Федеральное государственное бюджетное учреждение науки

«Институт системного программирования им В.П. Иванникова Российской академии наук»

На правах рукописи

Гонахчян Вячеслав Игоревич

Адаптивная стратегия рендеринга динамических трехмерных сцен

Специальность 05.13.11 — математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация
на соискание ученой степени
кандидата технических наук

Научный руководитель:
д.ф.-м.н., профессор, Семенов Виталий Адольфович

Москва — 2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
ГЛАВА 1. Обзор методов и программных интерфейсов рендеринга 10
1.1 Современные программные интерфейсы рендеринга 10
1.1.1 Модель графического конвейера 11
1.1.2 Схемы рендеринга 14
1.1.3 Составление командного буфера 15
1.2 Методы удаления невидимых поверхностей 16
1.2.1 Методы, сохраняющие информацию о видимости во время предобработки . 17
1.2.2 Методы, вычисляющие видимость в плоскости изображения 19
1.2.3 Методы с отбором объектов-преград 20
1.2.4 Методы на основе аппаратных проверок видимости 22
1.3 Методы на основе оценки времени рендеринга 23
1.4 Методы на основе пространственного индексирования 26
1.4.1 Одномерные методы индексирования 28
1.4.2 Многомерные методы индексирования 29
1.4.3 Пространственные методы индексирования 32
1.5 Сравнение и основные выводы 35
ГЛАВА 2. Модель производительности графического конвейера 37
2.1 Исследуемая модель графического конвейера 37
2.2 Исследуемая схема рендеринга 38
2.3 Описание динамической сцены 39
2.4 Оценка потребляемой памяти 40
2.5 Оценка времени рендеринга 42
2.5.1 Время выполнения рендеринга на CPU и GPU процессорах 43
2.5.2 Время выполнения этапов графического конвейера 46
2.5.3 Вычисление покрываемого объектом количества фрагментов 48
2.5.4 Генерация тестовых наборов для вычисления параметров модели 50
ГЛАВА 3. Предложенные методы 56
3.1 Техника составления командных буферов 56
3.1.1 Цикл жизни командного буфера 56
3.1.2 Способы составления командных буферов 57
3.1.3 Фрагментация и кэширование командных буферов совместно с использованием окто-дерева 58
3.2 Метод оценки количества аппаратных проверок видимости для эффективного выполнения рендеринга 63
3.2.1 Отбор узлов окто-дерева 64

3.2.2 Вычисление количества проверок видимости ... 65
3.3 Адаптивная стратегия рендеринга ... 73
3.3.1 Рассматриваемые способы реализации процессов рендеринга ... 74
3.3.2 Критерии применения способов рендеринга ... 75
3.4 Метод генерации сцен ... 82
ГЛАВА 4. Программная реализация модели производительности и стратегии рендеринга 86
4.1 Принципы и организация библиотеки программ для рендеринга динамических сцен ... 86
4.2 Программные модули, реализующие модель производительности графического конвейера ... 88
4.3 Особенности программной реализации адаптивной стратегии рендеринга 99
ГЛАВА 5. Вычислительные эксперименты ... 109
5.1 Тестовые наборы сцен ... 109
5.2 Применение методов пространственного индексирования при удалении невидимых поверхностей ... 115
5.3 Анализ эффективности аппаратных проверок видимости ... 119
5.4 Анализ производительности составления и отправки командных буферов 120
5.5 Тестирование модели производительности графического конвейера ... 122
5.6 Анализ производительности адаптивной стратегии рендеринга ... 135
5.7 Сравнение производительности разработанной и текущей графических библиотек ... 141
ЗАКЛЮЧЕНИЕ ... 143
СПИСОК ЛИТЕРАТУРЫ ... 144

ВВЕДЕНИЕ

Актуальность исследования

Рендеринг трехмерных сцен является одной из ключевых задач компьютерной графики и широко применяется в таких предметных областях, как научная визуализация, автоматизация проектирования, инженерии и производства (CAD/CAM/CAE), компьютерные игры и анимация, виртуальная и дополненная реальность. В связи с перманентным ростом сложности сцен повышаются требования и к эффективности программных и аппаратных средств рендеринга. Несмотря на обширные исследования в этой области и большое количество опубликованных работ, задачам рендеринга больших динамических сцен уделяется относительно мало внимания. Вместе с тем, класс приложений, оперирующих с подобными сценами, чрезвычайно широк, что определяет актуальность темы исследований. Особую важность тема приобретает в связи со стремительным развитием технологий информационного моделирования зданий и сооружений (BIM), предусматривающих, в частности, интерактивные графические средства динамического моделирования процессов возведения сложных строительных объектов и реализации масштабных инфраструктурных программ.

В работе рассматриваются задачи отображения динамических трехмерных сцен с использованием современных видеокарт (GPU) и графических интерфейсов к ним (OpenGL, DirectX, Vulkan). Видеокарты обеспечивают высокоэффективную поточно-параллельную обработку графических элементов (треугольников, пикселей) для достижения требуемой частоты генерации изображений. Графические интерфейсы позволяют задать описание трехмерной сцены и передать его на видеокарту, установить программы для геометрических преобразований, проецирования треугольников на экранную плоскость, расчета функций освещения, а также сгенерировать итоговые изображения.

Однако при отображении сложных сцен, содержащих большое количество графических элементов и предполагающих использование сложных моделей материалов и освещения, видеокарты часто не справляются с большим объемом вычислений. Подобные ограничения делают невозможными разработку и применение интерактивных графических приложений, предполагающих генерацию и вывод изображений с частотой, соответствующей скорости отклика на пользовательские события.

Одним из традиционных подходов к повышению производительности рендеринга является уменьшение числа растеризуемых графических элементов за счет

предварительного анализа и удаления невидимых поверхностей. Распространенные методы, применяемые при отображении динамических сцен, описаны в работах авторов: Coorg, S., Teller, S., Hudson, T., Manocha, D., Bittner, J., Wimmer, M., Mattausch, O. Наиболее перспективными среди них являются методы на основе аппаратных проверок видимости (hardware occlusion queries), которые в ряде случаев позволяют повысить эффективность отображения сцен. Однако регулярное использование аппаратных проверок может приводить к деградации производительности, принимая во внимание дополнительные расходы на подготовку и осуществление самих проверок. Поэтому необходимыми становятся оценки эффективности применения аппаратных проверок с учетом вычислительных затрат и изменений в сцене.

Для ускорения методов удаления невидимых элементов используются структуры пространственной декомпозиции сцены, такие как окто-деревья и k-d-деревья. Иерархический обход деревьев с отбраковкой невидимых узлов позволяет существенно сократить количество отображаемых объектов. При этом следует учитывать, что затраты на обновление пространственных структур в процессе рендеринга больших динамических сцен могут оказаться неоправданно высокими.

Важными факторами, влияющими на эффективность процесса рендеринга динамических сцен, являются способы составления и отправки командных буферов. Как правило, описание объектов сцены формируется в виде буферов команд рендеринга, которые записываются в основную память один раз и отправляются на видеокарту по шине. Однако при отображении динамических сцен и использовании проверок видимости требуется повторная запись, которая может занимать продолжительное время. При незначительных локальных изменениях на каждом временном шаге моделирования динамической сцены можно сократить затраты на составление и отправку буферов за счет их фрагментации и кэширования в основной памяти. В этом случае требуется обновление только тех буферов, объекты которых подверглись изменениям на текущем временном шаге.

Таким образом, для эффективного рендеринга больших динамических сцен с использованием современных видеокарт перспективным представляется применение комбинированной стратегии, сочетающей в себе следующие методы и техники:

- удаление невидимых объектов с использованием методов пространственной декомпозиции и индексирования;
- аппаратные проверки видимости с отложенной загрузкой результатов;

– фрагментация и кэширование командных буферов с учетом локальных изменений в сцене.

Поскольку данные методы выполняются одновременно в вычислительной системе, важно сбалансированное использование ее ресурсов при конвейерной обработке и передаче графических данных. В частности, должна быть сбалансирована загрузка CPU и GPU процессоров. Вместе с тем, вариативность в количестве и сложности индивидуальных объектов сцены, характере и интенсивности динамики делает подобную балансировку крайне сложной или даже невозможной при фиксировании конкретных методов и техник. Обеспечить высокую производительность вычислительной системы на широком классе задач рендеринга больших динамических сцен представляется возможным в результате адаптивного управления графическим конвейером, предусматривающего выбор и настройку альтернативных базовых методов и техник с учетом доступных ресурсов системы и особенностей отображаемой сцены на конкретном интервале модельного времени в конкретной пространственной области.

Целью работы является исследование и разработка эффективной стратегии рендеринга динамических трехмерных сцен, сочетающей в себе методы удаления невидимых объектов, отложенные аппаратные проверки видимости, кэширование командных буферов, а также реализующей адаптивный выбор методов и техник для ускорения процессов рендеринга и обеспечения возможностей разработки перспективных интерактивных графических приложений. Стратегия должна быть применима к широкому классу сцен, объекты которых геометрически представлены треугольными сетками, а их положение и материалы задаются индивидуально в виде соответствующих кусочно-линейных функций, явно зависящих от временного параметра. Предполагается также, что в рассматриваемых, так называемых, псевдо-динамических сценах превалируют локальные изменения, затрагивающие на каждом временном шаге моделирования относительно небольшое число объектов сцены.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. На основе сравнительного анализа существующих методов удаления невидимых поверхностей, техник составления командных буферов и реализации аппаратных проверок видимости выделить наиболее перспективные, применимые к рассматриваемому классу псевдо-динамических трехмерных сцен.

2. Рассмотреть возможные способы ускорения базовых методов и техник на основе структур пространственной декомпозиции и индексирования сцен. Выделить пространственные структуры, допускающие быстрые инкрементальные обновления с учетом локального характера изменений в псевдо-динамических сценах.
3. Разработать и исследовать модель производительности графического конвейера применительно к задачам рендеринга динамических трехмерных сцен, позволяющую оценивать требуемые ресурсы в зависимости от применяемых базовых методов и характеристик отображаемой сцены. В частности, модель должна устанавливать целесообразность применения проверок видимости.
4. Разработать адаптивную стратегию рендеринга динамических трехмерных сцен, реализующую выбор и настройку базовых методов в процессе отображения сцены на основе предложенной модели производительности графического конвейера.
5. Провести серию вычислительных экспериментов с синтетическими и реальными тестовыми наборами сцен для подтверждения релевантности примененной модели производительности графического конвейера и эффективности предложенной адаптивной стратегии.
6. Апробировать разработанную стратегию в составе системы визуального моделирования и планирования индустриальных проектов.

Научная новизна состоит в получении следующих результатов:

1. Предложена модель производительности графического конвейера для однопроходной схемы рендеринга динамических трехмерных сцен. Модель позволяет оценивать требуемые ресурсы (время обработки и передачи графических данных, объем основной и графической памяти) в зависимости от применяемых базовых методов и характеристик отображаемой сцены. Для получения релевантных оценок на оборудовании пользователя параметры модели калибруются с использованием тестовых наборов.
2. Предложена адаптивная стратегия рендеринга динамических трехмерных сцен, реализующая выбор и настройку базовых методов удаления невидимых объектов, техник отложенных аппаратных проверок видимости и кэширования командных буферов на основе модели производительности графического конвейера непосредственно в процессе отображения сцены.

Для ускорения вычислений стратегия предусматривает использование регулярных окто-деревьев в качестве структуры пространственной декомпозиции и индексирования динамической сцены.

3. Предложен метод генерации динамических трехмерных сцен, который позволяет синтезировать семейства сцен с разными характеристиками, определяемыми количеством и сложностью индивидуальных объектов, пространственной разреженностью сцены, интенсивностью событий и их пространственно-временной когерентностью. Сгенерированные синтетические сцены могут использоваться для организации репрезентативных наборов тестов и тестирования методов и программных средств компьютерной графики.

Практическая ценность полученных результатов заключается в возможности применения предложенной стратегии и методов рендеринга для эффективного отображения широкого класса динамических трехмерных сцен, а также для разработки интерактивных графических приложений в таких областях как научная визуализация, автоматизация проектирования, инженерии и производства (CAD/CAM/CAE), компьютерные игры и анимация, виртуальная и дополненная реальность.

Разработанная стратегия и методы программно реализованы в составе системы визуального пространственно-временного (4D) моделирования и планирования индустриальных проектов, обеспечивая прирост производительности при отображении больших динамических сцен в 2–9 раз. В настоящее время данная система успешно применяется в более чем трех сотнях ведущих индустриальных компаний в тридцати шести странах мира, в том числе, и в Российской Федерации.

Методология и методы исследования

Результаты диссертации были получены с использованием подходов и методов вычислительной геометрии, компьютерной графики, теории нелинейной оптимизации.

Основные положения, выносимые на защиту:

1. Модель производительности графического конвейера для однопроходной схемы рендеринга динамических трехмерных сцен.
2. Адаптивная стратегия рендеринга динамических трехмерных сцен.
3. Метод генерации динамических трехмерных сцен.

Апробация работы

Основные результаты работы докладывались на следующих конференциях:

- 27-я Международная конференция по компьютерной графике, обработке изображений и машинному зрению, системам визуализации и виртуального окружения. (GraphiCon 2017) (24–28 сентября 2017 года, г. Пермь).
- 12-я Международная конференция по компьютерной графике, научной визуализации, машинному зрению и обработке изображений (CGVCVIP 2018) (18–20 июля 2018 года, г. Мадрид).
- 13-я Международная конференция по компьютерной графике, научной визуализации, машинному зрению и обработке изображений (CGVCVIP 2019) (16–18 июля 2019 года, г. Порту).
- 29-я Международная конференция по компьютерной графике и машинному зрению (GraphiCon 2019) (23–26 сентября 2019 года, г. Брянск).

Публикации

По теме диссертации опубликовано 7 работ [1–7], в том числе 2 статьи [1–2] в реферируемых научных журналах из списка изданий, рекомендованных ВАК РФ. Работа [3] входит в международные системы цитирования Web of Science и Scopus. Работы [4; 5] индексированы в Scopus. В работах [1; 2; 5–7] все научные результаты принадлежат автору. В работе [3] автор разработал и описал метод составления и использования командных буферов в сочетании со структурами пространственного индексирования сцены. В работе [4] автор разработал и описал метод генерации динамических сцен.

Личный вклад автора

Все представленные в работе результаты получены лично автором.

Объем и структура работы

Представленная работа состоит из введения, пяти глав основного содержания, заключения и списка используемых источников, включающего 94 работы. Общий объем работы составляет 150 страниц, включая 42 рисунка и 34 таблицы.

ГЛАВА 1. ОБЗОР МЕТОДОВ И ПРОГРАММНЫХ ИНТЕРФЕЙСОВ РЕНДЕРИНГА

В данной главе дается обзор основных методов, которые используются для реализации эффективного рендеринга. В разделе 1.1 рассматриваются возможности современных программных интерфейсов рендеринга. Наибольшее внимание уделено принципам работы современных графических конвейеров (graphics pipeline). Описываются основные схемы рендеринга с использованием GPU процессоров. В разделе 1.2 приводится обзор методов удаления невидимых поверхностей. В разделе 1.3 рассматриваются методы на основе оценки времени выполнения рендеринга. В разделе 1.4 рассматриваются методы на основе пространственного индексирования объектов сцены, которые имеют большое значение для производительности рендеринга динамических сцен. В разделе 1.5 приводятся выводы, которые позволяют определить дальнейшее направление исследований.

1.1 Современные программные интерфейсы рендеринга

В данной работе рассматривается выполнение рендеринга с использованием аппаратных возможностей современных видеокарт. Видеокарты состоят из потоковых процессоров, выполняющих поточно-параллельную обработку команд (SIMD), разделяемой памяти и общей памяти, блоков вывода для объединения результатов и записи в кадровый буфер. В результате работы графического конвейера по полигональному представлению трехмерной сцены создается двухмерное изображение.

Для взаимодействия с видеокартами используются программные интерфейсы рендеринга: OpenGL, DirectX, Metal, Vulkan [8; 9; 10]. Основные функции программных интерфейсов рендеринга:

- загрузка буферов с вершинами;
- загрузка буферов с нормалями;
- загрузка буферов с индексами графических элементов;
- загрузка текстур;
- компиляция и загрузка шейдерных программ;
- задание конфигураций графического конвейера;
- загрузка буферов с командами рендеринга для обработки на конвейере GPU;
- загрузка результатов выполняемых команд в основную память.

1.1.1 Модель графического конвейера

Модели современных графических конвейеров, которые используются в различных программных интерфейсах рендеринга, имеют схожие этапы, поэтому ограничимся рассмотрением модели из программного интерфейса Vulkan [10]. Основные этапы рендеринга показаны на рисунке 1.1. Сначала приведем краткое описание работы графического конвейера, затем рассмотрим каждый этап более подробно. Для каждого объекта сцены записывается команда рендеринга. Команда рендеринга содержит информацию о состоянии конвейера GPU и смещение в памяти, по которому хранятся вершины (индексы вершин) объекта. При выполнении прямого рендеринга команды составляются на CPU процессоре и посылаются по шине на GPU процессор. Далее устанавливается новое состояние графического конвейера или продолжается использование старого состояния. При обработке вершин выполняется преобразование для перевода вершин в плоскость изображения. Далее происходит растеризация для определения фрагментов пикселей, которые входят в графический элемент. При обработке фрагментов происходит вычисление цвета каждого фрагмента с учетом источников освещения и текстур. Видимость фрагментов вычисляется с помощью метода z-буфера. Далее происходит объединение результатов, запись в кадровый буфер и вывод на экран.

На этапе сбора входных данных (Input Assembler) происходит чтение вершин из одного или нескольких буферов для создания графических элементов. Существует два способа сбора графических элементов: неиндексированный, индексированный. При неиндексированном сборе вершины зачитываются из буфера вершин. При индексированном сборе сначала проводится чтение индексов, которые задают смещение в буфере, затем чтение вершин и нормалей. Вершины объединяются в графические элементы: точка, линия, треугольник, полоса треугольников (triangle strip), веер треугольников (triangle fan) и др. Вершинам и графическим элементам присваиваются идентификаторы vertexID, primitiveID, instanceID. В случае использования отдельного буфера для матриц преобразования (instancing) выполняется рендеринг одних и тех же графических элементов с последовательным применением различных матриц.

