто возникает потребность в подобного рода возможностях для расширения доступной функциональности рендер-системы пользовательским кодом, и в тесной интеграции рендер-систем с многочисленными клиентскими приложениями, включающими пакеты 3D моделирования, системы автоматизированного проектирования, средства создания и наложения текстур, специальные приложения для настройки материалов и освещения, композирования и пр. При этом пользовательский код часто является специфичным для каждого конкретного проекта в связи с творческим характером этих областей применения. Поэтому интеграция подобных пользовательских расширений должна производится в короткие сроки, в том числе, за счет простоты их отладки и низкой трудоемкости интеграции в рендер-систему. Это является одной из причин, по которой CPU рендер-системы, обладающие высокой степенью гибкости, относительно легко интегрируются с другими CPU приложениями и продолжают широко использоваться в этих областях несмотря на значительные преимущества GPU рендер-систем по скорости.

Таким образом, разработка методов и подходов к построению расширяемых систем фотореалистичного рендеринга на GPU, которые бы позволили избавиться от перечисленных выше проблем является актуальной задачей.

Целью работы является разработка подходов и архитектурных решений для систем фотореалистичного рендеринга на GPU, допускающих расширение функциональности системы (включая добавление новых математических моделей) путем интеграции с готовыми инструментами приложений-клиентов (таких как приложения для 3D моделирования) и использования пользовательского кода расширений на GPU, без необходимости внесения изменений в существующий программный код рендер-системы и его полной перекомпиляции.

Основные задачи исследований

- Анализ требований к современным системам фотореалистичного рендеринга.

- Разработка архитектуры GPU рендер-системы, позволяющей обеспечить её интеграцию с клиентскими приложениями с возможностью использовать реализованную в них функциональность по созданию программируемых компонентов рендер-системы (таких как процедурные текстуры).
- Разработка подхода к исполнению кода пользовательских расширений для GPU рендер-системы.
- Реализация и апробация разработанных решений в составе GPU рендер-системы.

Научная новизна

- Предложена архитектура рендер-системы с использованием дополнительного программного слоя на основе объектной базы данных, решающего задачи организации интеграции и инфраструктуры. Предложенная новая программная архитектура позволяет:
    - работать с 3D сценами, не помещающимися в оперативную память вычислительной машины пользовательского приложения;
    - изменять и добавлять новые математические модели материалов, источников света и других компонентов 3D сцен без внесения изменений в код инфраструктурного слоя;
    - выполнять сериализацию, импорт и экспорт объектов 3D сцен между разными сценами и различными приложениями;
    - отслеживать изменения, производимые в сцене пользователем, что дает возможность эффективной передачи данных, в том числе и при распределенной работе со сценой - между разными приложениями (например, рендер-системой и 3D редактором) передаются только изменения, а не вся сцена целиком;
    - организовать эффективную отладку и поиск ошибок за счет механизма отслеживания изменений, который позволяет локализовать действия пользователя, повлекшие за собой сбой в рендер-системе или клиентском приложении.

- Предложен метод для разработки и исполнения пользовательского кода расширений GPU рендер-системы, позволяющий модифицировать отдельные этапы процесса синтеза изображений с использованием кода расширений без необходимости внесения изменений в существующий программный код рендер-системы и его полной перекомпиляции. Разработанный метод позволил реализовать ряд процедурных текстур, а также расширить функциональность рендер-системы такими возможностями, как проективное текстурирование (projective texture mapping), на рис. 2 показаны примеры.
- Разработан алгоритм оценки уровня детализации для предрассчитанных процедурных текстур. Алгоритм позволяет до начала рендеринга вычислить разрешение процедурных текстур, и позволяет снизить затраты памяти без потерь качества изображения. Алгоритм применен для синтеза процедурных текстур, получаемых средствами клиентских приложений (таких как 3D-редакторы). Тем самым была обеспечена прямая поддержка инструментов клиентских приложений. Также разработанный алгоритм позволяет определять разрешение текстур-изображений, что позволяет достичь экономии памяти в 1.6-4 раза без видимых потерь качества.

Рис. 2 Пример работы разработанного механизма создания пользовательских процедурных текстур и проективного текстурирования - полосы и надпись на изображении слева получены с помощью проекции изображения на модель с отсутствующими текстурными координатами.

Теоретическая значимость

Разработанная программная архитектура обеспечивает решение проблем интеграции, инфраструктуры, распределенного рендеринга и отладки, и применима при разработке GPU рендер-систем фотореалистичного рендеринга. Разработанный алгоритм оценки разрешения процедурных текстур позволяет обеспечить экономию памяти GPU и интегрировать существующие сторонние инструменты создания процедурных текстур. Для алгоритма была доказана теорема о том, что он позволяет вычислить разрешение, обеспечивающее соотношение в минимум 1 тексель текстуры на 1 пиксель синтезированного рендер-системой изображения. Алгоритм также применим и для обычных текстур-изображений. Предложенный метод разработки и исполнения пользовательских расширений может быть использован для поддержки новых математических моделей в рендер-системах на GPU, в том числе использующих возможности аппаратного ускорения трассировки лучей.

Рис. 3 Примеры изображений, созданных пользователями разработанных решений в ПО Autodesk 3ds Max

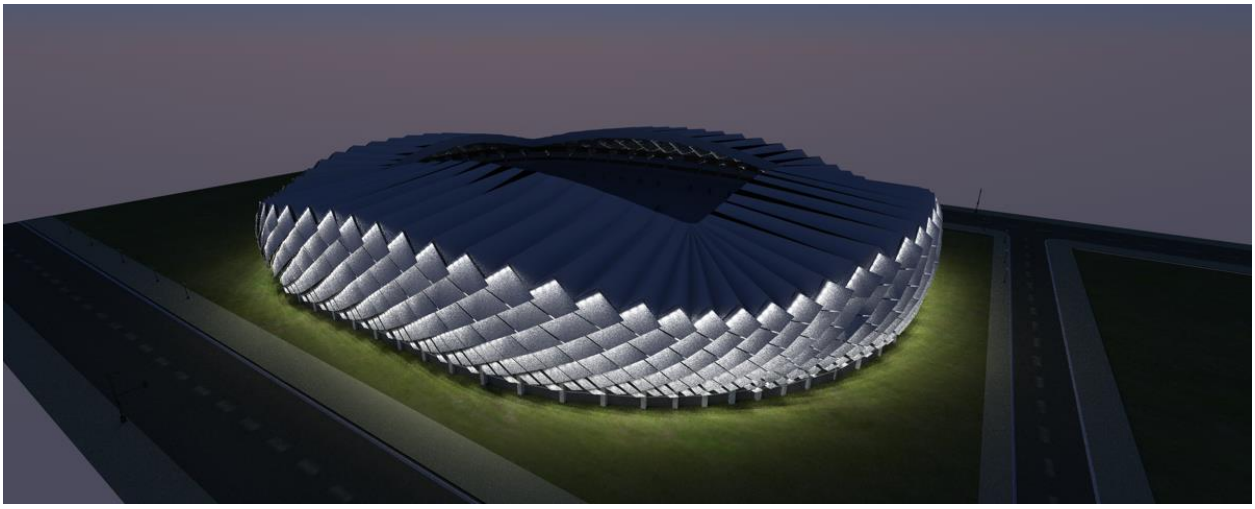


Рис. 4 Проект освещения нового стадиона в г. Батуми, выполненный компанией «Интилед» с использованием разработанной системы.

Практическая значимость

- Разработанная программная архитектура и алгоритмы внедрены в открытую GPU систему фотореалистичного рендеринга Hydra Render, разработанную в Институте прикладной математики имени М. В. Келдыша РАН.
- Возможности предложенной архитектуры дополнительного программного слоя позволили обеспечить интеграцию рендер-системы с рядом программных продуктов:
    - пакет 3D моделирования Autodesk 3ds max, рендер-система используется конечными пользователями (3D художниками) через механизм интеграции с 3ds max (рис. 3);
    - программным продуктом для светодизайна LightCAD компании «Интилед» в качестве фотореалистичной рендер-системы по умолчанию (рис. 4);
    - программный продукт 3d моделирования с открытым исходным кодом Blender.
- Разработанный метод исполнения пользовательских расширений апробирован для создания процедурных текстур в составе проекта для синтеза обучающих данных для нейросетей (рис. 2, 5, 6). Интеграция

компонентов программного комплекса была осуществлена с использованием предложенной архитектуры дополнительного программного слоя.

- Проведено сравнение разработанного подхода с конкурирующими решениями создания расширений для GPU рендер-систем, показавшее состоятельность предложенных подходов.

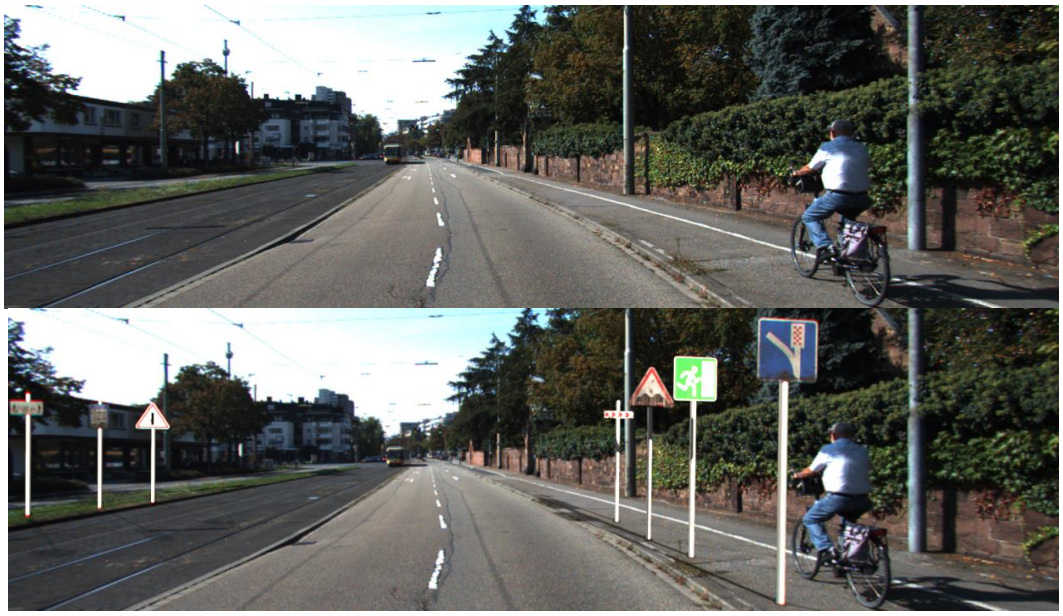


Рис. 5 Наверху – исходная фотография, внизу - с наложенными поверх изображениями дорожных знаков, синтезированными с помощью разработанных в работе средств. Такое наложение используется для повышения качества распознавания нейросетевыми моделями дорожных знаков, которые редко встречаются в реальных условиях, а также имеющих различного рода изменения внешнего вида, например, загрязнения.

Рис. 6 Наверху - исходная фотография, внизу с наложенными изображениями автомобилей, синтезированными с помощью разработанных в работе средств.

Методы исследования

При создании программной архитектуры рендер-системы использовались методы и алгоритмы реализации систем управления версиями и объектно-ориентированных баз данных. Для оценки разрешения предрассчитанных процедурных текстур использовались методы математического анализа. При создании метода разработки и исполнения пользовательских расширений использовалась теория синтаксического анализа и компиляции.

Достоверность и обоснованность результатов

Обоснованность результатов обеспечивается с помощью проведенной экспериментальной оценки результатов использования предложенных методов и алгоритмов при их интеграции в GPU рендер-систему с открытым исходным кодом, в которой был осуществлен рендеринг набора тестовых 3D сцен. Для предложенного алгоритма оценки разрешения предрассчитанных процедурных текстур была доказана теорема, что вычисленное им разрешение позволяет достичь соотношения минимум 1 тексель текстуры на 1 пиксель синтезированного рендер-системой изображения. Для предложенной программной архитектуры и метода разработки пользовательских расширений также было проведено сравнение результатов с аналогами.

Основные положения выносимые на защиту:

1. Программная архитектура рендер-системы с использованием дополнительного программного слоя, решающего задачи организации интеграции и инфраструктуры.
2. Алгоритм оценки уровня детализации для предрассчитанных процедурных текстур, основанный на выполнении специализированного предварительного рендеринга сцены. Алгоритм интегрирован в предложенную программную архитектуру.
3. Метод разработки и исполнения пользовательских расширений для GPU рендер-системы. Метод интегрирован в предложенную программную архитектуру.

Апробация работы

Основные положения работы были доложены на:

1. Международная конференция по компьютерной графике, обработке изображений и машинному зрению, системам визуализации и виртуального окружения Графикон, Томск, Россия, 24-27 сентября, 2018.
2. Международная конференция по компьютерной графике и машинному зрению Графикон, Брянск, Россия, 23-26 сентября, 2019
3. Международная конференция по компьютерной графике и машинному зрению Графикон, Брянск, Россия, 23-26 сентября, 2019
4. 14th International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing (CGVCVIP), Zagreb, Croatia, 23-25 July, 2020
5. Научный семинар им. М.Р. Шура-Бура в ИПМ им. М.В. Келдыша РАН, 10 декабря 2020.

Публикации

По результатам диссертации имеются 8 печатных работ [1-8] в рецензируемых журналах перечня ВАК, из них 6 работ в Scopus [2-5, 7-8] и 5 в Web of science [2-3, 5, 7-8].

Личный вклад автора

Содержание диссертации и основные положения, выносимые на защиту, отражают персональный вклад автора в опубликованных работах.

Диссертационная работа соответствует паспорту специальности (ПС) 2.3.5 - математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей. Работа посвящена разработке программной архитектуры GPU рендер-системы решающей задачи организации интеграции и инфраструктуры (направления 3 и 7 ПС). Был предложен и реализован метод разработки и исполнения кода пользовательских расширений в GPU рендер-системе (направления 1 и 2 ПС). Предложен и реализован алгоритм оценки разрешения предрассчитанных процедурных текстур (направление 7 ПС).

Структура и объем диссертации

Диссертация состоит из введения, обзора литературы, пяти глав, заключения, библиографии. Общий объем диссертации 156 страниц, из них 156 страниц основного текста, включая 36 рисунков и 5 таблиц. Библиография включает 167 наименований.

Глава 1. Обзор предметной области

1.1 Требования к современным приложениям компьютерной графики

Цель фотореалистичного рендеринга состоит в том, чтобы для некоторой 3D сцены синтезировать изображение, которое будет визуально неотличимо от фотографии той же самой сцены. Решение этой задачи востребовано в широком диапазоне различных отраслей, включающих архитектурную и интерьерную визуализацию, светодизайн, промышленный дизайн, создание анимационных и кинофильмов и др. В последнее время также активно развивается направление использования фотореалистичного рендеринга для синтеза наборов изображений и видеопоследовательностей для задач машинного обучения в компьютерном зрении [9-13].

Для синтеза изображения программные системы фотореалистичного рендеринга принимают на вход описание трехмерной сцены, включающее такие компоненты как:

- геометрическое представление (полигональное, криволинейные поверхности подразбиения, объемные примитивы и др.);
- параметры моделей оптических свойств поверхностей компонентов сцены – материалы, при этом некоторые из параметров могут быть заданы с помощью текстур или программ;
- параметры моделей источников света;
- расположение и характеристики виртуального наблюдателя.

Создание 3D сцен в современных индустриальных приложениях компьютерной графики, таких как производство кинофильмов и видеороликов, компьютерные игры и тренажерные комплексы, представляет собой сложный процесс, в котором задействовано множество специалистов и различное программное обеспечение. Весь процесс работы на сценой можно представить в виде конвейера [14], [15, с. 3-8], включающего в себя задачи 3D моделирования, текстурирования, анимации и физического моделирования, настройки освещения,

рендеринга, композирования и др. (рис. 7) В областях, где сложность сцен сравнительно ниже, таких как архитектурная визуализация или визуализация в промышленном дизайне, некоторые из стадий конвейера могут отсутствовать, а оставшиеся могут быть реализованы в меньшем числе программных продуктов и с вовлечением меньшего числа специалистов в этот процесс (возможно и единственным 3D-художником).

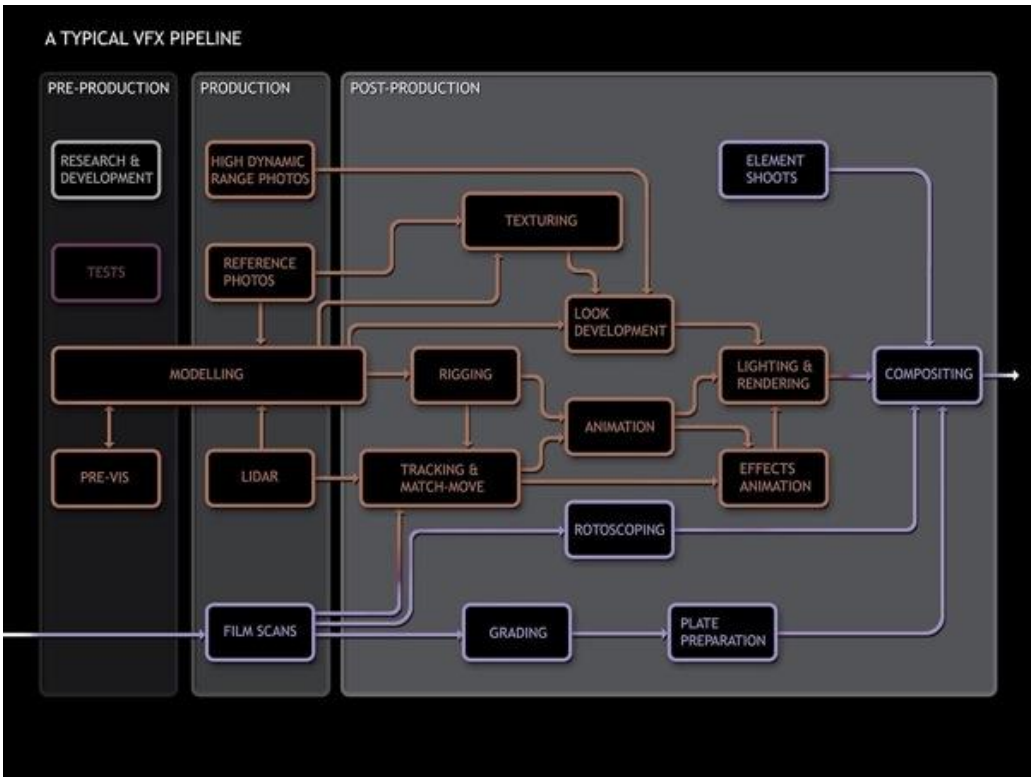


Рис. 7 Пример типичного конвейера рабочего процесса при создании визуальных эффектов в киноиндустрии [14].

При этом, каждая из стадий конвейера по созданию 3D сцены может также быть представлена в виде своего отдельного конвейера. Формально, название «конвейер» здесь становится уже не совсем корректным, т.к. работа выполняется одним специалистом итеративно, по отдельным стадиям. В частности, конечный пользователь рендер-системы выполняет многократные итеративные изменения параметров моделей материалов, источников света и, возможно, параметров алгоритма рендеринга. Высокая стоимость подобного ручного труда 3D-художников приводит к тому, что в современном программном обеспечении компьютерной графики для создания 3D сцен (Blender, 3ds Max, Maya, Houdini,

Cinema 4D и др.) на первое место встает удобство работы пользователя, которое выражается в ряде требований к программному обеспечению и необходимости тесной интеграции рендер-системы с этими приложениями. При этом из необходимости тесной интеграции следует то, что требования, обеспечивающие удобство работы пользователя, распространяются и на рендер-систему.

Анализируя литературные источники, представленные выше схемы конвейеров работы по созданию 3D сцен, а также опыта и отзывов пользователей рендер-систем и приложений для создания 3D сцен, можно сформулировать следующий набор требований:

1. **Интерактивность** процесса создания 3D сцены (концепция What You See Is What You Get – WYSIWYG, «что видишь, то и получишь»). Данное требование проистекает из характера работы 3D-художника - после выполнения очередного изменения параметров моделей (материалов, источников света и пр.) запускается рендер-система для синтеза чернового изображения, на основе которого оценивается соответствие результата поставленной цели или художественному замыслу и выполняется последующее редактирование. Таким образом, пользователь для эффективной работы должен в процессе редактирования сцены видеть результат максимально приближенный к финальному.
2. **Отсутствие ограничений на память**. Вся 3D сцена может не помещаться в оперативную память одного компьютера. При этом процесс редактирования и сопутствующий ему процесс рендеринга (хотя бы в режиме предварительного просмотра) не должны обладать задержками [16].
3. **Изменчивость параметров и расширяемость** обоих систем (и редактора и рендера). Разнообразие применений фотореалистичного рендеринга приводит к разнообразию требований к функциональности рендер-систем. При этом даже в пределах одной отрасли разные проекты могут значительно отличаться по составу используемых моделей и алгоритмов компьютерной графики как при создании контента, так и при рендеринге. Например, моделирование меха, волос, физических процессов и пр. Часто студии

визуальных эффектов разрабатывают свои расширения как для ПО 3D моделирования, так и для рендер-систем [17]. Поэтому если рендер-система планируется к использованию в разных отраслях и задачах, то она должна позволять добавлять новые модели и различные специфические пользовательские расширения. При этом желательно изолировать реализацию подобных расширений от кода самой рендер-системы, чтобы упростить их отладку и разработку в целом.

4. **Сериализация, импорт и экспорт**. Создание сложной сцены представляет собой длительный процесс и требует передачи данных между разными приложениями и, в конце концов, передачи данных рендер системе (рис. 7), а в рамках одного приложения - постоянного возвращения к своим наработкам [15, с. 3-8, 46]. Поэтому необходима возможность импортировать и экспортировать материалы и геометрические объекты из ранее созданной сцены в новую или обратно.

5. **Механизмы отладки и тестирования**. При работе над сложным проектом неизбежно появление ранее неизвестных ошибок, которые проявляются лишь при определённой последовательности действий в 3D редакторе или рендере. При этом источник ошибки может располагаться как в одной, так и в другой программной системе. Для эффективной отладки необходимо уметь определять, где именно произошла ошибка – в рендер-системе или 3D редакторе. Необходимость отладочных инструментов в сложном конвейере для создания 3D сцен подчеркнута в [15, с. 246, 278]. Кроме того, разработчики рендер-системы могут не иметь доступа к ПО 3D редактора и в этом случае возникает потребность в раздельном тестировании расчетного ядра рендер-системы на сериализованных данных сцены. Таким образом, нужно обеспечить разделение 3D редактора и рендер-системы, а также сохранение истории изменений 3D сцены и действий пользователя, и иметь возможность воспроизводить сценарии работы пользователя с рендер-системой.

6. **Распределённый рендеринг** и явная передача изменений. Рендеринг видеопоследовательностей, больших наборов изображений или просто тяжелых сцен на практике зачастую происходит на нескольких вычислительных узлах [15, с. 314-323]. Таким образом, изменения, производимые в редакторе, должны быть видны рендеру не только на одном компьютере, но и на других вычислительных узлах, задействованных в рендеринге. Передача всей сцены целиком по сети в этом случае неэффективна. Поэтому необходимо уметь отслеживать и передавать только изменения.

7. **Скорость расчета**. Чем быстрее расчёт – тем более сложные сцены и модели могут быть использованы в проекте. Кроме того, в таких приложениях, как генерация видеопоследовательностей или наборов данных для машинного обучения, необходимо синтезировать тысячи изображений (например, в [9] авторы использовали 4000 синтезированных изображений в дополнение к реальным фотографиям) и влияние скорости расчета одного изображения становится ещё более значительным. Распространенное решение для увеличения скорости расчета – реализация рендеринга на графическом процессоре, что становится ещё более актуально с появлением аппаратного ускорения трассировки лучей в современных GPU [7]. Однако, разработка на GPU более сложна и трудоемка.

Отметим, что перечисленные требования оказывают влияние друг на друга и в определенном смысле противоречат друг другу. Например, обеспечение требований отсутствия ограничений на память и наличия средств отладки, очевидно, приведёт к снижению скорости расчета. Также и требование расширяемости и изменчивости параметров вступает в противоречие со скоростью расчета, в особенности при реализации рендер-системы на GPU в виду ограничений, свойственных процессу разработки на графических процессорах. С другой стороны, обеспечение требования расширяемости вызывает проблемы для требования поддержки механизмов сериализации, импорт и экспорта данных.

Например, добавляя каждую новую модель материалов, необходимо отразить их параметры в сохраняемом описании сцены, что приводит к необходимости расширить его формат.

Также следует отметить, что обозначенная совокупность требований характерна именно для рассматриваемой области - приложений для создания 3D сцен и фотореалистичных рендер-систем, поскольку в других типичных приложениях компьютерной графики данные требования встречаются лишь частично. Рассмотрим некоторые примеры.

Компьютерные игры. Современные компьютерные игры, симуляторы и тренажерные комплексы являются чрезвычайно сложными программными системами, объём визуализируемых 3D сцен в которых почти всегда не помещаются в оперативную память или память GPU. Но в данном типе приложений обычно отсутствует требование изменчивости параметров. Все отображаемые сцены оптимизируются и подготавливаются для отображения заранее (зачастую совместными усилиями художника и программиста), а используемые математические модели - типы геометрии, материалы, источники, алгоритмы отображения, жёстко фиксированы. Изменить какую-либо 3D модель или текстуру в процессе игры невозможно и не требуется. В редакторах это, с другой стороны, является основной функциональностью, как и возможность расширять модели материалов и источников произвольными параметрами. Однако, если рассматривать современные так называемые «игровые движки», такие как Unity3D, Unreal Engine, Godot, то они уже в значительной мере повторяют по своей функциональности 3D-редакторы. Тем самым они уже приближаются к приложениям для создания 3D сцен и некоторые фотореалистичные рендер-системы интегрируются с игровыми движками [18] для решения таких задач как предварительный расчет освещения, создание анимационных видеороликов и т.д.

Информационные системы. В приложениях, традиционно работающих с базами данных выполняются многие из описанных выше требований, однако, как правило, не выполняется требование интерактивности. Такие приложения ориентированы в основном на быстрый поиск семантически значимой информации

и на транзакции, обеспечивающие целостность базы при одновременной работе нескольких пользователей. Быстро добавлять информацию в базу данных большими порциями в этих приложениях не требуется. Помимо этого, требование распределённости для баз данных также будет отличаться. Пользователю базы данных, как правило, не нужно уметь получать на своей машине состояние всей базы целиком (или какой-то значительной её части). Также не нужно дублировать состояние базы данных на различных машинах (разве что частично в целях повышение отказоустойчивости). В рендер-системах ситуация обратная, – чтобы выполнить синтез фотореалистичного изображение рендер-системе необходимо иметь доступ ко всей сцене целиком. Причём все машины, участвующие в распределённом рендеринге, должны получить одно и то же состояние сцены. С другой стороны, рендер-системе не нужен механизм транзакций, поскольку экспорт контента из редактора в рендер-систему всегда идёт из единственного приложения (что, правда не исключает многопоточности самого процесса экспорта).

Более подробное рассмотрение некоторых перечисленных требований может быть найдено в [19], где авторы, полагаясь на собственный опыт, заключили что «движок» рендер-системы должен существовать в форме некоторого API к базе данных и описали требуемую функциональность такой базы данных. Реализация, удовлетворяющая описанным требованиям, однако, не была описана, оставляя таким образом открытое пространство для будущих исследований.

В рамках настоящей работы разработан механизм интеграции рендер-системы с приложениями для создания 3D сцен, позволяющий обеспечить поддержку вышеперечисленных требований.

1.2 Задача фотореалистичного синтеза изображений

1.2.1 Уравнение рендеринга

Рендер-система – программная система, решающая задачу синтеза изображения. Если говорить о фотореалистичном рендеринге, то это означает, что

рендер-система должна вычислить цвет каждого пикселя изображения так, чтобы результат максимально близко соответствовал наблюдаемому в реальном мире. Математически данная задача формулируется в виде уравнения рендеринга [20]:

$$L_o(x, \omega_0) = L_e(x, \omega_0) + \int_{\Omega} L_i(x, \omega_i) f(x, \omega_0, \omega_i)(\omega_i \cdot n) d\omega_i \qquad (1\text{-}1)$$

где x – точка в сцене, n – нормаль в этой точке, ω_0 – единичное направление исходящего света, ω_i – единичное направление приходящего света, $(\omega_i \cdot n)$ – величина скалярного произведения между направлением приходящего света и нормалью, Ω - область определения всех возможных направлений приходящего света, $L_e(x, \omega_0)$ – свет излучаемый в точке x в направлении ω_0, $L_i(x, \omega_i)$ – свет, приходящий в точку x вдоль направления ω_i, $f(x, \omega_0, \omega_i)$ – двунаправленная функция отражательной способности (ДФОС, англ. bidirectional reflectance distribution function, BRDF), определяющая какая доля излучения, пришедшего в точку x по направлению ω_i будет отражена по направлению ω_0.

Большинство индустриальных рендер-систем решают уравнение рендеринга с использованием метода Монте-Карло в рамках алгоритма трассировки путей [20] или его модификаций [21-24]. Уравнение рендеринга при этом принимает следующий вид:

$$L_o(x, \omega_0) = L_e(x, \omega_0) + \frac{1}{N} \sum_{j=1}^{N} \frac{L_i(x, \omega_j) f(x, \omega_0, \omega_j)(\omega_j \cdot n)}{g(\omega_j)} \qquad (1\text{-}2)$$

где $g(\omega_j)$ – плотность вероятности, с которой происходит случайная выборка направлений ω_j.

Двунаправленная функция отражательной способности характеризует оптические свойства поверхности и может быть представлена различными параметризованными моделями (например, [25-30]) или в табличной форме в виде заранее измеренных значений [31, 32]. Более общий вариант - *двунаправленная функция рассеивания* (ДФР, англ. bidirectional scattering distribution function, BSDF), помимо отражения света также учитывает пропускание света (например, через полупрозрачную поверхность).

1.2.2 Входные данные

В описании сцены, которое рендер-система принимает на вход из, например, внешнего приложения 3D редактора, задаются материалы для геометрических моделей объектов в сцене. Материал содержит в себе информацию о том, какую модель ДФР необходимо использовать и значения параметров для этой модели. При этом параметры могут быть заданы в виде функции – текстуры, которая обычно имеет вид изображения или программы (процедурной текстуры), которая обычно представляет собой некоторую *математическую модель*. Текстуры позволяют изменять свойства модели материала в зависимости от значения каких-либо параметров – в случае текстур-изображений это координаты в специальном текстурном пространстве, а в случае процедурных текстур это могут быть любые данные, доступные в процессе рендеринга. Виды объектов реального мира, оптические свойства которых может обработать рендер-система в процессе синтеза изображения, определяются поддерживаемыми рендер-системой моделями ДФР, а также возможностями задания параметров этих моделей. Значительное разнообразие моделей материалов приводит к тому, что практически невозможно заранее реализовать в рендер-системе все возможные варианты. Поэтому возможность создания пользовательских расширений для моделей материалов является наиболее важной для обеспечения расширяемости рендер-системы в целом.

Ещё один компонент описания 3D сцены – геометрические модели. Наиболее распространенным является полигональное представление геометрии в виде треугольной сетки, называемое «меш». Другие виды моделей – конструктивная сплошная геометрия, сплайны и пр. могут быть сконвертированы в полигональное непосредственно перед рендерингом или в процессе синтеза изображения. Также следует отметить, что способ задания геометрии с точки зрения большинства реализаций фотореалистичного рендеринга (т.е. основывающихся на трассировке лучей) влияет в первую очередь на операцию определения пересечения луча и геометрического примитива. Таким образом, для поддержания различных

способов представления геометрии, рендер-системе достаточно позволять задавать пользовательские программы для определения пересечений.

Для синтеза изображения рендер-системе также необходима информация об источниках света. Существует несколько основных моделей источников света – точечные, бесконечно удаленные, имеющие конечную площадь и имеющие бесконечную площадь [33, с. 719-741]. Каждая из моделей характеризуется набором параметров, некоторые из которых могут быть заданы текстурой или с помощью специальных IES-файлов. В случае источников света разнообразие в целом не настолько велико, как для моделей материалов или даже геометрических моделей. Реализация в рендер-системе перечисленных выше основных моделей обычно покрывает все применения кроме наиболее специальных, – например в области светотехники возникают задачи моделирования источников света, представляющих собой сложные системы линз и зеркал. Подобные модели представляют значительную трудность для большинства алгоритмов синтеза фотореалистичных изображений. Однако, существуют подходы, позволяющие аппроксимировать подобные сложные источники с помощью группы точечных источников с разными распределениями, а также подходы на основе предварительного расчета [33, с. 742-744].

В рамках данной работы разработано решение, позволяющее создавать пользовательские расширения для моделей материалов в виде программ – процедурных текстур. Этот же механизм позволяет задавать пользовательские геометрические примитивы.

1.3 Особенности разработки рендер-систем на GPU

1.3.1 Технологии программирования

Как было отмечено ранее, разработка рендер-системы на GPU позволяет обеспечить её высокую производительность. Это во многом связано с тем, что алгоритм трассировки путей, на котором основывается фотореалистичный рендеринг, может эффективно использовать возможности массивно-параллельных

вычислений, предоставляемые графическим процессором. Однако, использование GPU существенно усложняет разработку, увеличивая её трудоемкость, время и стоимость. Это связано с необходимостью использования специальных технологий программирования (таких как OpenCL, Vulkan, CUDA), ограниченных возможностей по отладке программ – традиционные средства отладки на GPU не доступны, разработчику приходится полагаться на специальные программные средства (например, RenderDoc [34] и Nvidia NSight [35]), функциональность которых может зависеть от используемой технологии программирования и от модели графического процессора. На сложность отладки программ также влияет и то, что некоторые ошибки могут проявляться только на определенных моделях графического процессора или даже только в определенных версиях драйвера.

Также следует отметить, что при разработке на GPU во многих случаях происходит потеря кроссплатформенности программного обеспечения за счёт использования платформозависимых технологий программирования (например, CUDA или OptiX [36]), а также из-за отсутствия алгоритмических оптимизаций в системах трансляции/компиляции современных технологий программирования GPU, что приводит к тому, что оптимизации оказываются «зашитыми» в реализацию алгоритма и при этом являются специфическими для определенных моделей (или групп моделей) графических процессоров. Все это приводит к появлению технологической зависимости от производителя оборудования или же к необходимости поддержки со стороны разработчика рендер-системы нескольких реализаций отдельных компонентов (под разное аппаратное обеспечение) или же всей рендер-системы, что кратно влияет на стоимость и время разработки программной системы. Например, CPU рендер-система VRay и GPU рендер-система VRay GPU являются разными продуктами, не имеющими полной совместимости друг с другом, при этом VRay GPU поддерживает только GPU от Nvidia [37].

1.3.2 Подходы к реализации рендер-системы на GPU

Несмотря на то, что трассировка лучей на GPU сама по себе является ограниченной и компактной задачей, которую можно решать эффективно различными способами, проблема в корне меняется, когда на основе трассировки лучей необходимо построить расширяемую программную систему с большим количеством различных возможностей. На данный момент известно три основных подхода к реализации рендер-системы на основе трассировки путей на GPU:

1) «Убер-ядро» (uber-kernel, «убер-кернел») – подход, при котором код организуется вручную или автоматически (обычно последнее) в виде конечного автомата внутри одного вычислительного ядра. Автомат используется для того, чтобы сократить регистровое давление, поскольку каждое состояние в верхнем операторе switch получает в свое распоряжение все доступные для программы (вычислительного ядра) регистры. Основные недостатки этого подхода – существенные потери производительности на ветвлениях (когда разные потоки выполняют разные состояния), и влияние различных состояний на производительность друг друга, т.к. итоговое вычислительное ядро требует столько регистров, сколько нужно самому тяжелому состоянию [38, 39]. Этот недостаток, очевидно, снижает возможности по расширяемости рендер-системы – вновь разработанные расширения будут все больше увеличивать количество состояний. Кроме того, подход на основе убер-ядра может негативно сказаться на интерактивности работы рендер-системы и усложнить отладку. В случае добавления новой функциональности в рендер-систему, например, алгоритма процедурной закраски, её код будет также вставлен в убер-ядро. Это приведет к необходимости перекомпиляции всего убер-ядра, которая может потребовать ощутимого для конечного пользователя времени. А смешение кода расширений с основным кодом рендер-системы приводит к сложности поиска причин потенциальных ошибок.