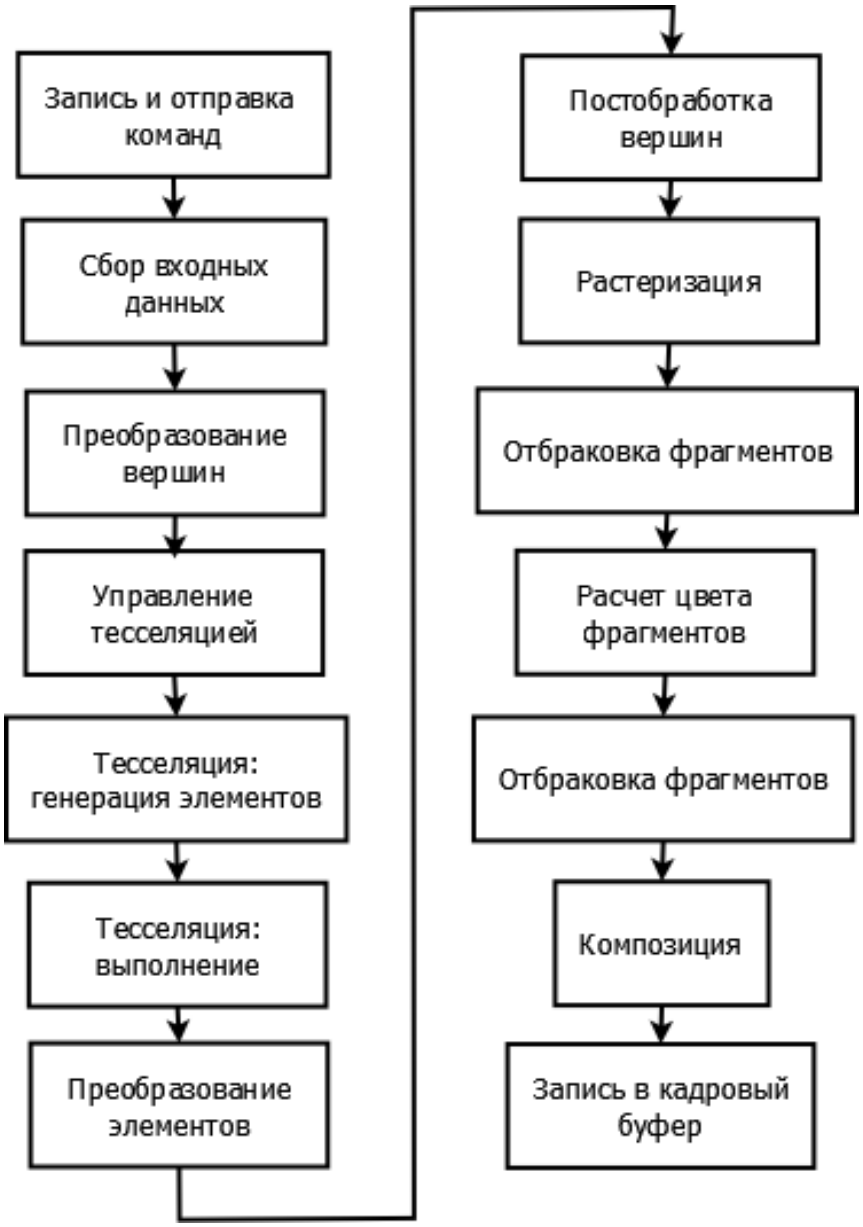

Рисунок 1.1 — Основные этапы современного графического конвейера.

Для каждой вершины, полученной на этапе сбора входных данных, выполняется преобразование. Сначала происходит проверка наличия вершины в кэш памяти. Если она отсутствует в кэше, то выполняется программа по обработке вершин, которую заранее загрузил пользователь [11]. Выполняется перевод вершины в плоскость изображения. Для этого происходит умножение вершины на матрицу объекта, матрицу камеры, матрицу перспективной проекции.

Во время тесселяции происходит создание новых графических элементов для повышения уровня детализации объектов. Вершины, поступающие на вход, называются управляющими. На этапе управления тесселяцией (tessellation control) происходит определение уровней тесселяции с помощью программы, заданной поль-

зователем. Затем производится генерация графических элементов согласно заданным уровням тесселяции. В завершение выполняется преобразование вершин созданных графических элементов (tessellation evaluation). Этапы графического конвейера, связанные с тесселяцией, выполняются только в том случае, когда заданы программы для управления и выполнения тесселяции. Зачастую этот этап используется для генерации ландшафтов [12].

На этапе преобразования элементов (Geometry Shader) выполняется обработка графических элементов для создания новых элементов или отбраковки элементов целиком. Можно преобразовать графические элементы одного типа в другой тип (например, точку в треугольник). Выходные данные можно направить далее на растеризацию или записать в буфер. Этот этап можно использовать для реализации разных методов: генерация треугольников по вершинам (particle system), генерация утолщенных линий, создание теневых объемов.

Рассмотрим операции, которые выполняются на этапе постобработки вершин. Обработанные вершины можно записать в буфер Transform Feedback. Это позволяет запустить повторное выполнение конвейера с обработанными данными или скачать данные в основную память для обработки на центральном процессоре. Далее выполняется отсечение графических элементов (clipping). Если элемент выходит за пределы экрана, то невидимая часть отсекается. Для новых вершин также вычисляются новые значения атрибутов. Если элемент целиком лежит за пределами экрана, то он удаляется. Далее выполняется оконное преобразование, которое задает смещение окна и интервал глубины (depth range).

Во время растеризации по проекции графического элемента создается растровое представление. Узел экранной сетки с координатами (x, y, z) и дополнительными атрибутами, которые назначаются на этапе по расчету цвета фрагментов, называется фрагментом. Для точки происходит создание фрагментов вокруг центра точки в соответствии с заданными размерами. Растеризация линии происходит в соответствии с заданным параметром ширины. Для треугольника проводится вычисление фрагментов, которые находятся внутри его границы. По вычисленной площади треугольника со знаком определяется, является ли треугольник лицевым (front-facing) [13]. Зачастую происходит отбраковка нелицевых треугольников (backface culling).

Перед проведением дорогостоящих вычислений цвета фрагментов может выполняться предварительная отбраковка фрагментов, если это допустимо. Во время

отбраковки по глубине (depth test) отбрасываются фрагменты со значением глубины, которое больше записанного в буфере значения. Во время отбраковки относительно прямоугольников (scissor test) выполняется проверка принадлежности координат фрагмента заданным пользователем прямоугольникам. Также на этом этапе могут проводиться проверки видимости (occlusion queries). Проверки видимости используются для подсчета количества видимых фрагментов.

Во время этапа по расчету цвета фрагментов проводится вычисление цвета на основе модели освещения и заданных источников освещения. На этом этапе может происходить обращение к текстурам для выборки цветов, нормалей и других параметров обрабатываемых фрагментов.

Далее выполняется отбраковка фрагментов. Этот этап пропускается, если он был выполнен ранее.

Во время композиции проводится объединение фрагментов с одинаковыми координатами на экране с использованием функций для интерполяции значений. В качестве весовых коэффициентов используются альфа-компоненты текущего и записываемого фрагментов. Далее выполняется запись в кадровый буфер и вывод на экран.

1.1.2 Схемы рендеринга

Выделяют следующие схемы выполнения рендеринга [14]:

- однопроходная,
- многопроходная,
- схема с отложенным проходом по вычислению цвета фрагментов (deferred rendering).

В однопроходной схеме выполняется обход объектов сцены, запись командного буфера, отправка буфера на выполнение. Для этого требуется компиляция шейдерных программ с различными комбинациями источников и моделей освещения.

В многопроходной схеме выполняется отдельный обход объектов сцены для каждого источника освещения, запись командного буфера, отправка буфера на выполнение. Эта схема используется для сокращения затрат за счет реализации эффективных шейдерных программ для разных типов источников освещения. В частности, производительность улучшается из-за отсутствия условных переходов и сокращения количества инструкций в шейдерных программах.

В схеме с отложенным проходом по вычислению цвета фрагментов рендеринг выполняется в два прохода [15]. Первый проход используется для составления буфера глубины. При этом используются шейдерные программы с минимальными затратами, отключается запись в кадровый буфер. Второй проход используется для вычисления цвета на основе модели освещения и заданных источников освещения. Достоинством этого способа является сокращение затрат на вычисление цвета фрагментов, которые в конечном счете оказываются невидимыми.

1.1.3 Составление командного буфера

На производительность рендеринга влияют техники составления командного буфера. Современные видеокарты могут выполнять запись командных буферов непосредственно на GPU процессоре (indirect rendering). При этом отсутствуют затраты на пересылку команд по шине. Составление командного буфера может выполняться на центральном процессоре (direct rendering). Это позволяет гибко выполнять отбор объектов, которые необходимо включить в итоговый буфер.

Рассмотрим развитие возможностей программных интерфейсов рендеринга для ускорения записи и отправки команд рендеринга. В программном интерфейсе OpenGL до версии 1.4 команды рендеринга составляются в режиме immediate mode [16]. Процедура glBegin вызывается в начале записи команды. Затем следует описание всех индексов, вершин, текстурных координат для вывода объекта. В конце вызывается процедура glEnd. Таким образом, производится задание атрибутов всех вершин для каждого кадра. Недостатком этой схемы является повышенный расход ресурсов процессора при составлении большого буфера команд. В OpenGL 1.5 добавлены объекты VBO, которые позволяют хранить вершины в видеопамяти. Проводится запись вершин и индексов объектов в область основной памяти, затем загрузка буфера в видеопамять. При составлении команд записывается ссылка на вершинный буфер с указанием смещения. Благодаря этому сокращается время записи команд. Расширение OpenGL NV_command_list позволяет включить в командный буфер также состояние конвейера и выполнить компиляцию итогового буфера. Это повышает эффективность рендеринга сцен САПР с большим количеством объектов [17]. Программный интерфейс Vulkan позволяет более эффективно работать с командными буферами и при необходимости отключать валидацию состояния [10].

Рассмотрим статьи по работе с командными буферами в Vulkan. В статье [18] отмечено, что в Vulkan выделяются этапы по составлению и исполнению команд.

Это позволяет избежать простоев, связанных с синхронизацией CPU и GPU процессоров. Запись команд проводится с указанием конфигурации конвейера. Благодаря этому командные буфера не зависят друг от друга, допускается отправка в любом порядке. Запись буферов можно выполнять параллельно в нескольких потоках. Это может повысить производительность рендеринга, если вычислительные процессы на CPU являются узким местом. В статье [19] измерено время рендеринга при повышении количества отображаемых объектов и показано, что Vulkan позволяет более эффективно использовать ресурсы CPU и GPU по сравнению с OpenGL.

Исследование производительности составления командных буферов было проделано в работе [5]. Показано, что для сокращения затрат на запись и отправку нужно выполнять фрагментацию и кэширование командных буферов в основной памяти. Для этого выделяются группы объектов, инициализируются вспомогательные буфера (secondary command buffer), в которые записываются команды для этих групп объектов. Преимущество в поддержании буферов для групп объектов заключается в том, что при локальных изменениях в сцене необходимо выполнить запись только для затронутых буферов.

1.2 Методы удаления невидимых поверхностей

Методы удаления невидимых поверхностей (occlusion culling) используются для сокращения вычислительных затрат в процессе рендеринга [20; 21]. Выделяют следующую классификацию методов:

- Выполнение предобработки. Предобработка объектов проводится для сохранения информации о видимости объектов. Затем в зависимости от положения камеры используется заранее подготовленное множество видимых объектов.
- Определения видимости объектов в плоскости изображения или в трехмерном пространстве. Видимость в плоскости изображения, как правило, определяется по проекции объекта на экран, а видимость в трехмерном пространстве по ограничивающим параллелепипедам AABB.
- Точность результатов. Консервативные методы завышают количество видимых объектов, но получают результаты быстрее. Приближенные методы не гарантируют нахождение всех видимых полигонов.
- Объединение объектов-преград. Рассмотрим три объекта A, B, C. Может быть так, что A и B по отдельности не закрывают C, а вместе закрывают.

Объединение объектов-преград позволяет получить более точные результаты видимости.

1.2.1 Методы, сохраняющие информацию о видимости во время предобработки

В сценах, состоящих из зданий, можно выделить ячейки, которые соответствуют комнатам, площадкам и коридорам. Для каждой ячейки вычисляется множество потенциально видимых объектов (PVS, Potentially visible set). Видимость объектов определяется через так называемые порталы.

Рассмотрим метод вычисления множества объектов PVS, описанный в работах Airey [22; 23]. Создается дерево BSP, которое содержит информацию о ячейках. При этом выделяются наиболее эффективные плоскости деления. Затем проводится вычисление порталов путем вычитания полигонов, лежащих на границе, из границы ячейки. Для каждого портала вычисляются видимые полигоны и добавляются в PVS. Точное решение этой задачи достаточно трудоемкое, поэтому предлагается определять порталы, которые не находятся в тени (shadow volume) обработанных полигонов. Это дает консервативно завышенное количество видимых полигонов. Но даже при использовании описанных техник алгоритм имеет временную сложность $O(n^3)$.

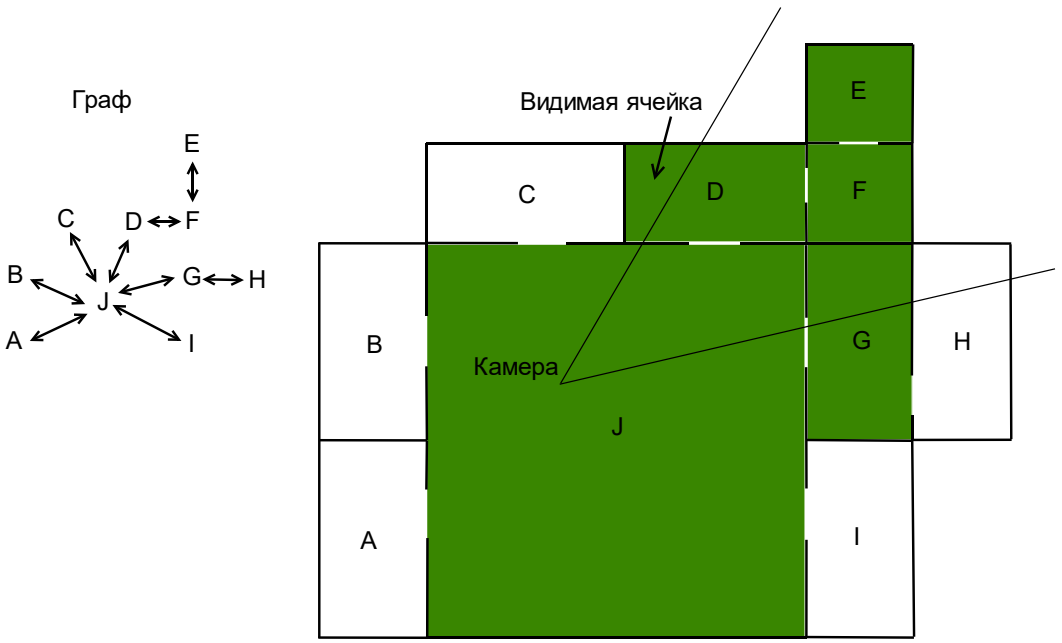

Рисунок 1.2 — Использование ячеек и порталов для определения видимых объектов. Слева показан граф смежности ячеек. Справа показана двухмерная проекция сцены, состоящей из комнат и дверных проемов. Зеленым цветом выделены видимые ячейки.

В работах [24; 25; 26] предлагается метод вычисления видимости с использованием графа с ячейками и порталами (рисунок 1.2). Во время предобработки производится вставка полигонов модели в BSP дерево, выделяются выпуклые ячейки. Главные непрозрачные поверхности, такие как стены, используются для определения границ ячеек. Объекты с маленьким количеством полигонов считаются неблокирующими и пропускаются на данном шаге. Объекты-порталы, такие как двери, определяются на границах ячеек и используются для создания графа с ячейками и порталами. Граф смежности показывает ячейки, которые напрямую соединяются через порталы. Видимость между двумя точками в пространстве определяется путем проведения линий и вычисления пересечений с границами ячеек. Ячейки, достижимые из данной ячейки путем проведения линий, добавляются во множество PVS. Во время прохода по сцене допускается уточнение видимости с помощью области видимости камеры [27]. Ячейка считается видимой, если выполняются условия:

- Ячейка лежит в области видимости камеры.
- Все ячейки вдоль проводимых линий лежат в области видимости камеры.
- Все порталы вдоль проводимых линий лежат в области видимости камеры, существует линия, которая проходит через все порталы.

Выполняется рендеринг объектов всех видимых ячеек на GPU процессоре. Преимуществом метода является то, что видимость вычисляется только с использованием геометрии ячеек и порталов без дополнительных затрат на объекты внутри ячеек. Это позволяет ускорить вычисление PVS.

В работе [28] описан метод составления PVS с использованием расширенных проекций (Extended projections). Создается иерархия ячеек, для каждой из которых записываются карты с проекциями объектов-преград. При этом площадь проекции занижена, чтобы получать консервативные оценки видимости. Затем проверяется видимость узлов структуры пространственного разбиения относительно составленных карт. Для объединения карт нескольких ячеек предлагается выполнять повторную проекцию на плоскость. Это повышает эффективность метода за счет обработки маленьких объектов-преград.

Структура пространственного разбиения на основе BSP-дерева позволяет выполнить обход сцены в порядке back-to-front при любом положении камеры [29]. Этот метод получил широкое распространение при отображении статических сцен. Недостатком метода является создание новых полигонов при пересечении объектов секущей плоскостью.

В работе [30] предложен метод ускорения рендеринга динамических сцен на основе ограничивающего объема с учетом движения (temporal bounding volume). Ограничивающий объем включает все положения объекта в рамках анимации сцены. Это ускоряет обработку подвижных объектов, если известна информация о движении объектов. Но остается проблема обработки объектов без детерминированного характера движения, с изменением видимости со временем.

1.2.2 Методы, вычисляющие видимость в плоскости изображения

Основными методами вычисления видимости в экранной плоскости являются метод z-буфера и трассировка лучей. Ограничимся рассмотрением только моделей с непрозрачными поверхностями. Тогда задача сводится к поиску первой видимой поверхности для каждого пикселя изображения.

При использовании метода z-буфера выполняются следующие шаги: инициализация z-буфера, растеризация полигона, определение видимости и вычисление цвета для каждого фрагмента [13]. Отметим, что метод не учитывает пространственную и временную когерентность. Производится вывод каждого полигона по отдельности и не используются результаты из предыдущего кадра. Для сокращения объема входных данных используются методы отсечения объектов, не попадающих в область видимости камеры (Frustum culling) [31], а также удаления задних граней (Back-face culling) [32].

В методах трассировки лучей зачастую используется пространственное разбиение, которое позволяет найти полигоны в заданной области пространства, выполнить обход полигонов в заданном порядке [33]. Благодаря этому сокращаются затраты на поиск первой поверхности, пересекающейся с лучом. Таким образом, в трассировке лучей учитывается пространственная когерентность, но не применяется временная и картинная когерентность.

В работе [34] предлагается использовать окто-дерево для выполнения проверок видимости для областей пространства. Для этого применяется Z-пирамида — иерархическое разбиение плоскости изображения. При построении пирамиды буфер глубины делится на 4 части так, что в каждой из них хранится самое дальнее значение глубины. Процесс деления продолжается до пикселей. Z-пирамида позволяет быстро исключить невидимые треугольники, выполнив сравнение минимального значения z треугольника и максимального значения z в области. Недостатком метода является затратное обновление пирамиды при добавлении новых объектов, которое снижает эффективность при работе с большими сценами.

В работе [35] предлагается использовать маски покрытия (coverage mask) для определения видимости. В основе предлагаемого метода лежит алгоритм Варнока [36]. На вход подается массив полигонов в порядке удаления от камеры (front-to-back). Происходит рекурсивное деление изображения до тех пор, пока видимость полигонов не может быть определена для каждого квадранта. В [35] предлагается ускорить алгоритм Варнока за счет использования иерархии битовых масок (covered, vacant, active). Предлагается заранее составлять битовые маски для всех возможных вариантов пересечения ребра и сетки заданного размера (рассматривается размер 8x8). Итоговая маска полигона получается объединением масок для его ребер. Маски организованы иерархически для быстрой отбраковки полигонов, которые попадают в заполненные ячейки. Если часть полигона оказывается видимой, то происходит обновление иерархии масок. Для дальнейшего ускорения метода предлагается использовать окто-дерево, которое также хранит в листьях BSP-деревья. Тогда осуществляется обход октантов в порядке удаления от камеры и проверка на видимость октантов без обновления иерархии масок. В случае видимости листового октанта выполняется обход его BSP-дерева. В результате получается метод, который использует все типы когерентности: пространственную, временную, картинную. Главное преимущество этого метода по сравнению с методом иерархического z-буфера заключается в низких требованиях по памяти и в отсутствии перезаписи пикселей. Однако требуется специализированное аппаратное обеспечение для эффективной реализации.