2) «Разделенное ядро» (separate kernel) – подход, при котором код организуется (как правило вручную) в виде нескольких вычислительных ядер, общающихся между собой явно через буферы данных в памяти [38]. Этот подход

решает основные недостатки убер-ядра, и, благодаря явному разделению на ядра, позволяет сохранять производительность критичных участков кода. Однако он обладает повышенной трудоемкостью разработки (из-за необходимости явной передачи данных, что особенно заметно при наличии сортировки или уплотнения потоков [40]). Кроме того, повышаются накладные расходы на запуск и ожидание ядер, и на передачу данных. Поэтому такой подход может замедлять работу программы на простых сценах, когда накладные расходы становятся соизмеримы с полезной работой ядер.

3) «Трассировка путей волновыми фронтами» (wavefront pathtracing) – это сложный подход, в котором работа и данные для лучей группируются в отдельные очереди [39]. Очереди выполняются в разных ядрах, а результат сохраняется в нужное место памяти также путем отдельных вызовов вычислительных ядер. Благодаря группировке по условным шейдерам, wavefront pathtracing меньше теряет на ветвлениях, чем подход убер-ядра, а в отличие от подхода раздельного ядра позволяет поддержать рекурсию. Однако сортировка и уплотнение потоков для лучей в этом подходе строго обязательны, поэтому его накладные расходы еще выше, чем в предыдущем случае.

Перечисленные сложности приводят к тому, что наличие исходно заложенных в архитектуру рендер-системы механизмов отладки и тестирования, а также возможностей к созданию расширений становится ещё более важным.

В рамках настоящей работы разработан механизм создания пользовательских расширений для GPU рендер-системы, выполняющий минимальные преобразования их кода и позволяющий отделить расширения от основной части рендер-системы, что значительно упрощает процесс отладки и тестирования. Разработанные решения были протестированы в системе, реализующей подход разделенного ядра [41], однако они не зависят от методов реализации трассировки лучей и могут использоваться и с другими перечисленными подходами.

1.4 Аппаратное ускорение трассировки лучей

Помимо технологий программирования и подходов к реализации трассировки путей на GPU, описанных в предыдущем разделе, на архитектуру рендер-системы значительное влияние может оказывать использование аппаратного ускорения трассировки лучей. Из-за потенциально значительных преимуществ в производительности поддержка этих технологий является актуальной для разработчиков рендер-систем. Однако, она требует использования определенных программных интерфейсов к аппаратным возможностям внутри рендер-системы и неизбежно влияет на организацию её структуры. Обзор и результаты представленные в данном разделе были впервые опубликованы в [7].

1.4.1 Существующие подходы

Идея аппаратного ускорения трассировки лучей не является принципиально новой и имеет долгую историю развития. Первыми специализированными аппаратными решениями, связанными с трассировкой лучей, были PCI-карты для визуализации объемных данных, в которых было реализовано марширование по лучу (ray marching) и затенение по Фонгу (например, [42, 43]). Другой заметной ранней реализацией была архитектура SaarCOR [44] и ее обновленная версия в ПЛИС [45]. Чип SaarCOR реализовал весь алгоритм трассировки лучей – данные сцены и камеры помещались в отдельную DRAM память, соединенную с чипом. SaarCOR использовал глубокую конвейеризацию для сокрытия высокой латентности доступа к памяти по аналогии к современными GPU: пока одни группы лучей загружают данные, другие, для которых данные уже загружены на чип, могут вычислять пересечение с треугольниками. Основной недостаток этих решений связан с их узкой специализацией, - они предоставляют фиксированную аппаратную функциональность без возможностей программирования.

Альтернативой специализации стало размещение большого числа обычных процессоров на одной плате с PCI-e интерфейсом [46-49]. Такие решения обладают наибольшей гибкостью и могут быть использованы не только для ускорения

трассировки лучей. Но сами по себе они не стали популярны в основном из-за их высокой стоимости.

Наконец, существует группа работ, направленных на разработку аппаратных расширений для графических процессоров (либо разработку похожих массивно-параллельных программируемых систем). Одно из первых программируемых решений такого типа было представлено в работе [50]. Обход дерева и поиск пересечений был реализован в специальном блоке с фиксированной функциональностью, в то время как пользовательские программы (шейдеры) выполнялись на так называемом Shader Processing Unit (SPU), очень похожим по архитектуре на ранние процессорные ядра GPU. Данную работу можно считать прообразом современных решений. Как и SaarCOR, работа [50] использовала трассировку пакетов, из-за чего скорость сильно падала на расходящихся в разные стороны (т.н. некогерентных) лучах. Такая же проблема наблюдается во многих GPU реализациях трассировки лучей [51, 52].

Одно из решений проблемы случайного доступа к памяти предложено в работе [53]. Этот подход подразумевает разделение потока запросов к памяти как минимум на 2 потока – поток данных для лучей (ray stream), и поток данных для сцены (BVH дерево, scene stream). Можно сказать, что традиционный подход сокрытия латентности памяти при помощи глубокой конвейеризации, широко используемый в GPU, в работе [53] расширяется таким образом, чтобы загруженный один раз в кэш трилет (фрагмент BVH дерева) был пройден всеми лучами, которые в данный момент обрабатываются на графическом процессоре. Авторы уверяют что таким образом им удается избегать случайного доступа.

Кроме некогерентных лучей для GPU существует еще проблема нерегулярного распределения работы. Когда в SIMD группе потоков (warp) остается мало активных потоков/лучей, эффективность SIMD процессора GPU существенно снижается. Для решения этой проблемы в работах [51, 52] было использовано уплотнение потоков и регенерация путей, а в [40] была предложена техника блочной регенерации.

Группа работ направлена на оптимизации, связанные с ускоряющими структурами данных. В [49, 51, 54] была использована идея группировки BVH дерева в т. н. трилетах (treelets) – небольших фрагментах BVH дерева. Основное отличие работы [54] состоит в том, что в трилетах можно хранить данные об ограничивающих объемах в BVH с пониженной точностью в 5 бит на 1 плоскость (вместо 32 бит для стандартного типа float). Благодаря этому снижается нагрузка на память и улучшается эффективность работы кэша GPU. Кроме того, предложенное авторами решение является относительно дешевым по количеству используемых транзисторов.

Существуют работы, ориентированные на аппаратную реализацию трассировки лучей для мобильных систем, где важен такой параметр как энергопотребление системы [55, 56]. Эти работы были нацелены в основном на реализацию классической трассировки лучей [57] и предоставляют фиксированную функциональность. Также, в отличие от многих работ, рассмотренных выше, используют MIMD архитектуру с VLIW процессорами, чтобы уменьшить потери энергии/эффективности во время вычислений для расходящихся лучей.

Итого, за последнее время было разработано множество аппаратных реализаций трассировки лучей. Более полный обзор можно найти в работе [58]. Однако, первой такой технологией, которая стала доступна широкой общественности стала Nvidia RTX / Microsoft DXR. Рассмотрим её отдельно более подробно.

1.4.2 Технология RTX

Исходно платформозависимая, данная технология была стандартизирована в качестве расширений для Vulkan API [59, 60] и поддержку других производителей GPU [61] . На стороне аппаратной реализации в GPU от Nvidia с архитектурами Turing и Ampere технология RTX поддерживается специальными т.н. RT ядрами (RT cores), реализующими операции обхода ускоряющей структуры (BVH дерева) и поиска пересечений с ограничивающими объемами и треугольниками, в

архитектуре Ampere также добавлена аппаратная поддержка интерполяции позиции примитивов во времени для ускорения расчета эффекта размытия в движении [62]. Решение от AMD, реализованное в GPU с архитектурой RDNA2 для трассировки лучей, судя по всему, исполняется на процессорах для вычислений общего назначения (т.н. «workgroup processors», WGP) и активно использует общий кэш – Local Data Share (LDS).

С программной стороны трассировка лучей в Vulkan API доступна в виде отдельного вида конвейера, а также может быть использована с помощью функций, доступных в традиционных графическом и вычислительном конвейерах – «вставная» (англ. inline) трассировка лучей или ray query. И в том, и в другом случае необходимо выполнить построение ускоряющих структур с использованием предоставляемого API.

В работе [7] были проведены программные эксперименты с реализаций трассировки путей с использованием RTX и её сравнение с открытой программной реализацией трассировки путей на OpenCL в Hydra Renderer. Сравнения показали значительный прирост производительности (2-5 раз) в том числе на сложных сценах, что говорит об актуальности использования этой технологии в рендер-системах. Также оказалось, что технология RTX устроена внутри намного сложнее, чем просто аппаратные модули поиска пересечений. Среди вероятных используемых внутренних механизмов: упорядочивание случайного доступа к памяти во время трассировки расходящихся лучей и механизм создания работы на GPU, включающий в себя передачу данных между разными вычислительными ядрами через кэш на чипе.

Что касается возможностей интеграции RTX в существующие рендер-системы и её влияния на архитектуру рендер-систем, то ситуация оказывается неоднозначной. Эксперименты в [7] показали, что RTX является сложной технологией, которую трудно эффективно реализовать программно. При этом сама технология доступна на ограниченном наборе графических процессоров [61]. Таким образом, разработчикам рендер-системы необходимо поддерживать несколько внутренних реализаций (т.н. бекэндов, англ. backend) – использующих

RTX и нет. При этом, в настоящий момент технология RTX доступна в таких программно-аппаратных интерфейсах как DirectX12, Vulkan и OptiX. Из них только Vulkan представляет собой кроссплатформенную технологию – DirectX12 привязан к ОС Windows, а OptiX к аппаратному обеспечению Nvidia. А трудоёмкость разработки программ на Vulkan превышает трудоёмкость разработки на CUDA или OpenCL (использовавшихся для разработки GPU рендер-систем до появления RTX) до 10 раз [63]. Это привело к тому, что многие существующие индустриальные системы расчёта освещения перешли на OptiX: Octane, VRay, IRay, RedShift, Cycles и многие другие. Для рендер-систем, реализованных на OpenCL и желающих сохранить кроссплатформенность, возможным выходом является использование технологий подобных [64] для переноса существующей кодовой базы.

Предлагаемый в рамках данной работы подход по созданию расширений использует OpenCL C, код расширений подвергается минимальной обработке и обособлен от основного кода рендер-системы. Благодаря этому, разработанное решение является кроссплатформенным, а при использовании таких инструментов как [64] (для трансляции кода расширений в SPIR-V) может быть интегрировано с рендер-системами, использующими Vulkan и технологию RTX.

1.5 Заключение

В данной главе представлен обзор требований, которым должны отвечать современные приложения компьютерной графики, в том числе рендер-системы, - интерактивности процесса создания 3D сцены, отсутствие ограничений на память, расширяемости, поддержки импорта и экспорта данных, наличие механизмов отладки и тестирования, поддержки распределенного рендеринга и высокой скорости расчета. Можно сказать, что рассмотренные требования отражают взгляд на задачу разработку приложений компьютерной графики (и рендер-систем в частности) со стороны конечных пользователей. Были рассмотрены особенности разработки рендер-систем на графических процессорах, продемонстрированы проблемы, возникающие на пути реализации рассмотренных требований с учетом

возможностей современного аппаратного обеспечения. Несмотря на сложности, имеющиеся в GPU разработке, данное направление является перспективным в связи с развитием графических процессоров и появлением широкодоступной технологии аппаратного ускорения трассировки лучей. При этом актуальной задачей является обеспечение кроссплатформенности программных систем, что проявляется в возможности использования рендер-системой графических процессоров разных производителей и имеющих разных уровень поддержки технологий аппаратного ускорения.

Настоящая работа направлена на разработку архитектурных решений для GPU рендер-систем, позволяющих обеспечить выполнение рассмотренных в данной главе требований. Поддержка данных требований, с одной стороны, повышает удобство работы для конечного пользователя, а с другой – снижает трудоемкость разработки рендер-системы, её интеграции с клиентскими приложениями (такими как 3D-редакторы) и обеспечивает возможность создания специализированных расширений, что позволяет расширить область применения.

Глава 2. Архитектура дополнительного программного слоя

В данной главе описана предложенная программная архитектура, позволяющая обеспечить интеграцию рендер-систем и клиентских приложений. Предлагаемое решение основывается на том, что между 3D редактором и рендер-системой должен находиться специальный слой ПО, обеспечивающий максимально прозрачную и простую реализацию требований, рассмотренных в разделе 1.1. Данный промежуточный слой предоставляет программный интерфейс (API – application programming interface) для клиентских приложений, обеспечивающий возможности передачи данных 3D сцены. А также определяет интерфейс для рендер-систем с заранее предопределенным порядком взаимодействия промежуточного программного слоя с расчётным ядром. Таким образом, рендер-система не контактирует напрямую с клиентскими приложениями, что позволяет упростить реализацию рендер-системы, исключив из неё решение различных инфраструктурных задач. Представленные результаты были впервые опубликованы в [2].

2.1 Существующие решения

2.1.1 Файловые форматы

Наиболее простой способ передачи 3D контента между приложениями, имеющий долгую историю, – это использование текстовых и/или бинарных файлов с жёстко определённым форматом. Например, сравнительно простой, но ограниченный формат OBJ [65] или более сложный и гибкий FBX [66]. В данном подходе обмена данными всегда будет иметь место некоторый компромисс между гибкостью/функциональностью и сложностью формата. Однако, основная проблема заключается в том, что заранее невозможно предусмотреть все возможности и параметры, которые могут понадобиться в будущем разработчикам клиентских приложений (например, 3D редакторов) или рендер-систем. Т.е. какие

будут параметры у материалов, появятся ли новые типы геометрических примитивов, источников света и т.д.

При этом разнообразные приложения компьютерной графики могут требовать специфические возможности (например, поддержку анимации в случае киноиндустрии или конструктивной сплошной геометрии для САПР). Примером такого специфичного формата является Alembic [67], зародившийся в анимационной и киноиндустриях. Основная задача формата Alembic – эффективное хранение сложной анимации, полученной различными способами - анимацией по ключевым кадрам, флюидной, тканевой или какой-либо другой симуляцией. Для того, чтобы решить обозначенную выше проблему появления новых параметров, данный формат позволяет дополнительно хранить произвольные пользовательские данные. Однако, получатель такого файла, должен уметь эти данные обнаружить и правильно интерпретировать.

Идея расширяемости формата вышла на новый уровень в формате OpenCollada. В отличие от жёстко заданных форматов хранения сцен, OpenCollada по сути является расширяемой технологией с чётко определённым стандартом хранения объектов (называемым «COLLADA») в XML [68]. Благодаря открытому исходному коду данного формата сцен и использованию XML описаний, разработчики имеют возможность расширять экспорт и импорт, добавляя свои параметры - как на стороне 3D редактора, так и на стороне рендер-системы.

Также следует упомянуть формат glTF, разработанный Khronos Group [69], который был создан в ответ на распространение технологий компьютерной графики в браузерах (в частности, WebGL) и подхода «Physically based rendering» (PBR, «физически корректный рендеринг») в компьютерной графике реального времени. Под PBR здесь в первую очередь имеется в виду определенные модели материалов, которые пришли на смену устаревшему набору коэффициентов *ka, kd, ks* (ambient – свет «окружающей среды», имитация глобального освещения, diffuse – «диффузный цвет», specular – «цвет отражений») из описания материалов в формате MTL, использовавшемуся в паре с форматом OBJ. Формат glTF позиционируется как «JPEG для 3D», т.е. простой и легковесный формат для

обмена данными. Описание сцены в данном формате представлено как следующий набор:

- .gltf файл, содержащий иерархию объектов («граф сцены»), описание моделей материалов и виртуальных камер в формате JSON;
- .bin файл(ы), содержащий сериализованные геометрические модели и анимацию;
- файлы текстур в различных форматах (.png, .jpg, .ktx2).

Проблема расширяемости в данном формате решается за счет механизма расширений (аналогично графическим API – OpenGL, Vulkan), свойства объектов в JSON могут включать опциональное свойство «extension», включающее описание всех расширений.

Рассмотрим недостатки использования «просто» файлов для решения задачи интеграции. Ни один из рассмотренных форматов хранения 3D сцены не удовлетворяет и в принципе не может удовлетворять требованиям, предъявляемым к приложениям компьютерной графики, рассмотренным в первой главе. Само собой, файл обеспечивают требование сериализации (импорта/экспорта) и позволяют передавать сцену между различными приложениями, при условии наличия соответствующего загрузчика и наличия некоторого стандарта или *соглашения* об интерпретации данных (в частности, пользовательских данных и расширений). В некоторой степени покрывается и требование на механизмы отладки и тестирования – можно проверить соответствие экспортированного файла определенным правилам, – например, «схемам» (schema) JSON в случае glTF [70]. Наконец, современные файловые форматы представления сцен (glTF, Alembic, OpenCollada) отвечают требованию на изменчивость параметров. Таким образом, остаются неохваченными возможности для работы с большими сценами, отслеживания изменений и хранения их истории, эффективного обмена информацией по сети. Наконец, один из основных недостатков – невозможность быстрой передачи изменений без перезаписи файла целиком.

2.1.2 Передача через DRAM и динамически загружаемые плагины

Другой распространенный способ интеграции основан на передаче структур данных через разделяемую или обыкновенную оперативную память из клиентского приложения (например, 3D редактора) в рендер-систему. Также сюда стоит отнести взаимодействие по средством тесно-интегрированных динамически загружаемых плагинов. В этом случае также во время работы могут напрямую использоваться виртуальные функции клиентского приложения, минуя операции импорта и экспорта. Например, рендер-плагин может вызывать некоторую функцию *Shade* для модели материала, которая реализована в редакторе и, таким образом, исключить необходимость передачи большого объёма данных между приложением пользователя и расчётным ядром рендер-системы. С одной стороны, отсутствие необходимости выполнять импорт и экспорт данных безусловно является преимуществом. Но, с другой стороны, такой подход ограничивает производительность, масштабируемость и не гарантирует корректность, т.к. отсутствует какая-либо уверенность в том, что эта функция *Shade* реализована достаточно эффективно и дает тот результат, который необходим разработчику рендер-плагина. Основной же недостаток связан с тем, что при таком способе интеграции внутреннее устройство рендер-системы начинает сильно зависеть от клиентского приложения [71]. Это приводит к тому, что рендер-систему будет значительно сложнее интегрировать в другие клиентские приложения, которые могут предоставлять кардинально отличающиеся друг от друга интерфейсы и функциональность. Также, при таком тесном контакте рендер-системы и клиентского приложения усложняется решение задач отладки и тестирования, т.к. становится трудно отделить возможные проблемы в клиентском приложении от ошибок и проблем в рендер-системе.

Данные подходы к интеграции обладают очевидной простотой и, как правило, довольно высоким быстродействием в плане обмена данными между клиентским приложением и рендер-системой, что позволяет обеспечить требование интерактивности. Также возможно обеспечение требования расширяемости/изменчивости параметров. Остальные же требования остаются

неподдержанными. Использование GPU рендер-систем и распределенного рендеринга оказывается невозможным, т.к. в этом случае рендер-система работает в другом адресном пространстве и не может взаимодействовать с клиентским приложением через разделяемую память и вызывать его виртуальные функции. Тесная интеграция с конкретным клиентским приложением не подразумевает какой-либо сериализации данных для обмена с другими приложениями. Проблемы с отладкой при данном подходе были рассмотрены выше.

2.1.3 Системы Pixar USD и Multiverse

Разработка компании Pixar под названием USD (Universal Scene Description) появилась в открытом доступе как проект с открытым исходным кодом в 2016 году [72]. Данная технология прежде всего предназначена для организации одновременной работы коллективов художников над одним большим проектом (таким как 3D контент для полнометражного анимационного фильма). Эта технология предполагает, что контент создаётся разными людьми в разных приложениях.

Сцена в формате USD представляет из себя набор файлов в формате подобном JSON, а также бинарных файлов во внутреннем формате или других форматах (таких как Alembic), поддержка которых осуществляется путем создания расширений. Программный интерфейс USD предоставляет возможности для иерархической организации компонентов 3D сцен через механизмы ссылок, который в терминах данной технологией называется «композицией» (composition). С каждым из компонентов могут быть ассоциированы параметры, возможно изменяющиеся во времени, а компонент может представлять из себя ссылку на файл или компонент в другом файле.

Основная задача, которую решает данная технология, заключается в обеспечении возможности быстрого создания 3D сцены разными людьми в разных приложениях с разными версиями компонент сцены. Например, в одном приложении производится моделирование некоторой сцены-локации, в другом – создание и анимация модели персонажа, в третьем – создание текстур, в четвертом

– настройка моделей материалов и т.д. После чего создается композиция – финальная сцена, которая ссылается на USD файлы, созданные в этих разных приложениях. При этом, например, у созданных 3D моделей (или любых других компонентов сцены) может быть несколько вариантов – несколько художественных решений, между которыми можно легко переключаться в финальной композиции для выбор наиболее подходящего. Ссылочная система и механизм отложенной загрузки позволяют работать с большими сценами [72]. При экспорте из клиентского приложения, данные (например, геометрия) могут быть сохранены как в бинарном, так и в текстовом формате с целью отладки.

В состав системы USD входит фреймворк (англ. framework - программное обеспечение предоставляющее инструменты для разработки и объединение компонентов программного проекта) для передачи данных в рендер-систему, который принимает на вход иерархическое описание сцены и формирует из него «плоское» представление. При формировании «плоского» представления, производится конвертация объектов сцены в данные во внутреннем формате по типам компонента. В результате получается готовый набор данных, который передается рендер-системе. При этом возможно производить отслеживание изменений – изменился ли тот или иной компонент сцены. Предоставляется рендер-система на основе OpenGL API, которая используется в Pixar для предварительного интерактивного рендеринга в процессе работы над сценой.

Технология USD удовлетворяет многим описанными в начале статьи требованиям. Входящий в её состав фреймворк для рендеринга потенциально позволяет использовать эту технологию как прослойку между 3D редактором (или точнее USD-сценой, экспортированной из редактора) и рендером. Однако, данный фреймворк является довольно сложным и требует от разработчиков рендер-системы определения целого ряда интерфейсов, – начиная от так называемого «рендер-делегата» («render delegate»), которые обеспечивает основное взаимодействие рендера со внутренним представлением сцены в USD, до интерфейсов буфера кадра, компонентов сцен (меша, модели материала и др.) и прочих. Пример, интеграции простейшего рендера на основе трассировки лучей,

предоставляемый разработчиками содержит ~3.5 тысячи строк кода. При этом исходный код самих интерфейсов фреймворка составляет ~19.5 тысяч строк, а всей системы USD – более 640 тысяч строк. Ещё одна проблема USD заключается в том, что у объектов отсутствуют глобальные идентификаторы и вся иерархия сцены представлена в виде текстовых имен и путей – по аналогии с представлением путей в файловой системе. Поэтому изменение имени какого-то из узлов сцены вызывает проблему разрешения ссылок в композициях и необходимо запускать специальную рекурсивную процедуру обхода дерева файлов, чтобы получить новую иерархию сцены. Наконец, возможность перезаписать любой файл затрудняет реализацию сетевого рендера, поскольку появляется ситуация, когда файл на одной машине имеет новое состояние, а на другой - старое.

Система Multiverse [73] исходно создавалась для обеспечения поддержки интерактивной работы с большими сцен в формате Alembic в 3D редакторе Autodesk Maya. Однако в более поздних версиях данной решение трансформировалось в программную прослойку между 3D редактором и системой USD с реализацией некоторой дополнительной функциональности. Как плагин для 3D редактора, Mulitverse предоставляет интерфейс для работы с USD описаниями сцены, их импортом, экспортом и редактированием, а также обеспечивает подключение ряда коммерческих рендер-систем к этой системе. Востребованность такой прослойки, по сути, подтверждает тезис о сложности работы с USD напрямую.

Система Multiverse тесно интегрирована в ПО Maya и обеспечивает возможность обмена данными с другими приложениями по созданию 3D сцен (Blender, Houdini, Katana, Unreal Engine и др.) через USD. Таким образом, использование Multiverse, как решение проблем интеграции, возможно только в том случае, если весь процесс создания 3D сцен построен вокруг ПО Maya. Пользователям других систем необходимо другое решение.

2.1.4 Резюме по существующим решениям

Все рассмотренные выше методы и технологии призваны в той или иной мере решать проблему «разных форматов» или «разного представления» данных, которую в некотором смысле можно назвать фундаментальной. Фундамент этой проблемы заключается в том, что большинство операции экспорта контента из редактора в рендер-систему долгие и ресурсоёмкие по определению. Поэтому ускорить или как-то упростить эти операции затруднительно. Например, загрузка большой текстуры с диска будет долгой практически при любых возможных оптимизациях, а преобразование сплайновых поверхностей в полигональную сетку требует времени и памяти, и является необходимым этапом экспорта т.к. большинство рендер-систем не умеют работать со сплайновыми поверхностями напрямую.

Очевидное решение заключается в том, чтобы не экспортировать повторно то, что уже было экспортировано хотя бы раз, а во время длительных процессов экспорта стараться не блокировать вызывающую программу редактора. Чтобы это было возможно, все изменения, вносимые в визуализируемую сцену, должны быть:

- явными,
- чётко выделенными,
- асинхронными (чтобы редактор мог реагировать на действия пользователя во время экспорта).

2.2 Общая архитектура и представление 3D сцены

Архитектура предлагаемого интегративного слоя может быть описана как объектная база данных с возможностями системы управления версиями и стратегией отсутствия перезаписи при внесении изменений. Рассмотрим данное определение по частям.

2.2.1 Объектная база данных

Пользователь в процессе создания 3D сцены наполняет её объектами разных типов – геометрическими моделями, источниками света, моделями материалов,

текстурами и др. Предлагаемое решение предоставляет программный интерфейс, позволяющий создавать новые объекты и изменять описание существующих. Набор из объектов и их взаимоотношений и составляет 3D сцену. Описание некоторого состояния сцены выглядит как один XML файл с названием «state_001.xml» в некоторой директории «myscene» плюс набор бинарных или текстовых файлов в произвольном формате, находящихся в директории «myscene/data».

Описание сцены внутри XML файла состоит из библиотеки данных и набора собственно сцен. Библиотека данных подразделяется на библиотеки по типам компонентов:

- текстуры,
- модели материалов,
- модели источников света,
- геометрические модели,
- виртуальные камеры,
- настройки рендер-систем.

Внутри определенной библиотеки содержится XML описание параметров объекта соответствующего типа в виде атрибутов и узлов. В качестве значений параметров объекта могут быть ссылки на внешние файлы, содержащиеся в директории «data». Таким образом, XML-описание не хранит в себе крупные объекты и бинарные данные, такие как текстуры и геометрические модели, сами по себе, а ссылается на них по имени файла.

Листинг 1 демонстрирует пример XML-описания геометрической модели (строки 3-13) внутри библиотеки геометрических моделей. В атрибуте «loc» (2 строка) задана ссылка на файл модели, остальные атрибуты в узле «mesh» содержат различные характеристики модели – формат файла («type»), размер файла («bytesize»), число вершин («vertNum»), число треугольников («triNum»), параллельный осям ограничивающий параллелепипед («bbox») и другие. В дочерних узлах (строки 6-11) представлены атрибуты – позиции («positions»),

нормали («normals»), касательные («tangents»), текстурные координаты («texcoords»), индексы вершин («indices»), индексы материалов («matindices»). Для каждого из атрибутов задан тип данных («type»), размер массива атрибутов («bytesize»), сдвиг относительно начала файла («offset»), тип атрибута («apply») – например, вершинный («vertex») или по примитивам («primitive»).

```
1. <geometry_lib total_chunks="8">
2. …
3. <mesh id="1" name="Teapot01" type="vsgf" bytesize="3379192"
4.       loc="data/chunk_00002.vsgf" offset="0" vertNum="53028"
5.       triNum="25600" dl="0" path="" bbox="-0.9525 1.0903 0 1.00013 -0.635
0.635">
6. <positions  type="array4f" bytesize="848448" offset="24" apply="vertex" />
7. <normals    type="array4f" bytesize="848448" offset="848472" apply="vertex" />
8. <tangents   type="array4f" bytesize="848448" offset="1696920" apply="vertex" />
9. <texcoords  type="array2f" bytesize="424224" offset="2545368" apply="vertex" />
10. <indices    type="array1i" bytesize="307200" offset="2969592" apply="tlist" />
11. <matindices type="array1i" bytesize="102400" offset="3276792"
12.             apply="primitive" />
13. </mesh>
14. </geometry_lib>
```

Листинг 1: Пример XML-описания для геометрической модели. Формат бинарного файла меша «chunk_00002.vsgf» описан в дочерних узлах «positions», «normals» и др., для которых задан тип данных, размер и смещение относительно начала файла.

Объекты в XML-описании могу также ссылаться и друг на друга. В листинге 2 представлен примеры описания текстуры (строки 3-5) и модели материала (строки 10-25), ссылающегося на неё. Текстура в своем атрибуте «loc» ссылается на место расположения файла текстуры во внутреннем формате после её добавления в библиотеку текстур, а в атрибуте «path» на исходное расположение файла изображения, который был задан пользователем в клиентском приложении. Материал, представленный в примере, состоит из двух ДФР – диффузной («diffuse», строки 11-18) и отражающей («reflectivity». строки 19-24). В диффузной составляющей узел цвета («color», строка 12) содержит узел-потомок - текстуру («texture», строка 13). Этот XML-узел текстуры внутри материала по сути представляет собой некоторой аналог так называемого сэмплера (sampler) в современных графических API (Vulkan, OpenGL и др.) и в данном примере содержит информацию о:

- к какой текстуре происходит обращаться – атрибут «id» указывает на идентификатор текстуры в библиотеке текстур (т.е. на атрибут «id» в описании самого объекта текстуры, строка 3);
- текстурной матрице, применяемой к текстурным координатам («matrix»);
- способу обработки текстурных координат («addresing_mode_v», «addressing_mode_u»);
- параметре гамма-коррекции («input_gamma»);
- альфа-канале («input_alpha»).

```
1. <textures_lib total_chunks="832">
2. …
3. <texture id="1" name="texture1.png" path="/home/user/textures/texture1.png"
4.  loc="data/chunk_00001.image4ub" offset="8" bytesize="262144" width="256"
5.  height="256" dl="0" />
6. …
7. </textures_lib>
8. …
9. <materials_lib>
10. <material id="23" name="Material #126" type="hydra_material">
11.  <diffuse brdf_type="lambert">
12.    <color val="1 1 1">
13.      <texture id="1" type="texref" matrix="1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1"
14.       addressing_mode_u="wrap" addressing_mode_v="wrap"
input_gamma="2.20000005"
15.       input_alpha="rgb" />
16.    </color>
17.    <roughness val="0" />
18.  </diffuse>
19.  <reflectivity brdf_type="phong">
20.    <color val="0.5 0.5 0.5" />
21.    <glossiness val="0.5" />
22.    <fresnel val="1" />
23.    <fresnel_ior val="1.33000004" />
24.  </reflectivity>
25. </material>
26. </materials_lib>
```

Листинг 2: Пример ссылки одного объекта в XML-описания на другой – модель материала ссылается на текстуру.

Остальные типы объектов в библиотеках XML-описания сцены выглядят подобным образом. Следует отметить, что предлагаемое решение предоставляет интерфейс для задания произвольных узлов, атрибутов и их возможных значений

в соответствии со стандартом XML. Пользователи промежуточного программного слоя – разработчики рендер-системы самостоятельно могут определить конкретный поддерживаемый ими набор узлов, атрибутов и их значений.

Кроме библиотек объектов, XML-описание содержит также набор сцен (листинг 3). В данном случае сцена представляет из себя набор ссылок (в англоязычной терминологии – instance, инстансов) на геометрические объекты (строки 8-15 и 20-21) и источники освещения (строки 16-19), находящихся в библиотеке данных. Каждая такая ссылка содержит идентификатор объекта (атрибут «mesh_id») и матрицу преобразования (атрибут «matrix»), задающую перенос, поворот, коэффициенты масштабирования, определяющую местоположение данной копии объекта в сцене. Другими словами, данная матрица задает преобразование из системы координат модели/объекта в мировую систему координат. Также, описание сцены может содержать списки переназначения материалов, которые определяют отображение из индексов материалов, заданных для геометрической модели (обычно в виде атрибута, заданного на каждый полигон), в новый список индексов материалов. Инстансы могут ссылаться на эти списки (атрибут «rmap_id»), что позволяет создавать копии одного объекта с разными материалами.

```
1.  <scenes>
2.   <scene id="0" name="my scene" discard="1" bbox=" -1.27 1.27 -1.27 1.27 -1.27
3.    1.40939">
4.     <remap_lists>
5.       <remap_list id="0" size="6" val="0 4 1 5 2 3 " />
6.       <remap_list id="1" size="2" val="0 6 " />
7.     </remap_lists>
8.     <instance id="0" mesh_id="0" rmap_id="0" scn_id="0" matrix="1 0 0
9.      0 0 1.151 0 -1.044 0 0 1 0 0 0 0 1 " />
10.    <instance id="1" mesh_id="1" rmap_id="1" scn_id="0"
11.     matrix="0.999 0 -0.0149 0 0 1 0 -1.27 0.0149 0 0.999 0 0 0 0 1" />
12.    <instance id="2" mesh_id="2" rmap_id="1" scn_id="0" matrix="1 0 0
13.     0.985 0 1 0 -1.27 0 0 1 0.512 0 0 0 1 " />
14.    <instance id="3" mesh_id="3" rmap_id="-1" scn_id="0" matrix="1 0 0
15.    -0.826 0 1 0 -1.27 0 0 1 0.809 0 0 0 1 " />
16.    <instance_light id="0" light_id="0" matrix="1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
1"
17.     lgroup_id="-1" />
18.    <instance_light id="1" light_id="1" matrix="1 0 0 0 0 1 9.313 1.244 0 -
9.313 1
```

```
19.      0 0 0 0 1 " lgroup_id="-1" />
20.     <instance id="6" mesh_id="5" rmap_id="-1" matrix="1 0 0 0 0 1 9.313 1.244 0
21.     -9.313 1 0 0 0 0 1 " light_id="1" linst_id="1" />
22.   </scene>
23. </scenes>
```

Листинг 3: Пример XML-описания сцены. Каждый инстанс ссылается на геометрическую модель («mesh_id»), список переназначения материалов («rmap_id») и исходную сцену («scn_id»). Исходная сцена может отличаться от текущий если используется функциональность объединения нескольких сцен вместе.

В примере, представленном в листинге 3, инстанс с идентификатором 0 (строки 8-9) ссылается на список переназначение также с идентификатором 0. Данный список (строка 5) содержит значения (атрибут «val») равные «0 4 1 5 2 3». Это означает, что в данной копии геометрической модели с идентификатором 0 (атрибут «mesh_id») материал с идентификатором в библиотеке материалов равным 0 будет заменен на материал с идентификатором 4, материал с идентификатором 1 на материал 5, а материал 2 на материал 3. Источники света могут иметь ассоциированную с ними геометрическую модель – в листинге 3 последний инстанс (строки 20-21) через атрибут «linst_id» ссылается на инстанс источника света с идентификатором в 1.

Следует отметить, что ни XML файл с описанием объектов сцены, ни сами объекты не обязаны в действительности находиться на диске. В момент передачи из клиентского приложения в рендер-систему большинство из них могут находиться в разделяемой памяти ОС (рис. 8).

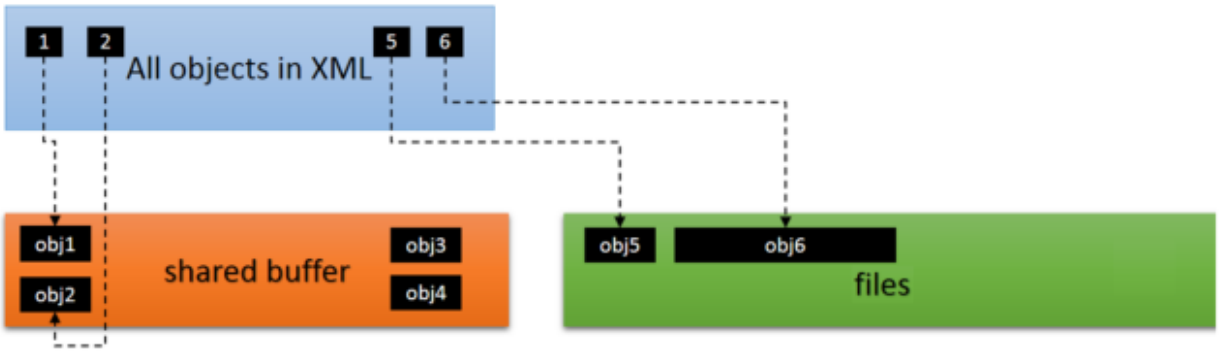


Рис. 8 Схема хранения объектов в промежуточном слое. Часть объектов может храниться на диске («files»), а другая часть - в кэше на основе разделяемой памяти ОС («shared buffer»).

2.2.2 Работа со сценой как управление версиями

Работа с 3D сценой с точки зрения разработанного программного интерфейса выглядит подобно системе управления версиями git (рис. 9).

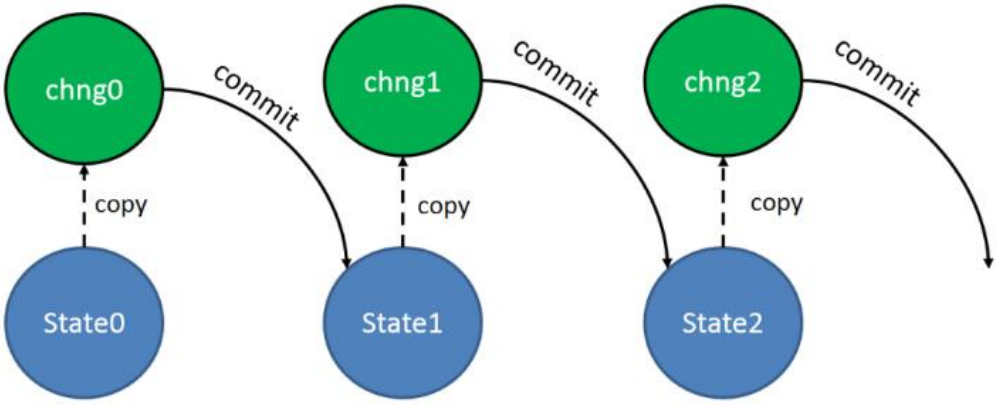


Рис. 9 Процесс работы с 3d сценой со стороны интеграционного слоя.

Новая сцена исходно имеет пустое состояние («State0» на рис. 9). В процессе работы над сценой пользователь добавляет новые и редактирует существующие объекты. Подобными действиями может быть изменено или создано любое количество любых объектов. Все эти действия накапливаются в XML-файле изменений («chng0» на рис. 9). При этом не требуется, чтобы сам файл был обязательно сохранен на диск. Для фиксации текущего состояния сцены и передачи его рендер-системе необходимо вызвать функцию *Commit*, которая подобна команде *git commit* и служит для фиксации изменений. Эта функция создаст новый файл состояния сцены («State1» на рис. 9), а рендер-системе будут переданы новые и измененные объекты и их параметры. Таким образом, новое состояние формируется из суммы предыдущего состояния и накопленных изменений на момент вызова *Commit*.

При дальнейшей работе над сценой процесс повторяется - новые изменения будут накапливаться в файлах изменений (верхняя строчка на рис. 9) до очередного вызова *Commit*, который будет создавать новые «замороженные» состояния сцены (нижняя строчка на рис. 9). Как правило, пользователь будет вызывать создание нового состояния каждый раз при вызове рендер-системы для синтеза некоторого

изображения. Таким образом, становится возможным отследить историю создания 3D сцены.

2.2.3 Хранение данных, стратегия отсутствия перезаписи

Для работы с компонентами 3D сцены большого размера, такими как геометрические модели и текстуры, предлагаемый промежуточный программный слой использует концепцию Append-буфера с бесконечным размером. Основные операции для такого буфера: добавление линейных блоков данных в конец (*append*), а также поиск и копирование блоков. При этом вновь добавленные данные, т.е. находящиеся в конце буфера, располагаются в оперативной памяти (рис. 10). Суммарно они занимают некоторое настраиваемое значение N мегабайт с конца буфера. Вся остальная часть буфера хранится на диске в виде набора файлов в директории *data*. При этом для объектов сцены, которые были вновь отредактированы пользователем, будет создан новый блок данных, который будет также помещен в конец буфера. А старая версия такого же объекта останется в буфере на прежнем месте. Подобное поведение отвечает принятой стратегии отсутствия перезаписи (no-overwrite).

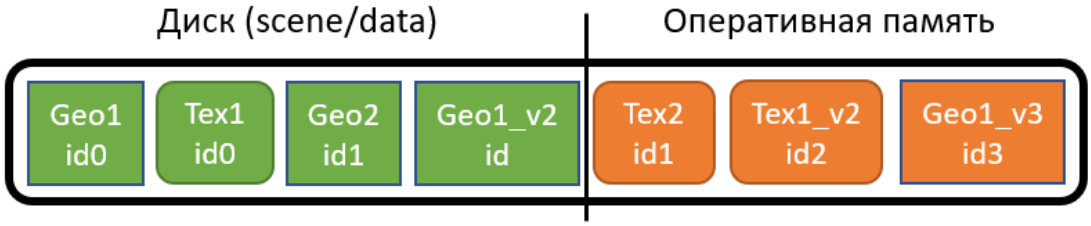


Рис. 10 Расположение объектов в Append-буфере.

В примере, показанном на рис. 10, геометрическая модель «Geo1», в процессе работы над сценой претерпела ряд изменений, при этом каждая последующая версия была сохранена в новые блок данных, в результате чего в буфере хранится три копии данной модели – «Geo1», «Geo1_v2», «Geo1_v3». При этом в XML описании сцены объект библиотеки геометрии будет содержать ссылку на последнюю версию на текущий момент времени. Более ранние версии так и будут существовать в буфере и по мере его наполнения будут вытеснены в часть,

располагающуюся на диске. Благодаря такой стратегии, объекты, с которыми работает пользователь в данный момент, почти всегда будут находиться в оперативной памяти. Это позволяет обеспечить сформулированное в первой главе требование интерактивности работы со сценой. Старые состояния геометрии и текстур, располагающиеся на диске, при этом либо уже были переданы рендер-системе ранее, либо уже не нужны, т.к. существует новая версия.

Стратегия отсутствия перезаписи, с одной стороны, обладает очевидным недостатком, заключающимся в перерасходе дисковой памяти на создание полных копий объектов. Но этот недостаток может быть в дальнейшем исправлен. Один из возможных вариантов - использование стратегии «Copy-On-Write» и разбиение больших буферов на страницы [74, с. 295]. Данный подход требует реализации на нижнем уровне, в частности в файловой системе вычислительных узлов компьютерной сети, и в целом является внешним по отношению к разработанной системе.

2.2.4 Распределенный рендеринг

Стратегия отсутствия перезаписи позволяет обеспечить надежную реализацию требования по поддержке распределенного (сетевого) рендеринга. Если разрешить перезапись любого файла, тогда существует возможность, при которой на одном из вычислительных узлов в сети этот файл будет иметь старое состояние, а на другой - новое. Стратегия отсутствия перезаписи гарантирует, что такого не может произойти никогда. Если файл данных для некоторого объекта существует, то он существует только в единственно возможном состоянии (т.к. любое изменение файла приведет к созданию новой копии). Если файла по какой-то причине нет на данном вычислительном узле, то возможно два варианта:

1. Файл ещё не был передан, рендер-система ожидает его передачи.
2. Файл не будет передан, поскольку уже существует новое состояние для того же самого объекта.

Во втором случае рендер-система переходит к следующему состоянию 3D сцены и ожидает необходимые для него данные.

При этом следует отметить, что сетевой фотореалистичный рендеринг может быть реализован эффективно с неограниченной масштабируемостью производительности, что было подтверждено моделированием, результаты которого были впервые опубликованы в работе [1].

Была построена модель, в которой вычислительные узлы объединяются в древовидную структуру (рис. 11). Корнем дерева является вычислительная машина, инициировавшая задачу рендеринга, «заказчик». В узлах располагаются машины, предоставляющие свои вычислительные ресурсы для решения задачи - «исполнители». Машина-заказчик отсылает вычислительную задачу своим соседям-исполнителям и те, в свою очередь, получив запрос, начинают выполнять расчеты. Исполнители далее также отсылают вычислительную задачу уже своим соседям-исполнителям, выступая для них в роли заказчика. После выполнения некоторого минимального объема вычислений, заданного в качестве исходного параметра машиной-инициатором всего процесса, и по запросу машины на более высоком уровне дерева, вычислительные узлы передают текущие результаты вычислений вверх по иерархии. Изображения, полученные от дочерних узлов и в результате локальных расчетов, объединяются в одно (при помощи усреднения сумм Монте-Карло выборок) в каждом узле графа и передаются на уровень выше. Таким образом, вышестоящая в иерархии машина “видит” каждого исполнителя как единую мощную машину, не имея информации о том, что на самом деле происходит ниже в иерархии.

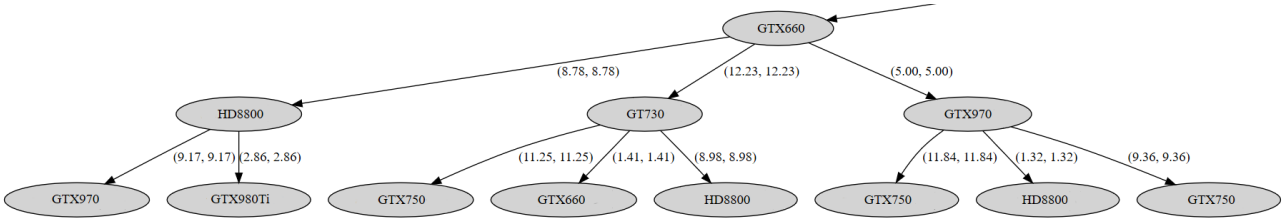


Рис. 11 Фрагмент примера модельной сети. В узлах расположены вычислительные машины с различными GPU, в качестве весов ребер заданы скорости передачи данных между узлами.

Один из экспериментов состоял в моделировании однородной сети с топологией бинарного дерева, фиксированными каналами связи с высокой

пропускной способностью (100 мегабит/сек. ≈ 12 мегабайт/сек.), одинаковой вычислительной мощностью узлов и довольно большим интервалом передачи обновлений. Моделирование проводилось до достижения 32768 случайных выборок на пиксель для разного числа вычислительных узлов – 31, 63 и 127 (глубина дерева 4, 5 и 6 соответственно). Для такого достаточно большого числа выборок на пиксель, являющегося типичным для расчета финального изображения в задаче фотореалистичного рендеринга, с достаточно большим интервалом отправления результатов (раз в 20 секунд), скорость расчета освещения соответствует идеальному сценарию, как если бы вся производительность сети была бы доступна на одной вычислительной машине (рис. 12). Есть лишь отставание, связанное с необходимостью передать сцену по сети.

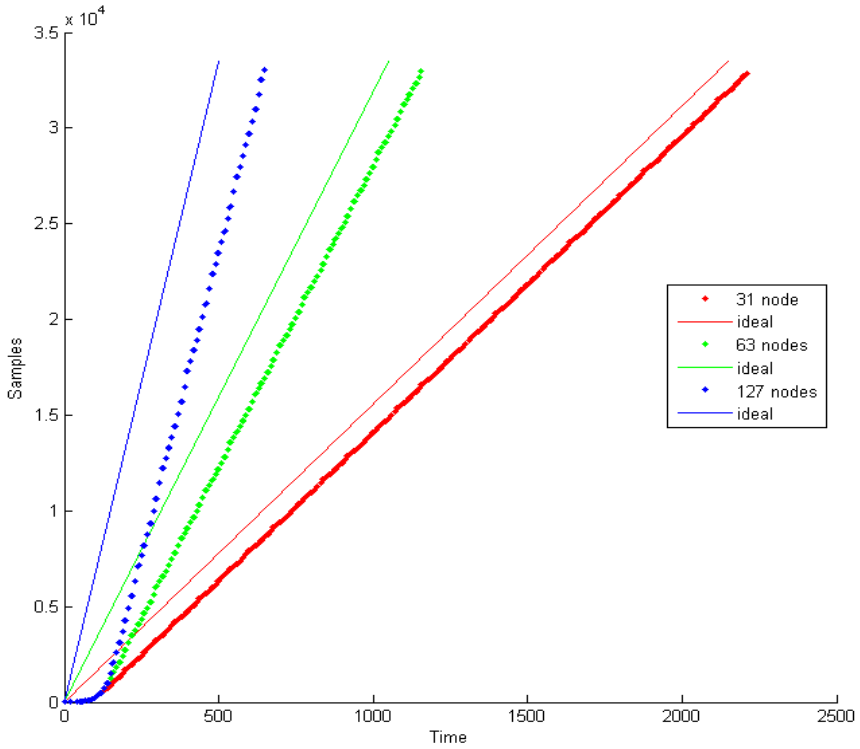
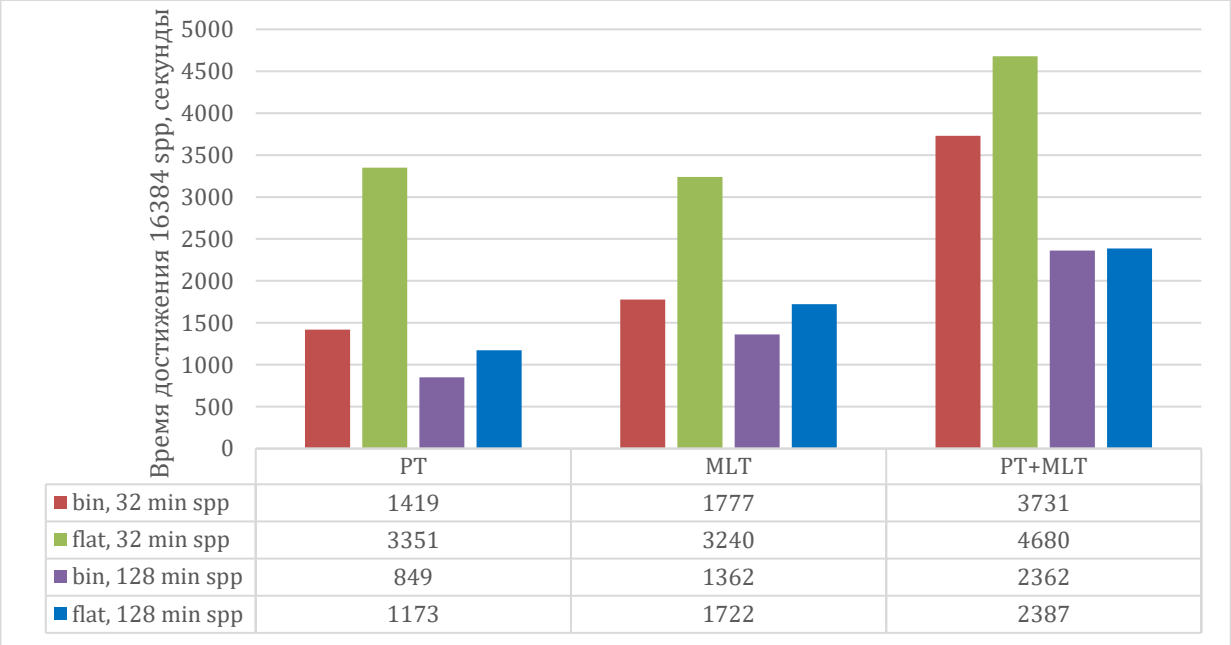


Рис. 12 Рост числа случайных выборок на пиксель для топологии бинарного дерева разной глубины. По оси X – время в секундах. По оси Y – число случайных выборок на пиксел. Графики предоставлены для сетей из 31 (красный), 63 (зелёный) и 127 (синий) вычислительных машин. Сплошные линии соответствуют производительности идеального случая, а точечные – моделированию. Размер передаваемой 3D сцены – 500 мегабайт, разрешение рендеринга 1920 на 1080, скорость расчета каждой машины - 0.5 выборок на пиксел в секунду.

Влияние сети на время расчета можно уменьшить, увеличив временной интервал между передачей промежуточных результатов. Если вычислительные узлы отправляют промежуточные результаты расчета каждые 32 случайные выборки на пиксель, то разница между бинарной топологией сети (красные столбцы на рис. 13) и плоской (зеленые столбцы на рис. 13) составляет порядка двух-трёх раз. При увеличении интервала отправки результатов до 128 случайных выборок на пиксель, разница между бинарной (фиолетовые столбцы на рис. 13) и плоской (синие столбцы на рис. 13) топологиями значительно уменьшается, что связано с уменьшением нагрузки на сеть. Когда по сети требуется передавать больше данных, как в случае расчета первичного освещения одним алгоритмом, а вторичного – другим, разница между топологиями практически нивелируется.



	PT	MLT	PT+MLT
bin, 32 min spp	1419	1777	3731
flat, 32 min spp	3351	3240	4680
bin, 128 min spp	849	1362	2362
flat, 128 min spp	1173	1722	2387

Рис. 13 Время достижения 16384 случайных выборок на пиксель для разных топологий, алгоритмов (MLT – Metropolis Light Transport, PT – Path Tracing) и разного интервала передачи промежуточных результатов. По оси Y – время в секундах. Красные столбцы - бинарная сеть из одинаковых машин, интервал передачи – 32 случайные выборки на пиксель. Зеленые столбцы – плоская сеть (все узлы подключены напрямую к машине-заказчику), интервал передачи – 32 случайные выборки на пиксель. Фиолетовые столбцы - бинарная сеть из одинаковых машин, интервал передачи – 128 случайных выборок на пиксель. Синие столбцы - плоская сеть, интервал передачи – 128 случайных выборок на пиксель. Каждая сеть состоит из 63 узлов и имеет суммарную производительность 31.5 случайных выборок на пиксел в секунду, Размер передаваемой 3D сцены – 500 мегабайт, разрешение рендеринга 1920 на 1080.

Таким образом, при обеспечении хорошей связи между вычислительными узлами возможно добиться практически идеальной скорости расчета по сравнению с несетевым решением. Кроме того, скорость расчета может быть увеличена за счет большего интервала между передачей промежуточных результатов расчетов по сети.

Для снижения задержек, связанных с передачей данных самой сцены по сети, в том числе в процессе итеративной работы с ней конечного пользователя (например, 3D художника), в предлагаемом решении был реализован механизм отслеживания изменений, описанный далее.

2.3 Интеграция клиентских приложений и работа с 3D сценой

Для формирования описания сцены разработанный промежуточный программный слой предоставляет API. Этот интерфейс включает 3 основных функции для каждого типа объектов сцены — *Create, Open* и *Close* (аналогично файловой системе ОС). Вызов *Create* создаёт новый пустой объект, вызов Open открывает объект для редактирования, и вызов *Close* сохраняет внесенные изменения.

При вызове *Open* указывается режим открытия объекта, предусмотрены три варианта:

- *HR_OPEN_EXISTING* – отрыть существующий непустой объект для редактирования;
- *HR_WRITE_DISCARD* – открыть новый (пустой) объект на редактирование или очистить непустой объект (удалить все параметры) перед открытием на редактирование;
- *HR_OPEN_READ_ONLY* – открыть объект для чтения параметров, любые сделанные изменения не будут применены.

Открытие объекта автоматически создаёт либо копию его текущих XML параметров (режим *HR_OPEN_EXISTING*), либо пустую копию (*HR_WRITE_DISCARD*). Далее пользователь работает с этой копией параметров, редактируя её тем или иным образом. При этом работа происходит с XML

представлением объекта, хранящимся в оперативной памяти. Задача редактирования XML описания решается с помощью функциональности, предоставляемой библиотекой pugixml [75]. После окончания работы с объектом необходимо закрыть его с помощью вызова *Close*. После этого новое состояние объекта считается “готовым к принятию” и после вызова функции *Commit* сформирует новое состояние сцены.

В листинге 4 представлен пример работы с объектом модели материала с использованием предложенного API, в листинге 5 представлено полученное XML-описание.

```
1.  HRMaterialRef mat = hrMaterialCreate(L"MyMaterial");
2.  hrMaterialOpen(mat, HR_WRITE_DISCARD);
3.  {
4.    xml_node matNode = hrMaterialParamNode(mat);
5.    xml_node diff = matNode.append_child(L"diffuse");
6.
7.    diff.append_attribute(L"brdf_type").set_value(L"lambert");
8.    diff.append_child(L"color").append_attribute(L"val") = L"1.0 1.0 0.0";
9.  }
10. hrMaterialClose(mat);
11. …
12. hrMaterialOpen(mat, HR_OPEN_EXISTING);
13. {
14.   xml_node matNode = hrMaterialParamNode(mat2);
15.   xml_node diff = matNode.child(L"diffuse");
16.
17.   diff.attribute(L"brdf_type").set_value(L"oren_nayar");
18.   diff.append_child(L"roughness").append_attribute(L"val") = 0.85;
19. }
20. hrMaterialClose(mat);
```

Листинг 4: Создание и редактирование модели материала с использованием предложенного API.

```
1.  <material id="0" name="MyMaterial" type="hydra_material">
2.   <diffuse brdf_type="lambert">
3.     <color val="1.0 1.0 0.0" />
4.   </diffuse>
5.  </material>
6. …
7.  <material id="0" name=" MyMaterial " type="hydra_material">
8.   <diffuse brdf_type="orennayar">
9.     <color val="1.0 1.0 0.0" />
10.     <roughness val="0.85" />
11.  </diffuse>
12. </material>
```

Листинг 5: XML-описание материала, созданное после выполнения кода в листинге 4. Строки 1-5 содержат исходное описание материала, строки 7-12 – после редактирования.

Сначала выполняется создание нового материала, при котором клиентский код получает уникальный идентификатор «mat» для дальнейшей работы с этим материалом (листинг 4, строка 1), и созданный материал открывается на редактирование в режиме *HR_WRITE_DISCARD* (листинг 4, строка 2). Далее необходимо получить XML узел описания материала, для чего используется вызов *hrMaterialParamNode*, принимающий уникальный идентификатор материала (листинг 4, строка 4). После чего происходит создание и редактирование XML узлов и атрибутов с использованием функций библиотеки pugixml (листинг 4, строки 5-9). Произведенные изменения финализируются при закрытии объекта (листинг 4, строка 11). В последующем коде (листинг 4, строки 13-21) объект материала снова открывается на редактирование, но уже в другом режиме, не производящем очистку ранее созданных параметров, которые затем изменяются.

Схема работы с параметрами выглядит в целом аналогично для других типов объектов. Для геометрических моделей и текстур также предоставляется интерфейс для загрузки данных – из некоторого буфера в памяти или из файла на диске. При этом для файлов на диске предусмотрена возможность отложенной загрузки. Это означает, что данные будут загружены в оперативную память только в момент передачи их рендер-системе. Таким образом, если добавленные в сцену текстура или геометрия окажутся не нужны для рендеринга текущего кадра, то они вообще не будут загружены.

Для сохраненного на диске XML описания 3D сцены допустимо и ручное редактирование при возникновении такой необходимости. Например, для решения задач отладки возможно удалить из описания какие-либо объекты или отредактировать их параметры.

Отличительной и важной чертой предлагаемой технологии является явное отслеживание и явная запись изменений. Таким образом, если текущее состояние объекта хранится в файле описания сцены «state_001.xml», тогда новое состояние объекта будет хранится в отдельном файле, содержащим изменения «change_001.xml».

В процессе работы над сценой пользователь может изменить любое количество любых объектов с использованием описанного программного интерфейса, в том числе изменить состояние одного объекта несколько раз. Список изменений пополняется при каждом вызове функции *Close* для объекта сцены. Этот список отражается в файле изменений («change_001.xml»), содержащим список всех изменённых и новых объектов с полным описанием их параметров. Объекты, которые со времени фиксации текущего состояния («state_001.xml») не изменились, не будут скопированы в файл изменений. Сами XML-файлы при этом не обязаны быть сохранены на диск и могут располагаться в оперативной памяти в динамическом представлении.

Для передачи произведенных изменений рендер-системы, пользователь вызывает операцию *Commit*, которая сформирует новое состояние в файле «state_002.xml». Оно будет содержать:

1. объекты из предыдущего состояния «state_001.xml», которые не подверглись изменениям,
2. новые объекты из «change_001.xml»,
3. изменённые объекты из «change_001.xml», которые заместят свои оригиналы из «state_001.xml».

Чтобы сформировать это содержание выполняется алгоритм 1, объединяющий объекты из предыдущего состояния и новые объекты с учетом их зависимостей.

Алгоритм 1. Формирование нового состояния сцены

Входные данные:

$\widetilde{M}, \tilde{B}, \tilde{L}, \tilde{T}$ – множества идентификаторов объектов сцены (мешей, материалов, источников света, текстур соответственно), переданных рендер-системе ранее, при формировании предыдущего состояния, если это первое формируемое состояние, то $\widetilde{M} = \emptyset, \tilde{B} = \emptyset, \tilde{L} = \emptyset, \tilde{T} = \emptyset$;

$\overline{M}, \overline{B}, \overline{L}, \overline{T}$ – множества идентификаторов измененных объектов сцены (мешей, материалов, источников света, текстур соответственно), полученные в результате вызовов *Open*/*Close* для объектов сцены;

I_M – множество всех экземпляров («инстансов») мешей в 3D сцене, каждый элемент множества представляет собой кортеж $\{id, \mathcal{M}, mat\_ids\}$, где id – уникальный идентификатор соответствующего объекта (меша), $\mathcal{M}$ – матрица трансформации, mat_ids – массив идентификаторов материалов, используемых данным экземпляром;

I_L – множество всех экземпляров («инстансов») источников света в 3D сцене, каждый элемент множества представляет собой кортеж $\{id, \mathcal{M}\}$, где id – уникальный идентификатор соответствующего объекта (источника света), $\mathcal{M}$ – матрица трансформации;

Выходные данные:

M, B, L, T – множества идентификаторов объектов сцены (мешей, материалов, источников света, текстур соответственно) в новом состоянии;

$drawSeq$ – отображение идентификатора меша на список его экземпляров («инстансов») в сформированном состоянии.

```
1  M_new = M̄, B_new = B̄, L_new = L̄, T_new = T̄
2  drawSeq = ∅
3  foreach i ∈ I_M do:
4      if i.id ∉ M̃ then:
5          M_new = M_new ∪ {i.id}
6      end if
7      drawSeq = drawSeq ∪ {i.id → i}
8      B_new = B_new ∪ {j | j ∈ i.mat_ids}
9  end for
10 foreach l ∈ I_L do:
11     if l.id ∉ L̃ then:
12         L_new = L_new ∪ {l.id}
13     end if
14 end for
15 foreach m ∈ M_new do:
16     Q = GetMaterialsUsedByMesh(m)
17     B_new = B_new ∪ {j | j ∈ Q}
18     /* Материал может быть сложным и состоять из нескольких дочерних материалов,
       каждый из которых, в свою очередь, также может быть сложным. */
```

```
    foreach j ∈ Q do:
        B_new = B_new ∪ {k | k ∈ GetAllChildMaterialsRecursive(j)}
    end for
end for
foreach b ∈ B_new do:
    T_new = T_new ∪ {j | j ∈ GetTexturesUsedByMaterial(b)}
end for
foreach l ∈ L_new do:
    T_new = T_new ∪ {j | j ∈ GetTexturesUsedByLight(l)}
end for
M = M̃ ∪ M_new, B = B̃ ∪ B_new, L = L̃ ∪ L_new, T = T̃ ∪ T_new
return M, B, L, T, drawSeq
```

Множества M, B, L, T отражают следующее состояние сцены («state_002.xml») внутри рендер-системы после операции *Commit*. При этом следует ещё раз отметить, что рендер-системе будут переданы только новые и измененные объекты, идентификаторы которых содержатся в множествах $M_{new}, B_{new}, L_{new}, T_{new}$, и которым соответствует файл изменений «change_001.xml».

2.4 Интеграция рендер-систем

Все изменения передаются в рендер-систему при выполнении операции *Commit*. Поэтому интеграционный слой имеет возможность выполнить упорядочивание изменений перед их передачей. Вне зависимости от того, в каком порядке пользователь создавал и редактировал объекты 3D сцены, рендер-система получит их всегда в одном и том же строго определенном порядке. Это позволяет в значительной мере упростить интеграцию рендер-системы с предлагаемым решением, т.к. она может полагаться на заранее известную и предопределенную последовательность вызовов от интеграционного слоя независимо от того, в каком порядке производятся вызовы API в прикладном приложении. Упрощённая

иллюстрация данного процесса изображена на рис 14. Следует отметить, что упорядочивание вызовов означает с одной стороны, что функции интерфейса рендер-системы для передачи данных от промежуточного слоя (рис. 14, справа) будут вызваны в определенном порядке. А с другой стороны, что каждая из этих функций, например, «UpdateMaterial» для передачи материала, будет вызвана в порядке возрастания идентификатора объекта, – сначала для материала 1, затем для материала 2 и т.д. Такой подход позволяет работать со сложносоставными объектами, например, деревьями материалов: сначала будут переданы данные для листьев, а затем уже для корня. Идентификатор корневого материала всегда будет больше идентификаторов листьев по построению – создать материал, смешивающий несколько других можно только же имея идентификаторы смешиваемых материалов.

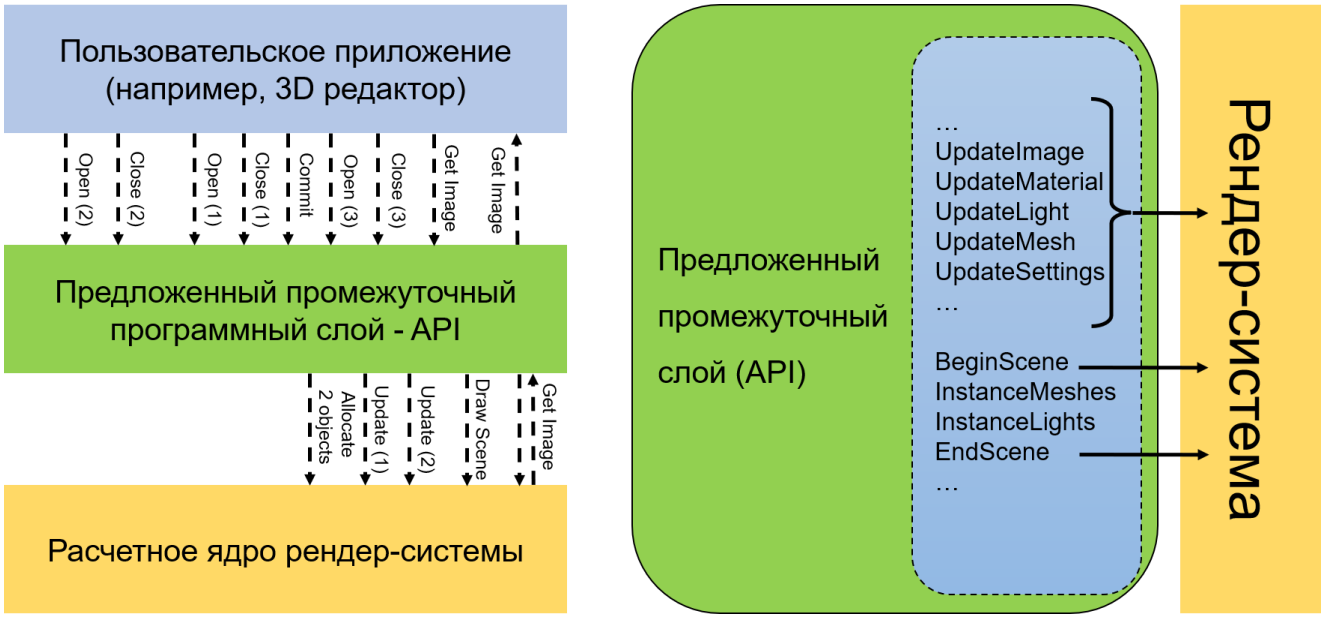


Рис. 14 Слева – упрощенная схема взаимодействия пользовательского приложения, разработанного API и расчетного ядра рендер-системы. В скобках цифрами показан идентификатор объекта, - в независимости от того, как пользователь работает со сценой, рендер-система получит объекты в порядке возрастания идентификатора. Справа – некоторые основные функции в интерфейсе рендер-драйвера в порядке их вызова промежуточным слоем.

Для подключения некоторой рендер-системы к интеграционному слою, она должна определить свою реализацию определенного в API интерфейса для рендер-систем путем публичного наследования и «зарегистрироваться» в интеграционном

слое. Интерфейс для рендер-систем в разработанном API называется «рендер-драйвером». При этом рендер-система может определить только те методы интерфейса, которые отвечают поддерживаемым ею возможностям. Интерфейс рендер-драйвера включает следующие группы методов:

1. загрузка и обновление компонентов библиотеки сцены - геометрия, модели материалов, текстуры, источники света, виртуальные камеры, настройки рендер-системы;
2. инстанцирование (т.е. расстановку копий объектов) геометрии и источников света;
3. методы, вызываемые непосредственно перед и после обновления того или иного вида компонентов библиотеки и инстанцирования;
4. выполнение отрисовки текущего состояния сцены;
5. запрос на передачу из рендер-системы синтезированного изображения в LDR (англ. low dynamic range, узкий динамический диапазон) или HDR (англ. high dynamic range, широкий динамический диапазон) формате;
6. запрос списка доступных устройств для осуществления рендеринга и «включения» определенного устройства;
7. передача рендеру некоторой произвольной команды, заданной в виде строки;
8. некоторые дополнительные методы.

Из приведенного списка методы из 1-3 групп вызываются интеграционным слоем в четко определенной последовательности в процессе передачи изменений. Остальные методы косвенно доступны пользователям интеграционного слоя (например, приложению 3D редактора) через набор функций самого интеграционного слоя.

Регистрация рендер-системы в интеграционном слое производится путем передачи в специальный фабричный класс *RenderDriverFactory*, определенный в интеграционном слое, структуры, содержащей информацию о поддерживаемой рендер-системой функциональности, указатель ну функцию создания экземпляра реализации рендер-драйвера и имя рендер-системы.

2.5 Механизмы отладки и тестирования

Разделение рендер-системы и клиентского приложения с помощью промежуточного интеграционного слоя открывает ряд дополнительных возможностей для отладки и тестирования. При такой программной архитектуре, рендер-система перестает каким-либо образом зависеть от особенностей клиентских приложений. Разработчикам рендер-системы достаточно знаний о последовательности вызовов, передаваемых из промежуточного слоя, для решения задачи интеграции.

В приложениях компьютерной графики, использующих такие графические API как OpenGL, DirectX, Vulkan и др., широко используется подход на основе отслеживания вызовов API при отладке приложений [76]. Суть данного подхода состоит в том, чтобы записать последовательность вызовов API, а затем воспроизвести её, возможно на другом устройстве или даже под управлением другой ОС [77, 78]. Однако в этом случае не происходит никакого упорядочивания вызовов. В случае же предлагаемого в данной работе решения, благодаря наличию упорядочивания появляется возможность создавать раздельные наборы тестов для интеграционного слоя и рендер-систем. Первый набор проверяет «вход», т.е. функциональность самого API и интеграционного слоя. А второй набор тестирует «выход», т.е. функциональность рендер-системы.

Кроме того, подобный подход к тестированию отличается и от других способов тестирования систем компьютерной графики, в которых реализуется запись и воспроизведение действий пользователя в конкретном клиентском приложении как, например, в работе [79]. Разработанное решение, благодаря архитектуре промежуточного программного слоя, записывает только «вход», поступающий из клиентского ПО в API программного слоя, что позволяет тестировать рендер-систему (или несколько рендер-систем на одних и тех же входных данных) без привязки к конкретному ПО.

В предлагаемом решении для тестирования функциональности интеграционного слоя возможно проверять корректность полученных XML

описаний состояний и списков изменений для той иной сцены, созданной в клиентском приложении. Другая возможность состоит в использовании отладочных рендер-драйверов. В составе интеграционного слоя уже реализовано несколько специальных рендеров, служащих для задач отладки - текстовый рендер-драйвер и ряд рендер-драйверов на основе OpenGL 1.x. Текстовый рендер-драйвер позволяет «распечатать» в текстовом формате содержимое экспортированных в API геометрических моделей и тем самым проверить его на правильность формата, наличие нужных данных и др. Базовый OpenGL рендер-драйвер позволяет произвести простую визуализацию сцены, что позволяет оценить наличие всех экспортированных геометрических объектов и их положение в мировой системе координат. Модификации этого базового рендера позволяют визуализировать дополнительную информацию – направления осей координат, векторы нормалей и касательных к вершинам полигональных моделей и др. Данные отладочные рендер-драйверы могут быть использованы, например, для проверки корректности экспорта геометрических данных сцены из клиентского приложения. Другой сценарий использования связан с предоставлением механизма визуализации и тестирования для разрабатываемого клиентского приложения – 3D редактора, реализации различных геометрических алгоритмов и т. д.