В работе [37] предлагается использовать аппаратные возможности GPU процессоров для генерации масок покрытия. В этих масках хранится только альфа компонента, а глубина отсутствует. На этапе предобработки выполняется отбор больших объектов. Затем составляются маски покрытия для больших объектов, которые лежат перед заданной плоскостью z-plane. Для этого проводится рендеринг в заданном разрешении, загрузка кадрового буфера в основную память и рекурсивное составление уменьшенных изображений. При отображении остальных объектов, которые лежат за плоскостью z-plane, выполняется отбраковка относительно полученной иерархии масок покрытия.

1.2.3 Методы с отбором объектов-преград

В работах [38; 39] предложен метод выполнения проверок видимости с использованием больших объектов (large convex occluders). Если камера находится за большим объектом в области между разделительными плоскостями (supporting

planes), то объекты вдали от камеры между плоскостями оказываются невидимыми (рисунок 1.3). Применение этого метода к структурам пространственного разбиения позволяет повысить производительность. Однако подбор ограниченного количества таких объектов зачастую вызывает трудности, а анализ видимости объектов с учетом множественных преград (occluder fusion) может требовать значительных вычислительных ресурсов.

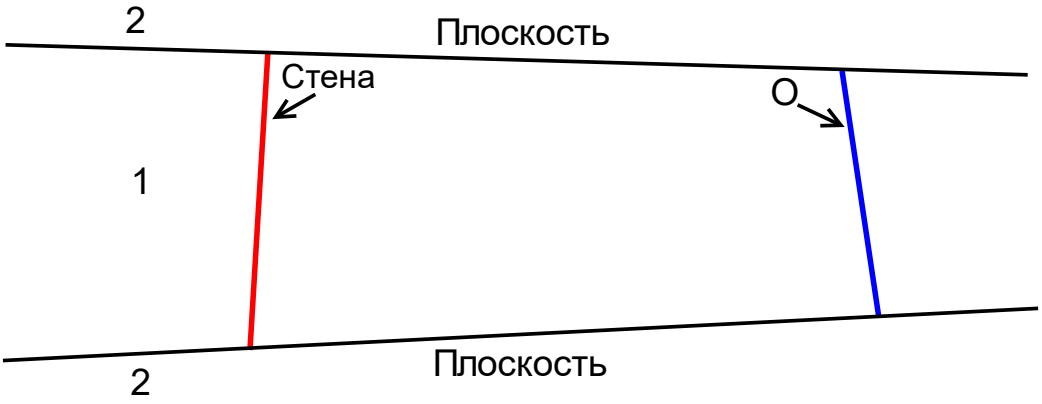

Рисунок 1.3 — Плоскости, которые задают визуальные события о смене видимости объекта O. Стена загораживает объект O, если камера находится в области 1.

В работе [40] предложено определять видимость с помощью теневых объемов (Shadow frusta). Для лучших объектов-преград производится построение пирамид с вершиной в месте положения камеры. Затем проверяется видимость узлов структуры пространственного разбиения относительно каждой пирамиды. Если узел оказывается невидимым относительно одной из пирамид, то объекты внутри узла считаются невидимыми.

В работе [41] предложено развитие метода [34] с использованием векторных инструкций SSE. На этапе предобработки проводится отбор больших объектов-преград. Для каждого кадра выполняется растеризация больших объектов для составления буфера глубины на центральном процессоре. Затем относительно полученного буфера глубины выполняется проверка видимости ограничивающих параллелепипедов AABB объектов. В статье [2] предложен алгоритм, который выполняет проверки видимости относительно буфера глубины, загруженного с GPU процессора. Это позволяет использовать все выведенные на экран объекты в качестве преград.

1.2.4 Методы на основе аппаратных проверок видимости

В работе [42] предлагается расширение OpenGL для выполнения проверок видимости на GPU процессоре. Сначала задаются части экрана (тайлы), в рамках которых нужно определить видимость объектов. Затем выключается запись в кадровый буфер и проводится рендеринг ограничивающих параллелепипедов объектов для определения их видимости. Во время этапа по отбраковке относительно буфера глубины определяется видимость фрагмента с заданной глубиной. В случае положительного результата координаты фрагмента передаются в блок Occlusion Unit, который занимается подсчетом количества всех фрагментов внутри тайла, видимых фрагментов, занимаемой площади. Далее происходит объединение результатов и запись в основную память. Получив результаты проверки видимости, приложение принимает решение об отображении самого объекта.

В 2002 году было разработано расширение OpenGL NV_occlusion_query, которое позволяет использовать аппаратные возможности видеокарт для выполнения проверок видимости относительно буфера глубины [43]. Каждая проверка видимости имеет свой уникальный идентификатор. Расширение допускает одновременное выполнение большого количества проверок видимости. При выполнении прямого рендеринга (direct rendering) необходимо загрузить результаты проверок видимости в основную память. Это вызывает задержку, связанную с передачей данных из видеопамяти в основную память. Во избежание простоя используются результаты предыдущих проверок видимости в текущем кадре [44; 45; 2].

Рассмотрим методы для сокращения количества проверок видимости. Некоторые сцены содержат большое количество объектов с маленьким количеством треугольников. В таком случае, ресурсы для проверки видимости объекта могут быть сопоставимы с ресурсами для выполнения рендеринга. Тогда целесообразно использовать пространственный индекс и выполнять проверки видимости для узлов дерева [6]. Зачастую на выполнение аппаратных проверок видимости расходуются вычислительные ресурсы, а объект (узел дерева) при этом оказывается невидимым. В худшем случае большинство проверок видимости выполняется впустую. Некоторые методы используют приблизительные вероятностные критерии для оценки эффективности проверок видимости и выполняют проверку видимости только в случае значительного потенциального выигрыша производительности [46]. Однако для этого требуются ресурсы на поддержку дополнительного буфера глубины, который используется для оценки вероятности видимости объекта (узла).

Существуют методы, которые выполняют генерацию команд рендеринга на GPU процессоре [47]. Буфер глубины, полученный в предыдущем кадре, используется для предварительного определения видимости объектов. Затем проводится повторная проверка невидимых объектов. Для этого выполняется повторное проецирование ориентированных ограничивающих параллелепипедов и обновление буфера глубины. Затем z-буфер используется для проведения предварительной отбраковки фрагментов (early depth testing). Для видимых фрагментов выполняется запись о видимости соответствующего объекта. С использованием полученной информации о видимости производится генерация буфера команд в вычислительном шейдере. Главным достоинством таких методов является отсутствие синхронизации с центральным процессором, что позволяет использовать результаты проверок видимости без задержек. Однако эти методы хранят информацию об объектах в буфере с заданным смещением, что вызывает проблемы при работе со сложными динамическими сценами, в которых количество объектов в узлах меняется со временем.

1.3 Методы на основе оценки времени рендеринга

Производительность рендеринга зависит от скорости выполнения этапов графического конвейера. Для выбора наиболее эффективного метода отображения требуются методы оценки вычислительных затрат рендеринга.

В работе [48] предложен адаптивный метод рендеринга с заданной частотой кадров. Отображение каждого объекта выполняется с подходящим уровнем детализации (level of detail) для поддержания целевой частоты кадров. Как правило, уровень детализации выбирается на основе расстояния до объекта и размера объекта, но это не всегда дает равномерную частоту кадров. Частота кадров может сильно меняться, когда появляются новые объекты или изменяются их уровни детализации. Для устранения этого недостатка можно адаптивно изменять пороговые размеры для вывода уровней детализации. Эта техника хорошо справляется со сценами, в которых видимость объектов слабо меняется при переходе к следующему кадру. В некоторых сценах количество объектов значительно меняется при движении камеры. Для таких сцен требуется выполнить предварительную оценку затрат на отображение. Исходя из этой оценки, можно выбрать подходящие уровни детализации объектов. Для каждого объекта осуществляется выбор уровня детализации и алгоритма затенения (равномерное затенение, затенение по Гуро, использование текстурных карт). Задача сводится к поиску значений, которые осуществимы с целевой частотой кадров и соответствуют максимальному качеству изображения.

Рассматривается упрощенная модель графического конвейера, которая включает в себя обработку графических элементов и обработку фрагментов (рисунок 1.4). Предполагается, что отправка команд рендеринга происходит мгновенно, на GPU процессоре выполняется только рассматриваемое приложение. Время работы конвейера определяется временем самого медленного этапа. Для оценки времени обработки графических элементов предлагается использовать линейную комбинацию количества полигонов и вершин. Получаем формулу:

$$Cost(O, L, R) = max\begin{pmatrix} C_1 Poly(O,L) + C_2 Vert(O,L) \\ C_3 Pix(O) \end{pmatrix},$$

где O — объект,

L — уровень детализации объекта,

$Poly(O, L)$ — количество полигонов объекта O,

$Vert(O, L)$ — количество вершин объекта O,

C_1, C_2, C_3 — константы рендеринга, которые предлагается определять экспериментальным путем. Для оценки вклада объекта в итоговое изображение используется следующая эвристика. Чем больше размер объекта на экране (количество выводимых пикселей), тем важнее этот объект. Также важна точность отображения, которая оценивается по отклонению от эталонного изображения объекта:

$$Accuracy(O, L, R) = 1 - Error = 1 - \frac{BaseError}{Samples(L,R)^m},$$

где $Samples(L, R)$ — количество вершин при использовании затенения по Гуро, количество полигонов при использовании плоского затенения,

m — значение экспоненты, которое зависит от алгоритма затенения (1 для плоского затенения, 2 для затенения по Гуро),

$BaseError$ — подобранное значение ошибки. В работе используется $BaseError = 0.5$ для штрафования плоского затенения с одним полигоном ($Samples(L,\ R)^m = 1$).

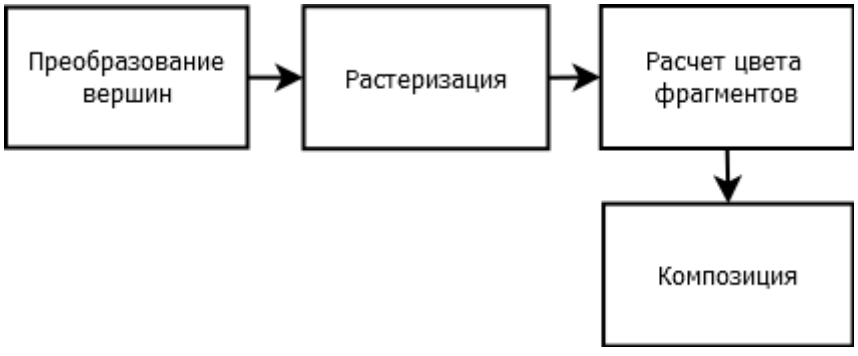

Рисунок 1.4 — Основные этапы рендеринга объектов сцены.

Помимо точности воспроизведения цвета имеют значение следующие факторы: важность объекта (стены важнее гвоздей), отклонение от центра экрана (focus), скорость движения объекта (motion), резкие переходы к другим уровням детализации (hysteresis). Каждый из этих факторов изменяется от 0 до 1. Получаем следующую эвристику для оценки вклада объекта в итоговое изображение: $Benefit(O, L, R) = Size(O) * Accuracy(O, L, R) * Importance(O) * Focus(O) * Motion(O) * Hysteresis(O, L, R)$.

Задача по оптимальному выбору $Cost$ и $Benefit$ для объектов сцены является NP-сложной. Она относится к классу задач о рюкзаке с множественным выбором (Continuous Multiple Choice Knapsack Problem), в котором множество объектов делится на подмножества кандидатов и допускается добавление в рюкзак одного объекта из каждого подмножества [49; 50]. В рассматриваемом случае время отображения задает размер рюкзака, разные способы отображения объекта задают подмножества, функции $Cost$ и $Benefit$ задают размер и пользу каждого объекта. Задача заключается в выборе объектов, которые поместятся в рюкзак и принесут максимальную пользу. В работе предлагается реализация жадного алгоритма, который подбирает максимальное соотношение $Benefit/Cost$ и имеет временные затраты $O(n * \log(n))$.

В работе [51] предложена модель производительности для определения времени рендеринга трехмерной сцены. Рассматривается общая формула для вычисления времени рендеринга:

$t = RT(SG, RA, HW, ST)$,

где SG — время обхода объектов сцены (scene graph),

RA — процедура по отображению объекта (rendering action),

HW — аппаратное обеспечение (hardware),

ST — текущее состояние операционной системы и аппаратного обеспечения (state). Для упрощения этой формулы вводится обозначение для массива объектов $X = (x_1, \dots, x_n)$. Считается, что каждый объект описывается своими атрибутами и геометрическим представлением $x_i = (g_i, a_i)$. Предполагается, что процедура по отображению объекта определяется его атрибутами. Тогда формула выглядит следующим образом: $t = RT(X, HW, ST)$. Для каждого кадра выполняется обход объектов сцены X, установка состояния GPU процессора согласно атрибутам a_i, посылка геометрии g_i. Драйвер отправляет составленные команды в командный буфер (очередь FIFO), GPU процессор читает отправленные команды и выполняет этапы графического конвейера для преобразования, растеризации, вычисления

цвета и вывода на экран треугольников объектов. Графические элементы состоят из вершин и индексов, геометрическое представление хранится либо в памяти AGP, либо в видеопамяти. Предлагаются следующие формулы для оценки временных затрат CPU и GPU процессоров:

$$RT = ET_{system} + \max\left(ET^{CPU}, ET^{GPU}\right),$$

$$ET^{CPU} = ET_{nr}^{CPU} + ET_{r}^{CPU} + ET_{mm}^{CPU} + ET_{idle}^{CPU},$$

$$ET^{GPU} = ET_{fs}^{GPU} + ET_{r}^{GPU} + ET_{mm}^{GPU} + ET_{idle}^{GPU},$$

где nr — затраты, не относящиеся к рендерингу,

r — затраты на рендеринг,

fs — затраты на установку состояния,

mm — затраты на обращения к памяти,

idle — время простоя.

В наилучшем сценарии, CPU и GPU процессоры работают параллельно без простоев (с загруженностью 100%). Зачастую этот сценарий является недостижимым. Задержка при выполнении обхода сцены на центральном процессоре вызывает простой в ожидании команд. Долгая обработка элементов на графическом конвейере вызывает простой центрального процессора. Для эффективного рендеринга необходимо сократить время простоя.

Предложен следующий метод оценки времени рендеринга. Пусть задано количество вершин, треугольников, пикселей. Тогда время рендеринга вычисляется как максимум времени преобразования вершин, растеризации треугольников, вычисления цвета пикселей. Для получения консервативной оценки предлагается суммировать времена работы разных этапов конвейера.

1.4 Методы на основе пространственного индексирования

Рассматриваемые в работе сцены имеют иерархическую организацию, в которой однотипные объекты объединяются в сборки. Такой способ группировки объектов помогает при выполнении задач по моделированию проектов, но не подходит для ускорения задач рендеринга, в которых требуется объединение объектов по степени близости. Для этого используются методы пространственного разбиения и индексирования объектов. Кластеризация имеет значение для выполнения следующих задач рендеринга: определение объектов, попадающих в область видимости камеры, удаление невидимых поверхностей, составление командных буферов. Данный раздел посвящен обзору методов пространственного индексирования

с целью выделения метода, который подходит для ускорения рендеринга целевого семейства сцен.

Рассмотрим свойства пространственных данных, чтобы лучше понять требования, предъявляемые к пространственным методам индексирования. Во-первых, пространственные данные имеют сложную структуру. Объект может состоять из одной точки или большого количества полигонов. Невозможно записать такие данные в структуры фиксированного размера. Во-вторых, данные обычно меняются динамически. Пространственный индекс должен сохранять эффективность после проведения вставок и удалений. В-третьих, пространственный индекс может занимать много памяти. Географические карты отнимают гигабайты свободного пространства. Возникает необходимость хранения во внешней памяти. В-четвертых, нет общепринятой алгебры для работы с пространственными данными [52; 53; 54; 55]. Это означает, что нет общепринятых операторов. Выбор операторов зависит от области применения. Хотя есть операторы, которые используются практически везде (например, пересечения). Многие пространственные операторы не являются замкнутыми. Пересечение двух полигонов может давать в результате набор точек, ребер или непересекающихся полигонов. Это имеет значение, когда операторы применяются последовательно. Также операторы имеют высокую вычислительную сложность по сравнению с операторами отношения. Важным классом геометрических операторов, которые необходимо поддерживать на физическом уровне, являются операторы пространственного поиска. Для выполнения поиска объектов в пространстве требуется быстрый доступ к информации о положении объектов. Для поддержки таких операций требуются специальные методы доступа к данным. Основная проблема в разработке таких методов заключается в том, что не существует метода упорядочения пространственных объектов, которые сохраняют близость в пространстве. Иными словами, нет такого отображения из трехмерного пространства в одномерное пространство, при котором близкие объекты в трехмерном пространстве остаются близкими в одномерном пространстве. В традиционных базах данных эта проблема решена и существуют эффективные методы доступа к данным [56; 57]. Распространенным способом обработки многомерных данных является последовательное применение этих структур данных для каждого измерения. К сожалению, этот подход во многих случаях не является эффективным [58].

Методы доступа к многомерным данным подразделяются на точечные методы (point access methods) и пространственные методы (spatial access methods). То-

чечные методы работают с точками многомерного пространства. Пространственные методы работают с объектами, которые имеют протяженность (линии, полигоны, полиэдры). Методы индексирования обычно работают более эффективно, когда объекты имеют более простое представление. Зачастую используют ограничивающий объем (параллелепипед или сферу). Индекс содержит только ограничивающий объем и идентификатор объекта. В качестве упрощенного представления объекта можно также использовать объединение ограничивающих объемов.

1.4.1 Одномерные методы индексирования

Одномерные методы индексирования являются основой для многомерных методов. Основными методами являются линейное хеширование [59; 60], расширяемое хеширование [57], B-дерево [61]. B-дерево работает эффективно с равномерным и неравномерным распределением данных. Методы хеширования теряют эффективность, когда данные неравномерные. Это особенно сильно проявляется в методах хеширования, которые сохраняют порядок (order preserving hashing).

При линейном хешировании сначала создается хеш-таблица с ячейками (buckets), в которой количество ячеек является степенью двойки. При достижении количества элементов, соответствующему коэффициенту наполнения хеш-таблицы, происходит добавление новой ячейки и повторная вставка элементов текущей ячейки в хеш-таблицу. При этом для новой ячейки и первой ячейки используется новый делитель. Ячейка имеет определенный размер. Если в ячейку вставляется новый элемент и размер ячейки превышает допустимый размер, его помещают в дополнительную структуру данных.

При расширяемом хешировании также создается таблица ячеек. Каждой ячейке соответствует уникальный индекс. При вставке по хеш коду вычисляется индекс ячейки в таблице. В отличие от линейного хеширования, при переполнении ячейки происходит увеличение размера таблицы в два раза, назначение новых индексов ячейкам и повторная вставка элементов текущей ячейки в таблицу.

B-деревья — сбалансированные деревья, в которых хранятся данные из интервала. В отличие от хеш-таблиц в B-деревьях данные организованы иерархически. Каждому узлу соответствует страница диска D и интервал I. Дочерние узлы разделяют интервал I на непересекающиеся интервалы. Листья содержат указатели на хранимые данные. При достижении максимального количества элементов в узле производится деление. Процесс деления распространяется вверх по дереву. Если

данные имеют равномерное распределение, то хеш-таблица более эффективно справляется с запросами по точному совпадению, вставками и удалениями.

1.4.2 Многомерные методы индексирования

Рассмотрим основные многомерные структуры данных. Ранние многомерные методы индексирования не использовали внешнюю память, поэтому они менее пригодны для больших пространственных баз данных. Для иллюстрации работы методов используется множество объектов, которое содержит 10 точек и 10 полигонов, которые равномерно заполняют область двухмерного пространства (рисунок 1.5). Модель полигона также включает в себя центр и ограничивающий прямоугольник.