Для тестирования функциональности рендер-системы одним из возможных решений является загрузка и рендеринг описаний простых сцен при помощи предлагаемого API. После чего производится попиксельное сравнение полученного изображения с изображением-эталоном и рассчитывается среднеквадратическая ошибка (MSE, mean squared error) или другая метрика. Если ошибка не превышает некоторого предопределенного порога, то тест считается пройденным. Порог при этом выбирается вручную при составлении теста в зависимости от алгоритмов расчета в рендер-системе и их настроек. Для тестирования рендер-систем также могут использоваться и специализированные отладочные рендер-системы интеграционного слоя (рис. 15). В частности, доступны рендер-драйверы для визуализации ускоряющей структуры BVH (bounding volume hierarchy, иерархия ограничивающих объемов) и для

визуализации сохраненных лучей/путей. При этом рассмотренный ранее механизм «подключения» рендер-системы к интеграционному слою позволяет разработчикам с легкостью создавать новые варианты отладочных рендер-систем.

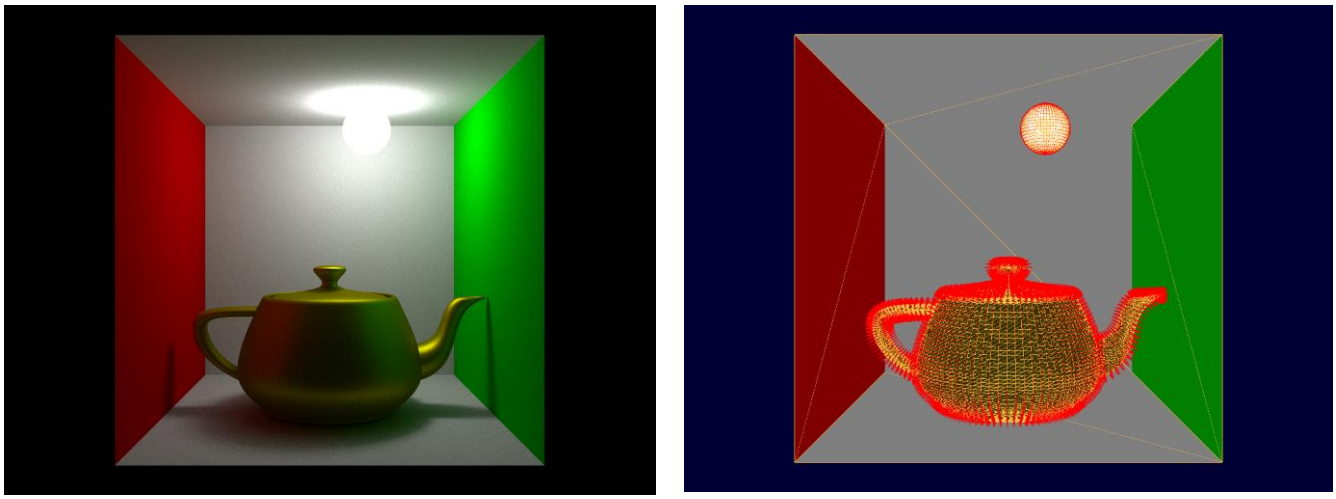


Рис. 15 Пример работы отладочной рендер-системы. Слева - фотореалистичный рендеринг, справа - отладочный рендеринг той же сцены.

2.6 Сравнение и выводы

2.6.1 Существующие рендер-системы

Если рассмотреть существующие рендер-системы и какие решения для интеграции используются в них, то большинство индустриальных решений для задач фотореалистичного рендеринга (кино- и анимационные фильмы, архитектурная и интерьерная визуализация и др.), имеют закрытый исходный код и могут предоставлять API и/или SDK (software development kit, «комплект для разработки программного обеспечения») для самого рендера, позволяющий выполнять разработку расширений, используя механизм плагинов [80]. Кроме того API/SDK может также не иметь свободного распространения [81], поэтому сложно оценить используемые внутри решения.

Рассмотрим распространенные рендер-системы с открытым исходным кодом.

Открытая рендер-система PBRT, являющаяся эталонной реализацией подходов, описанных в одноименной книге [33], и ориентированная на исследования в области алгоритмов и моделей фотореалистичного рендеринга,

принимает на вход текстовое описание сцены в собственном формате. Таким образом, пользователи данной системы должны самостоятельно составлять описание сцены, т.е. реализовывать собственное решение для её интеграции с тем или иным приложением. Для расширения функциональности системы пользователь может сделать собственную реализацию того или иного абстрактного класса объектов (геометрической модели, камеры, расчетного алгоритма и др.). Что во многом соответствует описанному ранее механизму плагинов.

Другая открытая рендер-система, ориентированная на исследовательские задачи в компьютерной графике - Mitsuba [82]. В данной системы описание сцены может быть выполнено в виде XML-файла или структуры данных dict языка Python. При этом готовых инструментов для создания такого описания нет и подразумевается, что пользователь самостоятельно будет его формировать любыми удобными средствами. Для расширения функциональности рендер-системы реализован механизм динамически загружаемых плагинов для различных компонентов 3D сцены и алгоритмов (синтеза изображения, генерации случайных выборок и др.).

В случае рендер-системы научной визуализации Intel OSPRay [83], можно отметить наличие некоторого сходства с разработанным подходом. В ней API также предоставляет функции для создания объектов различных типов (модели материалов, геометрические модели, камеры и пр.). Используя полученную ссылку на объект пользователь может изменять объект. Изменения фиксируются вызовом функции Commit для данного объекта, которая выступает аналогом функции *Close* в предлагаемом решении. При этом операция *Commit* для всей сцены отсутствует. Текущее состояние сцены меняется при вызове *Commit* для каждого отдельного объекта. Поскольку OSPRay является CPU рендер-системой, нет необходимости выполнять дорогостоящую операцию передачи данных на графический процессор. Поэтому текущее состояние сцены доступно для рендеринга в любой момент. При этом документация отмечает, что после запуска процесса рендеринга, недопустимо выполнять операцию *Commit* для объектов, т.к. это ведет к неопределенному поведению. Таким образом, в OSPRay создание сцены идет «непрерывным»

способом, без фиксации промежуточных состояний и изменений. Что не позволяет получить связанные с отслеживанием состояния сцены преимуществ, имеющихся в разработанном решении, таких как упрощение отладки и тестирования.

Другая отличительная особенность системы OSPRay состоит в более низкоуровневом управлении компонентами сцены. Например, пользователь самостоятельно создает объект буфер кадра, самостоятельно его очищает при необходимости. Это позволяет в том числе смешивать результаты работы разных рендеров в составе OSPRay, например, основной визуализации и вспомогательной геометрии. Это с одной стороны дает пользователю больше возможностей для настройки процесса рендеринга, а с другой стороны приводит к тесной интеграции рендеров и API и практически не позволяет отделить рендер-систему от API.

Ещё один пример низкоуровневого управления сценой в OSPRay состоит в имеющейся возможности создания в качестве объектов сцены отдельных массивов данных, которые затем могут использоваться, например, разными геометрическими моделями. Это в первую очередь связано с ориентацией системы на научную визуализацию, где единый набор данных может быть визуализирован различными методами. А поскольку наборы данных могут быть очень большими необходимо избегать их дублирования. Однако, это влечет за собой также и необходимость ручного освобождения ресурсов, для чего в API предоставляется функция *Release*.

Специальных механизмов для создания расширений не предусмотрено, однако относительно низкоуровневый API позволяет в некоторых пределах адаптировать рендер-систему под нужды пользователя.

В целом, предлагаемый API имеет определенное внешнее сходство с API рендер-системы OSPRay, однако на более глубоком уровне присутствуют принципиальные отличия. Реализация OSPRay соответствует ориентации этой рендер-системы на использование CPU и задачу научной визуализации.

Таким образом, для существующих распространенных открытых рендер-систем обычно отсутствуют специальные механизмы создания 3D сцен и интеграции со сторонними приложениями. Исключением является CPU рендер-

система OSPRay, которая предоставляет многофункциональный API, ориентированный на конкретную предметную область – научную визуализацию.

В качестве способа расширения возможностей существующими открытыми рендер-системами обычно используется механизм динамически загружаемых плагинов или его вариант, где пользователь добавляет новые классы непосредственно в код рендер-системы (как в PBRT).

Индустриальные решения могут предоставлять API, однако детали его реализации и границы использования неизвестны в виду закрытости данных рендер-систем.

2.6.2 Сравнение с существующими решениями

Предлагаемое решение обеспечивает все требования, сформулированные в первой главе:

1. Интеграционный слой может использоваться как с CPU, так и с GPU рендер-системами.
2. Расширяемость достигается за счет хранения описания самих объектов в динамическом формате (XML). Предлагаемая программная архитектура позволяет легко добавлять новые параметры или модели – достаточно создать новые узлы или атрибуты в XML. Внесения каких-либо изменений в код программного слоя не требуется. Остается только поддержать эти расширения в самой рендер-системе.
3. Интерактивность обеспечивается механизмом экспорта и передачи только изменений, сделанных в сцене, а также концепции append-буфера. Отслеживание изменений позволяет предотвратить повторный экспорт неизменившихся объектов и уменьшить объем информации, передаваемый рендер-системе. А механизм append-буфера хранит объекты, с которыми пользователь работает в данный момент, в оперативной памяти.
4. Также append-буфер обеспечивает отсутствие ограничений на память, сбрасывая на диск старые объекты.

5. Представление сцены во внутреннем формате интеграционного слоя позволяет обмениваться данными между всеми приложениями, поддерживающими этот формат (через API интеграционного слоя или напрямую).
6. Механизм подключения рендер-систем через интерфейс рендер-драйвера и упорядочивание вызовов этого интерфейса со стороны API интеграционного слоя позволяет тестировать рендер-систему и интеграцию с клиентским приложением без зависимости друг от друга. Разработано несколько отладочных рендер-драйверов для отслеживания корректности экспорта геометрических данных, а также отладки рендер-систем. Механизм передачи изменений позволяет отследить историю создания сцены пользователем и тем самым облегчить поиск места появления возможных ошибок и проблем.
7. Стратегия отсутствия перезаписи гарантирует то, что при распределенном рендеринге все вычислительные узлы будут работать с одинаковыми копиями компонентов сцены. Для алгоритмов фотореалистичного рендеринга было проведено исследование, подробно описанное в [1], показавшее перспективность распределенного рендеринга на GPU.
8. Для осуществления интеграции предоставляются два отдельных интерфейса для рендер-систем и клиентских приложений. Интеграция рендер-системы к предложенному промежуточному слою сразу дает ей доступ ко всем клиентским приложениям, интегрированным с ним и наоборот подобно случаю простого обмена файлами.

Среди существующих решений ни один подход не обеспечивает выполнения всех требований (обзор проведен в начале главы, общие результаты представлены в табл. 1). Наиболее близкой к предлагаемому решению является система USD от компании Pixar. Однако, это специализированная тяжеловесная система, ориентированная в первую очередь на задачи индустрии кино- и анимационных фильмов. Её основным недостатком является значительно более высокая

сложность использования для интеграции с существующими рендер-системами и другими приложениями компьютерной графики.

Требование	Обмен файлами	Динамические плагины	Pixar USD/ Multiverse	Предлагаемое решение
Скорость / GPU	+	-	+	+
Гибкость/расширяемость	+	+	+	+
Интерактивность	-	+	+	+
Отсутствие ограничений на память	-	-	+	+
Сериализация, импорт/экспорт	+	-	+	+
Механизмы отладки	+/-	-	+	+
Распределенный рендеринг	-	-	+/-	+
Простота интеграции	+	-	-	+

Таблица 1. Сравнение предлагаемого решения с существующими. Обозначения: + соответствие требованию, +/- частичное соответствие требованию, - несоответствие требованию.

Глава 3. Алгоритм оценки разрешения предрассчитанных процедурных текстур

3.1 Задача интеграции сторонних процедурных инструментов и существующие решения

Процедурная генерация является важной технологией в различных областях применения компьютерной графики поскольку она позволяет снизить стоимость ручного создания трехмерных сцен и их компонентов. Изменяя параметры алгоритмов генерации возможно автоматически получить множество различающихся между собой вариантов одной и той же текстуры или геометрической модели. Ручное же создание такого количества вариаций, очевидно, является значительно более затратным. Следует отметить высокую значимость процедурных подходов при использовании фотореалистичного синтеза изображений и видеопоследовательностей для формирования обучающих выборок для алгоритмов искусственного интеллекта в компьютерном зрении. Объемы подобных выборок весьма значительны, и в процессе решения задач может потребоваться изменять распределение объектов с разными свойствами в выборке, чтобы тестировать различные гипотезы. Поэтому полностью ручное создание компонентов сцен (и наполнение самих сцен) в данной области оказывается малоэффективным. Представленные результаты были впервые опубликованы в работе [3].

3.1.1 Процедурный подход

Процедурный подход в моделировании материалов в первую очередь основан на синтезе текстурных карт, которые в дальнейшем используются для задания распределенных в пространстве параметров различных компонентов сцен, в том числе моделей материалов и геометрических модели (например, с помощью карт смещения). Методы генерации процедурных текстур традиционно делят на явные и неявные [84, с. 12-14]. Явные методы заранее (т.е. до начала рендеринга) вычисляют всю текстуру целиком. Эти текстуры затем могут быть использованы

при расчете освещения как изображения. Неявные методы определяют текстуру как функцию от некоторых аргументов, например, координат в текстурном или мировом пространстве. Затем эта функция может быть вызвана рендер-системой в процессе расчета освещения по необходимости, в тот момент, когда потребуется осуществить выборку из такой текстуры. Другими словами, неявные методы в некотором смысле реализуют концепцию ленивых вычислений. Неявный подход подразумевает создание некоторого алгоритмического описания желаемого визуального результата.

Неявный подход в синтезе процедурных текстур используется уже довольно давно, – одним из первых подобных алгоритмов является шум Перлина [85], предложенный в 1985 г, и продолжает развиваться и сегодня (например, [86-88]). У неявных процедурных текстур нет фиксированного разрешения (оно потенциально бесконечное). В случае явных алгоритмов синтеза процедурных текстур, значения текселей влияют друг на друга и тем самым не могут быть рассчитаны независимо. Общей проблемой явных методов является значительный объем вычислений, который напрямую зависит от разрешения выходной текстуры. Алгоритм [89], по словам авторов, вычисляет текстуру в разрешении 256 на 256 пикселей за 10 минут на профессиональной видеокарте NVIDIA Tesla K40. В работе [90] сообщается, что время вычислений для различных алгоритмов текстурного синтеза на основе изображения-примера может достигать нескольких часов для больших разрешений.

3.1.2 Проблема определения разрешения

Разрешение текстуры оказывает влияние на занимаемый размер в памяти, время, необходимое для выполнения алгоритма синтеза, и на качество изображения при рендеринге. Таким образом, для фотореалистичного синтеза изображений значительную роль играет задача определения такого разрешения текстур, которое позволит получить максимальное визуальное качество при минимальной занимаемой памяти и минимальном объеме вычислений.

Данная задача актуальна и для обычных текстур-изображений. Конечный пользователь (3D-художник) в общем случае хранит текстуры в максимально большом разрешении для сохранения максимального качества и при создании не заботится о том, чтобы уменьшить размер текстуры, если она, например, накладывается на мелкие объекты. Таким образом, возникает ситуация, когда текстура высокого разрешения может быть назначена на объект, который занимает лишь незначительную часть изображения, рассчитанного рендер-системой (рис. 16). Очевидно, что это может вызывать значительный перерасход памяти.

Рис. 16 Пример сцены с мелкими объектами, на которые наложены текстуры высокого разрешения.

Простой подход, при котором разрешение текстур будет автоматически выбираться сравнимым с разрешением результирующего изображения, не решает проблемы. Так как он обрабатывает все текстуры в сцене одинаково, в

независимости от того, какой размер объекта на который они наложены, на финальном изображении. При этом текстуры на объектах, занимающих большую часть изображения, могут быть уменьшены слишком сильно, что приведет к падению качества синтезированного изображения, а текстуры на мелких объектах – недостаточно, что вызовет перерасход памяти.

Кроме того, размер текстуры влияет на процесс выборки из неё в процессе рендеринга. В идеальном случае текстура и синтезируемое изображение всегда имеют отношение пикселя к текселю один-к-одному – в каждом пикселе синтезируемого изображения «виден» один тексель текстуры. Или, другими словами, частота дискретизации синтезируемого изображения и текстуры совпадают, и фильтрацию применять не требуется. Однако, это возможно только в случае очень простой сцены, например, виртуальная камера «смотрит» на текстурированную плоскость, используется ортографическая проекция и разрешение текстуры на плоскости совпадает с разрешением рендеринга. В общем же случае, возникает необходимость выполнять увеличение текстуры (англ. magnification) с помощью интерполяционного фильтра или уменьшение (англ. minification) с помощью фильтра нижних частот. Соответственно, если текстура слишком большая, то это приведет к тому, что придется постоянно использовать фильтр нижних частот, который в зависимости от алгоритма, может быть довольно затратной в вычислительном плане операцией.

Если при расчете интеграла освещенности (1-1) в некотором пикселе синтезируемого изображения, ДФР f является параметризованной текстурами, то интеграл освещенности можно представить в следующем виде:

$$L_o(x,\omega_0) = L_e(x,\omega_0) + \int_{\Omega} L_i(x,\omega_i) f\left(x,\omega_0,\omega_i,\int \hat{f}(u,v)t_1(u,v)dudv,\ldots\right)(\omega_i\cdot n)d\omega_i \quad (3\text{-}1)$$

где $t_1(u,v)$ – функция, определяющая значение текстуры по текстурным координатам u,v, а $\hat{f}(u,v)$ – функция фильтрации.

Если этот пиксель синтезируемого изображения отображается на несколько пикселей текстуры (называемых текселями), то функция $\hat{f}(u,v)$ должна обеспечивать фильтрацию нижних частот, чтобы избежать эффекта наложения (алиасинга). В зависимости от реализуемого алгоритма фильтрации, для этого потребуется выполнять выборку нескольких значений из текстуры.

Ситуации, когда требуется выполнять увеличение текстуры с помощью интерполяции, в фотореалистичном рендеринге желательно избегать, так как это негативно сказывается на качестве синтезированного изображения.

Таким образом, задачей алгоритма определения разрешения (т.е. по сути определения частоты дискретизации) для предварительного расчета процедурной текстуры является поиск такого минимального разрешения, что во время рендеринга сцены никогда не потребуется использовать интерполяционный фильтр для увеличения текстуры. Для этого достаточно найти такое разрешение, чтобы по всему синтезируемому изображению для данной текстуры приходился минимум 1 тексель на 1 пиксель синтезируемого изображения.

Также следует отметить, что для некоторых видов текстурных карт, например, карт шероховатости поверхности и карт нормалей, фильтрация может привести к значительным потерям в детальности [91], что является проблемой при фотореалистичном рендеринге, так как детальность поверхностей объектов — одна из основ реалистичного изображения. Причина этой проблемы состоит в том, что, в частности, текстура шероховатости выступает в качестве параметра нелинейного компонента в составе ДФР. В некоторых ДФР на основе теории микрограней (англ. microfacet theory), параметр шероховатости используется в качестве показателя степени в описании функции распределения нормалей к микрограням, например, в модели Бекманна (Beckmann) [33, с. 533-548] :

$$\mathrm{D}(m) = \frac{e^{-\tan^2\theta/\alpha^2}}{\pi\alpha^2\cos^4\theta} \qquad (3\text{-}2)$$

где m – вектор нормали к поверхности, θ – угол между вектором нормали к поверхности и вектором нормали к микрограни, α – параметр шероховатости.

Заметим, что выражение (3-1) можно записать с другим порядком интегрирования:

$$L_o(x,\omega_0) = L_e(x,\omega_0) + \int \hat{f}(u,v) \int_{\Omega} L_i(x,\omega_i) f(x,\omega_0,\omega_i,t_1(u,v),\dots)(\omega_i \cdot n) d\omega_i du dv \quad (3\text{-}3)$$

В этом случае фильтрация будет происходить не для значений, выбранных из текстуры, а для результатов вычисления освещения в точке. Если текстура используется для параметризации линейного компонента ДФР f, например, диффузного цвета, то результаты вычисления (3-1) и (3-3) будут совпадать. Однако, для нелинейных компонентов (как упомянутая выше шероховатость) результаты будут отличаться. При этом в постановке (3-3) ДФР f будет вычисляться только для значений, которые реально присутствуют в текстурах $t_i(u,v)$. А для (3-1), за счет того, что сначала производится фильтрация текстуры, в качестве параметров для ДФР f могут выступать уже обработанные значения, которые не присутствуют в текстуре и, в зависимости от модели ДФР f, могут не иметь физического смысла.

При использовании постановки задачи расчета освещения (3-3), для оценки интеграла функции фильтрации может, также как и для интеграла освещенности, использоваться метод Монте-Карло с выборкой по значимости [92]. В индустриальных рендер-системах, например [93], используется такая оценка в пространстве экрана. Такой подход дает хороший результат для задачи уменьшения текстур (т.е. для фильтров низких частот) и позволяет авторам в [93] обойтись простой (и дешевой в плане вычислений) поточечной выборкой из текстуры. Однако для задачи увеличения (интерполяционных фильтров) для поточечной выборки возникают артефакты, приводящие к необходимости прибегать к более сложным (и вычислительно более дорогостоящим) фильтрам. Что также мотивирует стремление к избавлению от необходимости использования увеличения текстур в предлагаемом решении.

3.1.3 Интеграция сторонних процедурных инструментов

Другая задача связана с часто возникающей на практике необходимостью поддерживать существующие реализации методов неявного текстурного синтеза в проприетарных программных продуктах, в первую очередь в 3D редакторах. Для CPU-рендера в типичном случае это требует лишь вызова виртуальной функции клиентского приложения для получения выборки из текстуры, когда это потребуется в процессе вычислений. Однако, если адресное пространство, в котором работает рендер-система, отличается от адресного пространства 3D редактора, то подобный вызов функции в процессе рендеринга становится невозможным. Подобная ситуация возникает, например, если рендер-система работает на графическом процессоре или на другом компьютере в сети. Таким образом, для поддержки функциональности таких процедурных текстур, разработчикам рендер-системы нужно использовать другой подход. Один из вариантов – реализация аналогичных алгоритмов процедурного синтеза текстур в рендер-системе и сопоставление их параметров с таковыми в программном обеспечении, в которое производится интеграция рендер-системы. Однако, поскольку исходные коды алгоритмов текстурного синтеза в 3D редакторе неизвестны (а в большинстве случаев отсутствуют и ссылки на описание алгоритмов), подобное решение нельзя назвать «поддержкой существующей функциональности редактора» в полном смысле, а только её эмуляцией. Кроме того, для каждого из программных продуктов, в который интегрируется рендер-система, необходимо будет заново повторять реализацию его специфических процедурных текстур. Второй вариант - рассматривать неявные методы текстурного синтеза, которые необходимо поддержать, как явные - то есть производить предварительный расчет текстур, используя имеющуюся реализацию в клиентском приложения. В этом случае мы приходим к задаче определения необходимого разрешения текстур.

3.1.4 Существующие решения вычисления разрешения

Существующие подходы в большинстве случаев ориентированы на вычисление уровня детализации (и тем самым разрешения) для текстур, заданных изображениями для рендер-систем, работающих в реальном времени. Типичные решения задачи выбора разрешения текстур основаны на mip-текстурировании [94] и подразумевают хранение нескольких копий изображения с разными разрешениями, из которых подходящая выбирается в процессе рендеринга, например, на основе размера, спроецированного на экран текстурируемого объекта.

Есть методы, позволяющие проводить параллельный синтез разных mip-уровней для некоторых явных процедурных текстур [95, 96, 87]. Некоторые из существующих подходов нацелены на выполнение сравнительно небольшого числа предварительных вычислений для ускорения дальнейшего синтеза всей mip-пирамиды [97]. Усовершенствование mip-текстурирования, называемое clip-mapping [98] основано на том, что для большой текстуры не все данные в mip-пирамиде используются при рендеринге конкретного кадра, поэтому достаточно хранить только те части, которые являются видимыми. Более сложный подход - виртуальное текстурирование [99-101] основано на равномерном разбиении текстур на тайлы (прямоугольные фрагменты) и загрузке в память только тех тайлов, которые нужны для расчета конкретного кадра, также на основе информации о видимости. Виртуальное текстурирование получило аппаратную поддержку на GPU [102] и доступно в Vulkan API и как расширение OpenGL API.

3.1.5 Ограничения существующих подходов

Существующие решения для оценки разрешения текстур в основном ориентированы на рендеринг в реальном времени. Также, поскольку поставленная задача заключается в оценке разрешения для предрассчитанных процедурных текстур до начала рендеринга, существующие подходы сами по себе не позволяют её решить. Кроме того, реализация любого из существующих подходов предполагает тесную интеграцию с расчетным ядром рендер-системы и может

потребовать значительных изменений в нем. Например, если меши могут иметь больше одной назначенной текстуры, то использование виртуального текстурирование потребует реализации текстурных атласов [103]. Поскольку переписывание кода уже существующих и стабильно работающих частей сложной системы обычно является нежелательным, это можно считать недостатком для прямой реализации рассмотренных решений в некоторой существующей рендер-системе. Среди других возможных проблем имеющихся решений можно выделить возможное возникновение визуальных артефактов при фильтрации текстур [100, 101].

3.2 Предлагаемый алгоритм

3.2.1 Описание алгоритма

В основе предлагаемого решения лежит выделение всех необходимых предварительных вычислений из рендер-системы в отдельный этап быстрого предварительного рендеринга сцены, который может быть выполнен с помощью растеризации или трассировки лучей, за которым следует синтез текстур. На этом предварительном этапе определяется разрешение каждой текстуры в сцене, исходя из mip-уровней, которые рассчитываются с использованием модифицированной реализации подхода, предложенного в [102], и подхода, представленного в спецификации OpenGL API [104].

Следует отметить, что при реализации вспомогательного рендеринга через растеризацию, оценка разрешения будет корректна для текстур на объектах прямой видимости. Для объектов, видимых в отражениях или сквозь прозрачные объекты с преломлением, необходимо выполнять вспомогательный рендеринг с помощью трассировки лучей.

Предлагаемая общая схема вычисления разрешения представлена на рис. 17:

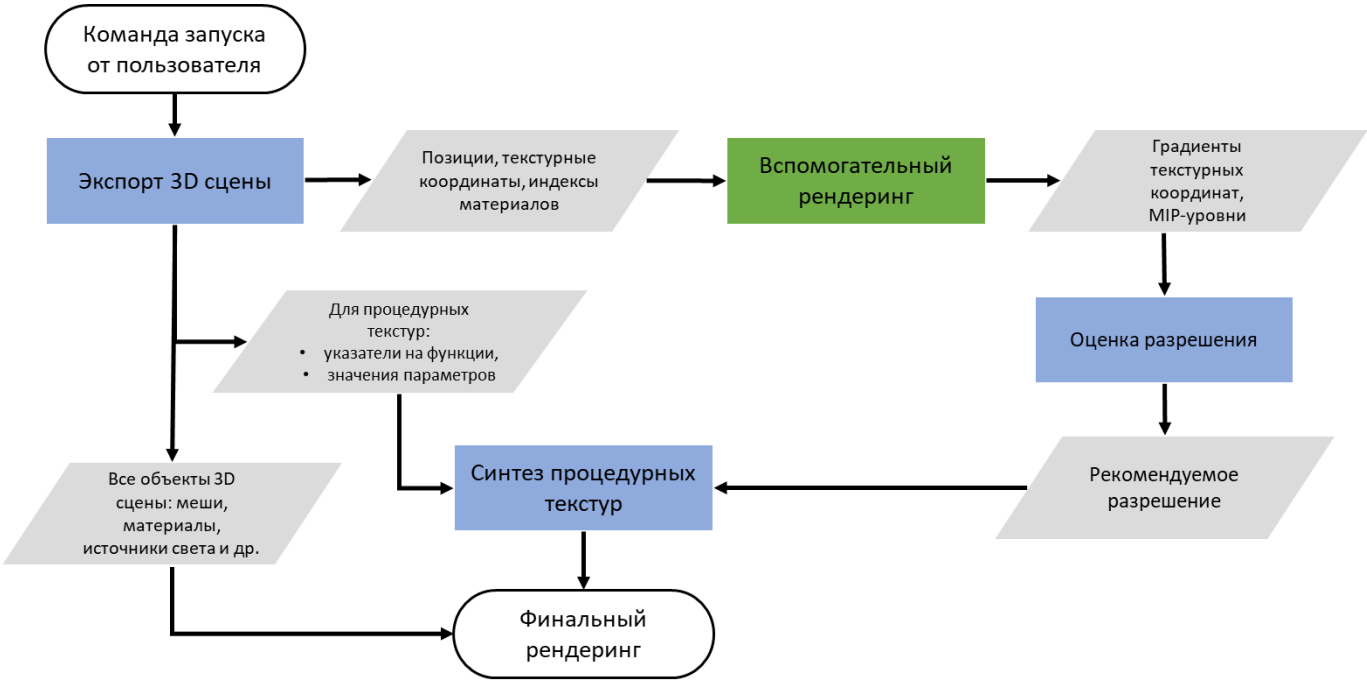


Рис. 17 Схема вычисления разрешения процедурных текстур.

В первую очередь необходимо сохранить указатели на функции, реализующие расчет процедурных текстур, использующихся в текущей 3D сцене, а также значения их входных параметров. Эта информация будет использована в дальнейшем для непосредственного синтеза текстур после определения требуемого разрешения. Следующий этап заключается в предварительном рендеринге геометрии сцены с информацией о назначенных материалах и текстурных координатах. Из полученных таким образом изображений можно определить градиент текстурных координат для всех текстурированных объектов, а на основе градиента вычислить mip-уровень. Выражения для вычисления градиента (3-4) схожи с таковыми для вычисления mip-уровня в [101] и спецификации графических API, например OpenGL [104].

$$G_x(uv) = \frac{R_{px}}{R_{rx}} * tex\_res * \frac{\partial u}{\partial x}$$
$$G_y(uv) = \frac{R_{py}}{R_{ry}} * tex\_res * \frac{\partial v}{\partial y} \quad (3\text{-}4)$$

где R_{px}, R_{py} – разрешение изображения для предварительной отрисовки, R_{rx}, R_{ry} – разрешение изображения для фотореалистичного рендеринга, tex_res – разрешение текстуры, u, v – текстурные координаты, x, y – координаты в пространстве изображения, полученного при предварительном рендеринге.

Однако, для процедурных текстур величина tex_res является искомой. Предположим, что она представляет собой некоторое произвольное большое разрешение, которое требуется уменьшить. Также следует учесть соотношение между разрешением предварительного рендеринга (R_{px}, R_{py}) и разрешения, в котором будет производится основной, фотореалистичный рендеринг (R_{rx}, R_{ry}). Значения этих разрешений могут отличаться, т.к. предварительный рендеринг не требует высокого разрешения, в то время как фотореалистичный рендеринг, очевидно, может быть осуществлен в произвольном разрешении.

После расчета градиентов, они используются для вычисления mip-уровня mip_lvl, опять же аналогично [104] и [101]:

$$mip\_lvl = log_2\left[max\left(G_x, G_y\right)\right] \tag{3-5}$$

Необходимое разрешение текстуры R тогда может быть рассчитано как:

$$R = \frac{tex\_res}{2^{mip\_lvl}} \tag{3-6}$$

Подставив выражения для вычисления градиентов из (3-4) в (3-6), а затем результирующее выражение для mip_lvl и, сократив tex_res, получим финальное выражение:

$$R = max\left(\frac{\frac{R_{rx}}{R_{px}} * 1}{\frac{\partial u}{\partial x}}, \frac{\frac{R_{ry}}{R_{py}} * 1}{\frac{\partial v}{\partial y}}\right) \tag{3-7}$$

При реализации алгоритма, нужно определить минимальный mip-уровень (т.е. соответствующий наибольшему разрешению) для каждой текстуры в сцене - одни и те же текстуры могут иметь разные mip-уровни при использовании в разных материалах и на разных мешах. Тем самым для каждой текстуры мы определяем один уровень из mip-пирамиды, который позволит обеспечить минимальное

разрешение, которое, в то же время, не потребует использовать интерполяционный фильтр для увеличения текстуры ни в одном из пикселей синтезируемого изображения. Исходя из этого mip-уровня рассчитывается разрешение текстуры, которое остается передать в сохраненные функции синтеза текстур.

В реализации алгоритма имеет смысл ограничить максимально возможное разрешение текстуры и максимально возможный mip-уровень, чтобы избежать слишком больших и слишком маленьких текстур соответственно.

Тогда алгоритм для определения разрешения всех текстур в сцене будет выглядеть следующим образом:

Алгоритм 2. Определение разрешения текстур

Входные данные:

I– изображение полученное на этапе предварительного рендеринга с разрешением R_{px}, R_{py}, каждый пиксель изображения содержит кортеж (x, y, uv, mat_id), где x и y – координаты пикселя в изображении I, uv - текстурные координаты объекта, видимого из виртуальной камеры, и mat_id - идентификатор его материала;

$materials$ – массив материалов, каждый из которых содержит массив пар (tex_id, M), где tex_id - идентификатор текстуры и M - матрица трансформации текстурных координат;

R_{rx}, R_{ry}– разрешение финального (фотореалистичного) рендеринга;

w_{max}, h_{max} – максимальное разрешение текстуры;

mip_{max} – максимальный mip-уровень.

Выходные данные:

$out\_resolution$ – массив рассчитанных разрешений текстур.

1 $mipbuf[\dots] \leftarrow 0$

2 **foreach** $pix \in I$ **do**:

3 $\quad uv \leftarrow pix.uv$

4 $\quad$ **foreach** $(tex\_id, M) \in materials[pix.mat\_id]$ **do**:

5 $\quad\quad \widehat{uv} \leftarrow M * uv$

6 $\quad\quad dx \leftarrow \frac{R_{px}}{R_{rx}} * w_{max} * \frac{\partial \widehat{uv}}{\partial pix.x}$

7 $\quad\quad dy \leftarrow \frac{R_{py}}{R_{ry}} * h_{max} * \frac{\partial \widehat{uv}}{\partial pix.y}$

8 $\quad\quad \Delta \leftarrow \max((dx)^2, (dy)^2)$

9 $\Delta \leftarrow \min\left(\max(\Delta, 1), 2^{mip_{max}*2}\right)$

10 $mip \leftarrow floor(\log_2\sqrt{\Delta})$

11 ***if*** $mip < mipbuf[tex_{id}]$ ***then***:

12 $mipbuf[tex\_id] \leftarrow mip$

13 ***end if***

14 ***end for***

15 ***end for***

16 ***foreach*** $mip_i \in mipbuf$ ***do***:

17 $width \leftarrow \frac{w_{max}}{2^{mip_i}}$

18 $height \leftarrow \frac{h_{max}}{2^{mip_i}}$

19 $out\_resolution[i] \leftarrow (width, height)$

20 ***end for***

21 ***return*** $out\_resolution$

Результаты работы алгоритма могут быть использованы как для определения разрешения процедурных текстур, так и обычных - заданных как изображения, для изменения их размера до достижения соотношения в минимум 1 тексель текстуры на 1 пиксель синтезированного изображения. Докажем это.

Определение. В изображении $I^{X\times Y}$ для некоторого исходного пикселя с дискретными координатами (x, y): $x, y \in \mathbb{N}_0, \forall x: 0 \le x \le W-1, \forall y: 0 \le y \le H-1$ соседними будем называть такие пиксели, что дискретное расстояние Чебышева до них от исходного пикселя будет меньше либо равно 1. При этом для пикселей, находящихся на границе изображения, т.е. с дискретными координатами (x, y) такими, что $x \in \{0, W-1\}, y \in \{0, H-1\}$, могут быть специальным образом заданы дополнительные соседние пиксели за границей изображения.

Теорема. Алгоритм 2 вычисляет разрешение текстуры (W, H) такое, что хотя бы для одной координаты достигается соотношение 1 тексель текстуры на 1 пиксель синтезированного рендер-системой изображения. Т.е. для любой пары соседних пикселей синтезированного изображения будет выполнена выборка из

пары соседних текселей текстуры, если в этих пикселях виден один и тот же объект 3D сцены.