Одним из самых распространенных методов индексирования многомерных данных является k-d-дерево (k dimensional tree) [62]. K-d-дерево — двоичное дерево, в котором производится рекурсивное деление множества объектов на два подмножества с помощью плоскости размерности $\mathrm{d} - 1$. Плоскости деления параллельны осям координат. Например, если $\mathrm{d} = 3$, то используются три плоскости, перпендикулярные плоскостям XY, YZ, XZ. Плоскости деления должны содержать хотя бы одну точку. Внутренние узлы содержат один или два дочерних узла и используются при поиске данных. На рисунке 1.6 показано результирующее дерево после вставки тестовых данных. Дерево поддерживает в качестве объектов только точки, поэтому полигоны представлены в виде центров. Первое деление производится в точке c_3, затем в точках p_{10} (левый узел), c_7 (правый узел). Порядок вставки элементов влияет на создаваемое k-d-дерево. Эта проблема решается путем выбора плоскости деления, которая делит множества объектов на два примерно равных множества [63]. Сложно поддерживать дерево сбалансированным, когда часто происходят вставки и удаления.

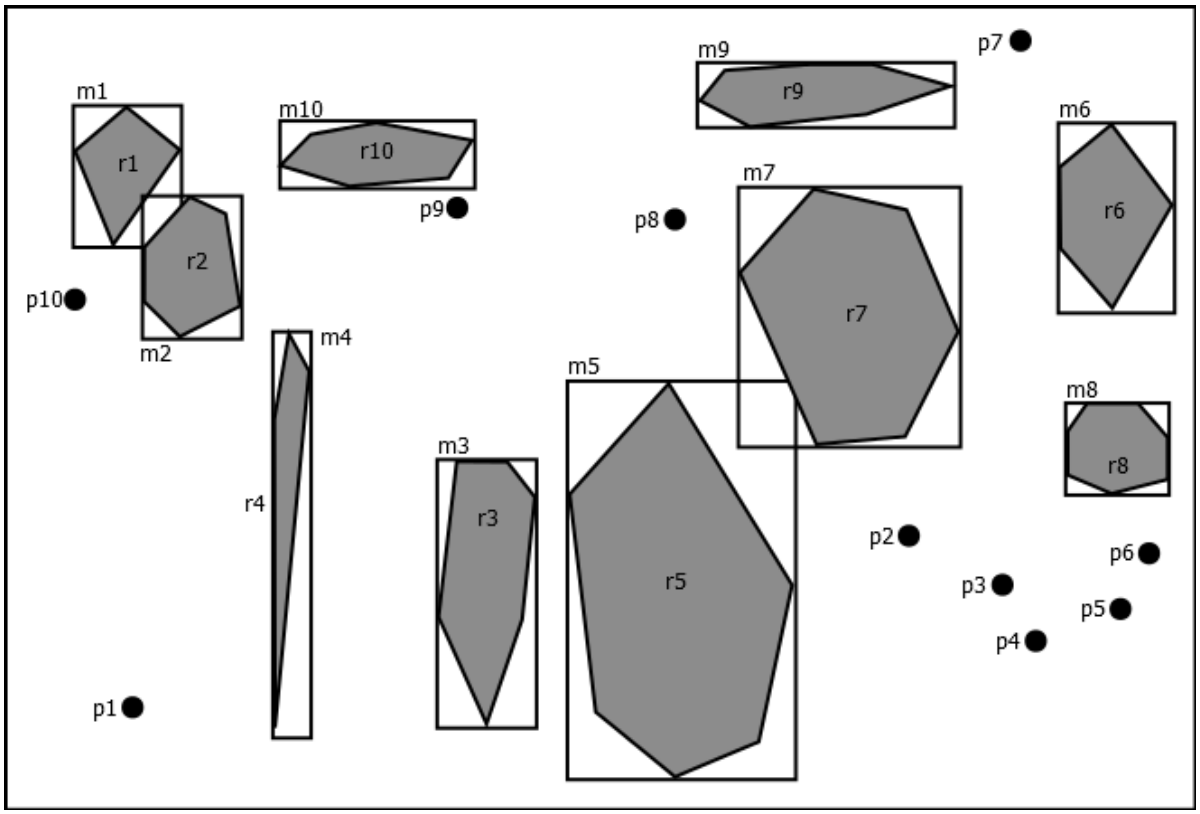

Рисунок 1.5 — Тестовые данные, которые используются для демонстрации работы методов индексирования.

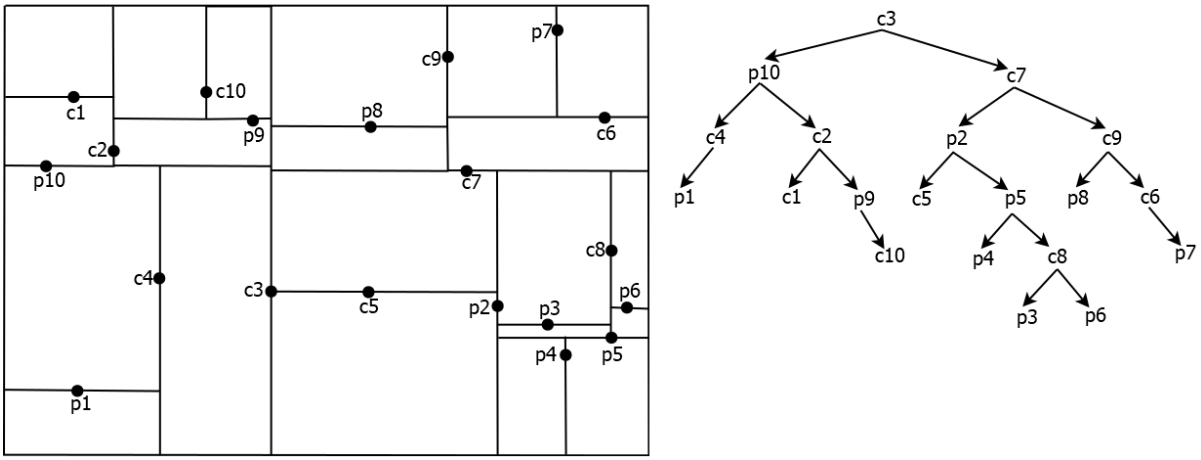

Рисунок 1.6 — K-d-дерево, которое получается в результате вставки тестовых данных.

BSP-дерево в отличие от k-d-дерева допускает деление вдоль плоскостей, которые не параллельны осям координат [64; 29]. Получается разбиение пространства, в котором плоскость деления является узлом, а каждое полупространство является дочерним узлом. На рисунке 1.7 показано BSP-дерево, которое получается в

результате вставки тестовых данных. BSP-деревья справляются с различными распределениями данных. Однако они не сбалансированы и могут иметь большую глубину.

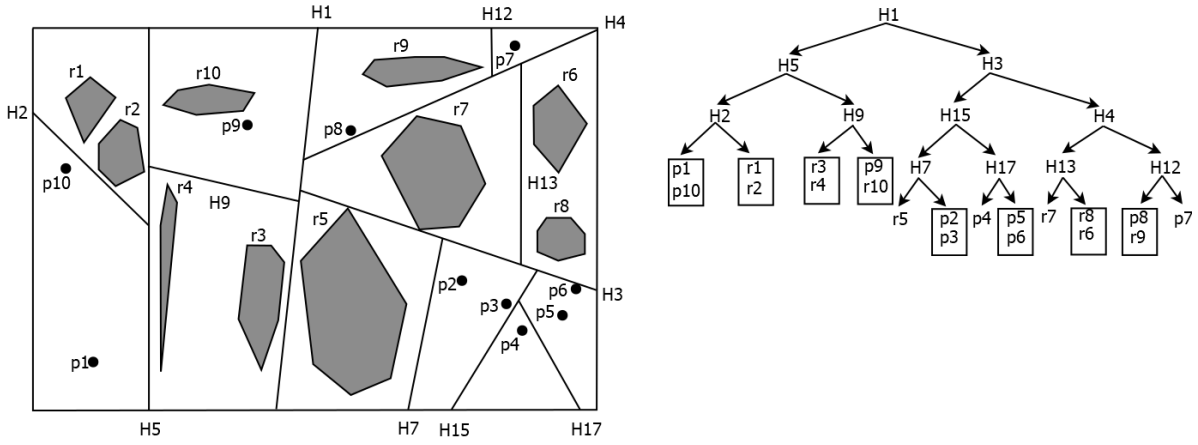

Рисунок 1.7 — BSP-дерево, которое получается в результате вставки тестовых данных.

Квадро-деревья в отличие от k-d-деревьев делят узел на четыре дочерних узла [65]. Дочерние узлы могут быть разного размера, но зачастую имеют одинаковый размер. Деление продолжается до тех пор, пока количество объектов больше порогового. Существуют разновидности квадро-деревьев для хранения точек (point quad-tree) и областей пространства (region quad-tree) [66]. На рисунке 1.8 показано квадро-дерево для точек, на 1.9 показано квадро-дерево, разбивающее области пространства.

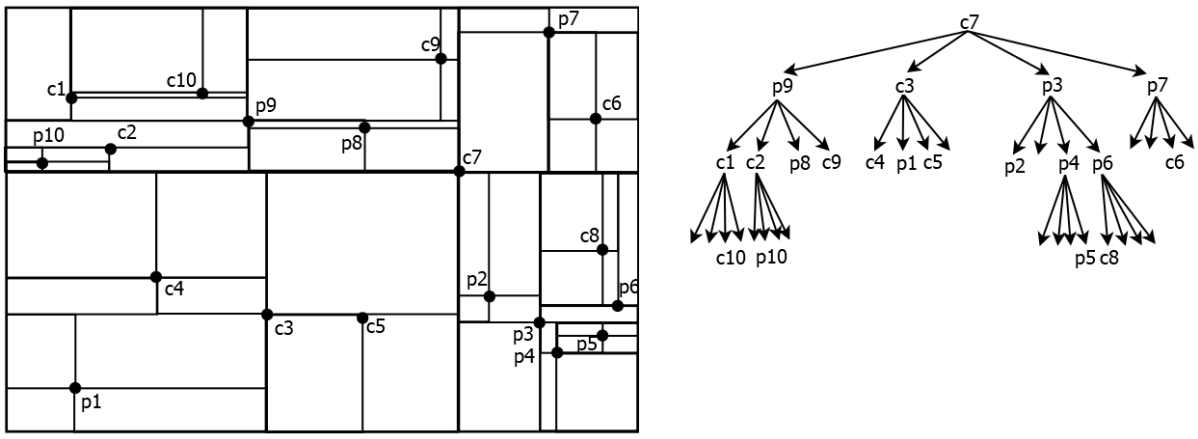

Рисунок 1.8 — Квадро-дерево для точек (point quad-tree), которое получается в результате вставки тестовых данных.

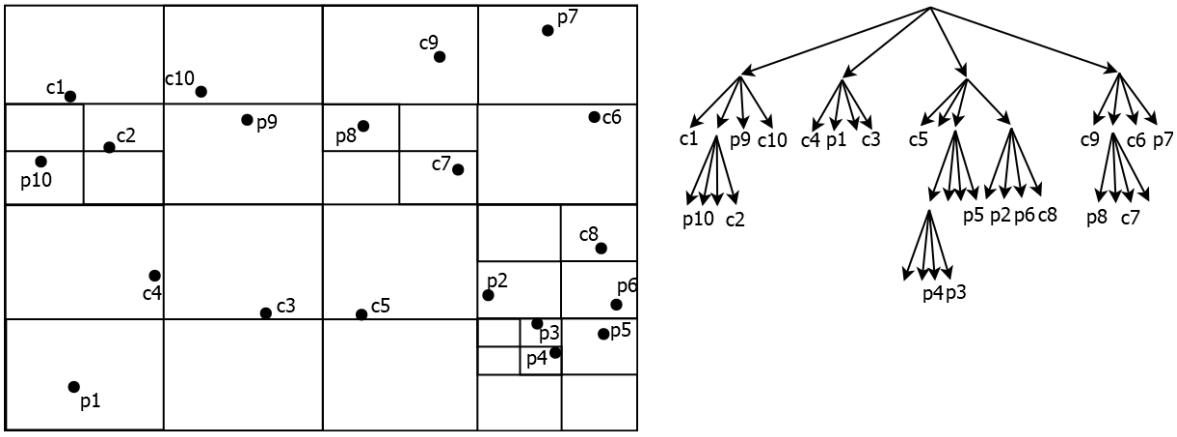

Рисунок 1.9 — Квадро-дерево для областей пространства (region quad-tree), которое получается в результате вставки тестовых данных.

Описанные выше многомерные методы были разработаны для основной памяти. В базах данных из-за большого объема данных необходимо использовать внешнюю память. При этом важно сократить количество обращений к диску. Рассмотрим, как при этом изменятся методы индексирования. Точки помещаются в ячейки (области пространства), которые соответствуют страницам на диске. Области пространства обычно имеют границы, параллельные осям координат. Доступ к ячейкам обычно осуществляется с помощью *n*-мерных хеш функций. Распространенные многомерные структуры данных, учитывающие доступ к внешней памяти, включают grid file [67], многомерное хеширование [68; 69; 70], hash tree [71], k-d-B-tree [72], hB-tree [73].

1.4.3 Пространственные методы индексирования

Описанные выше многомерные методы не применимы к пространственным данным (линиям, полигонам, полиэдрам), которые распространены в САПР, ГИС, виртуальной реальности. Для обработки объектов, которые не сосредоточены в одной точке, используются пространственные методы индексирования. Для перехода от многомерных методов к пространственным используются подходы [74]:

- отображение объектов;
- использование слоев;
- назначение объектов на области пространства.

При отображении объектов пространственные данные заменяются либо на точки другой размерности, либо на интервалы с помощью кривых, заполняющих пространство [75; 74; 76].

Рассмотрим использование слоев при индексировании пространственных данных. При вставке объект помещается в первый слой. Если объект пересекается с сеткой первого слоя, его вставляют во второй слой и т.д. Слои организованы иерархически, каждый слой определяет разбиение для всего пространства [77; 78].

Назначение объектов на области пространства допускает пересечения. Проводится назначение объекта целиком одной области пространства и при необходимости эта область пространства расширяется. При этом может возникнуть пересечение соседних областей. Это ухудшает производительность запросов поиска, потому что искомый объект может находиться во многих пересекающихся областях. Особенно остро эта проблема проявляется, когда выполняется вставка объектов с размером, значительно превышающим средний размер [79; 80; 81]. R-дерево — сбалансированная иерархия вложенных n-мерных параллелепипедов [79]. Каждый узел соответствует странице на диске, дочерние узлы содержатся в родительском узле. Узлы, находящиеся на одном уровне, могут пересекаться. Количество элементов в одном узле изменяется от m до M. Если количество элементов становится меньше m, то узел удаляется, а элементы распределяются между соседними узлами. Верхняя граница M определяется по размеру одной страницы диска. Корневой узел содержит, по крайней мере, два элемента. Узлы содержат минимальные ограничивающие параллелепипеды. Из-за этого поисковые запросы не дают точных результатов. Требуется отдельное обращение к диску для загрузки геометрии объектов [79; 80]. На рисунке 1.10 показано R-дерево, заполненное тестовыми данным.

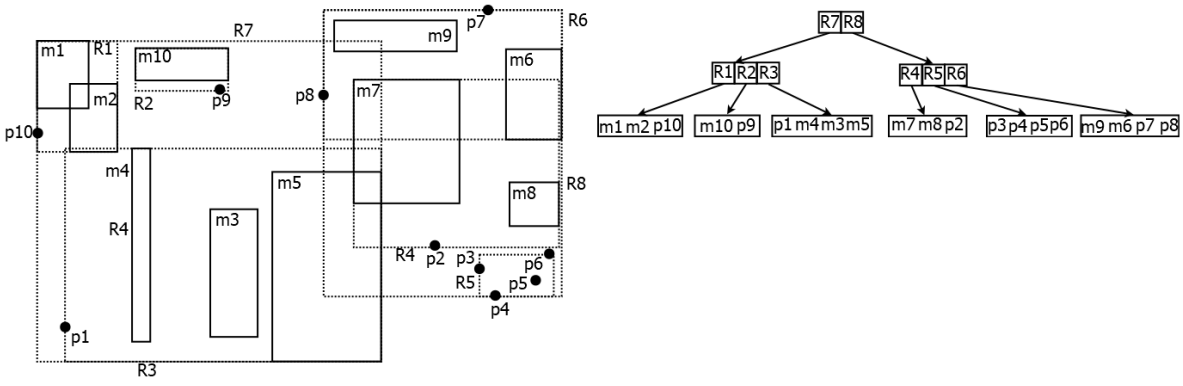

Рисунок 1.10 — R-дерево, которое получается в результате вставки тестовых данных.

BVH-дерево — двоичное дерево, которое было разработано для сокращения вычислительных затрат на проверки на пересечение лучей и ограничивающих объ-

емов в методах трассировки лучей [82; 83]. В качестве элементов дерева используют различные структуры: сфера, AABB, OBB, k-DOP, выпуклая оболочка (рисунок 1.11). Процесс построения BVH происходит сверху вниз, каждый шаг создается два новых объема. Существуют различные способы выбора плоскости разбиения. Можно выполнять деление самой длинной стороны, деление объектов на два подмножества одинакового размера. Наилучшая кластеризация достигается, когда деление производится на основе суммарной площади (Surface Area Heuristic) [84]. При построении дерева встречаются объекты, которые пересекаются с плоскостью разбиения. Такие объекты можно включить в оба узла или выполнить разбиение, создав новые объекты, которые не пересекаются с плоскостью деления.

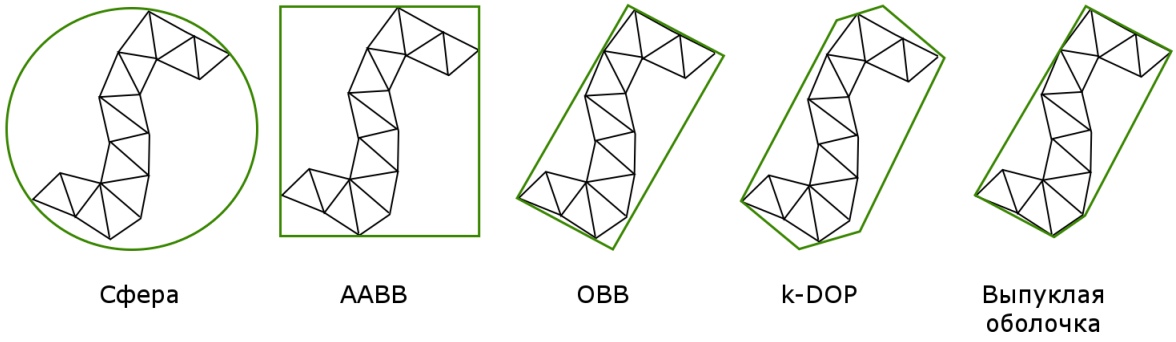

Рисунок 1.11 — Возможные ограничивающие объемы, которые используются в BVH-дереве.

BVH-дерево на основе SAH эвристики является эффективным методом индексирования, но имеет ряд недостатков. Во-первых, подсчет суммарной площади выполняется на каждом уровне. Во-вторых, сложно выполнить распараллеливание, когда деление производится сверху вниз. LBVH-дерево разработали для решения этих проблем [85]. При построении LBVH проводится сортировка вдоль заполняющей пространство кривой (Мортона, Гильберта, Пеано). Процедура создания пространственного индекса осуществляется с помощью сортировки вдоль кривой. Знание размера сетки позволяет легко выполнить распараллеливание на GPU.

Для равномерного деления пространства используется окто-дерево. При построении выполняется иерархическое деление на восемь узлов с помощью трех плоскостей, направленных вдоль координатных осей [86]. Когда ограничивающий параллелепипед AABB объекта пересекается с плоскостью деления, его либо оставляют на данном уровне (single reference octree), либо относят к нескольким узлам нижнего уровня (multiple reference octree). Хранение геометрии в листьях повышает качество кластеризации, но увеличивает расходуемый объем памяти.

В работах [87; 88; 89] разработан метод индексирования пространственных данных с использованием комбинированного индекса, который включает дерево представления сцены и дополнительные пространственные индексы для сложных составных объектов. Дерево сцены строится для некоторого момента времени и затем обновляется, когда на заданном временном интервале встречаются события. Выделение индексов для сложных объектов позволяет сократить затраты на перестроение дерева сцены при обработке временных событий. Это повышает производительность при вычислении столкновений объектов.

1.5 Сравнение и основные выводы

Были рассмотрены следующие схемы выполнения рендеринга: однопроходная, многопроходная, с отложенным проходом по вычислению цвета фрагментов. Вторую и третью схемы имеет смысл использовать, когда выполняется предобработка для объединения, упрощения объектов и сокращения количества команд, а расчет освещения является самым дорогим этапом графического конвейера. В данной работе рассматриваются динамические сцены, в которых каждый объект может иметь индивидуальное поведение в процессе рендеринга. Поэтому нет возможности выполнить описанную предобработку для сокращения количества команд. В связи с этим узким местом в процессе рендеринга также может быть составление буфера команд, преобразование вершин. Таким образом, для динамических сцен наиболее целесообразным является использование однопроходной схемы рендеринга, которая позволяет сократить время записи командных буферов и ресурсы, затрачиваемые на преобразование вершин.

Были рассмотрены особенности составления командного буфера на CPU и на GPU. В данной работе составление командного буфера проводится на CPU, чтобы можно было использовать аппаратные проверки видимости совместно с иерархической структурой пространственного разбиения.

Были рассмотрены методы удаления невидимых поверхностей. Большинство из них подходят только для статических сцен. Для отображения динамических сцен с интерактивной частотой кадров необходимо своевременно и эффективно проводить проверки видимости. Метод выполнения проверок видимости относительно буфера глубины GPU процессора на основе расширения NV_Occlusion_Query справляется с целевым семейством сцен, поддерживается в современных программных интерфейсах рендеринга.

Были рассмотрены методы оценки времени рендеринга трехмерных полигональных сцен. Существующие методы не учитывают работу по составлению и отправке буферов с командами рендеринга, которая может отнимать значительное время при большом количестве объектов, а также время выполнения проверок видимости. Эти аспекты учитываются в предлагаемой в данной работе модели производительности графического конвейера.

Были рассмотрены методы пространственного индексирования. Методы на основе BVH-деревьев, k-d-деревьев дают наилучшую кластеризацию объектов, позволяют быстро выполнить отбраковку невидимых групп объектов. Но они сталкиваются с трудностями при поддержании актуальных индексов для динамических сцен, состояние которых меняется в процессе отображения. Более того, выполнение отложенных проверок видимости для узлов BVH-деревьев, k-d-деревьев оказывается невозможным в связи с изменением границ узлов при обработке подвижных объектов. В разделе 5.2 приводится сравнение эффективности методов пространственного индексирования при выполнении рендеринга. Показано, что наиболее целесообразным является использование окто-дерева без множественных ссылок.

Зачастую для сокращения объема работы на этапах конвейера GPU подготавливаются альтернативные уровни детализации объектов (levels of detail), которые отображаются вместо исходного объекта в зависимости от расстояния до камеры. При этом подготавливаются отдельные буфера с геометрией объектов и командами рендеринга для каждого уровня детализации. Для этого требуются дополнительные затраты памяти и ресурсы процессора для поддержания актуального состояния буферов при изменении свойств объектов динамической сцены. Поэтому в данной работе альтернативные уровни детализации объектов не применяются.

ГЛАВА 2. МОДЕЛЬ ПРОИЗВОДИТЕЛЬНОСТИ ГРАФИЧЕСКОГО КОНВЕЙЕРА

В данной главе предлагается модель производительности графического конвейера, которая расширяет существующие модели, описанные в работах [48; 51]. Новизна заключается в том, что учитываются следующие аспекты рендеринга:

- затраты на запись и отправку командных буферов, в том числе с использованием рассматриваемых техник;
- затраты на отправку, выполнение, получение результатов аппаратных проверок видимости.

Выводятся формулы для расчета времени рендеринга с использованием альтернативных базовых методов и техник:

- удаление невидимых объектов с использованием методов пространственной декомпозиции и индексирования;
- отложенные аппаратные проверки видимости;
- фрагментация и кэширование командных буферов с учетом локальных изменений в сцене.

Полученные формулы позволяют своевременно выбирать наиболее эффективные способы реализации рендеринга в рамках однопроходной схемы в процессе отображения динамических сцен.

2.1 Исследуемая модель графического конвейера

Хотя графический конвейер содержит большое количество этапов, многие из них не являются необходимыми для выполнения рендеринга. Рассмотрим упрощенную модель графического конвейера, которая описывает рендеринг за один проход по объектам и графическим элементам. На рисунке 2.1 изображены основные этапы рендеринга, которые включают обработку объектов на CPU и GPU процессорах. На вход подается сцена, включающая множество объектов, которые представлены в виде треугольных сеток с заданным положением и материалом. Применяется структура пространственного разбиения для сокращения затрат при выполнении отсечений (frustum culling) и аппаратных проверок видимости (hardware occlusion queries). На первом этапе выполняется отбор узлов структуры, которые попадают в область видимости камеры. Затем на центральном процессоре проводится запись и отправка команд рендеринга. Команда рендеринга содержит информацию о конфигурации конвейера GPU процессора и смещение в памяти, по которому хранятся вершины (индексы вершин) объекта. Команды посылаются по

шине на GPU процессор. Принципы работы графического конвейера описаны в разделе 1.1.

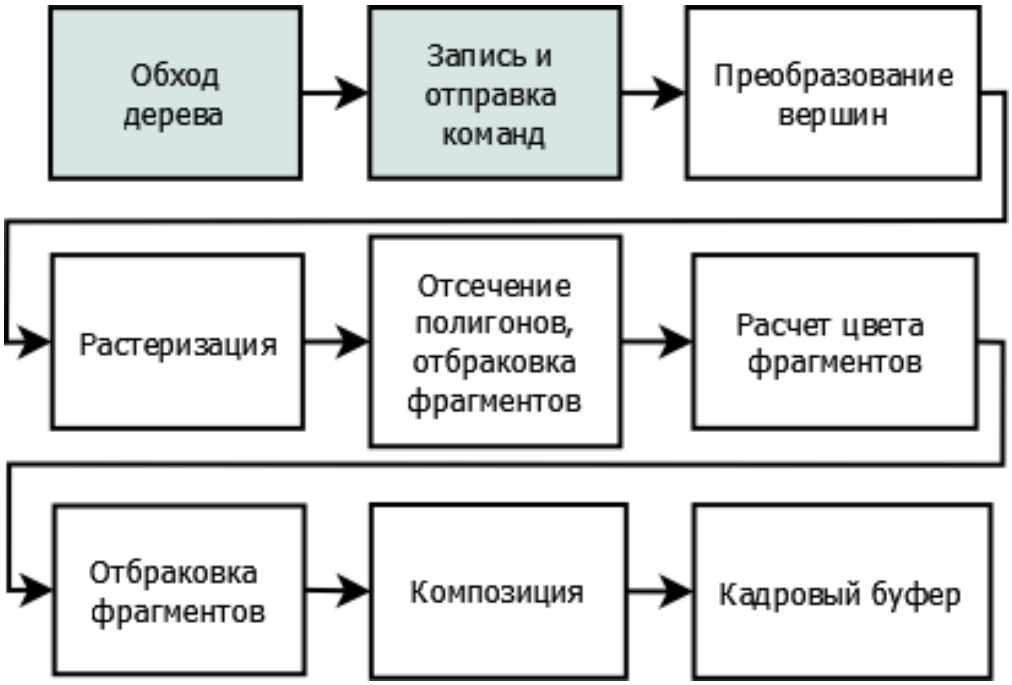

Рисунок 2.1 — Основные этапы рендеринга объектов сцены. Первые два этапа выполняются на центральном процессоре. Остальные этапы выполняются на GPU процессоре.

2.2 Исследуемая схема рендеринга

В данной работе рассматривается прямая однопроходная схема рендеринга динамических трехмерных сцен. Это означает, что записывается одна команда рендеринга для каждого объекта сцены, командный буфер составляется на центральном процессоре, осуществляется один проход по графическим элементам для генерации кадрового буфера. Следующие шаги описывают рассматриваемые в данной работе способы реализации процессов рендеринга:

1. Изменение видимости, положений, материалов объектов сцены при изменении анимационного времени t_{anim}.
2. Получение результатов аппаратных проверок видимости.
3. Ожидание готовности командного буфера.
4. Обход структуры пространственного разбиения и выполнение отсечений.
5. Запись команд рендеринга для отдельных объектов сцены.
6. Запись ссылок на вспомогательные буфера в главный командный буфер (более подробно этот метод составления командных буферов рассмотрен в разделе 3.1).
7. Запись буфера для выполнения аппаратных проверок видимости.

8. Отправка командного буфера на выполнение.

Шаги 1, 3, 8 выполняются для всех способов. Запись команд рендеринга может проводиться напрямую в главный командный буфер (шаг 5) или может выполняться запись командных буферов для групп объектов сцены (шаг 6). Для ускорения рендеринга может использоваться структура пространственного разбиения H (шаг 4), могут выполняться аппаратные проверки видимости (шаги 2, 7, 8).

2.3 Описание динамической сцены

Пусть кортеж S содержит описание динамической сцены:

$$S =< Mesh, MeshInst, Mat, Tex, Idx, Vert, TC, Norm, Vp, t_{anim}, TrnTl, VisTl, MatTl, batchSize, visFrags, d_{anim} >, \quad (2.1)$$

где $Mesh$ — множество треугольных сеток,

$MeshInst$ — множество объектов сцены, которые имеют заданное положение, материал, видимость,

Mat — множество материалов,

Tex — множество текстур,

Idx — множество индексов, которое задается в случае использования проиндексированных множеств вершин, нормалей, текстурных координат,

$Vert$ — множество вершин,

TC — множество текстурных координат,

$Norm$ — множество нормалей,

Vp — текущее положение камеры (viewpoint), которое задается с помощью матрицы размерности 4×4,

t_{anim} — время анимации, для которого определяются характеристики объектов сцены,

$TrnTl$ — множество временных событий, которые задают положение объектов сцены,

$VisTl$ — множество временных событий, которые задают видимость объектов сцены,

$MatTl$ — множество временных событий, которые задают материалы объектов сцены.

Также в кортеж S добавлены переменные, которые зависят от положения камеры Vp и анимационного времени t_{anim}:

$batchSize$ — среднее количество вершин в видимых объектах сцены во время t_{anim},

$visFrags$ — суммарное количество фрагментов видимых объектов сцены во время t_{anim} (принцип подсчета с помощью процедуры $CalcNumFrags$ описан ниже),

d_{anim} — средняя доля объектов сцены, свойства которых изменились за последние несколько кадров.

Кортеж S задает описание всех объектов сцены (видимых и невидимых). Для сцен с объектами, прошедшими проверки видимости, в момент времени t_{anim} при положении камеры Vp будем использовать обозначение S_{vis}, если не оговорено иначе. Под проверками видимости подразумевается метод удаления объектов, не попадающих в область видимости камеры, и метод выполнения аппаратных проверок видимости ограничивающих параллелепипедов AABB узлов дерева относительно буфера глубины (hardware occlusion query).

2.4 Оценка потребляемой памяти

Данные для выполнения рендеринга записываются в память. При использовании разных способов рендеринга расходуется различный объем памяти. Предварительная запись командных буферов, использование дополнительных уровней детализации объектов (levels of detail) повышают объем расходуемой памяти. Если памяти не хватает, то, как правило, выполняется завершение программы. Чтобы убедиться, что памяти достаточно для выполнения рендеринга, выведем формулы для вычисления требуемого объема памяти. Объем памяти, который занимает полигональное представление сцены, выражается с помощью формулы:

$$M_{geom}(S, HW) = |\mathrm{Vert}|\mathrm{M}(vert_1, HW) + |\mathrm{Idx}|\mathrm{M}(idx_1, HW) + |\mathrm{TC}|\mathrm{M}(tc_1, HW) + |\mathrm{Norm}|\mathrm{M}(norm_1, HW), \quad (2.2)$$

где HW — конфигурация вычислительной системы (CPU и GPU),

$|\mathrm{Vert}|$, $|\mathrm{Idx}|$, $|\mathrm{TC}|$, $|\mathrm{Norm}|$ — количества вершин, индексов, текстурных координат, нормалей,

$\mathrm{M}(a, HW)$ — объем памяти, который занимает элемент a.

Объем памяти, который занимают текстуры (кадровые буфера), выражается с помощью формулы:

$$M_{tex}(\mathrm{S}, \mathrm{HW}) = \sum_{i=1}^{|Tex|} W(tex_i)H(tex_i)BPP(tex_i, HW), \quad (2.3)$$

где $W(tex)$ — ширина текстуры tex,

$H(tex)$ — высота текстуры tex,

$BPP(tex)$ — объем памяти, который занимает один пиксель текстуры tex.

Объем памяти, который занимают буфера рендеринга с матрицами и материалами, выражается с помощью формулы:

$$M_{uniforms}(\mathrm{S}, \mathrm{HW}) = \sum_{i=1}^{|MeshInst|} M_{ubo}(Mat(inst_i), HW), \quad (2.4)$$

где $M_{ubo}(mat, HW)$ — объем памяти, который занимает буфер, хранящий значения переменных для шейдера, соответствующего данному материалу,

$Mat(inst)$ — материал объекта сцены $inst$.

В процессе рендеринга записываются буфера с командами рендеринга. Каждая команда содержит информацию для отображения объекта сцены с заданным положением и материалом. Объем памяти, который занимает буфер, содержащий команды для рендеринга объектов сцены, выражается с помощью формулы:

$$M_{cmds}(\mathrm{S}, \mathrm{HW}) = \sum_{i=1}^{|MeshInst|} M_{cmd}(Mat(inst_i), HW), \quad (2.5)$$

где $M_{cmd}(mat, HW)$ — объем памяти, который занимает команда рендеринга объекта сцены с материалом mat.

Пусть множество объектов сцены S разбито на подмножества $\mathrm{G} = \{g_i : \{inst_{i_1}, \dots, inst_{i_n}\}\}$. Будем считать, что для каждого подмножества g_i записывается отдельный командный буфер (secondary command buffer). Минимальный объем памяти, который занимает командный буфер, составляет $M_{\mathrm{sec\_buf}}(HW)$. Команды рендеринга подмножества g_i могут занимать больший объем памяти. В этом случае производится расширение командного буфера. Объем памяти, который занимают буфера, содержащие команды рендеринга для подмножеств g_i, выражается с помощью формулы:

$$M_{grouped\_cmds}(\mathrm{G}, \mathrm{HW}) = \sum_{i=1}^{|G|} \max\left(M_{\mathrm{sec\_buf}}(HW), M_{cmds}(g_i, \mathrm{HW})\right). \quad (2.6)$$

Для выполнения рендеринга необходимо задать конфигурацию графического конвейера, которая используется при отображении объекта сцены с заданным материалом. Зачастую используется одна конфигурация конвейера на материал. Конфигурации всех конвейеров, как правило, составляются заранее и хранятся в памяти для сокращения затрат в процессе рендеринга. Объем памяти, который занимают конфигурации графических конвейеров программного интерфейса, выражается с помощью формулы:

$$M_{rendering}(\mathrm{S},\mathrm{HW}) = \sum_{i=1}^{|Mat|} \mathrm{M}_{pipeline}(m_i, HW)\,, \tag{2.7}$$

где $\mathrm{M}_{pipeline}(m_i, HW)$ — объем памяти, который требуется для хранения описания конфигурации конвейера для вывода объектов сцены с материалом m_i.

Объем памяти, который требуется для хранения описаний объектов сцены, выражается с помощью формулы:

$$M_{instances}(\mathrm{S},\mathrm{HW}) = |MeshInst|M(inst_1, HW) + |Mat|M(mat_1, HW), \tag{2.8}$$

где $|MeshInst|$, $|Mat|$ — количества объектов сцены, материалов,

$M(inst, HW)$ — объем памяти, который занимает объект сцены при использовании вычислительной системы HW,

$M(mat, HW)$ — объем памяти, который занимает материал при использовании вычислительной системы HW.

При выполнении рендеринга может использоваться структура пространственного разбиения для кластеризации объектов сцены. Объем памяти, который требуется для хранения пространственного индекса, выражается с помощью формулы:

$$M_{hierarchy}(\mathrm{H},\mathrm{HW}) = |\mathrm{H}|M(n_1, HW), \tag{2.9}$$

где H — множество узлов структуры пространственного разбиения (дерева),

$M(n, HW)$ — объем памяти, который занимает узел структуры пространственного разбиения при использовании вычислительной системы HW.

Зачастую геометрия сцены, текстуры и буфера с матрицами и описанием материалов ($M_{geom}(S, HW)$, $M_{tex}(S, HW)$, $M_{uniforms}(S, HW)$) хранятся в видеопамяти. Остальные данные ($M_{cmds}(S, HW)$, $M_{grouped\_cmds}(G, HW)$, $M_{rendering}(S, HW)$, $M_{instances}(S, HW)$, $M_{hierarchy}(H, HW)$) хранятся в основной памяти (RAM). Когда недостаточно видеопамяти, можно выполнить запись в основную память. Таким образом, на этапе подготовки к рендерингу определяется доступная память и проводится запись данных либо в видеопамять, либо в основную память.

2.5 Оценка времени рендеринга

Графический конвейер выполняет работу для выявления треугольников, которые не попадают в область видимости камеры. Для каждой вершины треугольников выполняется преобразование для перевода в плоскость изображения. Только после растеризации выполняется отсечение треугольников, которые не попали в

область видимости камеры. Таким образом, расходуются лишние вычислительные ресурсы на отправку команд и преобразование вершин треугольников, которые не выводятся на экран. Подобная ситуация возникает и при отображении сложных сцен с большим количеством перекрывающихся поверхностей. Для определения видимости фрагментов используется метод z-буфера. Однако до применения метода z-буфера необходимо выполнить преобразование вершин, растеризацию, расчет цвета фрагментов. Таким образом, часть ресурсов тратится на обработку фрагментов, которые отсекаются методом z-буфера и в конечном итоге не попадают в кадровый буфер. Эта проблема особенно проявляется в сценах с большим количеством перекрывающихся поверхностей. Объем вычислений можно сократить с помощью методов удаления невидимых поверхностей, но они не всегда эффективны. Актуальной является разработка модели производительности рендеринга, в рамках которой можно оценить время выполнения различных этапов рендеринга. Это позволит определить сценарии, в которых методы удаления невидимых поверхностей являются эффективными.