Доказательство. Обозначим через $T^{W\times H}$ текстуру с разрешением W на H, $I^{R_{rx}\times R_{ry}}$ – финальное изображение, синтезированное основной рендер-системой, $\hat{I}^{R_{px}\times R_{py}}$ – изображение, полученное в результате предварительного рендеринга. Рассмотрим пиксель изображения $I^{R_{rx}\times R_{ry}}$ с дискретными координатами (x_1, y_1) и одного из его соседних пикселей с дискретными координатами (x_2, y_2) такими, что

$$\begin{aligned} x_2 - x_1 &= 1 \\ y_2 - y_1 &= 1 \end{aligned} \tag{3-8}$$

и в котором виден объект 3D сцены с текстурными координатами (u_1, v_1) и (u_2, v_2).

Координата пикселя x в изображении $I^{R_{rx}\times R_{ry}}$ и координата пикселя $\hat{x}$ в изображении $\hat{I}^{R_{px}\times R_{py}}$ соотносятся также, как и разрешения этих двух изображений:

$$\frac{\hat{x}}{x} = \frac{R_{px}}{R_{rx}} \tag{3-9}$$

Умножим выражения (3-8) на $\frac{R_{px}}{R_{rx}}$ и $\frac{R_{py}}{R_{ry}}$ соответственно:

$$\begin{aligned} \frac{R_{px}}{R_{rx}} * x_2 - \frac{R_{px}}{R_{rx}} * x_1 &= \frac{R_{px}}{R_{rx}} \\ \frac{R_{py}}{R_{ry}} * y_2 - \frac{R_{py}}{R_{ry}} * y_1 &= \frac{R_{py}}{R_{ry}} \end{aligned} \tag{3-10}$$

Тогда из (3-9) следует:

$$\begin{aligned} \widehat{x_2} - \widehat{x_1} &= \frac{R_{px}}{R_{rx}} \\ \widehat{y_2} - \widehat{y_1} &= \frac{R_{py}}{R_{ry}} \end{aligned} \tag{3-11}$$

Обозначим разности координат соседних пикселей в изображении $\hat{I}^{R_{px}\times R_{py}}$ как $\Delta\hat{x}$ и $\Delta\hat{y}$:

$$\Delta\hat{x} = \widehat{x_2} - \widehat{x_1}$$

$$\Delta\hat{y} = \widehat{y_2} - \widehat{y_1}$$

$$\Delta\hat{x} = \frac{R_{px}}{R_{rx}}$$
$$\Delta\hat{y} = \frac{R_{py}}{R_{ry}} \quad (3\text{-}12)$$

Тогда приближения производных текстурных координат по координатам изображения $\hat{I}^{R_{px} \times R_{py}}$ может быть записано как:

$$\frac{\partial u}{\partial x} \approx \frac{u_2 - u_1}{\Delta\hat{x}} = (u_2 - u_1) * \frac{R_{rx}}{R_{px}}$$
$$\frac{\partial v}{\partial y} \approx \frac{v_2 - v_1}{\Delta\hat{y}} = (v_2 - v_1) * \frac{R_{ry}}{R_{py}} \quad (3\text{-}13)$$

Из (3-7):

$$W = \frac{R_{rx}}{R_{px}} / \frac{\partial u}{\partial x} \quad (3\text{-}14)$$
$$H = \frac{R_{ry}}{R_{py}} / \frac{\partial v}{\partial y}$$

Подставим выражения (3-13) в (3-14):

$$W = \frac{1}{u_2 - u_1} \quad (3\text{-}15)$$
$$H = \frac{1}{v_2 - v_1}$$

В то же время координаты текселей могут быть получены с помощью текстурных координат по выражениям:

$$px_1 = u_1 * W, py_1 = v_1 * H$$
$$px_2 = u_2 * W, py_2 = v_2 * H \quad (3\text{-}16)$$

Тогда разность между координатами текселей:

$$px_2 - px_1 = u_2 * W - u_1 * W = W * (u_2 - u_1) \quad (3\text{-}17)$$
$$py_2 - py_1 = v_2 * H - v_1 * H = H * (v_2 - v_1)$$

Подставим (3-15) в (3-17) и получим:

$$px_2 - px_1 = \frac{1}{u_2 - u_1} * (u_2 - u_1) = 1 \quad (3\text{-}18)$$

$$py_2 - py_1 = \frac{1}{v_2 - v_1} * (v_2 - v_1) = 1$$

Если рассмотреть все остальные соседние пиксели, то разность координат в выражениях (3-8) может принимать значения -1 или 0. Для случая -1 результат будет аналогичный, – можно просто поменять местами в выражениях исходный пиксель и соседний (т.к. из определения для пикселя, являющегося соседними для исходного, исходный также будет являться соседним). А в случае, когда разность по одной из координат равна 0, эта координата не потребует рассмотрения, т.к. в (3-7) берется максимум.

Таким образом, соседним пикселям в изображении $I^{R_{rx} \times R_{ry}}$, синтезированном основной рендер-системой, соответствуют соседние тексели текстуры, **что и требовалось доказать**.

Отметим, что т.к. в выражении (3-7) берется максимум, то соотношение 1 пиксель к 1 текселю для неквадратных текстур будет достигнуто только для одной из координат.

Для разных видов текстур рекомендуется задавать разные максимально допустимые mip-уровни, например, чтобы генерировать карты отражений и смещения в более высоком разрешении, чем текстуры для диффузного цвета. Следует отметить, что рендер-система может выполнять масштабирование текстур в соответствии с рассчитанными предложенным подходом разрешениями только в том случае, когда есть необходимость в экономии памяти.

3.2.2 Вычисление производной текстурных координат

В финальном выражении для разрешения текстуры (формула 3-7) присутствуют частные производные текстурных координат по координатам полученного на этапе предварительной отрисовки сцены изображения – $\partial u/\partial x$, $\partial v/\partial y$. Существует несколько способов вычисления данных величины.

В случае реализации предварительного этапа как растеризации с помощью одного из графических API (например, OpenGL или Vulkan), можно воспользоваться доступными во фрагментном шейдере функциями $dFdx$ и $dFdy$,

позволяющими использовать аппаратные средства для вычисления значений частных производных переданного им параметра (в нашем случае - текстурных координат) по координатам в пространстве изображения. В этом случае частные производные вычисляются как разности значений дифференцируемого параметра в текущем фрагменте и его ближайших соседях [104]. Если предварительный этап реализован через растеризацию без использования графических API или с помощью трассировки лучей, то аналогичные вычисления могут быть реализованы разработчиком, при условии сохранения в каждом пикселе рассчитанного изображения информации о значении дифференцируемого параметра.

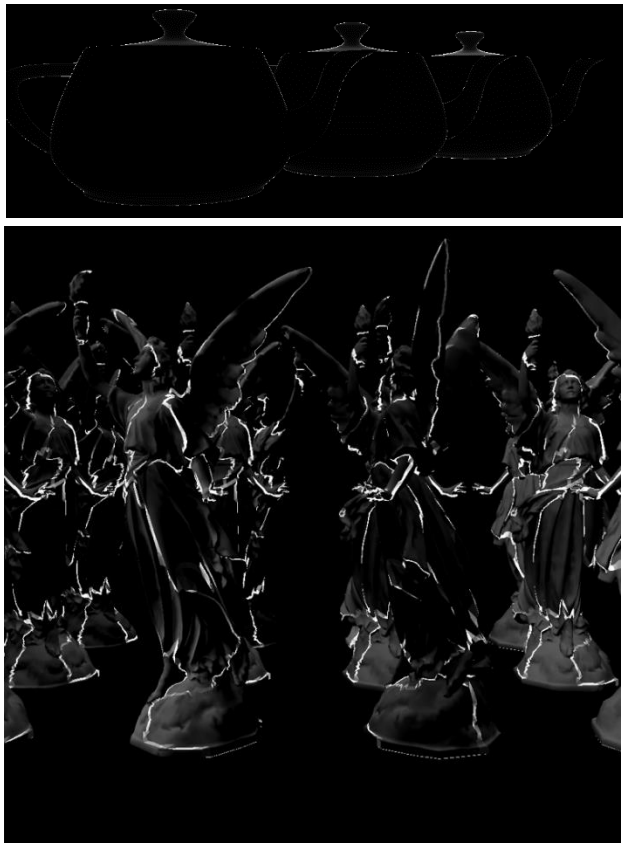


Рис. 18 Визуализация модуля градиента текстурных координат на 3D моделях оттенками серого.

Следует отметить, что как при использовании аппаратных функций $dFdx$ и $dFdy$, так и в случае собственной реализации описанного способа, предполагается, что дифференцируемая величина является непрерывной. На практике же будут появляться разрывы, например, на границах объектов (рис. 18, сверху). Или же, например, высокие значения производных на текстурных швах (рис. 18, снизу).

Однако, поскольку в предложенном решении происходит поиск минимального значения mip-уровней, высокие значения производной на различного рода границах будут отброшены и не повлияют на вычисление финального результата.

Второй подход вычисления частных производных текстурных координат основан на трассировке лучей с вычислением так называемых дифференциалов луча [105], [33, с. 605-608]. Идея метода заключается в том, что для каждого луча, трассируемого из виртуальной камеры, создается два вспомогательных луча со сдвигом в один пиксель по горизонтали и по вертикали. При этом все пересечения с геометрическими объектами вычисляются только для основного луча. Дополнительные лучи используются только для оценки частных производных, например, мировых или текстурных координат геометрического объекта, с которым было найдено пересечение основного луча. При этом принимается допущение, что поверхность геометрического объекта локально плоская. Таким образом, точки пересечения вспомогательных лучей могут быть вычислены как лежащие на плоскости, заданной точкой пересечения основного луча и вектором нормали в этой точке. Для сильно искривленных поверхностей и на границах объектов подобное допущение может иметь большую погрешность, но при этом гарантирует отсутствие разрывов.

Метод дифференциалов луча может быть использован при реализации предварительного расчета изображения как с помощью трассировки лучей, так и с помощью растеризации. В случае растеризации направление луча D из виртуальной камеры в конкретном пикселе (x, y) изображения может получено по следующему алгоритму (алг. 3):

Алгоритм 3. Определение направления луча из виртуальной камеры из матриц видового и проективного преобразований

Входные данные:

x, y – координаты пикселя в растеризованном изображении;

w, h – ширина и высота изображения;

M_{view} – матрица видового преобразования;

M_{proj} – матрица проективного преобразования.

Выходные данные:

D – вектор направления луча из виртуальной камеры через пиксель с координатами x, y.

1	$P1 = [0, 0, 0, 0]$
2	$P1[0] \leftarrow \frac{2 * (x + 0.5)}{w} - 1$
3	$P1[1] \leftarrow \frac{-2 * (y + 0.5)}{h} + 1$
4	$P1[2] \leftarrow 1$
5	$P2 \leftarrow \left(M_{view} * M_{proj}\right)^{-1} * P1$
6	$P2 \leftarrow \frac{P2}{P2[3]}$
7	$D \leftarrow \frac{P2}{\|P2\|}$
8	$\boldsymbol{return}\ D$

3.2.3 Ограничения и частные случаи

Отдельно следует рассмотреть несколько частных случаев, которые требуют внесения изменений в предложенный подход - наличие прозрачных объектов и текстурированных объектов, видимых в отражениях. Необходимые изменения будут отличаться в зависимости от используемого способа расчета частных производных текстурных координат.

Рассмотрим сначала случай первого варианта вычисления производных. При присутствии в сцене прозрачных объектов для корректной оценки разрешения текстур при вычислении первым из рассмотренных выше методов необходимо отдельно обрабатывать прозрачные и непрозрачные объекты. Например, сначала отрисовать все непрозрачные объекты и вычислить для них уровень детализации текстур, затем последовательно отрисовывать прозрачные объекты и выполнять вычисления для них. При этом возможные преломления в прозрачных объектах не будут учтены. В ситуации с текстурированными объектами, которые видимы в отражениях (возможно после многократных переотражений и преломлений) возможно только эвристическое решение - например, предположить, что отраженный объект не будет больше, чем разрешение экрана и задать разрешение

текстур на данном объекте соответствующим образом. Другая возможная эвристика - назначать отраженным объектам уровень детализации, полученный для наибольшего отражающего объекта (зеркала) в сцене.

Если же частные производные рассчитываются вторым из рассмотренных выше способов, то оба частных случая могут быть решены более эффективно, т.к. при трассировке дифференциалов луча возможно также рассчитать отражения и преломления для вспомогательных лучей [105], [33, с. 605-608]. Это можно считать значительным преимуществом подхода на основе дифференциалов луча. Однако, в этом случае необходимо сам этап предварительного расчета реализовывать на основе трассировки лучей.

3.3 Детали реализации

Разработанный алгоритм был реализован в интеграционном программном слое, описанном в главе 2. Экспорт сцены происходит обычным способом при помощь программного интерфейса интеграционного слоя. Для процедурных текстур в API были добавлены специальные функции для экспорта процедурных эффектов (листинг 6):

```
1.  HRTextureNodeRef hrTexture2DCreateBakedHDR(HR_TEXTURE2D_PROC_HDR_CALLBACK
2.     a_proc, void *a_customData, int customDataSize, int w = -1, int h = -1);
3.
4.  HRTextureNodeRef hrTexture2DCreateBakedLDR(HR_TEXTURE2D_PROC_LDR_CALLBACK
5.     a_proc, void *a_customData, int customDataSize, int w = -1, int h = -1);
6.
7.  HRTextureNodeRef hrTexture2DUpdateBakedHDR(HRTextureNodeRef currentRef,
8.    HR_TEXTURE2D_PROC_HDR_CALLBACK a_proc, void *a_customData,
9.    int customDataSize, int w, int h);
10.
11. HRTextureNodeRef hrTexture2DUpdateBakedLDR(HRTextureNodeRef currentRef,
12.   HR_TEXTURE2D_PROC_LDR_CALLBACK a_proc, void *a_customData,
13.   int customDataSize, int w, int h);
14.
15. typedef void(*HR_TEXTURE2D_PROC_LDR_CALLBACK)(unsigned char* a_buffer, int w,
16.   int h, void* a_customData);
17.
18. typedef void(*HR_TEXTURE2D_PROC_HDR_CALLBACK)(float* a_buffer, int w, int h,
19.   void* a_customData);
```

Листинг 6. Объявления функций API интеграционного слоя для экспорта процедурных эффектов.

На стороне пользовательского приложения необходимо определить функцию, которая будет выполнять вызов процедурного инструмента с сигнатурой, соответствующей типу *HR_TEXTURE2D_PROC_LDR_CALLBACK* или *HR_TEXTURE2D_PROC_HDR_CALLBACK* (листинг 6, строки 15-19), в зависимости от типа данных в текстуре. После этого пользователь передает указатель на эту функцию, а также буфер, содержащий данные, необходимые для вызова процедурного инструмента в API интеграционного слоя с помощью соответствующей функции *hrTexture2DCreate** (листинг 6, строки 1-5). При необходимости обновить данные для уже созданной процедурной текстуры используются вызовы *hrTexture2DUpdate** (листинг 6, строки 7-13). Таким образом, в интеграционном слое будет сохранена информация для синтеза процедурных текстур после вычисления необходимого разрешения.

Вспомогательный рендеринг, решающий задачу вычисления производных текстурных координат, был реализован как специальный рендер-драйвер. Он выполняет растеризацию сцены с помощью OpenGL API и вычисление градиентов и mip-уровней по первому из описанных выше методов. После экспорта сцены этот рендер-драйвер по необходимости запускается автоматически без создания окна и выполняет вычисление mip-уровней для каждой из текстур в сцене. После этого в интеграционном слое вычисляется минимальный mip-уровень для каждой из текстур, так как одна и та же текстура может быть назначена на разные геометрические объекты и таким образом иметь разные mip-уровни, в этом случае необходимо синтезировать текстуру с наибольшим разрешением и, соответственно, с минимальным mip-уровнем.

После этого вычисленные значения разрешения передаются в функции вычисления процедурных инструментов, ранее сохраненных в интеграционном слое. Наконец, синтезированные процедурные текстуры в нужном разрешении добавляются в XML-описание сцены и как бинарные файлы в директорию данных. Обновленное описание сцены передается основной рендер-системе.

Таким образом, основная рендер-система никак не зависит от процесса вычисления разрешения и синтеза процедурных текстур и ничего не должна делать

для какой-либо его поддержки. Схема взаимодействия программных компонентов с использованием разработанного интеграционного слоя представлена на рис. 19:

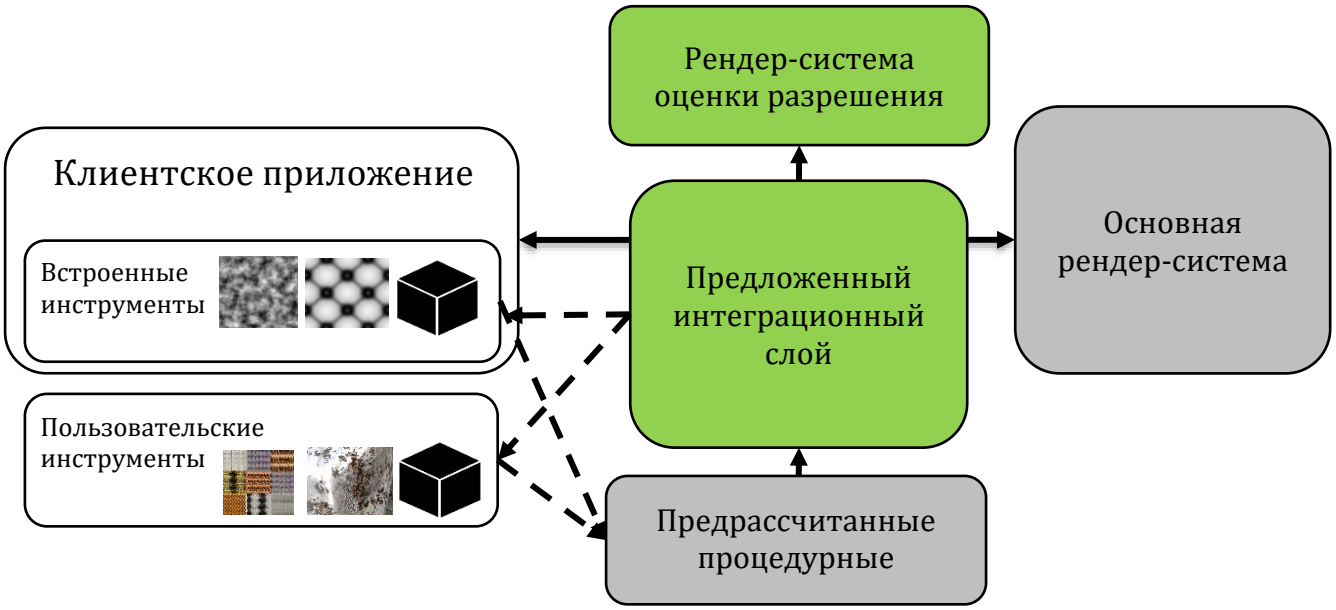


Рис. 19 Схема предварительного расчета процедурных эффектов с использованием разработанного интеграционного слоя и рендер-системы оценки разрешения.

3.4 Экспериментальная оценка

Тестирование предложенного решения было проведено на нескольких сценах с процедурными текстурами и текстурами-изображениями. Фотореалистичный рендеринг производился с помощью рендер-системы Hydra [41, 106] на GPU Nvidia GTX 1070.

Для проведения сравнения были использованы три сцены (рис. 20). В двух сценах «Arch» и «Alley» (рис. 20, сверху и справа внизу соответственно) преимущественно заданы процедурные текстуры, реализованные встроенными инструментами в 3D редакторе 3ds Max. А в сцене «Bathroom» (рис. 20, слева внизу) заданы текстуры-изображения высокого разрешения.

Текстуры-изображения в сценах использовались в их исходном разрешении, а процедурные текстуры в разрешении, выбранном вручную 3D-художником с запасом. Результат расчета данных сцен был принят за эталонный.

На следующем этапе первого эксперимента к тем же самым сценам был применен предложенный алгоритм оценки разрешения, использующий растеризацию с помощью графического конвейера для осуществления

предварительного расчета. После завершения работы алгоритма процедурные текстуры синтезировались в рассчитанном алгоритмом разрешении, а текстуры-изображения масштабировались до рассчитанного разрешения. Наконец был выполнен фотореалистичный рендеринг сцен и проведено сравнение с эталонными результатами. Сравнение показало уменьшение затрат памяти на текстуры от 1.6 до 4 раз при использовании предложенного алгоритма (рис. 21). При этом визуальные отличия с эталонными изображениями практически отсутствуют, приближенные фрагменты и разница представлены на рис. 22 и рис. 23.

Рис. 20 Тестовые сцены для разработанного алгоритма для уменьшения разрешения текстур. Верхняя (Arch) и нижняя правая сцены (Alley) - преимущественно процедурные текстуры, нижняя левая (Bathroom) - текстуры-изображения большого разрешения.

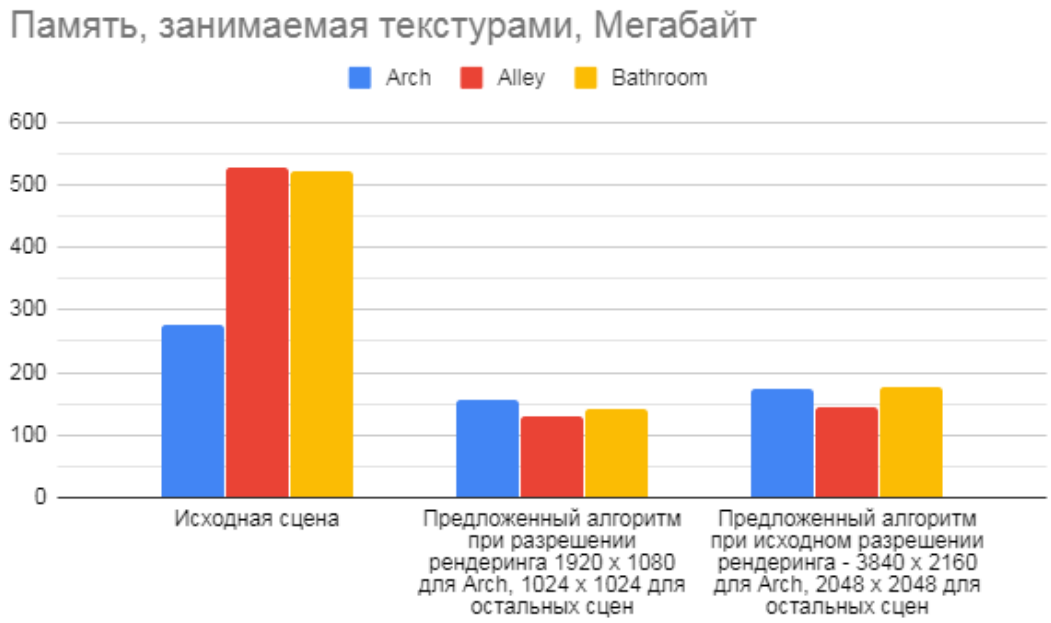


Рис. 21 Диаграмма, демонстрирующая уменьшение объема занимаемой памяти при использовании разработанного алгоритма на тестовых сценах от 1.6 до 4 раз. Разными цветами обозначены тестовые сцены. Левая группа столбцов – исходные значения, средняя и правая – при использовании алгоритма в разном разрешении рендеринга. Рендеринг производился на Nvidia GTX 1070.

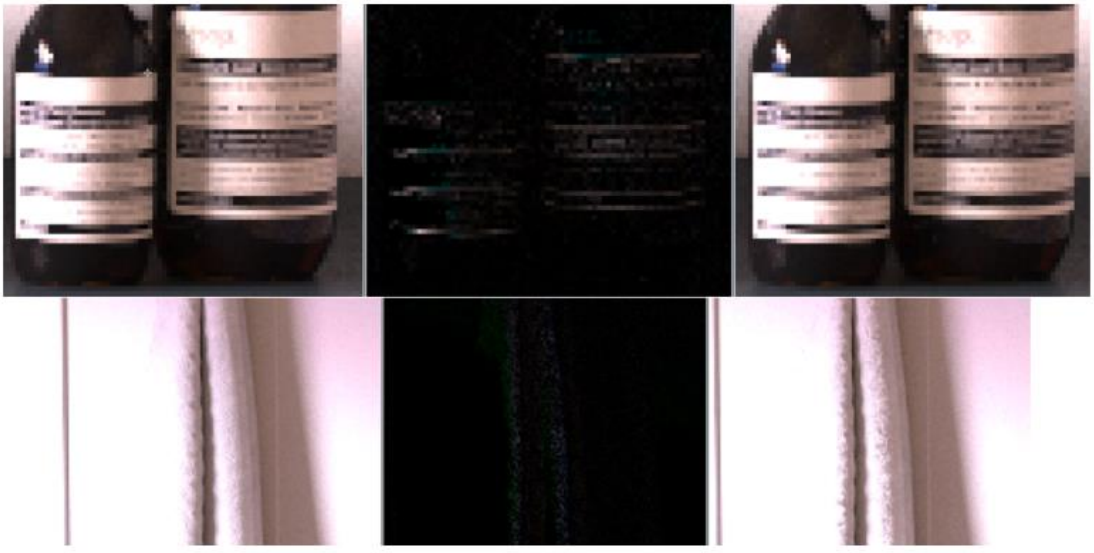


Рис. 22 Слева - эталон, справа – изображение с масштабированными по предложенному алгоритмы текстурами, по центру - разница этих изображений. Сцена Bathroom, в верхнем ряду - текстура-изображения для диффузного цвета, в нижнем ряду - карта нормалей, рассчитанная из текстуры-изображения.

Рис. 23 Слева - эталон, справа – изображение с масштабированными по предложенному алгоритмы текстурами, по центру - разница этих изображений. Сцена Alley, процедурная текстура на основе шума на объекте с неравномерным масштабированием текстурных координат.

Сцена	MSE (HDR)	MSE (LDR)	PSNR (LDR), дБ
Alley	1.67	5.929	40.401
Bathroom	3.59	21.563	34.794
Arch	3.02	20.626	34.987

Таблица 2 Среднеквадратические ошибки (MSE) и пиковое отношение сигнала к шуму (PSNR) между изображением-эталоном и изображением, для которого размер текстур был изменен по предложенному алгоритму для изображений с узким (LDR) и широким (HDR) динамическим диапазоном. Рендеринг на Nvidia GTX 1070, 2048 выборок на пиксель.

Численная оценка с помощью среднеквадратической ошибки между изображениями эталонов и полученных с использованием предложенного подхода (таблица 2) для изображений с широким динамическим диапазоном (HDR) также показывает малые значения отличий. Далее был произведен переход в узкий динамический диапазон (LDR), чтобы рассчитать значение PSNR (англ. peak signal-to-noise ratio, пиковое отношение сигнала к шуму), т.к. для HDR изображений использование данной метрика концептуально не является корректным из-за неравномерного восприятия яркости [107]. Полученные значения превышают значение в 30 дБ, которое обычно принимается пороговым для хорошего качества восстановления изображений [108, с. 6].

Вычисление среднеквадратической ошибки между двумя изображениями производилось по формуле:

$$MSE = \frac{\sum_{m,n=0}^{m<M,n<N}[I_1(m,n) - I_2(m,n)]^2}{M * N} \quad (3\text{-}16)$$

где I_1 и I_2 – матрицы изображений, M и N – размеры изображений по ширине и высоте.

Вычисление значения PSNR по формуле:

$$PSNR = 10\log_{10}\left(\frac{MAX_I^2}{MSE}\right) \quad (3\text{-}17)$$

где MAX_I – максимальное значение, принимаемое пикселем изображения, для LDR изображений с 8-бит на канал равно 255.

Таким образом, предложенный подход позволяет решить проблему поддержания сторонних процедурных инструментов в GPU рендер-системе путем выполнения предварительного синтеза этих текстур в разрешении, определяемым с помощью специального этапа рендеринга сцены. При этом предлагаемый подход позволяет обеспечить экономию памяти, требуемой для таких текстур, от 1.5 до 4 раз по сравнению с наивным подходом, при котором текстуры синтезируются в некотором фиксированном разрешении, заданном, например, исходя из разрешения рендеринга, или вручную 3D-художником, а текстуры-изображения используются в своем исходном разрешении.

Глава 4. Метод разработки пользовательских расширений для GPU рендер-системы

4.1 Расширяемость программного обеспечения

Отдельно следует рассмотреть требование расширяемости, которое в некотором смысле не является уникальным для приложений компьютерной графики, а представляет собой фундаментальную задачу разработки программного обеспечения. Значительный объем усилий по разработке приходится на эволюционные изменения – добавление, удаление и изменение некоторой функциональности. Под возможностью выполнения таких изменений «безболезненно», т.е. так, чтобы после их внесения не должна нарушаться работоспособность программы [109], и будем понимать требование расширяемости.

Существуют различные подходы и механизмы для обеспечения этого требования. Как наиболее примитивный и широко используемый прием можно выделить введение в программу конфигурационных параметров, влияющих на ход её исполнения. Сюда же можно отнести использование условных директив препроцессора в языках C/C++. При необходимости добавить новую функциональность разработчик добавляет новый вариант значения параметра и соответствующую ветвь выполнения программного кода. В той или иной степени данный подход используется практически в любой программе. Однако, его интенсивное использование для реализации поддержки расширений затруднено тем, что необходимо модифицировать исходный код программы, при том возможно в разных частях. Это быстро приводит к тому, что реализация новой функциональности часто оказывается разбросана по разным частям программы, что приводит к появлению зависимостей между ними, запутанности логики исполнения и, тем самым, снижает читаемость кода и затрудняет его поддержку и дальнейшую разработку. Кроме того, если необходима возможность разработки расширений пользователями программы, им нужно иметь доступ ко всему её исходному коду.

Чтобы отделить вновь добавляемую в программу функциональность также широко используются подходы объектно-ориентированного программирования (ООП), подразумевающие создание абстрактных интерфейсов для расширяемых компонентов программы, с возможностью дальнейшего определения новых реализаций, соответствующим вновь добавленной в программу функциональности с использованием механизмов наследования и виртуальных функций. В том числе и обобщенные решения проблем проектирования программ – паттерны проектирования (англ. design patterns), подробно рассмотренные в книге [110] (определение дано на с. 12-14). Подходы на основе ООП широко распространены, поддерживаются во многих языках программирования и позволяют отделить «базовый код» от кода расширений. Но привносят ощутимый объем поддерживающего кода и накладные расходы на реализацию. Кроме того, для обеспечения расширяемости программы необходимо заранее спроектировать точки роста программы (в англоязычной литературе обычно называемые hot spots – «горячие точки») и их интерфейсы, что чрезвычайно трудно сделать так, чтобы удовлетворить требования всех пользователей программы в будущем. А изменение определенных интерфейсов на более позднем этапе, когда уже существует целый набор их реализаций, оказывается достаточно трудоёмкой задачей. Наконец, механизмы объектно-ориентированного программирования обладают очень ограниченной поддержкой в системах программирования графических процессоров и зачастую работают неэффективно (причины подробнее рассмотрены в следующем разделе).

Также существуют более сложные подходы, основывающиеся на проектировании программы как структуры того или иного вида из компонентов или модулей так, чтобы обеспечить максимальное переиспользование компонентов и/или расширяемость. Первыми появились подходы, представляющих структуру программы из горизонтальных и вертикальных слоев [109, 111]. На их основе возникли такие решения как функционально-ориентированное программирование (англ. feature-oriented programming) [112-114] и аспектно-ориентированное программирование [115, 116]. Схожим подходом является и т.н. дельта-

ориентированное программирование [117], при котором программа представляется как базовый модуль и множество дельта-модулей, пошагово модифицирующих базовый. Все перечисленные подходы так или иначе основываются на использовании приемов ООП, а также требуют либо активного использования технологий обобщенного программирования, таких как шаблоны в языке C++ [118], либо расширения существующих языков программирования новыми конструкциями или же предлагают собственные языки [115, 116]. Это приводит к наличию у данной группы подходов, с одной стороны, ряда недостатков, свойственных ООП – сопровождающий код, накладные расходы, практически отсутствие поддержки для систем программирования GPU. А с другой стороны серьёзного недостатка, связанного с необходимостью использовать расширения существующих языков программирования или специальные языки, что приводит к невысокому распространению данных подходов за пределами исследовательских проектов в реальных индустриальных решениях. Наконец, перечисленные решения в первую очередь решают задачи развития и расширения программы её основными разработчиками и практически не ориентированы на создание расширений для базовой программы её конечными пользователями.

Распространенный подход для обеспечения расширяемости программы конечными пользователями – встраиваемые скрипты. В основе подхода лежит встраивание интерпретатора некоторого высокоуровневого языка программирования (Python, Lua, JavaScript и пр.), в котором предоставляется программный интерфейс к основной программе. Конечные пользователи (и сами разработчики) получают возможность писать скрипты на этом языке, которые могут использоваться для автоматизации некоторых рутинных действий, манипуляции конфигурационными параметрами программы, создания готовых сценариев использования и других задач. Возможности встраиваемых скриптов ограничиваются только программным интерфейсом. Однако, использование интерпретируемых языков приводит к потерям в производительности и не подходит для таких ресурсоемких задач, как фотореалистичный рендеринг. Для решения проблемы производительности может использоваться JIT-компиляция

(just-in-time компиляция). Но этот подход тоже имеет свои недостатки, в первую очередь связанные с высокой сложностью - JavaScript движок v8, использующий JIT-компиляцию, содержит более 3 миллионов строк кода [119].

4.2 Существующие решения для обеспечения расширяемости рендер-систем

Современные системы фотореалистичного рендеринга переходят на GPU реализации. С появлением доступной широкому пользователю технологии аппаратного ускорения трассировки лучей такой переход становится практически неизбежным. Однако, в течение длительного времени во многих приложениях фотореалистичного рендеринга (таких как кино- и анимационные фильмы, архитектурная визуализация и др.) использовались CPU рендер-системы, которые обладают значительными возможностями по расширяемости. В данном случае под «расширяемостью» будем понимать возможность для пользователей таких рендер-систем добавлять новую функциональность, такую как процедурное моделирование и текстурирование, специальные модели источников света и материалов. Рассмотрим существующие подходы для обеспечения таких возможностей с учетом необходимости поддержки GPU.

4.2.1 Плагины

Один из наиболее эффективных традиционных подходов — это объектно-ориентированные плагины [82, 120]. В этом случае разработчики рендер-системы предоставляют интерфейсы для таких компонентов как, например, модели источников света и материалов. Пользователи рендер-системы в дальнейшем могут делать свои реализации этих интерфейсов, которые затем используются как динамически загружаемые плагины к рендер-системе. У данного подхода есть хорошо известные недостатки и ограничения. Одна из самых серьезных проблем — низкая производительность [121, 122]. Рассматривая объектно-ориентированный подход (ООП) можно выделить следующие проблемы с производительностью на GPU [123]:

1. Расположение данных в памяти. Хотя ООП не требует какого-то особенного расположения объектов в памяти, в большинстве языков программирования и компиляторах один объект хранится как один непрерывный блок в памяти. Такое расположение может быть неидеальным для GPU, — для оптимизации доступа к памяти и эффективного использования кэшей, необходимо иметь возможность влиять на расположение данных в памяти [124].

2. Динамическое выделение памяти. Возможность создавать и удалять объекты в любой момент времени является одной из основных для ООП. Однако для GPU проблематично реализовать эффективный механизм динамического выделения памяти (из-за высокой стоимости межпоточного взаимодействия и синхронизации при большом количестве потоков). Это активное направление исследований [125-129], но при этом развитие аппаратного обеспечения происходит быстрее и лишь немногие решения работают на современных GPU [130]. Наконец, большинство решений так или иначе используют технологию Nvidia CUDA и тем самым не являются кроссплатформенными.

3. Виртуальные функции. Реализация механизма виртуальных функций обычно подразумевает переход (jump) по адресу в таблице виртуальных функций. Однако на GPU, переходы и вызовы функций довольно дороги, поэтому GPU компиляторы активно используют вставку (inline) функций [131, 132]. Однако переходы для виртуальных функций не могут быть вставлены подобным образом и в общем случае являются на порядок медленнее, чем обычные вызовы функций. Кроме того, вызов виртуальных функций может привести к расхождению потоков. В целом, графические процессоры не предназначены для динамической диспетчеризации [133].

4. Использование ООП может неявно увеличивать использование памяти, т.к. объект может хранить указатели на другие объекты, что требует 64 бит на современных архитектурах GPU. Без необходимости хранить указатель на

область памяти, во многих случаях можно было бы обойтись 32-битным целочисленным идентификатором.