2.5.1 Время выполнения рендеринга на CPU и GPU процессорах

Рассмотрим формулы для оценки времени выполнения приведенных способов реализации процессов рендеринга. Время выполнения рендеринга на центральном процессоре выражается с помощью формулы:

$$T_{CPU}\big(S, S_{vis}, H, H_{vis}, H_{upd}, H_q, h, g, q, HW\big) = T_{update}(S, HW) + [h = 1]T_{update}(H, HW) + [h = 0]T_{fc}(S, HW) + [h = 1](T_{traverse}(H, HW) + T_{fc}(H, HW)) + [g = 0]T_{buf}(S_{vis}, HW) + [g = 1](T_{buf}\big(H_{upd}, HW\big) + T_{\mathrm{sec\_}buf}(H_{vis}, HW)) + [q = 1](T_{buf}\big(H_q, HW\big) + T_{recv\_query}\big(H_q, HW\big)), \quad (2.10)$$

где S_{vis} — кортеж с информацией об объектах сцены, которые прошли проверки видимости и находятся в области видимости камеры,

H_{vis} — множество узлов структуры пространственного разбиения в области видимости камеры,

H_{upd} — множество узлов структуры пространственного разбиения, в которых произошли изменения после движения объектов сцены,

H_q — множество узлов, для которых выполняются аппаратные проверки видимости,

q — параметр, который определяет, используются ли проверки видимости,

$[q = 1]$ — выражение, которое равняется 1, если $q = 1$ (скобка Айверсона),

h — параметр, который определяет, используется ли структура пространственного разбиения H,

g — параметр, который определяет, используется ли составление и отправка вспомогательных буферов (secondary command buffers) для узлов структуры пространственного разбиения H,

$T_{update}(S, HW)$ — время изменения свойств объектов сцены с помощью структур $TrnTl$, $VisTl$, $MatTl$ при изменении анимационного времени,

$T_{update}(H, HW)$ — время обновления пространственного индекса при изменении анимационного времени,

$T_{fc}(H, HW)$ — время выполнения отсечений узлов, не попадающих в область видимости камеры,

$T_{traverse}(H, HW)$ — время обхода структуры пространственного разбиения H,

$T_{buf}(S, HW)$ — суммарное время составления и отправки командных буферов со всеми объектами сцены S,

$T_{\text{sec\_}buf}(H, HW)$ — суммарное время отправки вспомогательных командных буферов для узлов структуры пространственного разбиения H,

$T_{recv\_query}(H, HW)$ — время получения всех результатов проверок видимости.

Время выполнения рендеринга на GPU процессоре выражается с помощью формулы:

$$T_{GPU}(S, H_q, q, HW) = \max\left(T_{state}(S, HW), T_{vert}(S, HW), T_{frag}(S, Vp, HW)\right) + [q = 1]\max\left(T_{state}(H_q, HW), T_{vert}(H_q, HW), T_{frag}(H_q, Vp, HW), T_{exec\_query}(H_q, HW)\right), \quad (2.11)$$

где $T_{state}(S, HW)$ — суммарное время установки состояний конвейера для всех материалов, используемых при отображении сцены S,

$T_{vert}(S, HW)$ — суммарное время преобразования вершин объектов сцены S,

$T_{frag}(S, Vp, HW)$ — суммарное время обработки фрагментов объектов сцены S,

$T_{exec\_query}(H, HW)$ — время выполнения проверок видимости узлов структуры пространственного разбиения H.

Итоговое время рендеринга сцены S при использовании структуры пространственного разбиения H выражается с помощью формулы:

$$T\left(S, S_{vis}, H, H_{vis}, H_{upd}, H_q, h, g, q, HW\right) =$$
$$\begin{cases} T_{CPU}\left(S, S_{vis}, H, H_{vis}, H_{upd}, H_q, h, g, q, HW\right) + T_{GPU}\left(S_{vis}, H_q, q, HW\right), \text{один гл. ком. буфер} \\ \max\left(T_{CPU}\left(S, S_{vis}, H, H_{vis}, H_{upd}, H_q, h, g, q, HW\right), T_{GPU}\left(S_{vis}, H_q, q, HW\right)\right), \text{иначе.} \end{cases} \quad (2.12)$$

В формуле (2.12) выполняется расчет времени рендеринга для двух сценариев. В первом сценарии используется один главный командный буфер (при этом может использоваться много вспомогательных буферов). Запись команд возможна только тогда, когда чтение буфера завершено на GPU процессоре. Это означает, что необходимо выполнять синхронизацию доступа к буферу, при этом работа на CPU и GPU будет выполняться последовательно. Во втором сценарии используется несколько командных буферов и выполняется параллельная работа над разными буферами (рисунок 2.2).

Для вычисления затрат при использовании различных способов реализации процессов рендеринга в приведенные формулы добавлены параметры h, g, q. Если $h = 0, g = 0, q = 0$, получаем формулу для расчета времени рендеринга без использования структуры пространственного разбиения, без записи команд рендеринга для узлов дерева (фрагментации), без проверок видимости узлов дерева. Если $h = 1, g = 0, q = 0$, получаем формулу для расчета времени рендеринга с использованием структуры пространственного разбиения для выполнения отсечений, без записи команд рендеринга для узлов дерева, без проверок видимости узлов дерева. Если $h = 1, g = 1, q = 0$, получаем формулу для расчета времени рендеринга с использованием структуры пространственного разбиения для выполнения отсечений, с составлением команд рендеринга для узлов дерева, без проверок видимости узлов дерева. Если $h = 1, g = 1, q = 1$, получаем формулу для расчета времени рендеринга с использованием структуры пространственного разбиения для выполнения отсечений, с составлением команд рендеринга для узлов дерева, с проверками видимости узлов дерева.

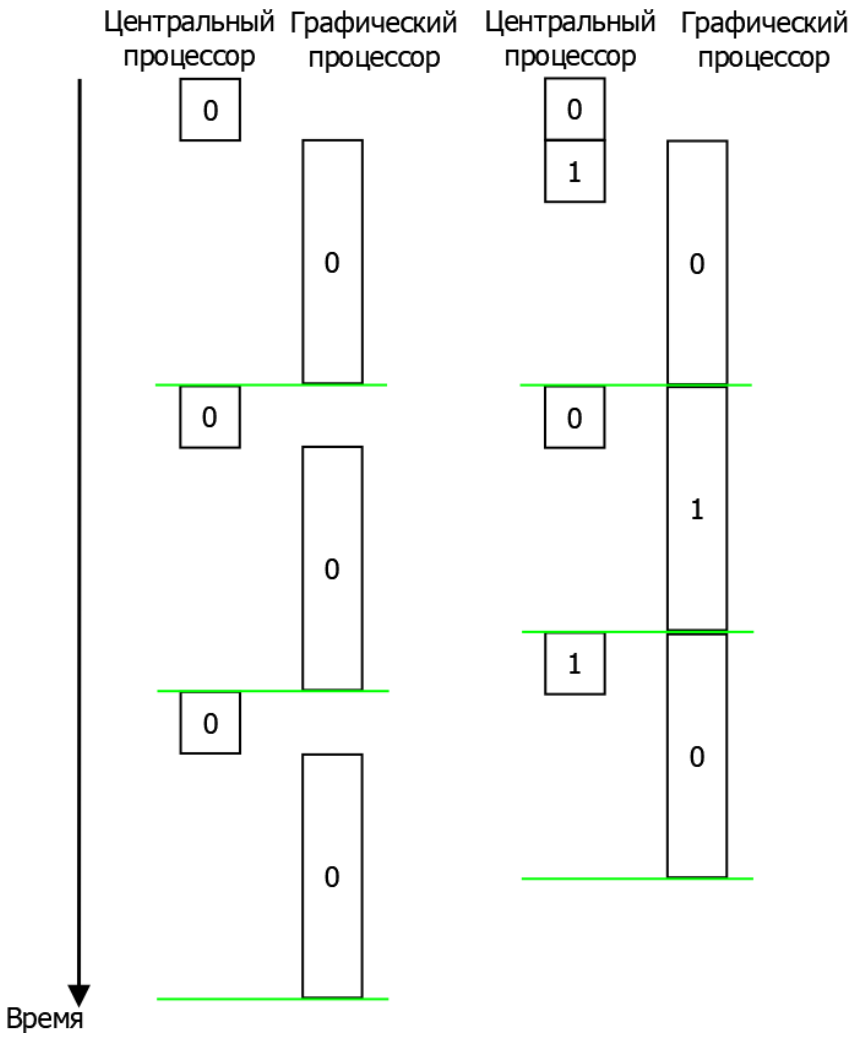

Рисунок 2.2 — Синхронизация доступа к главным командным буферам. Номерами помечены командные буфера. На CPU производится запись, на GPU — выполнение. Использование одного главного буфера (слева). Использование двух главных буферов для параллелизации записи и выполнения буфера команд (справа). Зеленая линия показывает момент, когда доступен для записи следующий командный буфер.

2.5.2 Время выполнения этапов графического конвейера

Оценим время выполнения различных этапов рендеринга. Время изменений свойств объектов сцены с помощью структур $TrnTl, VisTl, MatTl$ зависит от количества изменений. Время изменений вычисляется по формуле:

$$T_{update}(S, HW) = d_{anim}|MeshInst|t_{update}(HW), \quad (2.13)$$

где $t_{update}(HW)$ — среднее время записи свойств (матрицы преобразования, материала, видимости) для одного объекта сцены. Оно также включает время обращения к памяти UBO, в которой хранятся значения переменных шейдерных программ.

Предположим, что структура пространственного разбиения построена (узлы созданы и не меняются при добавлении объектов). Тогда время обновления дерева вычисляется по формуле:

$$T_{update}(H, HW) = d'_{anim}\ |MeshInst|\ height\ t_{intersect}(HW),$$

где $t_{intersect}(HW)$ — время вычисления узла для дальнейшего обхода дерева,

$height$ — высота дерева,

d'_{anim} — средняя доля объектов сцены с изменившимся положением за последние несколько кадров.

Время обхода структуры пространственного разбиения вычисляется по формуле:

$$T_{traverse}(H, HW) = |H| t_{traverse}(HW), \quad (2.14)$$

где $t_{traverse}(HW)$ — время обхода одного узла (обращение к памяти узла) на данной вычислительной системе.

Время выполнения отсечений узлов структуры пространственного разбиения вычисляется по формуле:

$$T_{fc}(H, HW) = |H| t_{fc}(HW), \quad (2.15)$$

где $t_{fc}(HW)$ — время отсечения одного узла на данной вычислительной системе.

Время записи командного буфера для всех объектов сцены вычисляется по формуле:

$$T_{buf}(MeshInst, HW) = |MeshInst| t_{buf}(HW), \quad (2.16)$$

где $t_{buf}(HW)$ — время записи и отправки одной команды на данной вычислительной системе.

Время отправки вспомогательных командных буферов вычисляется по формуле:

$$T_{\text{sec\_buf}}(H, HW) = |H| t_{\text{sec\_buf}}(numInstPerNode, HW), \quad (2.17)$$

где $t_{\text{sec\_buf}}(numInstPerNode, HW)$ — время отправки одного вспомогательного командного буфера с количеством команд $numInstPerNode$.

Времена выполнения этапов конвейера GPU вычисляются по формулам:

$$T_{state}(S, HW) = |Mat| t_{state}(HW), \quad (2.18)$$

$$T_{vert}(S, HW) =$$

$$\sum_{i=1}^{|MeshInst|} NumVerts(inst_i) t_{vert}(Mat(inst_i), batchSize, |MeshInst|, HW), \quad (2.19)$$

$$T_{frag}(S, Vp, HW) =$$

$$\sum_{i=1}^{|MeshInst|} NumFrags(inst_i, Vp) t_{frag}(Mat(inst_i), visFrags, HW), \quad (2.20)$$

$$T_{exec\_query}(H, HW) = |H| t_{exec\_query}(HW), \quad (2.21)$$

$$T_{recv\_query}(H, HW) = |H| t_{recv\_query}(HW), \quad (2.22)$$

где $t_{state}(HW)$ — время установки состояния графического конвейера на данной вычислительной системе,

$t_{vert}(mat, batchSize, numObjects, HW)$ — время работы вершинного шейдера данного материала с данным количеством вершин в объекте $batchSize$ и количеством объектов в сцене $numObjects$ (важность этих параметров объясняется ниже),

$NumVerts(inst)$ — количество вершин в объекте $inst$,

$t_{frag}(mat, visFrags, HW)$ — время работы фрагментного шейдера данного материала при обработке одного фрагмента (важность параметра $visFrags$ объясняется ниже),

$NumFrags(inst, Vp)$ — количество фрагментов объекта, которые видимы при данном положении камеры,

$t_{exec\_query}(HW)$ — время выполнения проверки видимости на этой вычислительной системе,

$t_{recv\_query}(HW)$ — время получения результата проверки видимости на данной вычислительной системе.

2.5.3 Вычисление покрываемого объектом количества фрагментов

Количество фрагментов, которые объект (узел дерева) занимает на экране, можно определить путем проекции его ограничивающего параллелепипеда AABB на экран. Количество видимых фрагментов рассчитывается по формуле:

$$NumFrags = w\,h \left(\frac{1}{\sqrt{w/h}\; 2\, tg\left(fov/2\right)} \right)^2 \frac{bb_x bb_y}{d^2}, \quad (2.23)$$

где w — ширина экрана,

h — высота экрана,

fov — вертикальный угол обзора,

bb_x — ширина видимой стороны ограничивающего параллелепипеда AABB объекта сцены,

bb_y — высота видимой стороны ограничивающего параллелепипеда AABB объекта сцены,

d — расстояние от камеры до ближайшей грани ограничивающего параллелепипеда AABB объекта сцены.

Вместо произведения $bb_x bb_y$ можно использовать среднюю площадь грани $A_{bb}/_6$. Псевдокод для вычисления $NumFrags(inst, Vp)$:

```
procedure CalcNumFrags(BoundingBox bb,
  Float occupiedArea,
  Integer w, Integer h,
  FloatArray camPos, FloatArray camFw, Float camFOV)
begin
  Float tanHalfFOV ← tan(camFOV / 2.0)
  Float screenRatio ← powf(1.0 / (sqrt(w / h) × 2 × tanHalfFOV), 2)
  Float d ← GetDistFromCam(camPos, camFw, bb)
  Float bbArea ← CalcArea(bb)/6.0
  Float coverage ← screenRatio × bbArea / ( d × d )
  coverage ← coverage × occupiedArea
  return coverage × w × h
end
```

Процедура принимает параметры, задающие ограничивающий параллелепипед AABB, размер экрана, свойства камеры. Параметр *occupiedArea* задает долю непустой площади объектов внутри узла. Если процедура применяется для объекта, то *occupiedArea* равняется 1. Значение переменной *screenRatio* можно вычислить один раз для кадра и сохранить для дальнейших расчетов. Отметим, что в случае выполнения сглаживания методом Supersampling передаются значения w, h с учетом числа выборок на пиксель. Если применяется сглаживание методом Multisampling, то количество обрабатываемых фрагментов не увеличивается [14].

2.5.4 Генерация тестовых наборов для вычисления параметров модели

Оценим скорость работы вычислительной системы HW при выполнении этапов рендеринга. Современные видеокарты содержат большое количество потоковых процессоров, которые выполняют параллельную обработку данных. Итоговое время вычислений зависит от объема работы (количества вершин, треугольников и объектов в сцене).

Производится генерация трехмерных сцен (см. раздел 3.4) с различными параметрами:

- количество вершин, приходящихся на объект ($batchSize$),
- количество объектов ($numObjects$),
- материал.

Параметр $batchSize$ имеет значение для производительности, потому что при выполнении рендеринга производится установка значений переменных для каждого объекта (uniform variables). Частое изменение этих переменных ухудшает производительность рендеринга. Произведение параметров $batchSize * numObjects$ определяет объем работы (суммарное количество вершин).

Пусть характерный размер объекта равняется a. Тогда длина, ширина и высота генерируемой сцены вычисляются, как $\sqrt[3]{numObjects}\ a$. Объекты равномерно заполняют весь объем сцены. Используется один материал для всех объектов. Предлагается выполнять генерацию таких сцен при увеличении значений параметров $batchSize$, $numObjects$ с некоторым шагом до тех пор, пока выполняются условия:

- Геометрия сцены помещается в доступную видеопамять.
- Командный буфер помещается в доступную основную память.

В результате получим множество тестовых трехмерных сцен, которое используется для оценки $t_{vert}(Mat(inst_i), batchSize, |MeshInst|, HW)$, $t_{buf}(HW)$. Производится рендеринг полученных сцен с таким положением камеры, когда все объекты попадают в область видимости камеры, с записью команд рендеринга объектов в порядке удаления от камеры. Время преобразования вершины тестовой сцены вычисляется, как отношение времени работы графического конвейера к количеству вершин в сцене.

Пусть дана некоторая трехмерная сцена, для которой необходимо выполнить оценку времени выполнения этапов графического конвейера. Выберем тестовую

сцену с наиболее близкими значениями параметров $batchSize$, $numObjects$. Сначала вычисляется среднее значение $batchSize$ для входной сцены, берутся тестовые сцены с наиболее близким значением $batchSize$. Затем среди них выбирается сцена с наиболее близким значением $numObjects$. В качестве значения $t_{vert}(Mat(inst_i), batchSize, |MeshInst|, HW)$ берется время обработки вершины при выполнении рендеринга выбранной тестовой сцены. В качестве значения $t_{buf}(HW)$ берется время записи в главный командный буфер и отправки одной команды при выполнении рендеринга выбранной тестовой сцены.

Для оценки $t_{exec\_query}(HW)$, $t_{recv\_query}(HW)$, $t_{\mathrm{sec}\_buf}(numInstPerNode, HW)$, $t_{traverse}(HW)$ и $t_{fc}(HW)$ можно использовать тестовую сцену с большим количеством объектов (для усреднения результатов). Также можно взять тестовую сцену с близким значением параметров $batchSize$, $numObjects$. Производится создание структуры пространственного разбиения (построение дерева, используемого в данной работе, описано в разделе 3.1). Производится рендеринг сцены и выполнение проверок видимости для узлов структуры пространственного разбиения. Осуществляется измерение времени работы графического конвейера при выполнении аппаратных проверок видимости. Тогда в качестве значения $t_{exec\_query}(HW)$ берется время одной проверки видимости. В качестве значения $t_{recv\_query}(HW)$ берется время получения результата одной проверки видимости. В качестве значения $t_{\mathrm{sec}\_buf}(numInstPerNode, HW)$ берется время отправки одного вспомогательного командного буфера с количеством команд $numInstPerNode$. В качестве значения $t_{traverse}(HW)$ берется время обхода структуры пространственного разбиения, поделенное на количество узлов. В качестве значения $t_{fc}(HW)$ берется время отсечений узлов структуры пространственного разбиения, поделенное на количество узлов.

Наиболее распространенное разрешение экрана современных мониторов 1920×1080 содержит 2073600 пикселей. Полноэкранный кадровый буфер содержит миллионы фрагментов, если объекты занимают весь экран. Таким образом, для оценки $t_{frag}(mat, visFrags, HW)$ достаточно произвести вывод пары треугольников на весь экран и измерить время работы конвейера GPU. Обозначим количество обработанных фрагментов при выполнении рендеринга сцены, как $visFrags$. Тогда время обработки фрагмента равняется времени работы конвейера, поделенному на значение $visFrags$. Также производится оценка времени обработки фрагмента в случаях, когда пара треугольников занимает только часть экрана: $3/4$, $1/2$, $1/4$,

1/8, 1/16. В качестве значения $t_{frag}(mat, visFrags, HW)$ для некоторой входной сцены берется время обработки фрагмента при выполнении рендеринга тестовой сцены с наиболее близким значением $visFrags$.

Для вычисления $t_{state}(HW)$ выполняется рендеринг одного треугольника, что вызывает задержку, связанную со сменой состояния конвейера. При этом количество обрабатываемых фрагментов треугольника должно быть незначительным ($NumFrags(inst, Vp) \leq 10$). Воспользуемся формулой (2.23) и получим расстояние от камеры, на котором выполняется это условие:

$$d \geq \frac{h}{2\, tg\left({}^{fov}/_{2}\right)} \sqrt{\frac{bb_x bb_y}{10}}. \tag{2.24}$$

Рассмотрим псевдокод для измерения времени выполнения этапов рендеринга. Процедура *MeasurePerfForMaterials* измеряет время выполнения рассматриваемых этапов рендеринга для всех материалов, используемых в сцене. Процедура *GenScenes* выполняет генерацию сцен для тестирования производительности с возрастающим количеством вершин и объектов с заданным материалом.

```
procedure MeasurePerfForMaterials(out Map materialToPerfMaps,
  out Scenes testScenes, MaterialArray matArray)
begin
  for i ← 0 to matArray.size()-1 do
    Material mat ← matArray[i]
    testScenes.common[mat] ← GenScenes(mat)
    testScenes.frag[mat] ← GenScenesForFrag(mat)
  done

  for i ← 0 to matArray.size()-1 do
    Material mat ← matArray[i]
    PerfMaps maps
    MeasurePerfForMaterial(maps, testScenes, mat)
    materialToPerfMaps[mat] ← maps
  done
 end
```

Процедура *MeasurePerfForMaterial* измеряет время выполнения этапов рендеринга для данного материала.

```
procedure MeasurePerfForMaterial(out PerfMaps maps,
  Scenes testScenes, Material mat)
begin
  MeasurePerfFc(maps.fc)
  MeasurePerfHierTraversal(maps.hierTraversal)
  MeasurePerfCmd(maps.cmd, testScenes.common[mat])
  MeasurePerfVert(maps.vert, testScenes.common[mat])
  MeasurePerfFrag(maps.frag, testScenes.frag[mat])
  MeasurePerfQuery(maps.recvQuery, res.execQuery,
          testScenes.common[mat])
  MeasurePerfSecondary(maps.secondary,
          testScenes.common[mat])
end
```

Процедура *MeasurePerfVert* производит измерение времени обработки вершин (вычислений в вершинном шейдере) для переданных тестовых сцен. Измерения проводятся большое количество раз, полученные результаты усредняются.

```
procedure MeasurePerfVert(out PerfMap infoToTimeMap,
  FileArray sceneFiles)
begin
  Integer i, j, k
  for k ← 0 to sceneFiles.size()-1 do
    String sceneFile ← sceneFiles[k]
    Scene scene
    LoadScene(scene, sceneFile)
    SceneInfo sceneInfo
    sceneInfo.numVerticesPerInst ← scene.allInstances[0].
                        mesh.drawCount
    sceneInfo.numVerticesTotal ← 0
    for i ← 0 to scene.numAllInstances-1 do
      MeshInstance inst ← scene.allInstances[i]
      sceneInfo.numVerticesTotal ← sceneInfo.numVerticesTotal +
                                   inst.mesh.drawCount
    done
```

```
Frustum f
BuildFrustum(f, scene)
VarsReset(InstancesRenderingTime)

Integer numRenderedFrames ← 100
for i ← 0 to numRenderedFrames-1 do
  Integer periodicFrameIdx ← GetPeriodicFrameIdx()
  WaitForRenderingFinish(periodicFrameIdx)

  CmdBuf cmdBuf
  GetCmdBuffer(cmdBuf, periodicFrameIdx)

  if i ≥ NumPrimBufs then
    Double instancesRenderingDuration ← GetPipelineDuration(
        TimeQueryObjStart, TimeQueryObjEnd, periodicFrameIdx)

    VarsAdd(InstancesRenderingTime, i,
              instancesRenderingDuration)
  endif

  for j ← 0 to scene.numAllInstances-1 do
    MeshInstance inst ← scene.allInstances[j]
    if IsInsideFrustum(inst.bb, f) then
      RecordCmd(cmdBuf, inst, scene.buffersDescr)
    endif
  done

  SubmitCommands(cmdBuf)
done

for i ← 0 to NumPrimBufs-1 do
  Integer periodicFrameIdx ← GetPeriodicFrameIdx()
  WaitForRenderingFinish(periodicFrameIdx)
done
```

```
    Double totalInstRenderTime ← VarsGetAvg(
                                        InstancesRenderingTime)
    Float vertRenderingTime ← totalInstRenderTime /
                                    sceneInfo.numVerticesTotal
    infoToTimeMap[sceneInfo] ← vertRenderingTime
    DeleteScene(scene)
  done
end
```

Аналогично измеряется время выполнения других вычислительных процессов:

- отсечение объектов с помощью метода Frustum Culling,
- обход структуры пространственного разбиения,
- запись буфера команд,
- обработка фрагментов,
- аппаратные проверки видимости,
- отправка вспомогательных командных буферов.

ГЛАВА 3. ПРЕДЛОЖЕННЫЕ МЕТОДЫ

В разделе 3.3 предложена адаптивная стратегия рендеринга динамических трехмерных сцен, реализующая выбор и настройку базовых методов удаления невидимых объектов, техник отложенных аппаратных проверок видимости и кэширования командных буферов на основе модели производительности графического конвейера непосредственно в процессе отображения сцены. В разделах 3.1, 3.2, 3.4 описаны вспомогательные техники и методы: техника составления и использования командных буферов совместно со структурами пространственного индексирования, метод оценки количества аппаратных проверок видимости, метод генерации тестовых наборов сцен.

3.1 Техника составления командных буферов

При работе с большими динамическими сценами следует обратить внимание на запись и отправку буферов с командами рендеринга. Этот этап может отнимать значительное время при отображении рассматриваемых сцен. В данном разделе рассматриваются принципы работы командных буферов. Предлагается техника составления и использования командных буферов совместно со структурами пространственного разбиения.

3.1.1 Цикл жизни командного буфера

Командные буфера содержат команды для установки состояния конвейера, динамического изменения состояния конвейера в процессе рендеринга, выполнения рендеринга, выполнения команд, находящихся в другом буфере. В каждом командном буфере записывается полное состояние конвейера, которое не зависит от других буферов. Командные буфера не могут унаследовать состояние конвейера, которое установил другой командный буфер. Благодаря этому GPU процессор может выполнять командные буфера в любом порядке.

Рассмотрим возможные состояния командных буферов на примере программного интерфейса Vulkan (рисунок 3.1). После создания и выделения памяти командный буфер переходит в исходное состояние. Далее его можно перевести в состояние для записи команд рендеринга. После записи всех команд командный буфер переходит в состояние готовности к выполнению. Затем выполняется отправка буфера в очередь команд для графического конвейера, командный буфер переходит в исполняемое состояние. После выполнения происходит переход обратно в состояние готовности к выполнению. Стоит обратить внимание, что командный буфер

можно выполнять неограниченное количество раз без необходимости перезаписи. Только в случае явного сброса командный буфер переходит в исходное состояние.

Рисунок 3.1 — Возможные состояния командного буфера и переходы между ними.

Отметим, что запись главного командного буфера и вычисления на GPU процессоре идут последовательно в случае использования одного главного командного буфера. Это связано с тем, что невозможно производить запись в командный буфер, если он выполняется на GPU процессоре.

3.1.2 Способы составления командных буферов

Рассматриваются две техники составления буферов: запись команд, запись ссылок на вспомогательные буфера. Первая техника заключается в записи команд рендеринга для каждого объекта. Вторая техника заключается в записи указателей на память, где хранятся предварительно записанные буфера. Разница заключается в различных временных затратах и требуемых объемах памяти. Предварительная подготовка буферов (кэширование) и запись главного буфера со ссылками на вспомогательные буфера происходит быстрее, но для хранения вспомогательных буферов может потребоваться дополнительная память (рисунок 3.2). Один вспомогательный буфер занимает, по крайней мере, одну страницу памяти. Таким образом,

использование большого количества вспомогательных буферов может оказаться неэффективным по памяти.

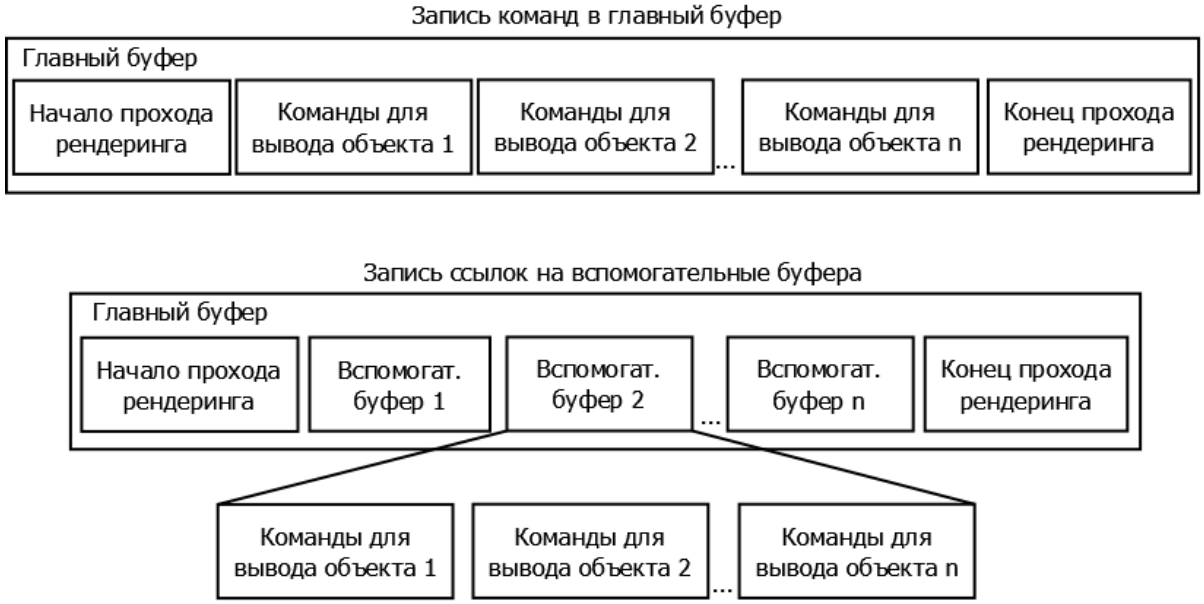

Рисунок 3.2 — Рассматриваемые техники составления буфера: запись команд (сверху), запись ссылок на вспомогательные буфера (снизу).

В разделе 5.4 приведены результаты тестирования производительности записи командных буферов. Показано, что для достижения наилучшей производительности необходимо выделять группы объектов и записывать вспомогательные буфера для этих групп.

3.1.3 Фрагментация и кэширование командных буферов совместно с использованием окто-дерева

При отображении сцен можно выполнять запись команд для рендеринга объектов сцены для каждого кадра без кэширования. Эта техника справляется с относительно маленьким количеством объектов в сцене. Однако при большом количестве объектов запись команд может отнимать значительное время. Предлагается техника составления и использования командных буферов совместно со структурами пространственного разбиения. Она позволяет сократить вычислительные затраты на запись командных буферов при выполнении рендеринга динамических сцен.

В данной работе в качестве структуры пространственного разбиения используется окто-дерево (single reference octree). Для каждого непустого узла создается буфер с командами рендеринга. При этом необходимо проверить, что $M_{grouped\_cmds}(\mathrm{H}, \mathrm{HW})$ укладывается в доступную память. При изменении свойств

объектов динамической сцены (видимость, материал) выполняется перезапись командных буферов для узлов, которые содержат данные объекты. При перемещении объектов также производится перезапись командных буферов для затронутых узлов. Сокращается время записи командных буферов, потому что перезапись выполняется не для всех объектов. Далее приводится псевдокод процедур для заполнения окто-дерева:

```
procedure AddItemToOctree(OctreeNode node, MeshInstance inst,
 Integer numInstPerNode, MeshInstanceArray allInstances,
 BitArray instIntersectingSecants,
 out Integer usedCmdMem, out Bool allocFailed)
begin
 Bool intersectsSecants ← NodeSecantsIntersectBb(node, inst.bb)
 if intersectsSecants then
  AddInst(node, inst, allocFailed)
  SetBit(instIntersectingSecants, inst.idx)
  return
 endif

 Bool hasSubnodes ← NodeHasSubnodes(node)
 Bool propagateInstancesToSubnodes ← hasSubnodes Or
      Size(node.instances) + 1 ≥ numInstPerNode + 1
 if not propagateInstancesToSubnodes then
  AddInst(node, inst, allocFailed)
  return
 endif

 if not hasSubnodes then
  Integer availMemory ← GetAvailMem() - usedCmdMem
  Integer cmdMemPerNode ← GetCmdMemPerNode(numInstPerNode)
  if 8 * cmdMemPerNode ≤ availMemory then
   usedCmdMem ← usedCmdMem + 8 * cmdMemPerNode
   CreateSubnodes(node, allocFailed)
   if allocFailed then return
  else
```

```
    allocFailed ← true
    return
   endif
  endif

  for nextInstIdx ∈ node.instances do
   If not IsBitSet(instIntersectingSecants, nextInstIdx) then
    RemoveItem(node.instances, nextInstIdx)
    PropagateItem(nextInstIdx, node, numInstPerNode,
        allInstances, instIntersectingSecants,
        usedCmdMem, allocFailed)
   endif
  done
  PropagateItem(inst.idx, node, numInstPerNode,
      allInstances, instIntersectingSecants,
      usedCmdMem, allocFailed)
end

procedure PropagateItem(Integer nextInstIdx, OctreeNode node,
 Integer numInstPerNode, MeshInstanceArray allInstances,
 BitArray instIntersectingSecants, out Integer usedCmdMem,
 out Bool allocFailed)
begin
 MeshInstance nextInst ← allInstances[nextInstIdx]
 Integer subnodeIdx ← GetSubnodeContainingBb(nextInst.bb, node)
 AddItemToOctree(node.subnodes[subnodeIdx], nextInst,
     numInstPerNode, allInstances, instIntersectingSecants,
     usedCmdMem, allocFailed)
end

procedure GetCmdMemPerNode(Integer numInstPerNode)
begin
 Integer cmdMem ← GetMemPerCmd()
 Integer secondaryBufMem ← GetMemPerSecondaryBuf()
```

```
  return max(secondaryBufMem, numInstPerNode × cmdMem)
end
```

Процедура *AddItemToOctree* производит добавление объекта *inst* в узел октодерева *node*. Параметр *numInstPerNode* задает допустимое количество объектов в узле. Параметр *allInstances* задает все объекты сцены (осуществляется доступ по индексу). Параметр *instIntersectingSecants* задает массив бит, который используется для определения, пересекается ли данный объект с секущими плоскостями узла. Параметр *usedCmdMem* задает объем используемой памяти для хранения командных буферов, которые не создаются в рамках этой процедуры. Отметим, что инициализация командных буферов для узлов проводится только после создания окто-дерева, расход памяти отслеживается с помощью параметра *usedCmdMem*. При выполнении процедуры задается значение параметра *allocFailed=true*, когда памяти недостаточно для хранения узлов дерева или командных буферов.

Если объект пересекается с секущими плоскостями данного узла, то он добавляется в этот узел, и процедура завершается. Иначе проводится проверка количества объектов в узле *node*. В случае достижения максимального количества объектов *numInstPerNode+1* выполняется заполнение дочерних узлов. Перед этим проверяется то, что доступен необходимый объем памяти для хранения восьми дочерних узлов и командных буферов с количеством команд *numInstPerNode*. Если памяти недостаточно, дочерние узлы не создаются, а объект добавляется в текущий узел. Если памяти достаточно, выполняется обход объектов узла *node*. Для объектов, которые не пересекаются с секущими плоскостями узла, выполняется процедура *PropagateItem*. Эта процедура вычисляет дочерний узел, который содержит данный объект, и вызывает процедуру *AddItemToOctree* для добавления этого объекта в дочерний узел.

Рассмотрим возможные сценарии при работе с памятью при заполнении октодерева. Выделение памяти может проводиться при вызове процедур *AddInst*, *CreateSubnodes*. В эти процедуры передается параметр *allocFailed*. Когда памяти недостаточно, проставляется значение параметра *true*. Также с помощью параметра *usedCmdMem* отслеживается расход памяти на командные буфера. Когда памяти недостаточно для хранения командных буферов, также устанавливается значение *allocFailed=true*. Тогда процедура завершается и в этом случае использование октодерева не допускается.

Количество узлов дерева зависит от значения параметра *numInstPerNode*. Рассмотрим формулы для оценки количества узлов в дереве в зависимости от *numInstPerNode*.

$$h = \log_8 \frac{|Inst|}{numInstPerNode}, \tag{3.1}$$

$$|H| = \sum_{i=0}^{\lfloor h \rfloor - 1} 8^i + \frac{|Inst|}{numInstPerNode}, \tag{3.2}$$

где h — предполагаемая высота окто-дерева при заданном количестве объектов в узле дерева,

$|H|$ — предполагаемое количество узлов окто-дерева.

Выведем условие того, что памяти достаточно для хранения окто-дерева:

$$|H|\left(M_{node} + \max\left(M_{\text{sec\_buf}}, numInstPerNode\ M_{cmd}\right)\right) \leq availMem. \tag{3.3}$$

Таким образом, чтобы уложиться в заданный объем доступной памяти, нужно задать начальное значение *numInstPerNode.* Затем изменять его и проверять условие (3.3) до тех пор, пока не будет выбрано подходящее значение *numInstPerNode*:

```
procedure GetNumInstPerNode(Integer numInst)
begin
 Integer numInstPerNode ← 5
 Bool enoughMemory ← false
 while not enoughMemory do
  Float h ← Log(8, numInst/numInstPerNode)

  Integer numNodes ← 0
  for Integer i from 0 to Floor(h)-1 do
   numNodes ← numNodes + Pow(8, i)
  done
  numNodes ← numNodes + numInst/numInstPerNode

  Integer availMemory ← GetAvailMem()
  Integer cmdMemPerNode ← GetCmdMemPerNode(numInstPerNode)
  enoughMemory ← (SizeOf(OctreeNode) + cmdMemPerNode) ×
          numNodes ≤ availMemory
  numInstPerNode++
```

```
 done
 numInstPerNode--
 return numInstPerNode
end
```

3.2 Метод оценки количества аппаратных проверок видимости для эффективного выполнения рендеринга

Аппаратные проверки видимости позволяют выполнить отбраковку большого количества невидимых треугольников, чтобы сократить время работы графического конвейера. Однако на выполнение аппаратных проверок видимости также расходуются вычислительные ресурсы. Необходимо оценивать эффективность аппаратных проверок видимости в процессе отображения динамической сцены.

В разделе 1.2 рассмотрены разные методы рендеринга с применением аппаратных проверок видимости [45; 46]. В них эффективность проверок видимости определяется с помощью оценок временных затрат на проверки видимости и вероятностных критериев. Проверка видимых узлов выполняется один раз для n кадров, используются групповые проверки видимости многих узлов. Таким образом, реакция на изменения в сцене отложена на некоторое количество кадров. Это не так критично для статических сцен, в которых высокая степень когерентности видимости объектов, но может ухудшить результаты для динамических сцен. Также эти методы рассчитывают на то, что на центральном процессоре есть свободные вычислительные ресурсы, которые можно потратить на определение эффективности проверок видимости. Дополнительные накладные расходы на сортировку узлов и поддержание структуры данных, которая является приближением z-буфера, не всегда оправданы.

В отличие от существующих методов предложенный в этом разделе метод предназначен для динамических сцен. В течение некоторого времени накапливаются данные о видимости объектов. Затем производится вычисление и адаптивное обновление количества проверок видимости в процессе отображения. Использование аналитических формул позволяет сократить затраты и получить релевантные оценки для текущего состояния сцены.