Таким образом, хотя объектно-ориентированный подход возможен на графическом процессоре, но его эффективность является низкой в связи с перечисленными проблемами. Кроме того, следует отметить ограниченную гибкость подхода на основе объектно-ориентированных интерфейсов: невозможно создать интерфейс, который удовлетворял бы всех пользователей в будущем.

4.2.2 Предметно-ориентированные языки

Другой распространенный подход – использование специальных языков для создания расширений. Одним из первых методов создания пользовательских операций затенения были «графы затенения» (shading trees) [134] – ориентированные ациклические графы с входными значениями в листьях, операциями (такими как сложение и умножение) в узлах и конечным значением в корне. Модели материалов, источников света, процедурные текстуры создаются с помощью разных графов, которые затем обрабатываются рендер-системой. Такие графы не могут определять циклы или условное выполнение. В редакторе потоков пикселей (pixel stream), предложенном в [135], пользовательская процедура выполняется для каждого пикселя с использованием некоторых произвольных данных для каждого пикселя в качестве входных. Эти два подхода послужили основой для языка RSL (RenderMan Shading Language) [136]. В RSL есть шейдеры (т.е. программы) разных типов (для источников света, оптических свойств поверхности, объемных эффектов), которые вызываются системой рендеринга.

Язык RSL был разработан как часть рендер-системы RenderMan, а более современный OSL (Open Shading Language) [137] изначально был разработан для рендер-системы Arnold. Обе эти системы рендеринга долгое время были исключительно CPU-рендерами, однако в настоящее время идет разработка версий для графического процессора [138, 139]. OSL был специально разработан для алгоритмов на основе трассировки лучей. Он основан на проекте LLVM и использует JIT-компиляцию. Шейдеры сначала компилируются в байт-код, а затем

транслируются в инструкции x86/x64 процессора. Подходы к созданию расширений в виде пользовательских программ, которые затем обрабатываются полноценными компиляторами, безусловно, самые мощные и гибкие. Их основные недостатки - высокая сложность, высокая стоимость разработки и трудоемкость отладки. Также, большинство существующих решений используют CPU: RSL, OSL, VEX [140] и другие [141]. Следует, однако, отметить, что GPU рендер-система Octane реализует подмножество OSL для процедурных текстур [142].

Существует несколько хорошо известных шейдерных языков ориентированных и на GPU: GLSL, HLSL, Cg, Metal. Некоторые подходы предлагают новые языки, которые компилируются в один или несколько из этих шейдерных языков [143], для улучшения переносимости между различными механизмами визуализации. В [144] авторы предлагают подход к построению шейдеров из модульных компонентов, написанных на предметно-ориентированном языке. В языке шейдеров, предложенном в [145], шейдеры сопоставляются более чем с одним этапом графического конвейера и реализуют концепцию наследования из объектно-ориентированного программирования, чтобы сделать шейдеры легко расширяемыми и повторно используемыми. В [146] предлагается фреймворк, подходящий для создания пользовательских шейдерных конвейеров. Эти конвейеры отображаются на серию вычислительных ядер, которые затем выполняются последовательно. В качестве целевого аппаратного обеспечения могут выступать как GPU, так и многоядерные CPU. Авторы демонстрируют применение для традиционного графического конвейера (растеризации) и алгоритма REYES.

Авторы [147] предлагают шейдерный язык для трассировки лучей (RTSL), который основан на GLSL и в некоторой степени на RSL. RTSL ориентирован на CPU рендер-системы и использует SIMD-инструкции.

Другое решение – движок трассировки лучей OptiX от Nvidia [36]. Он предоставляет пользователю возможность создавать программы разных типов – подход, аналогичный обычным графическим API. OptiX может быть аппаратно ускорен на оборудовании Nvidia с помощью технологии RTX. Как было

рассмотрено в первой главе, аппаратное ускорение трассировки лучей доступно и в графических API таких как Vulkan и DirectX12. Более ранним технологическим аналогом схожих идей с аппаратным ускорением было решение Caustic Graphics OpenRL [148], которое не достигло широкого коммерческого использования.

Ключевой недостаток использования предметно-ориентированных языков в том, что алгоритмы и знания написанные на них трудно переносить на другие области и трудно состыковать с остальной частью ПО, не использующей предметно-ориентированный язык. Кроме того, следует отметить сложность такого решения, связанную с необходимостью реализации инструментов программирования (англ. toolchain), необходимых для компиляции или интерпретации кода на предметно-ориентированном языке. В настоящее время есть несколько вариантов графического конвейера и конвейера трассировки лучей, в которых один алгоритм распределяется на нескольких шейдерных программ (от 2 до 5), что дополняется необходимостью настройки состояний графического конвейера или конвейера трассировки лучей в клиентском CPU коде. Это делает процесс написания программы неочевидным, поскольку нет одного связанного описания алгоритма, а вместо этого имеется множество разрозненных программ и настраиваемые связи между ними в отдельном месте.

4.2.3 Визуальное программирование

Недостатки предметно-ориентированных языков отчасти привели к тому, что многие приложения для создания 3D сцен и игровые движки предлагают пользователям специальные средства для создания своих шейдерных программ (например, [149, 150]). Эти средства можно отнести к визуальному программированию – пользователю предоставляется своего рода «конструктор» из отдельных блоков, представляющих разные операции, такие как вычисление BRDF, выборка из текстуры (как изображения, так и процедурной), арифметические операции, интерполяция, смешение цветов и другие. Каждый из таких блоков имеет набор входов и выходов для данных определенного типа. Пользователь составляет граф операций, соединяя блоки между собой, чтобы получить нужный

результат, который передается в специальный блок выходного значения. Например, такой граф может описывать подход к процедурному затенению объекта для получения эффекта ржавчины (рис. 24). В этом случае граф может состоять из комбинаций различных готовых (т.е. оформленных в виде блока) процедурных текстур шума, выступающих в качестве масок для смешения нескольких BRDF, также заданных готовыми блоками.

Возможности подхода на основе графов естественным образом ограничены количеством реализованных блоков. Но данный подход нельзя в полной мере назвать самостоятельным – пользовательские графы так или иначе должны быть преобразованы в код на некотором шейдерном языке, который затем поступит на вход рендер-системе.

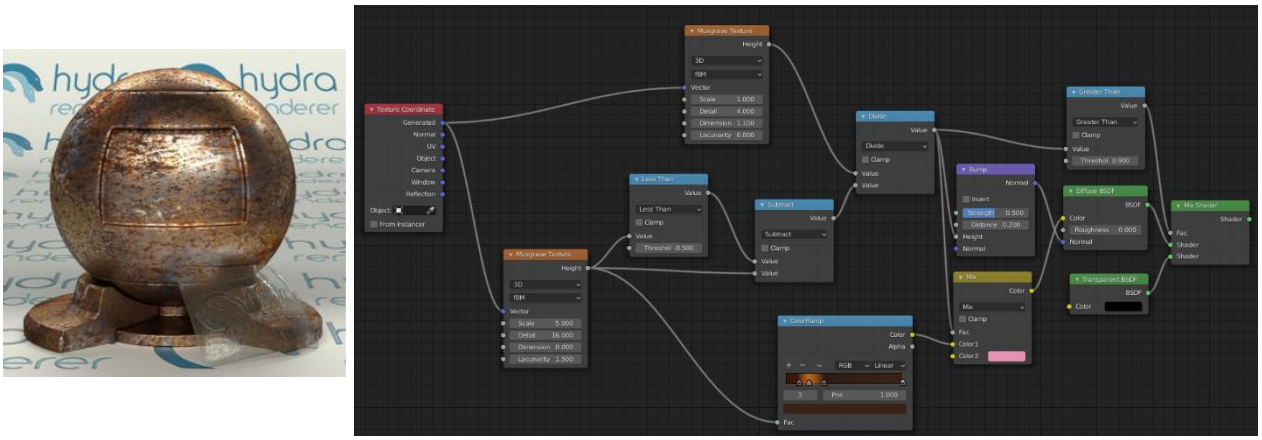


Рис. 24 Слева – визуализация объекта с наложенным комбинированным материалом из чистого металла и ржавчины, справа – фрагмент графа этого материала в редакторе Blender, описывающий материал только ржавчины.

4.2.4 Реализации в существующих рендер-системах

В коммерческой рендер-системе Octane [142] возможности создания пользовательских расширений с помощью шейдерного языка OSL ограничены процедурными текстурами, виртуальными текстурами и геометрическими примитивами, заданными как поля расстояний со знаком (signed distance fields, SDF).

В рендер-системе с открытым исходным кодом Cycles [151] расширения на языке OSL поддерживаются только для рендеринга на CPU. При рендеринге на GPU пользователи ограничены предоставленным набором блоков, которые могут

быть использованы в редакторе Blender для создания шейдерных графов. Пользовательские шейдерные графы экспортируются во внутреннее представление и исполняются (интерпретируются) компонентом Cycles, называющимся виртуальной машиной шейдеров (Shader Virtual Machine, SVM). SVM содержит реализацию каждого типа блока (узла графа) на языке C с ограничениями, позволяющими исполнять этот код как на CPU, так и на GPU с использованием OpenCL или OptiX в зависимости от производителя GPU.

В других GPU рендер-системах возможности создания расширений также являются ограниченными. Гибридная (CPU + GPU) рендер-система RenderMan XPU поддерживает шейдерный язык OSL, но с рядом ограничений, – в частности, она не позволяет пользователю писать т.н. замыкания для материалов (material closures), т.е. реализовывать ДФР [152]. Похожим образом реализована поддержка OSL в рендер-системе RedShift [153]. Таким образом, пользователям не предоставляется (без вмешательства в исходные коды самой рендер-системы) возможность создавать свои BRDF или модели источников света.

Таким образом, среди существующих рендер-систем наиболее распространенным является подход создания расширений на основе комбинирования готовых функциональных блоков, реализованных в рендер-системе, при помощи среды визуального программирования или языка OSL. Результат либо интерпретируется рендер-системой (как в Cycles), либо компилируется компилятором шейдерного языка. При этом механизм создания расширений преимущественно используется для создания процедурных текстур.

4.3 Предлагаемый метод

4.3.1 Общее описание

Представленный метод впервые опубликован в работе [4].

Предлагаемое решение схоже с подходом предметно-ориентированных (шейдерных) языков. Пользователям предоставляется возможность создавать в качестве расширений программы на подмножестве языка C99, которые затем

автоматически подвергаются обработке и исполняются в процессе работы рендер-системы в соответствующем месте алгоритма синтеза изображений. Основными отличиями предлагаемого решения от подхода шейдерных языков являются:

- использование подмножества языка C99 в качестве языка создания расширений (а не некоторого нового языка);
- ввод ограничений на разрабатываемые расширения.

Выбор языка расширений позволяет обеспечить кроссплатформенность предлагаемого решения и избежать недостатков шейдерных языков, описанных в начале главы. Кроме того, как было рассмотрено в разделе 1.4, важным фактором для рендер-систем является поддержка аппаратной трассировки. Однако, большинство рендер-систем, реализующие такую поддержку используют технологию Nvidia OptiX, доступную на аппаратном обеспечении только одного производителя. Если же к требованию поддержки аппаратной трассировки лучей добавить кроссплатформенность, то из технологий программирования GPU остается лишь графический API Vulkan. Данный API принимает на вход шейдеры в виде SPIR-V – бинарного промежуточного языка для представления шейдеров графического конвейера, вычислительных ядер и, с помощью расширений, конвейера аппаратной трассировки лучей. Использование SPIR-V теоретически позволяет Vulkan API использовать шейдеры, исходно написанные на разных языках и затем скомпилированные в SPIR-V. Помимо компиляторов для GLSL и HLSL активно разрабатывается и компилятор для OpenCL C [64]. Таким образом, предлагаемое решение в перспективе может использовать аппаратное ускорение трассировки лучей, а также поддерживает исполнение кода как на GPU, так и на CPU.

Предлагаемое решение ограничивает возможные расширения созданием процедурных текстур и поддержкой новых геометрических примитивов. Эти две задачи покрывают широкий круг задач и, как показал обзор существующих рендер-систем, обычно рассматриваются как достаточные. Кроме того имеется и ряд других причин для ввода ограничений на пользовательские расширения.

Во-первых, программирование и расширение функциональности моделей материалов или источников света является нетривиальной задачей для конечного пользователя (например, 3D художника). Это в основном связано с тем, что современные системы рендеринга используют сложные алгоритмы переноса света с многократной выборкой по значимости. Поэтому для каждой модели материала и источника света необходимо не только выполнять выборку значений, но и рассчитывать их плотность вероятности. При этом, если используются алгоритмы на основе двунаправленной трассировки лучей [22], то это нужно делать для двух случаев – прямой и обратной трассировки. Получение правильного решения для такой задачи крайне затруднительно для рядового пользователя рендер-системы, а неправильное решение приведет к некорректности всего расчета освещения.

Во-вторых, если пользовательские расширения затрагивают расчетное ядро рендер-системы или же рендер-система использует подход на основе «убер-ядра» (см. раздел 1.3), то это приводит к повторной компиляции для всей системы (подход OptiX) каждый раз, когда происходит изменение. Поскольку пользователь (в частности, 3D-художник) может часто редактировать модель материалов в процессе *итеративной работы* над 3D сценой, это может привести к значительным задержкам в отклике рендер-системы, что, в свою очередь, приводит к неудобствам в работе и нарушает принцип интерактивности (см. раздел 1.1). Наконец, когда пользовательские расширения оказываются слишком тесно связаны с рендер-системой, это приводит к усложнению отладки и поиску возможных ошибок при разработке расширений. Таким образом, ограничения на разрабатываемые расширения позволяют обеспечить обособленность кода расширений и избежать его перемешивания с кодом самой рендер-системы, основная, «фиксированная», функциональность которой остается неизменной.

4.3.2 Математическое представление предлагаемых пользовательских расширений

Процедурные текстуры. Математически, возможность создания процедурных текстур означает, что пользователь может параметризовать

вычисление ДФР в интеграле освещенности (1-1) по аналогии с обычными текстурами (3-1). Но для процедурных текстур в качестве аргументов может выступать произвольный набор параметров, не ограниченный лишь текстурными координатами. Кроме того, процедурные текстуры зачастую являются «сплошными» (англ. solid textures), т.е. определены в $\mathbb{R}^3$ (в отличии от «обычных» текстур-изображений, определенных в $\mathbb{R}^2$). При этом пользователь может задавать отображение U из некоторого набора параметров в текстурные координаты:

$$U: X \to Y, X = \{(x_1, \dots, x_n) \in \mathbb{R}^n\}, Y = \{(u, v, w) \in \mathbb{R}^3\} \qquad (4\text{-}1)$$

В качестве параметров $x_1, \dots, x_n$ могут выступать, например, координаты объекта в локальном или мировом пространстве 3D сцены, время или номер кадра для анимированных текстур и т.д.

С учетом сказанного, уравнение рендеринга (1-1) для ДФР, параметризованной k процедурными текстурами может быть записано следующим образом:

$$L_o(x, \omega_0) = L_e(x, \omega_0) + \int_\Omega L_i(x, \omega_i) f(x, \omega_0, \omega_i, T_1, \dots, T_k)(\omega_i \cdot n) d\omega_i$$
$$T_i = \int \hat{f}_i(U_i, x_1, \dots x_n) t_i(U_i, x_1, \dots x_n, a_1, \dots, a_m) du dv dw, 1 \le i \le k \qquad (4\text{-}2)$$

где T_i – значение, получаемое при вычислении и фильтрации процедурной текстуры, $\hat{f}_i$ – функция фильтрации процедурной текстуры, t_i – процедурная текстура, U_i – отображение вида (4-1), $x_1, \dots x_n$ – параметры для вычисления текстурных координат u, v, w и $a_1, \dots a_m$ – параметры процедурной текстуры.

Таким образом, пользователь при создании расширения определяет функцию t_i, которая реализует внутри себя некоторый алгоритм или математическую модель, вычисляющую соответствующий параметр ДФР в зависимости от параметров $a_1, \dots a_m$ и отображение U_i для получения текстурных координат. Опционально, пользователь может определить функцию $\hat{f}_i$, обеспечивающую фильтрацию значений процедурной текстур, чтобы избежать эффекта наложения (алиасинга). Если $\hat{f}_i$ не определена, то используется поточечная выборка (англ.

point sampling), т.к. в общем случае алгоритм вычисления процедурной текстуры может быть сложным и вычисление нескольких выборок является вычислительно затратным. Также отметим, что, как было рассмотрено в разделе 3.1.2, рендер-система может использовать постановку задачи расчета освещения, при которой фильтрация вынесена во внешний интеграл. В качестве параметров $a_1, \dots a_m$ могут выступать как некоторые числовые значения, так и другие текстуры (рис. 25).

Рис. 25 Сцена с простыми процедурными текстурами. На задней, нижней и верхней стенах процедурная текстура вычисляет диффузный цвет в ДФР Ламберта как смешение двух обычных текстур-изображений с использованием фактора окклюзии окружающей среды (ambient occlusion, AO) в качестве веса. На левом кубе вычисляется смешение двух константных цветов, на кубе по центру – смешение текстуры-изображения и константного цвета, на кубе справа – смешение двух текстур-изображений с локальными (модельными) координатами куба в качестве веса.

В представленном на рис. 25 примере модель материала задней стены, пола и потолка представляет собой ДФР Ламберта, в которой диффузный цвет представлен процедурной текстурой, которая смешивает две текстуры-изображения a_1 и a_2 используя в качестве веса фактор окклюзии окружающей среды $A \in \mathbb{R}^3, A \epsilon [0,1]$, представляющий собой оценку интеграла функции видимости по полусфере, построенной вокруг нормали к поверхности 3D модели в точке вычисления. В данном примере для этого фактора задается экспоненциальное затухание с показателем степени, являющимся параметром процедурной текстуры a_3. Отображение U_i не задано и используются заранее рассчитанные текстурные координаты (u, v) для соответствующих 3D моделей, на которые накладывается текстура. Таким образом, выражение для процедурной текстуры t_i имеет вид:

$$t_i(u, v, a_1, a_2, a_3) = (1 - A^{a_3}) * a_1(u, v) + (1 - (1 - A^{a_3})) * a_2(u, v) \quad (4\text{-}3)$$

Модель процедурной текстуры может иметь и гораздо более сложный вид. В частности, многие процедурные текстуры строятся на основе так называемых функций «шума» (англ. noise functions) [84], – непрерывных и не имеющих разрывов функций, исходно предложенных Кеном Перлином [85] и описанных им как «повторяемые псевдослучайные функции с ограниченным диапазоном частот».

$$f_{noise}: \mathbb{R}^n \to X, X = \{x \epsilon \mathbb{R}, -1 \leq x \leq 1\}, n \epsilon [1,2,3] \quad (4\text{-}4)$$

В [85] и в модификациях в [84], f_{noise} представляет собой гладкую функцию, с нулевыми значениями в каждой точке целочисленной решетки. При этом на этой решетке заданы градиенты функции f_{noise}, которые определяются псевдослучайным образом. Таким образом, значение функции f_{noise} в точке 3D сцены с некоторыми действительными координатами $(x, y, z) \epsilon \mathbb{R}$ вычисляется с помощью трилинейной интерполяции между вершинами трехмерной ячейки, в которую попадает эта точка. А для определения весов интерполяции используются градиенты в вершинах этой трехмерной ячейки.

Типично применение функции шума состоит в передаче ей в качестве аргументов координаты в мировом пространстве 3D сцены для получения

псевдослучайного значения, используемого для формирования некоторого паттерна в цвете объекта сцены. Для получения паттерна, имеющего детали, меняющиеся в пространстве в разном масштабе (или, другими словами, с разной скоростью), часто комбинируют несколько функций шума, используя т.н. «спектральный синтез», при котором результирующая функция определяется как сумма вкладов другой функции [84, с. 85]:

$$f_s(x) = \sum_{i=1}^{i \leq n} w_i f_{noise}(s_i x) \tag{4-5}$$

где w_i – веса, s_i – скалярные коэффициенты масштабирования.

Если функция f_{noise} имеет ограниченный диапазон частот, то и функция f_s также имеет ограниченный диапазон частот. Таким образом, f_s представляет собой сумму вкладов разных диапазонов частот. В качестве коэффициентов масштабирования в s_i рассматриваемом примере используется геометрическая прогрессия:

$$s_i = l * s_{i-1}, s_0 = 1 \tag{4-6}$$

где $l \in \mathbb{R}$ – некоторая константа, являющаяся параметром модели.

А веса w_i задаются как:

$$w_i = w_{i-1} * l^{-H} \tag{4-7}$$

где $H \in \mathbb{R}, H \in [0, 1]$ – некоторая константа, являющаяся параметром модели.

Тогда бóльшие частоты будут меньше влиять на форму результирующей функции f_s. Каждое следующее слагаемое $w_i f$ в сумме называется «октавой шума».

Модель материала ржавчины на рис. 26 представляет собой смешение двух ДФР – ДФР основного материала f_1 (серой матовой краски краски) и ДФР Ламберта ржавчины f_2. Процедурные текстуры используются в модели для двух задач – задания маски M для смешения f_1 и f_2, и для вычисления диффузного цвета в f_2.

Рис. 26 Сцена с процедурными текстурами, имитирующими ржавчину.

Уравнение освещенности с использованием этой модели материалов можно записать следующим образом:

$$L_o(x,\omega_0) = L_e(x,\omega_0) + \int_\Omega L_i(x,\omega_i) f(x,\omega_0,\omega_i,T_1,T_2)(\omega_i \cdot n) d\omega_i$$

$$f(x,\omega_0,\omega_i,T_1,T_2) = f_2(x,\omega_0,\omega_i,T_2) * T_1 + f_1(x,\omega_0,\omega_i) * (1 - T_1)$$

$$T_1 = \int \hat{f}_1(U_1,x_1,x_2) t_1(U_1,x_1,x_2,a_1,\dots,a_m) dudvdw \qquad (4\text{-}8)$$

$$T_2 = \int \hat{f}_2(U_2,x_1) t_2(U_2,x_1,a_1,\dots,a_m) dudvdw$$

Обе процедурные текстуры t_1 и t_2 основаны на использовании модели шума, описанной в выражениях (4-5), (4-6), (4-7) а также цветовых картах, отображающих числовые значения с помощью кусочно-линейной интерполяции константных значений цвета.

Отображения U_1 и U_2 используют координаты точки $x_1 = (\hat{x}_1, \hat{y}_1, \hat{z}_1) \epsilon \mathbb{R}^3$, где происходит вычисление текстуры, в мировом пространстве 3D сцены. Отображение U_2 использует эти координаты напрямую – они попадают в неизменном виде в качестве аргумента для суммарной функции шума f_s

(выражение 4-5). Таким образом, цвет ржавчины непрерывно меняется во всем пространстве 3D сцены. А отображение U_1 «нормализует» мировые координаты относительно размеров объекта, заданных в виде минимальных и максимальных координат вершин ограничивающего параллелепипеда 3D объекта $x_2 = \left(\hat{x}_2^{min}, \hat{x}_2^{max}, y_2^{min}, y_2^{max}, \hat{z}_2^{min}, \hat{z}_2^{max}\right) \epsilon \mathbb{R}^6$, на который накладывается текстура:

$$U_1: (x_1, x_2) \rightarrow (u, v, w)$$
$$u = \frac{\hat{x}_1}{\hat{x}_2^{max} - \hat{x}_2^{min}}, v = \frac{\hat{y}_1}{\hat{y}_2^{max} - \hat{y}_2^{min}}, w = \frac{\hat{z}_1}{\hat{z}_2^{max} - \hat{z}_2^{min}} \quad (4\text{-}9)$$

За счет этого в процедурной текстуре T_2 техническим художником может контролироваться локализация материала ржавчины на 3D объекте. А масштаб деталей в текстуре контролируется за счет введения дополнительных параметров – множителей для координат.

Таким образом, процедурные текстуры могут описывать достаточно сложные математические модели, параметры которых позволяют конечному пользователю контролировать результат их применения в сцене.

Геометрические примитивы. Возможность реализации пользователями поддержки новых геометрических примитивов означает возможность задания функции $Intersect$ поиска пересечения луча и нового геометрического примитива:

$$Intersect: X \rightarrow Y$$
$$X = \{(o, d) | o \epsilon \mathbb{R}^3, d \epsilon \mathbb{R}^3\}$$
$$Y = \{(i, t_1, \dots, t_k) | i \epsilon [0,1], t_j \in \mathbb{R}, t_j \geq 0, 1 \leq j \leq k\} \quad (4\text{-}10)$$

где o – координаты начальной точки луча; d – вектор направления луча; i – индикаторная переменная, принимающая значение 0, если примитив и луч (o, d) не имеют общих точек и значение 1 в противном случае; $t_1, \dots, t_k$ – значения расстояния от начала луча до каждой из k точек пересечения (или же значение расстояния только до ближайшей точки пересечения).

Часто это означает, что функция $Intersect$ должна реализовывать решение системы уравнений, включающей параметрическое уравнение луча и некоторые

уравнения для нового геометрического примитива. Например, если геометрический примитив задан неявной функцией:

$$\begin{cases} x = o_x + t * d_x \\ y = o_y + t * d_y \\ z = o_z + t * d_z \\ f(x, y, z) = 0 \end{cases}, t \geq 0 \quad (4\text{-}11)$$

Простым примером такого геометрического примитива может служить квадрика – сфера, конус, парабалоид и т.д. В этом случае пересечение может быть найдено с помощью подстановки параметрических уравнений луча в неявную функцию, которая даст квадратное уравнение относительно t.

Другой пример возможного пользовательского примитива – поле расстояний со знаком (англ. signed distance field), которое может быть задано знаковой функцией расстояния $f: \mathbb{R}^3 \to \mathbb{R}$. Пользователь может задать её с помощью уравнения, алгоритма или, например, в виде предварительно рассчитанной трехмерной сетки значений.

4.3.3 Программная реализация

В предлагаемом решении реализуется подход нескольких «убер-ядер» – все пользовательские расширения для геометрии в данной сцене собираются в одно вычислительное ядро, а для вычисления цвета (т.е. процедурных текстур) – в другой (рис. 27). В процессе работы рендер-системы код пользовательских расширений при необходимости исполняется в нужном месте алгоритма синтеза изображений. Таким образом, код расширений остается обособленным и не перемешивается с кодом самой рендер-системы, чья основная, «фиксированная» функциональность остается неизменной. Такое решение позволяет сделать модульную архитектуру рендер-системы – допускающую замену части алгоритма трассировки лучей без влияния на другие части.

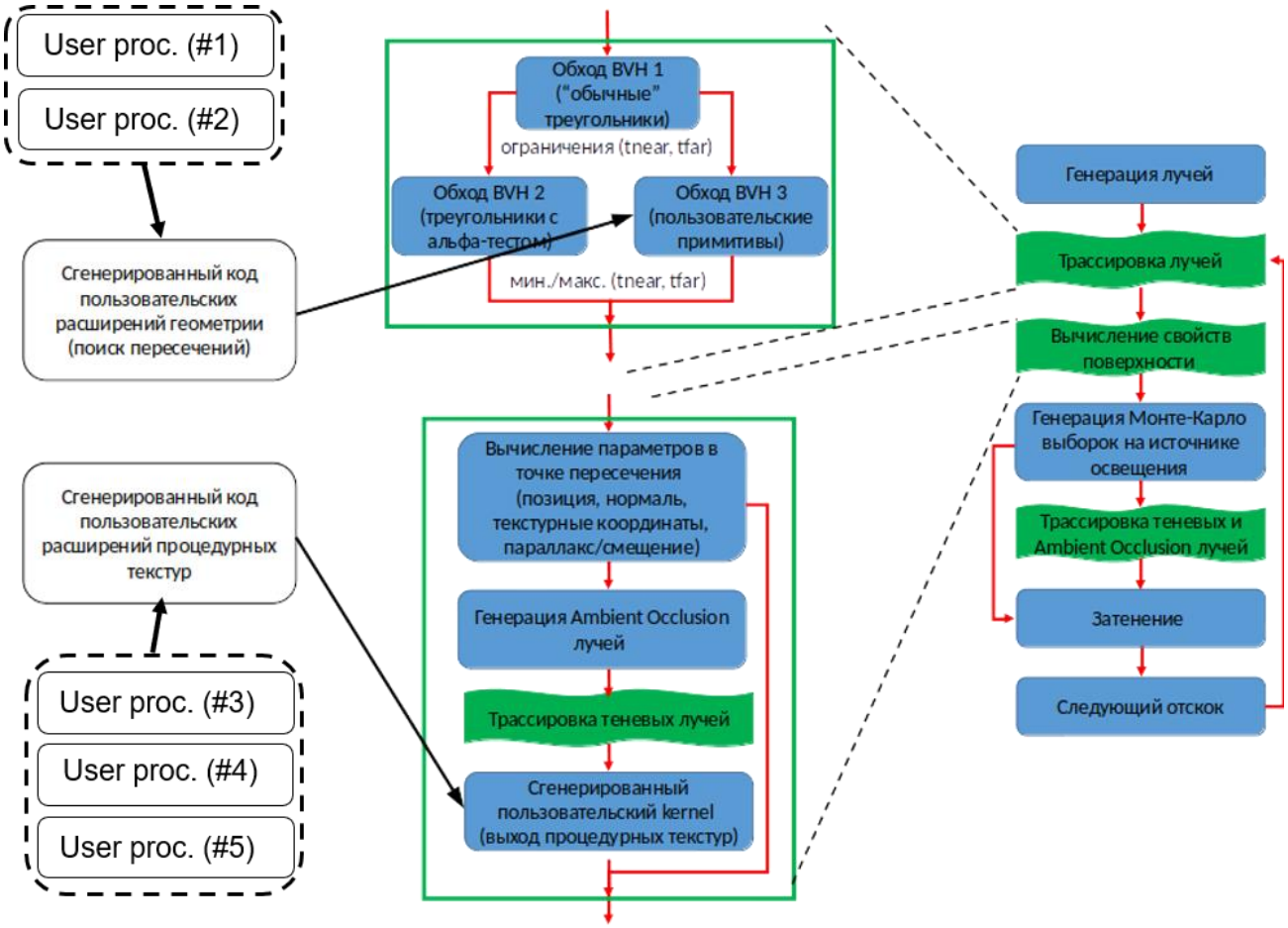


Рис. 27 Архитектура нескольких «убер-ядер» для рендер-системы с использованием пользовательских расширений. Пользователи определяют процедуры расчета цвета (user proc. 3, 4, 5) и поиска пересечений луча и нового геометрического примитива (user proc. 1, 2), из которых предлагаемое решение генерирует вычислительные ядра, которые являются опциональным этапом всего алгоритма синтеза изображений.

Новая функциональность может быть добавлена путем введения нового вычислительного ядра как дополнительного шага в алгоритме. Для поддержки процедурных текстур предлагаемое решение добавляет новое вычислительное ядро, который исполняет код процедурной текстуры при наличии пересечения с соответствующим объектом сцены и записывает результат в глобальную память, никак не влияя на другие шаги процесса синтеза изображения. Аналогичным образом интегрируются расширения для геометрических примитивов – новое вычислительное ядро исполняет код поиска пересечения луча с примитивом в процессе обхода отдельной ускоряющей структуры.

Рассмотрим механизм передачи аргументов в предложенном подходе на примере процедурных текстур (рис. 28). Для хранения аргументов пользовательских функций выделяется отдельная область памяти – «псевдостек». С точки зрения генератора кода она рассматривается как обычный стек, в который генератор может помещать аргументы. Но поскольку на момент генерации все параметры являются константными, «псевдостек» является просто областью глобальной памяти, допускающей только чтение. Это позволяет обновлять её со стороны CPU без необходимости перекомпиляции вычислительных ядер. Установка реального значения аргументов происходит при присоединении текстуры к материалу.

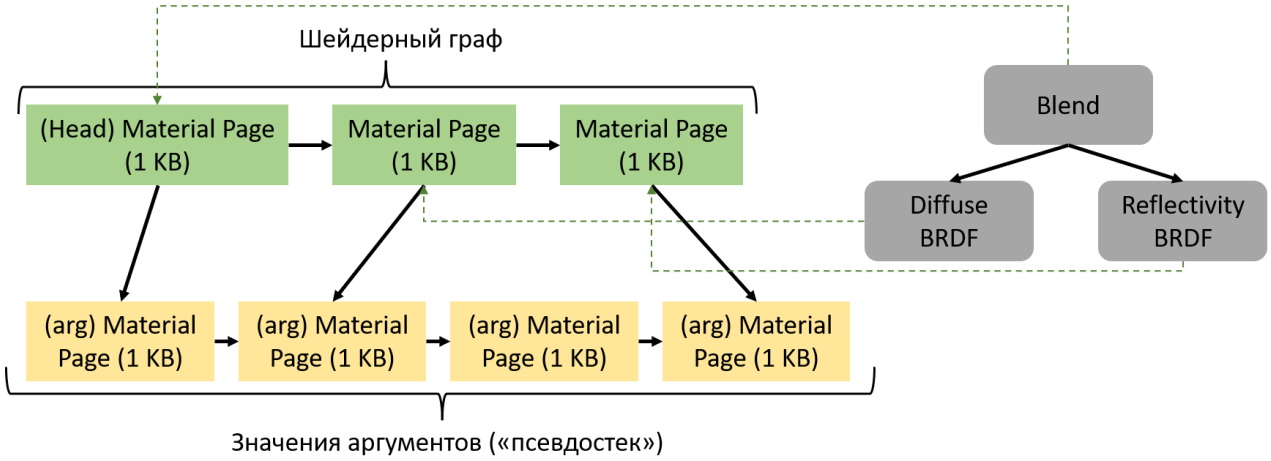


Рис. 28 Пример шейдерного графа (справа, серый цвет) и его расположение в памяти. Страницы для материалов (зеленые прямоугольники) представляют разные узлы шейдерного графа и могут ссылаться на разные части области хранения аргументов («псевдостека», желтые прямоугольники) в том же буфере.

Для исполнения в составе рендер-системы код пользовательских расширений подвергается обработке. Ко всем вызовам пользовательских функций, объявлениями и определениям добавляются уникальные префиксы в соответствии с идентификаторами процедурной текстуры в предложенном интеграционном программном слое (см. главу 2), чтобы избежать конфликтов имен. Также выполняется замещение встроенных вызовов и типов, которые позволяют коду пользовательского расширения получать атрибуты поверхности («readAttr»), глобальные настройки рендер-системы и производить выборку из изображений

(«texture2D»). В заголовок функции добавляется макрос «_PROCTEXTAILTAG_» (листинг 8, строка 3), который непосредственно перед компиляцией OpenCL ядра заменяется на конкретные аргументы функции, которые зависят от версии рендер-системы и обычно включают указатели на буфер, содержащий текстуры-изображения, и на буфер с глобальными константами. Цель этого макроса в том, чтобы не нужно было заново выполнять обработку пользовательского кода, если произошли какие-то внутренние изменения в реализации рендер-системы, которые влияют на процедурные текстуры.

Выполняемая обработка кода пользовательских расширений является достаточно простой и может быть реализована «вручную» на основе механизма регулярных выражений или же с использованием специальных библиотек, например Clang LibTooling [154], которая предоставляет возможность построения абстрактного синтаксического дерева для исходного кода на OpenCL C и его модификации. Предлагаемый алгоритм обработки представлен ниже (алг. 4):

Алгоритм 4. Обработка кода пользовательских расширений.

Входные данные:

code – строка, содержащая код пользовательского расширения на OpenCL подмножестве языка C99;

main_name – имя функции, являющейся точкой входа для пользовательского расширения;

id – целое число, уникальный идентификатор расширения (например, идентификатор текстуры).

Выходные данные:

$codeOUT$ – строка, содержащая преобразованный код пользовательского расширения на OpenCL подмножестве языка C99;

$argsOUT$ – массив структур, содержащих параметры аргументов функции, являющейся точкой входа для пользовательского расширения;

$call$ – строка вызова функции, являющейся точкой входа для пользовательского расширения, на OpenCL подмножестве языка C99.

Обозначения:

$\${a}$ – значение переменной a, представленное в виде строки;

/* … */ - комментарий.

```
1  args_out ← ∅, codeOUT ← code
2  returnType ← FunctionReturnType(main_name, code)
3  /* Получение аргументов функции, каждый аргумент - структура, содержащая имя, тип и
   размер аргумента (для аргументов, не являющихся массивами размер равен 1)
   args = {arg | arg = {name, type, size}} */
4  args ← FunctionArguments(main_name, code)
5  argsList ← {"${arg.type} ${arg.name}" | arg ∈ args}
6  main ← "${returnType} prtex${id}_${main_name}(${argsList}, _PROCTEXTAILTAG_)"
7  Replace(main_name, main, codeOUT) /* замена заголовка ф-ции main на обработанный */
8  /* обработка и замена заголовков остальных функций */
9  foreach funcName ∈ ExtractFunctionNames(code) do:
10     newName ← "prtex${id}_${funcName}"
11     Replace(funcName, newName, codeOUT)
12 end for
13 call ← "prtex${id}_${main_name}("
14 offset ← 0, i ← 0
15 foreach arg ∈ args do:
16     if arg.size > 1 then: /* если аргумент является массивом */
17         call ← "${call} (__global *)(stack + ${offset}),"
18     else:
19         call ← "${call} make_${arg.type}(stack + ${offset}, ${arg.size}),"
20     end if
21     offset ← offset + arg.size * sizeof(arg.type)
22     a = {arg.name, arg.type, arg.size, arg.size * sizeof(arg.type), offset}
23     argsOUT ← argsOUT ∪ {a}
24 end for
25 call ← "${call}_PROCTEXTAILTAG"
26 return codeOUT, argsOUT, call
```

Пример исходного пользовательского кода процедурной текстуры и результата обработки представлены в листингах 7 и 8 соответственно (пример синтезированного изображения с помощью данных процедурных текстур

представлен на рис. 25). Обработка кода пользовательских расширений позволяет спрятать детали реализации процедурных текстур в рендер-системе от пользователя и при необходимости её менять. В предлагаемом решении намеренно не вводится никаких новых языковых конструкций для пользовательского кода, чтобы он соответствовал C99 и мог быть легко интегрирован в какое-либо стороннее C/C++ приложение. В частности, это позволяет выполнять отладку пользовательских расширений с использованием широкодоступных и развитых средств разработки C/C++ приложений. За счет того, что обработка лишь незначительно изменяет исходный пользовательский код, как видно из примеров выше (листинги 7 и 8), является возможным модифицировать и отлаживать также и сгенерированный код как обычное OpenCL ядро.

```
1.  float4 main(const SurfaceInfo* sHit, float3 colorHit, sampler2D texHit,
2.              float3 colorMiss, sampler2D texMiss, float faloffPower)
3.  {
4.    const float2 texCoord = readAttr(sHit,"TexCoord0");
5.    float ao              = readAttr(sHit,"AO");
6.    const float4 col1     = to_float4(colorHit, 1.0f) * texture2D(texHit,
7.                                      texCoord, 0);
8.    const float4 col2     = to_float4(colorMiss,1.0f) * texture2D(texMiss,
9.                                      texCoord, 0);
10.   ao = 1.0f - pow(ao, faloffPower);
11.   return ao * col1 + (1.0f - ao) * col2;
12. }
```

Листинг 7. Пример пользовательского кода процедурной текстуры, вычисляющей цвет как смешение двух изображений с использованием фактора ambient occlusion.

```
1.  float4 prtex3_main(const SurfaceInfo* sHit, float3 colorHit, sampler2D
2.                     texHit, float3 colorMiss, sampler2D texMiss,
3.                     float faloffPower, _PROCTEXTAILTAG_)
4.  {
5.    const float2 texCoord = readAttr_TexCoord0(sHit);
6.    float  ao             = readAttr_AO(sHit);
7.    const float4 col1     = to_float4(colorHit, 1.0f) * texture2D(texHit,
8.                                      texCoord, 0);
9.    const float4 col2     = to_float4(colorMiss,1.0f) * texture2D(texMiss,
10.                                     texCoord, 0);
11.   ao = 1.0f - pow(ao, faloffPower);
12.   return ao * col1 + (1.0f - ao) * col2;
13. }
```

Листинг 8. Та же процедурная текстура, что и листинг 7, но после обработки.

4.3.4 Интеграция с промежуточным программным слоем

Для интеграции предложенного подхода создания пользовательских расширений с промежуточным программным слоем (глава 2) в программный интерфейс промежуточного слоя были добавлена функции для создания объектов, являющихся расширениями. Для процедурных текстур была добавлена функция *hrTextureCreateAdvanced*. Подобно функциям создания обычных текстур, она создает новый объект текстуры с уникальным идентификатором, но не осуществляет никаких других операций. Далее пользователю необходимо добавить в XML-описание созданного объекта текстуры дочерний узел «code», содержащий в качестве атрибутов путь к текстовому файлу с кодом процедурной текстуры, а также имя точки входа (листинг 9).

```
1.  HRTextureNodeRef texProc  = hrTextureCreateAdvanced(L"proc", L"my_dirt_tex");
2.  hrTextureNodeOpen(texProc, HR_WRITE_DISCARD);
3.  {
4.    xml_node texNode = hrTextureParamNode(texProc);
5.    xml_node code_node = texNode.append_child(L"code");
6.    code_node.append_attribute(L"file") = L"data/code/dirt_ao.c";
7.    code_node.append_attribute(L"main") = L"main";
8.
9.    xml_node aoNode = texNode.append_child(L"ao");
10.   aoNode.append_attribute(L"length")     = 1.0f;
11.   aoNode.append_attribute(L"hemisphere") = L"up";
12.   aoNode.append_attribute(L"local")      = 0;
13. }
14. hrTextureNodeClose(texProc);
```

Листинг 9. Пример создания объекта процедурной текстуры с помощью программного интерфейса промежуточного слоя.

В представленном примере в XML-описании процедурной текстуры также создается узел «ao» (строки 9-12), который сигнализирует рендер-системе о необходимости выполнения расчета окклюзии окружающей среды (Ambient Occlusion) для объекта, на который наложен материал, содержащий данную процедурную текстуру. Данный эффект используется, например, для моделирования загрязнений (рис. 36).

При вызове функции *hrTextureClose*, производится проверка узлов и атрибутов текстуры. Если текстура является процедурной и будет найден узел «code», то промежуточный слой загрузит код процедурной текстуры по заданному

в соответствующем атрибуте пути, после чего будет выполнена его обработка по алгоритму 4. После чего в XML-описание сцены в узел объекта текстуры в библиотеке текстур будет добавлена список аргументов, информация о расположении обработанного кода и строка вызова (<call>) процедурной текстуры, использующая «псевдостек» (рис. 28) для получения аргументов (листинг 10):

```
<texture id="9" name="my_dirt_tex" type="proc">
 <code file="data/code/dirt_ao.c" main="main" loc="data/proctex_00003.c">
  <generated>
   <arg id="0" type="float3" name="colorHit" size="1" wsize="3"
        woffset="0"/>
   <arg id="1" type="sampler2D" name="texHit" size="1" wsize="1"
        woffset="3" />
   <arg id="2" type="float3" name="colorMiss" size="1" wsize="3"
        woffset="4" />
   <arg id="3" type="sampler2D" name="texMiss" size="1" wsize="1"
        woffset="7" />
   <arg id="4" type="float" name="faloffPower" size="1" wsize="1"
        woffset="8" />
   <return type="float4" />
   <call>prtex3_main(sHit, make_float3(stack[0], stack[1], stack[2]),
           as_int(stack[3]), make_float3(stack[4], stack[5], stack[6]),
           as_int(stack[7]), stack[8], _PROCTEXTAILTAG_)</call>
  </generated>
 </code>
 <ao length="1" hemisphere="up" local="0" />
</texture>
```

Листинг 10. Сгенерированное XML-описание объекта процедурной текстуры.

После создания объекта процедурной текстуры необходимо привязать её к конкретному материалу сцены (аналогично обычным текстурам). Процедурная текстура, как и обычные текстуры, может использоваться разными материалами. При этом в разных материалах значения аргументов для процедурного алгоритма могут быть разные. Поэтому при привязке процедурной текстуры к материалу необходимо также указать значения аргументов для главной функции - точки входа (листинг 11).

```
hrMaterialOpen(mat, HR_WRITE_DISCARD);
{
 …

  auto texNode = hrTextureBind(texProc, colorNode);
  xml_node p1 = texNode.append_child(L"arg");
  xml_node p2 = texNode.append_child(L"arg");
  xml_node p3 = texNode.append_child(L"arg");
  xml_node p4 = texNode.append_child(L"arg");
```

```
10.   xml_node p5 = texNode.append_child(L"arg");
11.
12.   p1.append_attribute(L"id")   = 0;
13.   p1.append_attribute(L"name") = L"colorHit";
14.   p1.append_attribute(L"type") = L"float3";
15.   p1.append_attribute(L"size") = 1;
16.   p1.append_attribute(L"val")  = L"1 0 1";
17.
18.   p2.append_attribute(L"id")   = 1;
19.   p2.append_attribute(L"name") = L"texHit";
20.   p2.append_attribute(L"type") = L"sampler2D";
21.   p2.append_attribute(L"size") = 1;
22.   p2.append_attribute(L"val")  = texBitmap1.id;
23.
24.   …
25.  }
26.  hrMaterialClose(mat4);
```

Листинг 11. Пример привязки процедурной текстуры «texProc» к материалу и установки значений аргументов для неё.

Таким образом, в контексте архитектуры интеграционного программного слоя, схема создания пользовательских расширений на примере процедурных текстур состоит из следующих шагов:

1. Пользователь пишет код процедурной текстуры на C99.
2. С помощью функции *hrTextureCreateAdvanced* программного интерфейса пользователь создает объект процедурной текстуры, открывает его на редактирование и указывает путь к файлу, содержащему код текстуры (п.1) и имя точки входа (имя функции).
3. Пользователь привязывает созданный объект текстуры к конкретному материалу (или материалам) и задает в XML-описании аргументы, принимаемые на вход процедурной текстурой.
4. Интеграционный слой выполняет обработку кода процедурных текстур, сохраняет результат и записывает информацию, необходимую для вызова процедурной текстуры в XML-описание сцены.
5. Обработанный код всех процедурных текстур, присутствующих в сцене, объединяется в единое вычислительное ядро, который компилируется и загружается рендер-системой.

4.4 Сравнение и выводы

Было проведено сравнение разработанного решения с двумя методами:

1. «Наивная вставка», - подход, при котором осуществлялась вставка всех констант и кода процедурных текстур в одно вычислительное ядро. Т.е. если некоторая процедурная текстура использовалась несколько раз (в разных материалах, с разными параметрами), то её код также дублировался несколько раз, а для её аргументов создавались новые константы.
2. «RTX», - подход на основе конвейера трассировки лучей в Vulkan API, при котором код процедурных текстур добавлялся непосредственно в шейдер расчета материала (имитация подхода OptiX).

Предлагаемое решение было интегрировано в рендер-систему Hydra [41, 106]. В качестве тестового сценария использовалась 3D сцена Sponza с дополнительной 3D моделью (рис. 29). Сначала на эту 3D модель был назначен серый диффузный материал, после чего был произведен рендеринг каждым из подходов с замером времени. Далее серый материал 3D модели менялся на материал, использующий процедурные текстуры, и снова производился рендеринг. Замерялось падение производительности при добавлении в сцену процедурных текстур, а также, размер стекового кадра для вычислительных ядер OpenCL для предложенного подхода и подхода наивной вставки.

Для тестирования использовались два материала с процедурными текстурами. Первый материал моделировал эффект ржавчины и представлял собой смешение двух материалов – основного материала объекта (без процедурных текстур) и материала ржавчины. Для материала ржавчины использовались процедурные текстуры из комбинации функций шума и цветовых карт, более подробно описанные в разделе 4.3.2. Смешение материалов происходило по маске, задаваемой процедурной текстуры также из функций шума, а также значений окклюзии окружающей среды (англ. ambient occlusion). Второй материал моделировал эффект царапин и представлял собой смешение двух материалов без

процедурных текстур по маске, заданной процедурной текстурой на основе функции ячеистого шума («паттерн Вороного», по аналогии с диаграммой Вороного [155]). Результаты сравнения представлены в таблице 3.

Процедурная текстура	Наивная вставка	RTX	Предлагаемое решение
Ржавчина (прирост времени)	+46%	+33%	+21%
Царапины (прирост времени)	+9%	+4%	+15%
Ржавчина (размер стекового кадра)	604 байт	-	464 байт
Царапины (размер стекового кадра)	412 байт	-	336 байт

Таблица 3 Результаты сравнения. Прирост времени – изменение времени рендеринга в процентах в сцене при добавлении материала с процедурным текстурами по сравнению с серым диффузным материалом. Размер стекового кадра был получен с помощью флага «-cl-nv-verbose» компилятора OpenCL. Рендеринг производился на Nvidia RTX 2070.

Предложенный подход нескольких «убер-ядер» демонстрирует наименьшее увеличение времени рендеринга для сложного материала с процедурными текстурами (ржавчины). Для более простой процедурной текстуры (царапины) предлагаемое решение теряет в скорости больше, чем базовые методы, т.к. в этом случае вставка кода оказывается достаточно эффективной и накладные затраты на запуск отдельного вычислительного ядра процедурных текстур оказываются более значительными, чем его выполнение. Ожидаемо, предлагаемое решение демонстрирует меньший размер стекового кадра по сравнению с подходом наивной вставки. Данный параметр был выбран для сравнения в связи с тем, что регистровое давление является одним из факторов при выборе подхода к реализации рендер-системы на GPU (см. подраздел 1.3.2). В случае, если регистров не хватает для хранения всех переменных, они могут быть сохранены в память (т.н. «протекание регистров», англ. register spilling), что может значительно повлиять на эффективность выполнения кода.

Рис. 29 Тестовая сцена. Слева – базовое состояние с серым диффузным материалом на модели, по центру – модель с материалом ржавчины, справа вверху – приближенный фрагмент процедурной ржавчины, справа по центру и справа внизу – материалы ржавчины и царапин соответственно на другой модели и при другом освещении для наглядности.

Глава 5. Программное решение и практические применения

5.1 Интеграция GPU рендер-системы в программные продукты для создания 3D сцен

Реализация интеграционного слоя выполнена на языке C++ с отдельными компонентами на языке C (взаимодействие с ОС). Также используется ряд сторонних библиотек с открытым исходным кодом:

1. FreeImage [156] для работы с изображениями.
2. Pugixml [75] для работы с XML-файлами в памяти и на диске.
3. IESNA для работы с IES-файлами представления фотометрических источников света.
4. GLFW [157] для кроссплатформенной работы с OpenGL API в отладочных рендер-системах.
5. Corto [158] для сжатия геометрических моделей.
6. Mikktspace [159] для расчета касательных в геометрических моделях.
7. Pybind11 [160] для программного интерфейса интеграционного слоя на языке Python.

В предполагаемом сценарии использования интеграционного слоя пользователь скачивает подключает его как статическую библиотеку к своему проекту. Сборка интеграционного слоя может быть выполнена заранее или в качестве компонента пользовательского проекта. Интеграционный слой может использоваться под управлением ОС семейства Windows и дистрибутивов ОС GNU/Linux.

Разработанная программная архитектура промежуточного интеграционного слоя и описанные в предыдущих главах подходы внедрены в открытую GPU систему фотореалистичного рендеринга Hydra Renderer [41, 106].

Интеграционный слой используется для интеграции Hydra Renderer с несколькими программными продуктами.

Autodesk 3ds Max – программное обеспечение для 3D-моделирования, анимации и визуализации. ПО 3ds Max используется в том числе для таких задач как архитектурная и предметная визуализация. Рендер-система Hydra Renderer ранее была интегрирована с данным 3D редактором через экспорт сцены в формате COLLADA. При этом плагины для 3ds Max выполняли непосредственную запись файлов в данном формате, что приводило к появлению ошибок при неправильной записи. Также имелись все недостатки, свойственные механизму обмена файлами – полный экспорт всей сцены даже при небольших её изменениях, а следовательно длительный процесс экспорта и ожидание пользователя начала рендеринга, сложность отладки (трудность разделения в некоторых ситуациях ошибок рендеринга от ошибок экспорта). Добавление новых параметров, например, в модель материалов или в настройки рендер-системы требовало внесения изменений в нескольких различных точках программы плагина:

1. Графический интерфейс пользователя.
2. Экспорт данных во внутренние структуры данных для промежуточного хранения и обработки.
3. Запись внутренних структур данных в выходные файл(ы) формата COLLADA.

Перенос интеграции на предлагаемую архитектуру позволил заменить п.2 и 3 из представленного списка на передачу данных через программный интерфейс промежуточного слоя. Также были обеспечены описанные в первой главе требования. В частности, поддержано использование механизмов отслеживания изменений в сцене, позволившее избежать повторного экспорта неизменившихся компонентов, скрыты детали реализации хранения и передачи сцены от плагина, предоставлены механизмы отладки и тестирования (история изменений сцены, специальные рендер-системы для отладочной отрисовки). На рис. 30 представлены некоторые работы пользователей Hydra Renderer, созданных в 3ds Max.

Рис. 30 Примеры синтезированных изображений 3D сцен, созданных пользователями Hydra Renderer в 3ds Max.

LightCAD [161] – отечественный программный продукт для светодизайна и проектирования управляемых систем освещения, разработка компании Интилед – производителя систем освещения для фасадов, мостов, транспортной инфраструктуры, спортивных сооружений, высотных зданий и масштабных объектов [162]. Разработанный промежуточный слой был использован для осуществления интеграции Hydra Renderer в LightCAD в качестве системы фотореалистичного рендеринга (рис. 4, рис. 31). Следует отметить, что LightCAD

написан на языке программирования C#, что демонстрирует возможность использования интеграционного слоя с программным обеспечением на этом языке.

Рис. 31 Проект освещения торгового комплекса, выполненный компанией «Интилед» с использованием разработанной системы.

Fabric Engine [163] – платформа для создания инструментов для визуальных эффектов и компьютерных игр, состоящая из собственного языка программирования и среды визуального программирования. Сторонним разработчиком с помощью предложенного промежуточного слоя была реализована начальная интеграция Hydra Renderer в Fabric Engine (рис. 32). Несмотря на то, что разработка и распространение самой платформы была прекращена, этот пример демонстрирует использование программного интерфейса интеграционного слоя сторонними разработчиками.

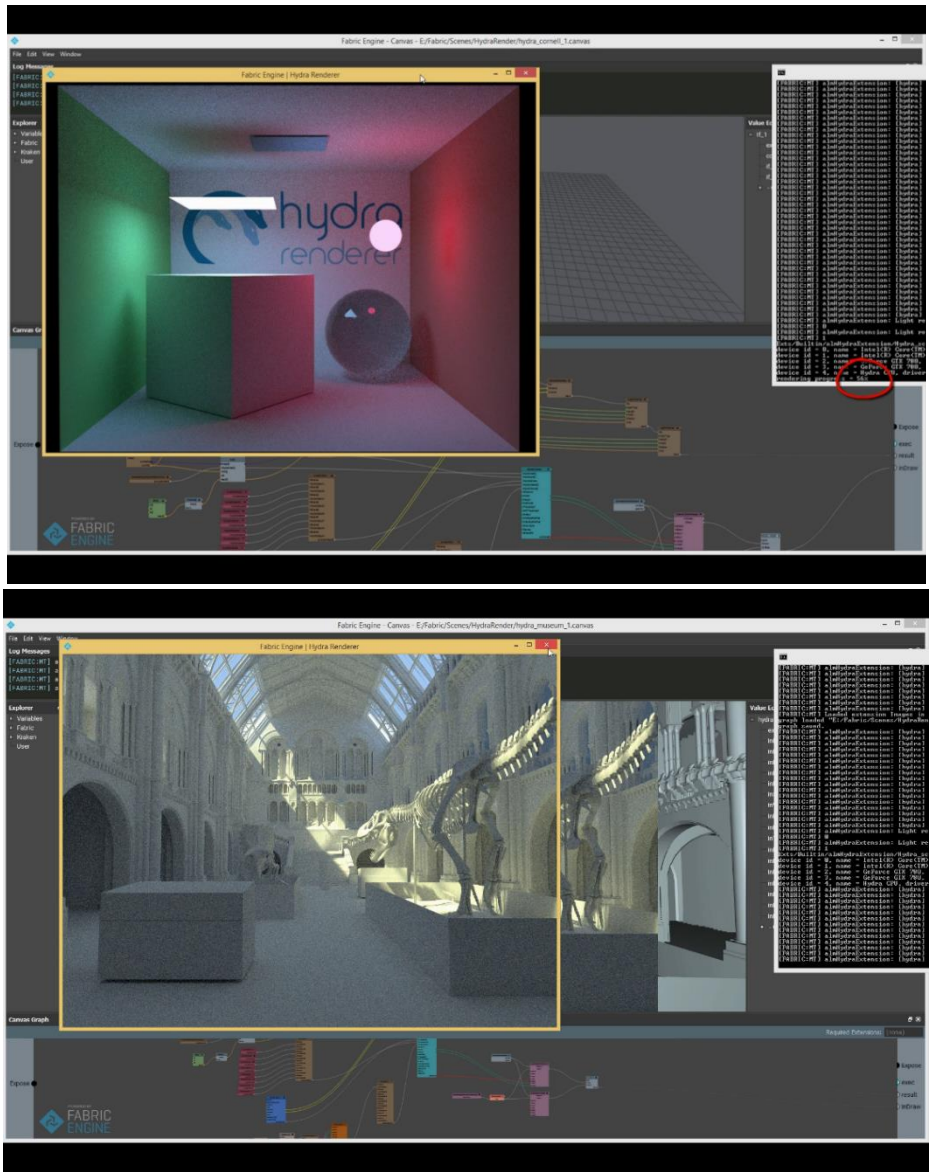


Рис. 32 Снимки экрана использования Hydra Renderer в Fabric Engine через разработанный интеграционный слой.

5.2 Программный комплекс синтеза наборов изображений для задач машинного обучения

Предложенная архитектура позволила построить на базе системы Hydra Render программный комплекс для синтеза обучающих данных для ряда задач компьютерного зрения (результаты опубликованы в [5, 6, 8]).

Для обучения современных алгоритмов компьютерного зрения требуются наборы данных изображений значительного объема - десятки и сотни тысяч изображений для обучения на статических изображениях и на порядок больше для анимации [164, 165]. Сбор подобных объемов данных требует определенных

технических возможностей и сопряжен со значительными временными и финансовыми затратами. Проблема количества данных также подразумевает необходимость достаточного представления разных объектов (классов), которые должна распознавать целевая модель, т.е. набор данных должен быть сбалансирован относительно представления в нем разных классов. Это может быть трудно достижимо, потому что определенные классы могут очень редко встречаться в наборах данных реального мира [5].

Помимо «просто» количества данных, важным является и наличие метаинформации о них. Например, в задачах распознавания в компьютерном зрении нужна разметка изображения по классам, на основе которой и будет обучена целевая модель. Разметка большого набора изображений вручную или полуавтоматически может быть затратной, кроме того, такая разметка часто не имеет необходимой точности, что связано с человеческим фактором и несовершенством средств разметки.

Во время проведения исследований специалисты-аналитики должны изменять входные наборы данных для проверки определенных гипотез. А из-за невозможности быстро получить новый набор данных (в связи с упомянутыми выше проблемами) аналитики могут рассматривать только подмножества существующего набора данных, что значительно ограничивает потенциал исследований.

Одно из решений заключается в использовании синтетических наборов данных, то есть полученных искусственным путем. В случае изображений – синтезированных с помощью алгоритмов рендеринга.

Проблема количества данных может быть решена с помощью алгоритмов настройки и выбора оптических свойств материалов и поверхностей, что позволяет быстро генерировать практически неограниченное количество обучающих примеров с любым распределением классов объектов. Кроме того, можно создавать обучающие примеры, которых очень мало или которые полностью отсутствуют наборах данных реального мира. Например, аварийные ситуации на дороге или на производстве, боевые действия, объекты, существующие только в виде дизайн-

проектов или прототипов. Проблема метаинформации при этом решается автоматически – она полностью доступна на этапе генерации данных. Рендер-система создает точные попиксельные маски для разметки изображений по отдельным объектам, а также позволяет получить и другую информацию, такую как карты глубины, карты нормалей, разметку по материалам поверхности и пр.

Для решения описанной проблемы была создан программный комплекс генерации наборов изображений, имеющий следующую архитектуру (рис. 33)

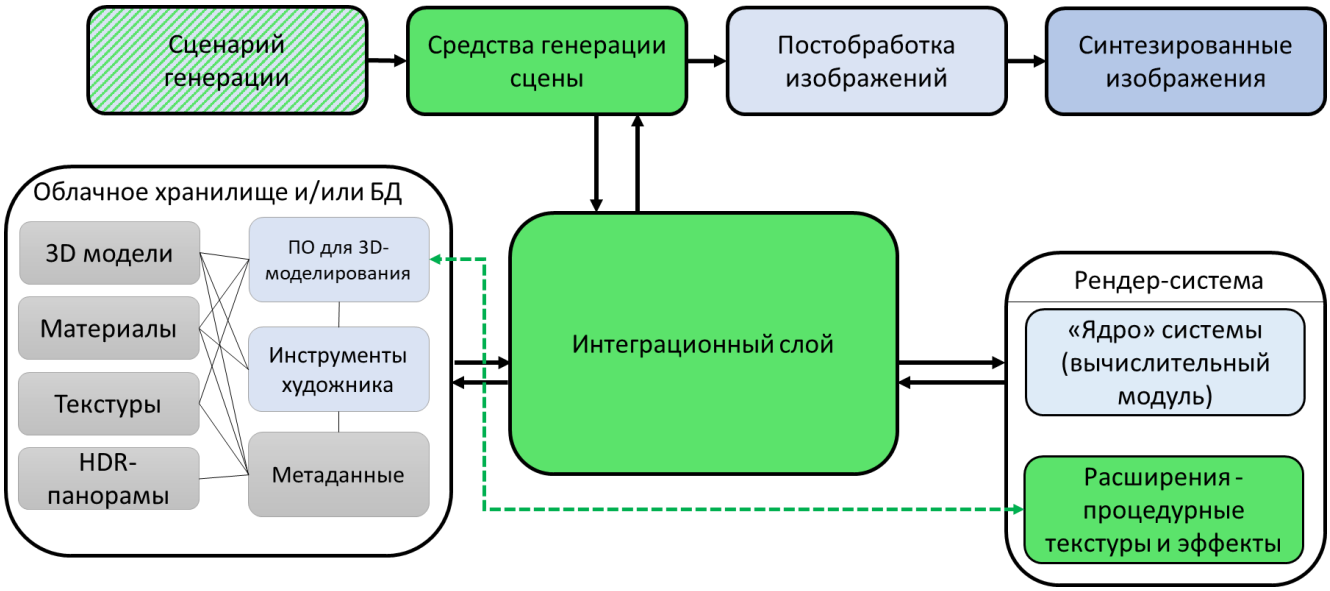


Рис. 33 Архитектура программного комплекса генерации синтетических данных.

Сценарий генерации определяет настройки для всей процедуры генерации, т.е. для всех остальных компонентов – классы 3D-моделей, типы освещения, настройки случайной генерации, выходные данные системы рендеринга, постобработка изображения, которая должна выполняться после рендеринга.

Облачное хранилище или база данных содержит базовые ресурсы для генерации сцен.

Инструменты художника – специально разработанные инструменты в виде расширения к ПО для 3D-моделирования, позволяющие настраивать ограничения для процесса рандомизации на этапе подготовке 3D-моделей и материалов.

Средства генерации сцены формируют запрошенное количество описаний сцен в соответствии с входным сценарием. Сгенерированное описание сцены предназначено для непосредственного использования системой рендеринга.

Система фотореалистичного рендеринга – рендер-система Hydra Renderer [41], включая расширения для процедурных текстур.

Инструменты постобработки изображений корректируют выходные изображения для получения дополнительных данных или для объединения их с фотографиями в случае использования подхода к генерации на основе дополненной реальности.

Компоненты комплекса взаимодействуют через предложенный в данной работе *интеграционный программный слой*:

- Программное обеспечение для 3D моделирования использует интеграционной слой для экспорта готовых компонентов сцен – 3D моделей, настроенных моделей материалов с текстурами.
- Средства генерации с помощью программного интерфейса интеграционного слоя формируют 3D сцену из компонентов, находящихся в базе данных и/или облачном хранилище, которая далее передается в рендер-систему.
- Рендер-система интегрирована с промежуточным слоем через программный интерфейс рендер-драйвера и осуществляет синтез изображения для сформированной средствами генерации сцены.
- Расширения рендер-системы для процедурных текстур добавляются в сцену при помощи специальных функций программного интерфейса промежуточного слоя.

Имеющийся у интеграционного слоя программный интерфейс для языка Python позволил реализовать средства создания сценариев генерации на этом языке, являющимся привычным и широко распространенным среди конечных пользователей – специалистов в области машинного обучения и компьютерного зрения.

Рис. 34 Примеры интерьерных сцен, сгенерированных с помощью программного комплекса.

На рис. 35 представлены примеры синтезированных изображений сцен, созданных с помощью программного комплекса.

Предложенный механизм определения разрешения для процедурных текстур позволил использовать готовые процедурные инструменты, встроенные в приложение для создания 3D-сцен.

Предложенный в 4-ой главе подход позволил реализовать ряд процедурных текстур в виде расширений для рендер-системы, реализующих математические модели таких явлений, как грязь, ржавчина, царапины, обледенение (рис. 35).

Рис. 35 Примеры процедурных текстур, реализованных с помощью подхода, предложенного в главе 4.

Также на основе механизма процедурных текстур было создано решение для случайной генерации автомобильных номеров по стандарту разных стран (рис. 36).

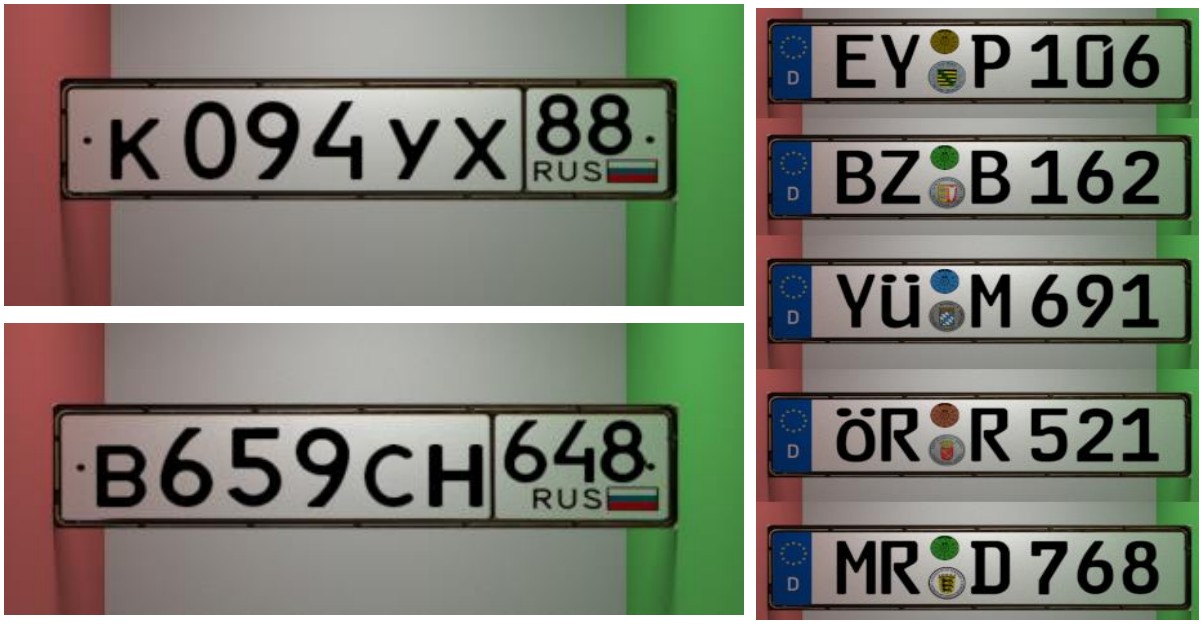


Рис. 36 Текстуры автомобильных номеров российского (слева) и немецкого (справа) образца, генерируемые с помощью механизма процедурных текстур.

Все разработанные процедурные текстуры параметризованы, что позволяет легко изменять внешний вид текстуры на этапе генерации сцены. Большинство процедурных эффектов основано на функциях шума, следовательно, параметры функции шума, такие как амплитуда, частота и постоянство (или коэффициенты, примененные к этим параметрам каким-либо образом), могут быть использованы как параметры сцены и задаваться случайными величинами, чтобы обеспечить визуальное разнообразие синтезируемых изображений (что является важным для задач компьютерного зрения). Одним из использовавшихся параметров процедурных текстур являются относительные размеры объектов. Этот параметр позволяет управлять распространением процедурного эффекта по поверхности объекта - например, процедурная текстура грязи может быть настроена так, чтобы она накладывалась только на нижние части модели. Подобная гибкость невозможна с обычными текстурами-изображениями.

Важно отметить, что использование процедурного подхода к текстурированию в некоторых случаях является критичным, потому что 3D-модели могут просто не иметь текстурных координат. И это достаточно частый случай даже для высококачественных 3D-моделей из открытых источников. Процедурные текстуры могут принимать мировые (или локальные) координаты точки поверхности в качестве входных данных, что может использоваться для отображения текстур. Кроме того, даже при наличии текстурных координат на модели, нет никакой гарантии, что текстуры, например, грязи или ржавчины будут выглядеть правильно на разных 3D-моделях из-за коэффициента масштабирования этих моделей при преобразовании из локального пространства в мировое. В таких случаях использование трехмерных процедурных текстур, зависящих от мировой позиции, или с помощью относительных локальных координат (полученных путем деления на размеры объекты по соответствующей оси) является единственным вариантом. Один из примеров использования такого подхода – так называемое проективное текстурирование (рис. 37).

Рис. 37 Пример текстурирования по относительным локальным координатам с помощью разработанного подхода создания процедурных текстур.

Апробация программного комплекса синтеза наборов данных.

Разработанная система генерации синтетических данных была апробирована на задачах распознавания дорожных знаков и автомобилей.

Рис. 38 Разметка траекторий на исходных изображениях (слева) и встраивание 3D-моделей на случайные позиции на траекториях (справа).

Для автомобилей для экспериментальной оценки использовался набор данных KITTI [166]. Были составлены обучающая и проверочная выборки по 200 кадров. В качестве исходного данных для генерации использовались 50 различных 3D-моделей автомобилей, процедурные текстуры для моделирования грязи, ржавчины и проективного наложения изображений на 3D-модели, для освещения использовался набор HDR-панорам. Позиции автомобилей выбирались на траекториях (рис. 38), построенных как кусочно-

линейные функции для всех 200 изображений (плоскость дороги была реконструирована, исходя из известных параметров камеры).

Для каждого изображения из обучающей выборки было проведено встраивание 2-5 случайных моделей автомобилей (рис. 38) объектов 20 раз с использованием разработанной системы генерации синтетических данных. В результате был получен расширенный обучающий набор из 4000 изображений. Также был сгенерирован набор данных со случайными горизонтальными отражениями (вероятность отражения 0.5) исходных изображений, чтобы понять преимущества нашего подхода по сравнению с простыми методами увеличения размера обучающих данных. Была обучены три нейросетевые модели (на основе Faster-RCNN и ResNet-50 FPN) на разных наборах данных – исходные изображения, исходные изображения + отраженные изображения, сгенерированные синтетические данные.

Для оценки качества детектора использовалась метрика mAP (mean average precision). Результирующие значения метрики mAP, демонстрирующие увеличение качества для синтетических данных, полученных с помощью разработанного подхода, показаны в таблице 4.

Набор данных	Значение метрики mAP
Исходный набор данных	38.67
Набор данных со случайными горизонтальными отражениями	40.30
Набор с синтетическими данными, сгенерированными разработанным комплексом	43.26

Таблица 4 Результаты использования синтетических данных, сгенерированных с помощью разработанного комплекса, на задаче распознавания автомобилей.

В задаче распознавания дорожных знаков [5] использование генерации синтетических данных с помощью предлагаемой системы позволило поднять точность распознавания редких дорожных знаков от 0% в базовом наборе

данных RTSD (Russian traffic sign dataset)[167] до 69.7% без использования нейросетевой пост-обработки методом CycleGAN и до 71% с её использованием (табл. 5). На рис. 39 показаны примеры сгенерированных изображений.

Рис. 39 Слева – разные категории изображений дорожных знаков: icon – текстура-символ знака, real – фотография, synt – наложение пиктограммы дорожного знака на фотографию, cgi – фотореалистичный рендеринг с помощью разработанного комплекса, cgi-gan – фотореалистичный рендеринг с помощью разработанного комплекса с последующей нейросетевой постобработкой. Справа – отдельные синтезированные изображения дорожных знаков с процедурными текстурами.

Набор данных	Точность на редких дорожных знаках	Точность на частых дорожных знаках
rtsd	0.0	94.9
rtsd + cgi	69.7	91.8
rtsd + cgi-gan	71.0	93.2

Таблица 5 Результаты использования синтетических данных, сгенерированных с помощью разработанного комплекса, на задаче распознавания дорожных знаков. rtsd – базовый набор данных [168], cgi – синтетические данные, полученные фотореалистичным рендерингом с помощью разработанного комплекса, cgi-gan – синтетические данные, полученные фотореалистичным рендерингом с помощью разработанного комплекса с нейросетевой постобработкой.

Заключение

Основные результаты данной работы:

1. Разработана новая программная архитектура расширяемой GPU рендер-системы, позволяющая существенно снизить трудоёмкость интеграции рендер-системы в клиентские приложения и отладки проблем, возникающих в процессе эксплуатации. Разработанная архитектура обеспечивает расширяемость рендер-системы, позволяя добавлять новые модели и параметры моделей, без внесения изменений в архитектуру.
2. Разработан подход создания расширений, позволяющий конечным пользователям реализовывать дополнительную функциональность для GPU рендер-систем с целью их адаптации к решению новых прикладных задач. Разработанный подход позволяет пользователям создавать новые геометрические примитивы, процедурные текстуры и проективное текстурирование, без необходимости внесения изменений непосредственно в рендер-систему и её перекомпиляции и с сохранением возможности отладки созданных расширений.
3. Разработан алгоритм оценки разрешения для процедурных текстур, также применимый и для текстур-изображений, позволяющий снизить потребление памяти в 1.6-4 раза без видимых потерь качества, а также обеспечить поддержку рендер-системой механизмов генерации процедурных текстур, имеющихся в клиентских приложениях.
4. Разработанные решения внедрены в систему фотореалистичного синтеза изображений на GPU и позволили осуществить её интеграцию и использование в широком круге задач - архитектурной визуализации, светодизайна, генерации синтетических данных для обучения алгоритмов искусственного интеллекта.

Литература

1. *Санжаров В.В., Фролов В.А.* Исследование масштабируемости распределённых рендер-систем на основе алгоритмов адаптивной трассировки путей и Metropolis Light Transport в гетерогенных сетях // Препринты Института прикладной математики им. МВ Келдыша РАН. – 2016. – №. 114. – С. 1-22. – DOI: 10.20948/prepr-2016-114
2. *Санжаров В.В., Фролов В.А.* Современные проблемы интеграции в приложениях компьютерной графики и пути их решения // Программирование. – 2018. – №. 4. – С. 36-45. – DOI: 10.31857/S013234740000699-3

   English translation: Frolov V.A., Sanzharov V.V. Modern problems of software integration in computer graphics applications and ways to solve them // Programming and Computer Software. — 2018. — Vol. 44, no. 4. — P. 233–239. — DOI: 10.1134/S0361768818040060
3. *Санжаров В.В., Фролов В.А.* Уровень детализации для предрассчитанных процедурных текстур // Программирование. – 2019. – №. 4. – С. 54-63. – DOI: 10.1134/S0132347419040071

   English translation: Sanzharov V.V., Frolov V.A. Level of detail for precomputed procedural textures // Programming and Computer Software. — 2019. — Vol. 45, no. 4. — P. 187–195. — DOI: 10.1134/s0361768819040078
4. *Sanzharov V., Frolov V., Pavlov I.* Restricted Extensions for GPU Photo-realistic Renderer // CEUR Workshop Proceedings. – 2019. Т 2485. – p. 37-42. – DOI: 10.30987/graphicon-2019-2-37-42
5. *Фаизов Б.В., Шахуро В.И., Санжаров В.В., Конушин А.С.* Классификация редких дорожных знаков // Компьютерная оптика. – 2020. – Т. 44. – №. 2. – С. 236-243. – DOI: 10.18287/2412-6179-CO-601
6. *Санжаров В.В., Фролов В.А., Волобой А.Г., Галактионов В.А., Павлов Д.С.* Система генерации наборов изображений для задач компьютерного зрения на основе фотореалистичного рендеринга // Препринты Института

прикладной математики им. МВ Келдыша РАН. – 2020. – №. 80. – С. 1-29. – DOI: 10.20948/prepr-2020-80

7. *Санжаров В.В., Фролов В.А., Галактионов В.А.* Исследование технологии RTX // Программирование. – 2020. – №. 4. – С. 65-72. – DOI: 10.31857/s0132347420030061

   English translation: Sanzharov V. V., Frolov V. A., Galaktionov V. A. Survey of Nvidia RTX technology// Programming and Computer Software. — 2020. — Vol. 46, no. 4. — P. 297–304. — DOI: 10.1134/s0361768820030068

8. *Frolov V., Faizov B., Shakhuro V., Sanzharov V., Konushin A., Galaktionov V., Voloboy A.* Image Synthesis Pipeline for CNN-Based Sensing Systems // Sensors. 2022. – Т. 22. – №. 6. – p. 2080. – DOI: 10.3390/s22062080

9. *Alhaija H. A. et al.* Augmented reality meets computer vision: Efficient data generation for urban driving scenes // International Journal of Computer Vision. – 2018. – Т. 126. – №. 9. – С. 961-972. – DOI: 10.1007/s11263-018-1070-x

10. *Hodaň T. et al.* Photorealistic image synthesis for object instance detection // IEEE International Conference on Image Processing (ICIP). – IEEE, 2019. – p. 66-70. – DOI: 10.1109/ICIP.2019.8803821

11. *Movshovitz-Attias Y., Kanade T., Sheikh Y.* How useful is photo-realistic rendering for visual learning? // European Conference on Computer Vision. – Springer, Cham, 2016. – p. 202-217. – DOI: 10.1007/978-3-319-49409-8_18

12. *Tsirikoglou A. et al.* Procedural modeling and physically based rendering for synthetic data generation in automotive applications // arXiv preprint arXiv:1710.06270. – 2017. – DOI: 10.48550/arXiv.1710.06270

13. *Zhang Y. et al.* Physically-based rendering for indoor scene understanding using convolutional neural networks // Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. – 2017. – p. 5287-5295. – DOI: 10.1109/CVPR.2017.537

14. Whitehurst A. The Visual Effects Pipeline [Электронный ресурс]. – URL: http://www.andrew-whitehurst.net/pipeline.html (дата обращения 10.10.2022)

15. Production Pipeline Fundamentals for Film and Games / Dunlop R. // CRC Press. – 2014. – 351 p. – ISBN: 9781317936237, 131793623X
16. *Wald I., Dietrich A., Slusallek P.* An interactive out-of-core rendering framework for visualizing massively complex models // ACM SIGGRAPH 2005 Courses. – 2005. – p. 17-es. – DOI: 10.1145/1198555.1198756
17. Ephere Inc., Ornatrix Hair, fur, feathers plugin for 3ds max Documentation [Электронный ресурс]. – URL: https://ephere.com/plugins/autodesk/max/ornatrix/docs/index.html (дата обращения 14.10.2022)
18. V-Ray for Unreal [Электронный ресурс]. – URL: https://www.chaosgroup.com/vray/unreal (дата обращения 14.12.2022)
19. *Дерябин Н.Б., Денисов Е.Ю.* Объектно-ориентированная инфраструктура систем компьютерной графики // Труды 17-ой международной конференции по компьютерной графике и зрению Graphi'Con, Россия, Московский Государственный Университет. – июнь 23-27, 2007. – с. 289-292.
20. *Kajiya J.T.* The rendering equation // Proceedings of the 13th annual conference on Computer graphics and interactive techniques. – 1986. – p. 143-150. – DOI: 10.1145/15922.15902
21. *Georgiev I. et al.* Light transport simulation with vertex connection and merging //ACM Trans. Graph. – 2012. – Т. 31. – №. 6. – p. 192:1-192:10. – DOI: 10.1145/2366145.2366211
22. *Veach E.* Robust Monte Carlo methods for light transport simulation.: дис. – Stanford University, 1998.
23. *Veach E., Guibas L.J.* Optimally combining sampling techniques for Monte Carlo rendering // Proceedings of the 22nd annual conference on Computer graphics and interactive techniques. – 1995. – p. 419-428. – DOI: 10.1145/218380.218498
24. *Волобой А.Г., Галактионов В.А., Дмитриев К.А., Копылов Э.А.* Двунаправленная трассировка лучей для интегрирования освещенности методом квази-Монте-Карло // Программирование. – 2004. – №. 5. – С. 25-34.

25. *Phong B.T.* Illumination for computer generated pictures // Communications of the ACM. – 1975. – Т. 18. – №. 6. – p. 311-317. – DOI: 10.1145/360825.360839
26. *Blinn J.F.* Models of light reflection for computer synthesized pictures // Proceedings of the 4th annual conference on Computer graphics and interactive techniques. – 1977. – p. 192-198. – DOI: 10.1145/563858.563893
27. *Cook R.L., Torrance K.E.* A reflectance model for computer graphics // ACM Transactions on Graphics (ToG). – 1982. – Т. 1. – №. 1. – p. 7-24. – DOI: 10.1145/357290.357293
28. *Ward G.J.* Measuring and modeling anisotropic reflection // Proceedings of the 19th annual conference on Computer graphics and interactive techniques. – 1992. – p. 265-272. – DOI: 10.1145/133994.134078
29. *Oren M., Nayar S.K.* Generalization of Lambert's reflectance model // Proceedings of the 21st annual conference on Computer graphics and interactive techniques. – 1994. – p. 239-246. – DOI: 10.1145/192161.192213
30. *Ashikhmin M., Shirley P.* An anisotropic Phong BRDF model // Journal of graphics tools. – 2000. – Т. 5. – №. 2. – p. 25-32. – DOI: 10.1080/10867651.2000.10487522
31. *Matusik W.* A data-driven reflectance model: дис. – Massachusetts Institute of Technology, 2003.
32. *Dupuy J., Jakob W.* An adaptive parameterization for efficient material acquisition and rendering // ACM Transactions on graphics (TOG). – 2018. – Т. 37. – №. 6. – p. 1-14. – DOI: 10.1145/3272127.3275059
33. Physically based rendering: From theory to implementation / Pharr M., Jakob W., Humphreys G. // 3rd edition, Morgan Kaufmann. – 2016. – 1266 p. – ISBN:9780128006450, 0128006455
34. RenderDoc. Open-Source graphics debugger [Электронный ресурс]. – URL: https://renderdoc.org/ (дата обращения 10.12.2022)
35. Nvidia Nsight Graphics software [Электронный ресурс]. – URL: https://developer.nvidia.com/nsight-graphics (дата обращения 10.12.2022)

36. *Parker S. G. et al.* Optix: a general purpose ray tracing engine // ACM transactions on graphics (ToG). – 2010. – Т. 29. – №. 4. – p. 1-13. – DOI: 10.1145/1778765.1778803
37. V-Ray GPU 3D GPU rendering software [Электронный ресурс]. – URL: https://www.chaosgroup.com/vray-gpu (дата обращения 10.12.2022)
38. *Frolov V.A., Kharlamov A.A., Ignatenko A.V.* Biased solution of integral illumination equation via irradiance caching and path tracing on GPUs // Programming and Computer Software. – 2011. – Т. 37. – №. 5. – p. 252-259. – DOI: 10.1134/S0361768811050021
39. *Laine S., Karras T., Aila T.* Megakernels considered harmful: Wavefront path tracing on GPUs // Proceedings of the 5th High-Performance Graphics Conference. – 2013. – p. 137-143. – DOI: 10.1145/2492045.2492060
40. *Frolov V.A., Galaktionov V.A.* Low overhead path regeneration // Programming and Computer Software. – 2016. – Т. 42. – №. 6. – p. 382-387. – DOI: 10.1134/S0361768816060025
41. Hydra Renderer. Open-source rendering system [Электронный ресурс]. – URL: https://github.com/Ray-Tracing-Systems/HydraCore (дата обращения 05.05.2023)
42. *Meißner M. et al.* VIZARD II: A reconfigurable interactive volume rendering system // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. – 2002. – p. 137-146. –DOI: 10.5555/569046.569065
43. *Pfister H. et al.* The volumepro real-time ray-casting system // Proceedings of the 26th annual conference on Computer graphics and interactive techniques. – 1999. – p. 251-260. – DOI: 10.1145/311535.311563
44. *Schmittler J., Wald I., Slusallek P.* Saarcor: a hardware architecture for ray tracing // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. – 2002. – p. 27-36. – DOI: 10.2312/EGGH/EGGH02/027-036
45. *Schmittler J. et al.* Realtime ray tracing of dynamic scenes on an FPGA chip // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. – 2004. – p. 95-106. – DOI: 10.1145/1058129.1058143

46. *Hall D.* The AR350: Today's ray trace rendering processor // Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware, Hot 3D Session. – 2001. – T. 1. – p. 2.

47. *Seiler L. et al.* Larrabee: a many-core x86 architecture for visual computing // ACM Transactions on Graphics (TOG). – 2008. – T. 27. – №. 3. – p. 1-15. – DOI: 10.1145/1360612.1360617

48. *Spjut J. et al.* TRaX: A multi-threaded architecture for real-time ray tracing // Symposium on Application Specific Processors. – IEEE, 2008. – p. 108-114. – DOI: 10.1109/SASP.2008.4570794

49. *Kopta D. et al.* An energy and bandwidth efficient ray tracing architecture // Proceedings of the 5th High-Performance Graphics Conference. – 2013. – p. 121-128. – DOI: 10.1145/2492045.2492058

50. *Woop S., Schmittler J., Slusallek P.* RPU: a programmable ray processing unit for realtime ray tracing // ACM Transactions on Graphics (TOG). – 2005. – T. 24. – №. 3. – p. 434-444. – DOI: 10.1145/1073204.1073211

51. *Aila T., Karras T.* Architecture considerations for tracing incoherent rays // Proceedings of the Conference on High Performance Graphics. – 2010. – p. 113-122. – DOI: 10.5555/1921479.1921497

52. *Novák J., Havran V., Dachsbacher C.* Path regeneration for interactive path tracing // Eurographics (Short Papers). – 2010. – p. 61-64. – DOI: 10.2312/egsh.20101048

53. *Shkurko K. et al.* Dual streaming for hardware-accelerated ray tracing // Proceedings of High-Performance Graphics. – 2017. – p. 1-11. – DOI: 10.1145/3105762.3105771

54. *Keely S.* Reduced precision hardware for ray tracing // Proceedings of High-Performance Graphics. – 2014. – p. 29-40. – DOI: 10.5555/2980009.2980013

55. *Nah J. H. et al.* RayCore: A ray-tracing hardware architecture for mobile devices // ACM Transactions on Graphics (TOG). – 2014. – T. 33. – №. 5. – p. 1-15. – DOI: 10.1145/2629634

56. *Lee W. J. et al.* SGRT: A mobile GPU architecture for real-time ray tracing // Proceedings of the 5th high-performance graphics conference. – 2013. – p. 109-119. – DOI: 10.1145/2492045.2492057
57. *Whitted T.* An improved illumination model for shaded display //ACM Siggraph 2005 Courses. – 2005. – p. 4-es. – DOI: 10.1145/1198555.1198743
58. *Deng Y. et al.* Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques // ACM Computing Surveys (CSUR). – 2017. – Т. 50. – №. 4. – p. 1-41. – DOI: 10.1145/3104067
59. Vulkan 1.2.x A Specification (with all registered Vulkan extensions). VK_KHR_ray_query extension [Электронный ресурс]. – URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#VK_KHR_ray_query (дата обращения 05.04.2023)
60. Vulkan 1.2.x A Specification (with all registered Vulkan extensions). VK_KHR_ray_tracing_pipeline extension [Электронный ресурс]. – URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#VK_KHR_ray_tracing_pipeline (дата обращения 05.04.2023)
61. Non-official Vulkan hardware database. Known driver version support for VK_KHR_ray_query [Электронный ресурс]. – URL: https://vulkan.gpuinfo.org/listdevicescoverage.php?extension=VK_KHR_ray_query (дата обращения 05.04.2023)
62. Nvidia Ampere GA102 GPU architecture whitepaper [Электронный ресурс]. – URL: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf (дата обращения 05.04.2023)
63. *Mammeri N., Juurlink B.* Vcomputebench: A vulkan benchmark suite for gpgpu on mobile and embedded gpus // IEEE International Symposium on Workload Characterization (IISWC). – IEEE, 2018. – p. 25-35. – DOI: 10.1109/IISWC.2018.8573477

64. clspv. A prototype compiler for a subset of OpenCL C to Vulkan compute shaders [Электронный ресурс]. – URL: https://github.com/google/clspv (дата обращения 08.08.2022)
65. Advanced Visualizer Manual. Appendix B1. Object Files (.obj) [Электронный ресурс]. – URL: https://fegemo.github.io/cefet-cg/attachments/obj-spec.pdf (дата обращения 08.08.2022)
66. FBX file format overview [Электронный ресурс]. – URL: https://www.autodesk.com/products/fbx/overview (дата обращения 08.08.2022)
67. Alembic. Open computer graphics interchange framework [Электронный ресурс]. – URL: http://www.alembic.io/ (дата обращения 08.08.2022)
68. OpenCOLLADA. A stream based reader and writer library for COLLADA[Электронный ресурс]. – URL: files http://www.opencollada.org/ (дата обращения 08.08.2022)
69. glTF (GL Transmission Format) [Электронный ресурс]. – URL: https://www.khronos.org/gltf/ (дата обращения 08.08.2022)
70. glTF 2.0 Specification, JSON schemas [Электронный ресурс]. – URL: https://github.com/KhronosGroup/glTF/tree/master/specification/2.0/schema (дата обращения 08.08.2022)
71. *Барладян Б.Х., Волобой А.Г., Шапиро Л.З.* Построение реалистичных изображений в системах автоматизированного проектирования // Труды 23-й Международной Конференции по Компьютерной Графике и Зрению Граф'Кон, Институт автоматики и процессов управления ДВО РАН. – 16-20 сентября 2013 года. – p.148-151
72. Universal Scene Description (USD) [Электронный ресурс]. – URL: https://graphics.pixar.com/usd/docs/index.html (дата обращения 11.12.2021)
73. Multiverse. Production-grade native USD solution [Электронный ресурс]. – URL: https://j-cube.jp/solutions/multiverse/ (дата обращения 11.12.2021)
74. Understanding the Linux Kernel: from I/O ports to process management / Bovet D. P., Cesati M. // O'Reilly Media, Inc. – 2005. – 1229 p. – ISBN: 0596554915, 9780596554910

75. pugixml. Light-weight C++ XML processing library [Электронный ресурс]. – URL: https://pugixml.org/ (дата обращения 04.03.2023)
76. apitrace. Set of graphics debugging tools [Электронный ресурс]. – URL: https://apitrace.github.io/ (дата обращения 29.04.2023)
77. Danylo Piliaiev. Testing Vulkan drivers with games that cannot run on the target device [Электронный ресурс]. – URL. - URL: https://blogs.igalia.com/dpiliaiev/gfxreconstruct-test-mobile-gpus/ (дата обращения 29.04.2023)
78. GFXReconstruct. Software for capture and replay of Vulkan API calls [Электронный ресурс]. – URL: https://github.com/LunarG/gfxreconstruct (дата обращения 29.04.2023)
79. *Волобой А.Г., Денисов Е.Ю., Барладян Б.Х.* Тестирование систем моделирования освещенности и синтеза реалистичных изображений // Программирование. – 2014. – Т. 40. – №. 4. – p. 13-22.
80. Arnold Renderer API Reference [Электронный ресурс]. – URL: https://help.autodesk.com/view/ARNOL/ENU/?guid=Arnold_API_REF_arnold_ref_index_html (дата обращения 30.04.2023)
81. V-Ray App SDK [Электронный ресурс]. – URL: https://www.chaos.com/vray/application-sdk (дата обращения 30.04.2023)
82. Jakob W., Speierer S., Roussel N., Nimier-David M., Vicini D., Zeltner T., Nicolet B., Crespo M., Leroy V., Zhang Z. Mitsuba 3 renderer, v. 3.1.1, 2022 [Электронный ресурс]. – URL: https://mitsuba-renderer.org/ (дата обращения 04.05.2023)
83. OSPRay - The Open, Scalable, and Portable Ray Tracing Engine [Электронный ресурс]. – URL: https://www.ospray.org/ (дата обращения 04.05.2023)
84. Texturing & modeling: a procedural approach / Ebert D. S. et al. // Morgan Kaufmann. – 2002. – 712 p. – ISBN: 1558608486, 9781558608481
85. *Perlin K.* An image synthesizer // ACM Siggraph Computer Graphics. – 1985. – Т. 19. – №. 3. – p. 287-296. – DOI: 10.1145/325165.325247

86. *Lefebvre L., Poulin P.* Analysis and synthesis of structural textures // Graphics Interface. – 2000. – T. 2000. – p. 77-86.

87. *Han C. et al.* Multiscale texture synthesis // ACM SIGGRAPH 2008 papers. – 2008. – p. 1-8. – DOI: 10.1145/1399504.1360650

88. *Gilet G. et al.* Local random-phase noise for procedural texturing // ACM Transactions on Graphics (TOG). – 2014. – T. 33. – №. 6. – p. 1-11. – DOI: 10.1145/2661229.2661249

89. *Gatys L., Ecker A. S., Bethge M.* Texture synthesis using convolutional neural networks // Advances in neural information processing systems. – 2015. – T. 28. – p. 262-270. – DOI: 10.5555/2969239.2969269

90. *Kolář M., Debattista K., Chalmers A.* A subjective evaluation of texture synthesis methods // Computer Graphics Forum. – 2017. – T. 36. – №. 2. – p. 189-198. – DOI: 10.1111/cgf.13118

91. *Yan L.Q. et al.* Rendering glints on high-resolution normal-mapped specular surfaces // ACM Transactions on Graphics (TOG). – 2014. – T. 33. – №. 4. – p. 1-9. – DOI: 10.1145/2601097.2601155

92. *Ernst M., Stamminger M., Greiner G.* Filter importance sampling // IEEE Symposium on Interactive Ray Tracing. – IEEE, 2006. – p. 125-132. – DOI: 10.1109/RT.2006.280223

93. *Lee M. et al.* Vectorized production path tracing // Proceedings of High Performance graphics. – 2017. – p. 1-11. – DOI: 10.1145/3105762.3105768

94. *Williams L.* Pyramidal parametrics // Proceedings of the 10th annual conference on Computer graphics and interactive techniques. – 1983. – p. 1-11. – DOI: 10.1145/800059.801126

95. *Wei L.Y., Levoy M.* Order-independent texture synthesis // arXiv preprint arXiv:1406.7338. – 2014. – DOI: 10.48550/arXiv.1406.7338

96. *Lefebvre S., Hoppe H.* Parallel controllable texture synthesis // ACM SIGGRAPH 2005 Papers. – 2005. – p. 777-786. – DOI: 10.1145/1186822.1073261

97. *Dong Y. et al.* Lazy solid texture synthesis // Computer Graphics Forum. – Oxford, UK : Blackwell Publishing Ltd, 2008. – Т. 27. – №. 4. – p. 1165-1174. – DOI: 10.1111/j.1467-8659.2008.01254.x
98. *Tanner C.C., Migdal C.J., Jones M.T.* The clipmap: a virtual mipmap // Proceedings of the 25th annual conference on Computer graphics and interactive techniques. – 1998. – p. 151-158. – DOI: 10.1145/280814.280855
99. *Lefebvre S., Darbon J., Neyret F.* Unified texture management for arbitrary meshes. Research Report RR-5210. – INRIA, 2004.
100. *Mittring M.* Advanced virtual texture topics // ACM SIGGRAPH 2008 Games. – 2008. – p. 23-51. – DOI: 10.1145/1404435.1404438
101. Sean Barett. Sparse Virtual Texturing [Электронный ресурс]. – URL: https://silverspaceship.com/src/svt/ (дата обращения 04.05.2023)
102. *Bilodeau B., Sellers G., Illesland K.* Partially Resident Textures on Next-Generation GPUs // Game Developers Conference. – 2012.
103. *Mayer A.J.* Virtual texturing: дис. – TU Wien, 2010.
104. OpenGL 4.6 API Specification [Электронный ресурс]. – URL: https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf (дата обращения 04.05.2023)
105. *Igehy H.* Tracing ray differentials // Proceedings of the 26th annual conference on Computer graphics and interactive techniques. – 1999. – p. 179-186. – DOI: 10.1145/311535.311555
106. Hydra Renderer. Hydra API Open-source rendering system [Электронный ресурс]. – URL: https://github.com/Ray-Tracing-Systems/HydraAPI (дата обращения 04.05.2023)
107. *Mantiuk R. K.* Practicalities of predicting quality of high dynamic range images and video // IEEE International Conference on Image Processing (ICIP). – IEEE, 2016. – p. 904-908. – DOI: 10.1109/ICIP.2016.7532488
108. JPEG2000: Image Compression Fundamentals, Standards and Practice / Taubman D.S., Marcellin M.W. // Springer Science & Business Media. – 2001. – 798 p. – ISBN: 079237519X, 9780792375197

109. *Горбунов-Посадов М.М.* Как растет программа // Препринт Института прикладной математики им. М.В. Келдыша РАН. – М. – 2000. – 16 с.

110. Elements of reusable object-oriented software / Gamma E. et al. // Addison-Wesley. – Reading, Massachusetts. – 1995. – 395 p. – ISBN: 0201633612, 978-0201633610

111. *Smaragdakis Y., Batory D.* Implementing reusable object-oriented components // Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203). – IEEE, 1998. – p. 36-45. – DOI: 10.1109/ICSR.1998.685728

112. *Apel S., Kastner C., Lengauer C.* Featurehouse: Language-independent, automated software composition // IEEE 31st International Conference on Software Engineering. – IEEE, 2009. – p. 221-231. – DOI: 10.1109/ICSE.2009.5070523

113. *Batory D., Sarvela J. N., Rauschmayer A.* Scaling step-wise refinement // IEEE Transactions on Software Engineering. – 2004. – Т. 30. – №. 6. – p. 355-371. – DOI: 10.1109/TSE.2004.23

114. *Rosenmüller M. et al.* Flexible feature binding in software product lines // Automated Software Engineering. – 2011. – Т. 18. – №. 2. – p. 163-197. – DOI: 10.1007/s10515-011-0080-5

115. *Kiczales G. et al.* An overview of AspectJ // European Conference on Object-Oriented Programming. – Springer, Berlin, Heidelberg, 2001. – p. 327-354. – DOI: 10.1007/3-540-45337-7_18

116. *Spinczyk O., Gal A., Schröder-Preikschat W.* AspectC++ an aspect-oriented extension to the C++ programming language // Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications. – 2002. – p. 53-60. – DOI: 10.5555/564092.564100

117. *Schaefer I. et al.* Delta-oriented programming of software product lines // International Conference on Software Product Lines. – Springer, Berlin, Heidelberg, 2010. – p. 77-91. – DOI: 10.1007/978-3-642-15579-6_6

118. *Smaragdakis Y., Batory D.* Implementing layered designs with mixin layers // European Conference on Object-Oriented Programming. – Springer, Berlin, Heidelberg, 1998. – p. 550-570. – DOI: 10.1007/BFb0054107

119. v8 JavaScript Engine [Электронный ресурс]. – URL: https://github.com/v8/v8 (дата обращения 01.12.2022)

120. *Жданов Д., Ершов С., Дерябин Н.* Объектно-ориентированная модель фотореалистичной визуализации сложных сцен // Научная визуализация. – 2013. – Т. 5. – №. 4. – p. 88-117.

121. *Albrecht T.* Pitfalls of object-oriented programming // Proceedings of Game Connect: Asia Pacific (GCAP). – 2009.

122. *Patel P.* Object Oriented Programming for Scientific Computing: дис. – Master's thesis, The University of Edinburgh, 2006.

123. *Springer M.* Memory-Efficient Object-Oriented Programming on GPUs // arXiv preprint arXiv:1908.05845. – 2019. – DOI: 10.48550/arXiv.1908.05845

124. *Siegel J., Ributzka J., Li X.* CUDA memory optimizations for large data-structures in the Gravit simulator // Journal of Algorithms & Computational Technology. – 2011. – Т. 5. – №. 2. – p. 341-362. – DOI: 10.1109/ICPPW.2009.78

125. *Adinetz A. V., Pleiter D.* Halloc: a high-throughput dynamic memory allocator for GPGPU architectures // GPU Technology Conference (GTC). – 2014. – Т. 152.

126. *Huang X. et al.* Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines //10th IEEE International Conference on Computer and Information Technology. – IEEE, 2010. – p. 1134-1139. – DOI: 10.1109/CIT.2010.206

127. *Spliet R. et al.* KMA: A dynamic memory manager for OpenCL // Proceedings of Workshop on General Purpose Processing Using GPUs. – 2014. – p. 9-18. – DOI: 10.1145/2588768.2576781

128. *Springer M., Masuhara H.* DynaSOAr: a parallel memory allocator for object-oriented programming on GPUs with efficient memory access // arXiv preprint arXiv:1810.11765. – 2018. – DOI: 10.48550/arXiv.1810.11765

129. *Gelado I., Garland M.* Throughput-oriented GPU memory allocation // Proceedings of the 24th symposium on principles and practice of parallel programming. – 2019. – p. 27-37. – DOI: 10.1145/3293883.3295727

130. *Winter M. et al.* Are dynamic memory managers on GPUs slow? A survey and benchmarks // Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. – 2021. – p. 219-233. – DOI: 10.1145/3437801.3441612

131. *Wu J. et al.* gpucc: an open-source GPGPU compiler // Proceedings of the 2016 International Symposium on Code Generation and Optimization. – 2016. – p. 105-116. – DOI: 10.1145/2854038.2854041

132. *Zhang M., Green R., Rogers T. G.* Characterizing the runtime effects of object-oriented workloads on GPUs // IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). – IEEE, 2018. – p. 109-110. – DOI: 10.1109/ISPASS.2018.00019

133. *Barik R. et al.* Efficient mapping of irregular C++ applications to integrated GPUs // Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. – 2014. – p. 33-43. – DOI: 10.1145/2581122.2544165

134. *Cook R. L.* Shade trees // Proceedings of the 11th annual conference on Computer graphics and interactive techniques. – 1984. – p. 223-231. – DOI: 10.1145/800031.808602

135. *Perlin K.* In the beginning: The pixel stream editor //Real-Time Shading SIGGRAPH Course Notes. – 2001.

136. *Hanrahan P., Lawson J.* A language for shading and lighting calculations // Proceedings of the 17th annual conference on Computer graphics and interactive techniques. – 1990. – p. 289-298. – DOI: 10.1145/97879.97911

137. *Gritz L. et al.* Open shading language // ACM SIGGRAPH 2010 Talks. – 2010. – p. 1. – DOI: 10.1145/1837026.1837070

138. RenderMan Advanced Technology from Pixar Animation Studios for Rendering VFX and Animation [Электронный ресурс]. – URL: https://renderman.pixar.com/ (дата обращения 08.12.2022)
139. Arnold GPU. Supported Features and Known Limitations [Электронный ресурс]. – URL: https://docs.arnoldrenderer.com/display/A5ARP/Supported+Features+and+Known+Limitations (дата обращения 08.08.2022)
140. VEX, high-performance expression language used in Houdini [Электронный ресурс]. – URL: https://www.sidefx.com/docs/houdini/vex/index.html (дата обращения 24.11.2021)
141. *Deryabin N. B., Zhdanov D. D., Sokolov V. G.* Embedding the script language into optical simulation software // Programming and Computer Software. – 2017. – Т. 43. – №. 1. – p. 13-23. – DOI: 10.1134/S0361768817010029
142. OctaneRender [Электронный ресурс]. – URL: https://home.otoy.com/render/octane-render/ (дата обращения 10.12.2021)
143. *Sons K. et al.* shade.js: Adaptive Material Descriptions // Computer Graphics Forum. – 2014. – Т. 33. – №. 7. – p. 51-60. – DOI: 10.1111/cgf.12473
144. *He Y. et al.* Shader components: modular and high-performance shader development // ACM Transactions on Graphics (TOG). – 2017. – Т. 36. – №. 4. – p. 1-11. – DOI: 10.1145/3072959.3073648
145. *Foley T., Hanrahan P.* Spark: modular, composable shaders for graphics hardware // ACM Transactions on Graphics (TOG). – 2011. – Т. 30. – №. 4. – p. 1-12. – DOI: 10.1145/2010324.1965002
146. *Patney A. et al.* Piko: a framework for authoring programmable graphics pipelines // ACM Transactions on Graphics (TOG). – 2015. – Т. 34. – №. 4. – p. 1-13. – DOI: 10.1145/2766973
147. *Parker S. G. et al.* RTSL: a ray tracing shading language // IEEE Symposium on Interactive Ray Tracing. – IEEE, 2007. – p. 149-160. – DOI: 10.1109/RT.2007.4342603

148. OpenRL SDK White Paper [Электронный ресурс]. – URL: http://imgtec.eetrend.com/sites/imgtec.eetrend.com/files/article/201402/1495-2162-1.pdf (дата обращения 08.12.2022)
149. Blender Shader Editor documentation [Электронный ресурс]. – URL: https://docs.blender.org/manual/en/latest/editors/shader_editor.html (дата обращения 04.05.2023)
150. Unity Shader graph [Электронный ресурс]. – URL: https://unity.com/ru/shader-graph (дата обращения 08.12.2022)
151. Cycles Open-Source Production Rendering [Электронный ресурс]. – URL: https://www.cycles-renderer.org/ (дата обращения 04.05.2023)
152. RenderMan documentation. OSL Patterns [Электронный ресурс]. – URL: https://rmanwiki.pixar.com/display/REN24/OSL+Patterns (дата обращения 11.08.2022)
153. Redshift render OSL shaders repository [Электронный ресурс]. – URL: https://github.com/redshift3d/RedshiftOSLShaders (дата обращения 11.08.2022)
154. Clang documentation. LibTooling [Электронный ресурс]. – URL: https://clang.llvm.org/docs/LibTooling.html (дата обращения 04.05.2023)
155. *Inigo Quilez* Voronoise - a combination of Voronoi, and Noise [Электронный ресурс]. – URL: https://iquilezles.org/articles/voronoise/ (дата обращения 04.05.2023)
156. FreeImage, an Open-Source library project [Электронный ресурс]. – URL: https://freeimage.sourceforge.io/ (дата обращения 04.05.2023)
157. GLFW, an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development [Электронный ресурс]. – URL: https://www.glfw.org/ (дата обращения 04.05.2023)
158. Corto, a library for compression and decompression meshes and point clouds [Электронный ресурс]. – URL https://github.com/cnr-isti-vclab/corto (дата обращения 04.05.2023)
159. Tangent Space Normal Maps generation [Электронный ресурс]. – URL: http://www.mikktspace.com/ (дата обращения 04.05.2023)

160. pybind11 — Seamless operability between C++11 and Python [Электронный ресурс]. – URL: https://github.com/pybind/pybind11 (дата обращения 04.05.2023)
161. LightCAD инструмент светового дизайнера для создания и управления системой динамического освещения на объекте [Электронный ресурс]. – URL: https://lightcad.com/ (дата обращения 04.08.2022)
162. Компания INTILED [Электронный ресурс]. – URL: https://intiled.ru/o-nas/about-intiled (дата обращения 04.08.2022)
163. Fabric Engine and KL LLVM for 3D Digital Content Creation [Электронный ресурс]. – URL: https://llvm.org/devmtg/2014-04/PDFs/Talks/FabricEngine-LLVM.pdf (дата обращения 04.12.2021)
164. *Karpathy A. et al.* Large-scale video classification with convolutional neural networks // Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. – 2014. – p. 1725-1732. – DOI: 10.1109/CVPR.2014.223
165. *Wu Z. et al.* Deep learning for video classification and captioning // Frontiers of multimedia research. – 2017. – p. 3-29. – DOI: 10.1145/3122865.3122867
166. *Geiger A. et al.* Vision meets robotics: The kitti dataset // The International Journal of Robotics Research. – 2013. – Т. 32. – №. 11. – p. 1231-1237. – DOI: 10.1177/0278364913491297
167. *Shakhuro V.I., Konouchine A.S.* Russian traffic sign images dataset // Computer optics. – 2016. – Т. 40. – №. 2. – p. 294-300. – DOI: 10.18287/2412-6179-2016-40-2-294-300