3.2.1 Отбор узлов окто-дерева

Рассмотрим типы узлов окто-дерева, для которых актуально выполнять проверки видимости. На рисунке 3.3 изображено окто-дерево, которое заполнено объектами сцены. В сцене есть преграда (один объект или множество объектов), за которой расположены невидимые объекты. На рисунке цветом выделены разные типы обрабатываемых узлов. Серым цветом выделены пустые узлы. Зеленым цветом выделены заполненные узлы дерева, находящиеся за преградой. Для таких узлов выполнение проверок видимости наиболее актуально. Синим цветом выделены заполненные граничные узлы, находящиеся за преградой. Они имеют меньший объем, содержат меньше объектов. Тем не менее, если выполнение проверки видимости вычислительно менее затратно, чем рендеринг объектов внутри узла, то желательно провести проверку видимости. Красным цветом выделены заполненные узлы, которые видимы. Хотя на проверки видимости красных узлов тратятся лишние ресурсы, есть также красные узлы, которые могут стать невидимыми при движении камеры или изменении свойств объектов сцены.

В связи с вышеизложенным предлагается проводить обход окто-дерева в ширину для приоритетного отбора октантов с наибольшим количеством объектов. При этом пропускаются узлы, для которых не выполняется критерий отбора. Предлагается собирать данные, задающие зависимость количества невидимых элементов в сцене от количества проверок видимости. Исходя из этого, выводится количество проверок видимости для эффективного рендеринга.

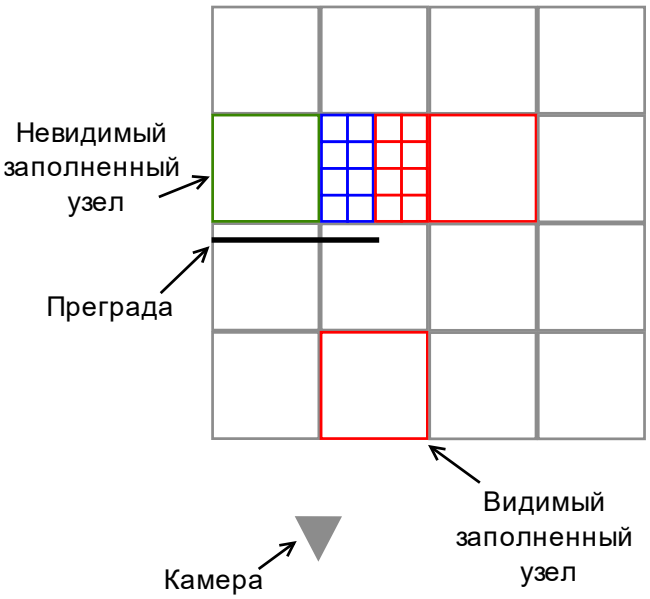

Рисунок 3.3 — Разные типы узлов окто-дерева при выполнении аппаратных проверок видимости.

В процессе рендеринга динамической сцены для каждого узла окто-дерева поддерживается количество вершин объектов в узле. Для родительских узлов количество вершин в узле суммируется с количеством вершин в дочерних узлах. Значение этой переменной обновляется при иерархической вставке объектов в дерево. При удалении объекта из дерева значение обновляется за счет обратных ссылок на родительские узлы.

Также в процессе рендеринга динамической сцены для каждого узла поддерживается доля занятой площади:

$$\text{occupiedArea}(node) = \min\left(1.0, \sum_{i=0}^{n} \frac{A_{bb}(inst_i)}{6}\right), \tag{3.4}$$

где n — количество объектов в узле $node$,

$A_{bb}(inst)$ — площадь ограничивающего параллелепипеда AABB объекта $inst$.

Узел считается затратным (heavy), если время рендеринга объектов в узле значительно выше времени выполнения проверки видимости (запись команды, выполнение проверки на GPU, получение результата). Количество вершин в узле поддерживается в актуальном состоянии. Количество фрагментов вычисляется по доли занимаемой площади $\text{occupiedArea}(node)$ с помощью процедуры $CalcNumFrags$. Таким образом, получаем условие вычислительной затратности узла:

$$t_{\text{sec\_buf}}(HW) + \max\left(t_{state}(HW), NumVerts(node)\hat{t}_{vert}, NumFrags(node, Vp)\hat{t}_{frag}\right) \geq \text{C}\left(t_{buf}(HW) + t_{exec\_query}(HW) + t_{recv\_query}(HW)\right), \tag{3.5}$$

где $\hat{t}_{vert}$ — среднее время преобразования вершины для всех материалов сцены,

$\hat{t}_{frag}$ — среднее время расчета цвета фрагмента для всех материалов сцены,

C — некоторая константа больше единицы.

Отметим, что в случае использования нескольких главных командных буферов работа на CPU и GPU выполняется параллельно, и в формуле (3.5) нужно использовать максимум времени работы на CPU и GPU.

3.2.2 Вычисление количества проверок видимости

Во время работы метода выполняются следующие шаги:

1. Отбор ограниченного количества $N_{queries}$ узлов дерева для выполнения проверок видимости. Постепенное накопление данных о видимости графических элементов и узлов дерева.

2. Вывод количества проверок видимости $N_{queries}^{efficient}$ для эффективного рендеринга на основе накопленных данных.
3. Отбор $N_{queries}^{efficient}$ узлов дерева для выполнения проверок видимости до тех пор, пока это выгодно (выполнение проверок позволяет сократить время рендеринга) и количество изменившихся в процессе анимации объектов незначительно.

Шаг 1 выполняется, когда еще нет информации о видимости объектов или состояние динамической сцены существенно изменилось, и требуется повторная оценка эффективности проверок видимости. На шаге 2 применяется модель производительности графического конвейера для оценки количества проверок видимости. Вычислительные затраты на шаге 2 пропорциональны размеру массива с информацией о количестве невидимых объектов ($O(N)$). Выполнение проверок видимости на шаге 3 используется для корректировки значения $N_{queries}^{efficient}$. Для этого выполняется шаг 2, используются только данные о видимости, полученные при выполнении рендеринга последнего кадра.

В течение определенного количества кадров N_{gather} производится накопление данных о видимости затратных узлов. Предполагается, что за это время количество невидимых объектов в сцене изменяется незначительно. Выполняется иерархический обход дерева в ширину, отбор затратных узлов, для которых проверки видимости еще не выполнялись. При этом количество проверок видимости $N_{queries}$ для каждого кадра ограничено, чтобы распределить вычислительные затраты на количество кадров N_{gather}. Когда накопление данных закончено, полученная информация используется для задания зависимости количества невидимых объектов от количества иерархических проверок видимости затратных узлов. Аналогично строятся зависимости для количества невидимых узлов, вершин и фрагментов от количества иерархических проверок видимости. Эта информация используется для определения эффективного количества проверок видимости при выполнении рендеринга сцены S в момент времени t_{anim}.

Определим количество проверок видимости для эффективного выполнения рендеринга с помощью предложенной модели производительности графического конвейера. Рассмотрим сценарий, когда этапы рендеринга на CPU и GPU выполняются последовательно, самым долгим этапом графического конвейера является преобразование вершин или расчет цвета фрагментов. Важны этапы рендеринга, объем работы на которых зависит от количества проверок видимости. Обход дерева

и отсечения методом Frustum Culling выполняются вне зависимости от количества проверок видимости, поэтому не участвуют в приведенных ниже расчетах. Тогда время рассматриваемых этапов рендеринга определяется по формуле:

$$T(x) = c(x)t_c + s(x)t_s + e(x)t_e, \quad (3.6)$$

где x — количество проверок видимости узлов структуры пространственного разбиения,

$c(x)$ — количество записываемых команд рендеринга,

$s(x)$ — количество отправляемых командных буферов,

$e(x)$ — количество графических элементов (вершин или фрагментов) на лимитирующем этапе конвейера,

$t_c = t_{buf}(HW)$ — время записи одной команды рендеринга,

$t_s = t_{\text{sec\_}buf}(numInstPerNode, HW)$ — время отправки вспомогательного командного буфера,

t_e — время выполнения лимитирующего этапа конвейера (преобразования вершины, расчета цвета фрагмента).

Накопленные данные о видимости узлов дерева позволяют определить функции $c(x)$, $s(x)$, $e(x)$. Количество невидимых узлов, объектов, вершин (фрагментов) монотонно растет с увеличением количества проверок видимости. Зачастую рост происходит быстрее при малых x, а затем останавливается при больших x (см. раздел 5.3). Приближенное описание таких данных задается с помощью логарифмической зависимости $c_1 + c_2 \ln(1 + x)$, где c_1 и c_2 — коэффициенты, которые вычисляются методом наименьших квадратов [90].

Накопив достаточное количество точек, определим коэффициенты регрессии и подставим в функции:

$$c(x) = \alpha_n(o_0 - o_1 - o_2 \ln(1 + x)) + x, \quad (3.7)$$

$$s(x) = (n_0 + 1 - s_1 - s_2 \ln(1 + x)), \quad (3.8)$$

$$e(x) = (e_0 + px - e_1 - e_2 \ln(1 + x)), \quad (3.9)$$

где α_n — средняя доля перезаписываемых командных буферов, которая определяется по динамике объектов сцены за последние несколько кадров,

o_0 — количество объектов в сцене на данный момент времени,

n_0 — количество непустых узлов в структуре пространственного разбиения,

e_0 — количество вершин (фрагментов) при выполнении рендеринга сцены,

p — количество вершин (фрагментов), приходящихся на один узел структуры пространственного разбиения,

o_1, o_2 — коэффициенты регрессии в зависимости количества невидимых объектов от количества проверок видимости,

s_1, s_2 — коэффициенты регрессии в зависимости количества невидимых узлов от количества проверок видимости,

e_1, e_2 — коэффициенты регрессии в зависимости количества невидимых вершин (фрагментов) от количества проверок видимости.

Определение минимума (3.6) вместе с ограничениями на количество проверок видимости $0 \le x \le n_0$ является задачей нелинейного программирования. Функция $T(x)$ является непрерывной и дифференцируемой на множестве ограничений, поэтому минимум является либо стационарной точкой, либо лежит на границе множества ограничений (теорема 14.1 [91]). Для нахождения стационарной точки решим уравнение $T'(x) = 0$ и получим:

$$x_1^* = \frac{\alpha_n o_2 t_c + s_2 t_s + e_2 t_e}{p t_e + t_c} - 1. \tag{3.10}$$

В случае выхода значения стационарной точки за пределы отрезка $[0, n_0]$ берется ближайшая точка на границе. Сравнив $T(0)$, $T(n_0)$, $T(x_1^*)$, найдем минимальное время выполнения рендеринга сцены и соответствующее количество проверок видимости x.

Рассмотрим сценарий с параллельной работой с буферами на CPU и GPU. Тогда время рендеринга определяется по формуле: $T(x) = \max\left(T_{CPU}(x), T_{GPU}(x)\right)$. Нужно определить минимум двух функций:

$$T_{CPU}(x) = c(x)t_c + s(x)t_s, \tag{3.11}$$

$$T_{GPU}(x) = e(x)t_e. \tag{3.12}$$

Уравнение $T'_{CPU}(x) = 0$ имеет решение: $x_2^* = \frac{\alpha_n o_2 t_c + s_2 t_s}{t_c} - 1$. Сравнив $T_{CPU}(0)$, $T_{CPU}(n_0)$, $T_{CPU}(x_2^*)$, найдем минимальное время выполнения рендеринга на CPU и соответствующее значение x_2. Уравнение $T'_{GPU}(x) = 0$ имеет решение: $x_3^* = \frac{e_2}{p} - 1$. Сравнив $T_{GPU}(0)$, $T_{GPU}(n_0)$, $T_{GPU}(x_3^*)$, найдем минимальное время выполнения рендеринга на GPU и соответствующее значение x_3. В случае пересечения $T_{CPU}(x)$ и $T_{GPU}(x)$ необходимо также использовать точки пересечения в качестве граничных точек. Найдем корни функции $T_{CPU}(x) - T_{GPU}(x)$ с помощью метода деления отрезка пополам [92], взяв в качестве начальных отрезков $[0, x_4^*]$, $[x_4^*, n_0]$, где $x_4^* = \frac{e_2 t_e - \alpha_n o_2 t_c - s_2 t_s}{p t_e - t_c} - 1$. Если функции пересекаются, то получим корни $x_{\mathrm{r}1}$, $x_{\mathrm{r}2}$. После этого сравним $T(0)$, $T(n_0)$, $T(x_2)$, $T(x_3)$, а также значения $T(x_{\mathrm{r}1})$, $T(x_{\mathrm{r}2})$ в случае

пересечения $T_{CPU}(x)$ и $T_{GPU}(x)$ на соответствующих отрезках, и возьмем количество проверок видимости x, которое соответствует минимуму $T(x)$.

Рассмотрим псевдокод предложенного метода. Процедура *RecordHeavyOcclQueries* выполняет запись команд с проверками видимости для затратных узлов окто-дерева. Параметр *cmdBuf* задает командный буфер, в который выполняется запись. Параметр *maxNumQueries* задает максимальное количество проверок видимости. Задается значение параметра *gradualCheck=true*, чтобы исключить узлы, видимость которых уже была проверена. Таким образом, производится постепенное накопление данных о видимости большинства узлов. Параметр *recordedNumQueries* задает количество записанных проверок видимости. Параметр *camBB* задает ограничивающий параллелепипед AABB камеры. Он используется, чтобы не выполнять проверку узлов, которые содержат камеру или пересекаются с камерой. Такие узлы считаются видимыми. Процедура *IsHeavy* выполняет проверку того, что рендеринг объектов в узле является затратным. Для узлов кэшируется результат процедуры *IsHeavy*, перерасчет выполняется в случае изменения количества объектов узла или свойств объектов узла.

```
procedure RecordHeavyOcclQueries(out CmdBuf cmdBuf,
 Octree octree, Bool gradualCheck, Frustum f, BoundingBox camBB,
 Integer maxNumQueries, out Integer recordedNumQueries)
begin
 Integer periodicFrameIdx ← GetPeriodicFrameIdx()
 recordedNumQueries ← 0
 OctreeNodeQueue queue
 Push(queue, octree.root)
 while Size(queue) ≠ 0 do
  OctreeNode node ← Pop(queue)
  Bool isHeavy ← IsHeavy(node)
  if isHeavy and
    not IsIntersecting(camBB, node.bb) and
    not IsContained(camBB, node.bb) and
    IsInsideFrustum(node.bb, f) and
    (not gradualCheck or not IsOcclStatDefinedForNode(node.idx)) then
     RecordOcclQueryCmd(cmdBuf, node.bbInst, octree.buffersDescr)
     node.isQueried[periodicFrameIdx] ← true
```

```
      recordedNumQueries ← recordedNumQueries + 1
      if recordedNumQueries ≥ maxNumQueries then break
    endif
    if isHeavy then Push(queue, node.subnodes)
  done
end
```

Процедура *CheckQueryResults* проводит получение результатов проверок видимости и подсчет количества невидимых узлов, объектов, вершин и пикселей, если задано значение параметра *createOcclStats=true*. Отметим, что процедура *GetNumVerts* возвращает количество вершин в данном узле без дочерних узлов. Параметр *maxNumQueries* позволяет ограничить количество проверок видимости, чтобы уложиться в заданное время. Полученные значения *numInvisNodes*, *numInvisibleObjects*, *numInvisibleVertices* и *numInvisibleFrags* используются для оценки эффективности проверок видимости.

```
procedure CheckQueryResults(Octree octree, Scene scene,
  Bool createOcclStats)
begin
  Integer numInvisibleNodes ← 0
  Integer numInvisibleObjects ← 0
  Integer numInvisibleVertices ← 0
  Integer numInvisibleFrags ← 0
  Integer periodicFrameIdx ← GetPeriodicFrameIdx()
  Integer screenWidth ← GetScreenWidth()
  Integer screenHeight ← GetScreenHeight()
  FloatArray camPos ← GetCamPos(scene)
  FloatArray camFw ← GetCamFw(scene)
  Float camFOV ← GetCamFov(scene)
  Integer numVisibleFragments
  OctreeNodeQueue queue
  Push(queue, octree.root)
  while Size(queue) ≠ 0 do
    OctreeNode node ← Pop(queue)
    if node.isQueried[periodicFrameIdx] then
```

```
    node.isQueried[periodicFrameIdx] ← false
    numVisibleFragments ← GetQueryResult(periodicFrameIdx,
                                node.idx)
    if numVisibleFragments = 0 then
     node.isOccluded[periodicFrameIdx] ← true
     if createOcclStats then
      Integer numObjects ← GetNumObjects(node)
      Integer numVerts ← GetNumVerts(node)
      numInvisibleNodes++
      numInvisibleObjects ← numInvisibleObjects + numObjects
      numInvisibleVertices ← numInvisibleVertices + numVerts
      numInvisibleFrags ← numInvisibleFrags + CalcNumFrags(node.bb,
        node.occupiedArea,
        screenWidth, screenHeight,
        camPos, camFw, camFOV)
     endif
    else
     node.isOccluded[periodicFrameIdx] ← false
     Push(queue, node.subnodes)
    endif
   endif
  done

  if createOcclStats then
   OcclStat occlStat
   occlStat.numInvisNodes ← numInvisibleNodes
   occlStat.numInvisibleObjects ← numInvisibleObjects
   occlStat.numInvisVertices ← numInvisibleVertices
   occlStat.numInvisibleFrags ← numInvisibleFrags
   AddOcclStat(occlStat)
  endif
end
```

Процедура *CalcNumQueries* вычисляет количество проверок видимости для эффективного рендеринга. Параметры *t_cmd*, *t_secondary* задают времена записи

команды, отправки буфера команд. Параметры *t_vert*, *t_frag* задают средние времена преобразования вершины, расчета цвета фрагмента для всех материалов сцены. Флаг *vertLimited* имеет значение true, если последние кадры самым затратным этапом графического конвейера был этап по преобразованию вершин. Параметр *numFragsForNode* задает среднее количество фрагментов, которое занимает на экране узел дерева. Параметр *alphaN* задает среднюю долю перезаписываемых командных буферов за последние кадры. Параметр *occlStats* содержит массив данных о количестве невидимых элементов сцены. Используется процедура *CalcRenderingTime*, которая возвращает время рендеринга, рассчитанное по формуле (3.6) при разных значениях x.

```
procedure CalcNumQueries(Float t_cmd,
 Float t_secondary, Float t_vert, Float t_frag,
 Bool vertLimited, Integer numFragsForNode,
 Float alphaN, OcclStatArray occlStats)
begin
 //s1 + s2 × log(1+x) — количество невид. узлов
 Float s1, s2
 LogFit_NumQueries_And_NumInvisNodes(s1, s2, occlStats)

 //o1 + o2 × log(1+x) — количество невид. объектов
 Float o1, o2
 LogFit_NumQueries_And_NumInvisObjects(o1, o2, occlStats)

 //v1 + v2 × log(1+x) — количество невид. вершин
 Float v1, v2
 LogFit_NumQueries_And_NumInvisVerts(v1, v2, occlStats)

 //f1 + f2 × log(1+x) — количество невид. фрагментов
 Float f1, f2
 LogFit_NumQueries_And_NumInvisFrags(f1, f2, occlStats)

 Integer x
 if vertLimited then
  x ← (alphaN × o2 × t_cmd + s2 × t_secondary +
```

```
        v2 × t_vert) / (8 × t_vert + t_cmd ) – 1
 else
  x ← (alphaN × o2 × t_cmd + s2 × t_secondary +
        f2 × t_frag) / (numFragsForNode × t_frag + t_cmd) - 1
 endif

 Integer numNodes ← GetNumNodes()
 Float T0 ← CalcRenderingTime(0, s1, s2, o1, o2, v1, v2, f1, f2,
              t_cmd, t_ secondary, t_vert, t_frag,
              vertLimited, numNodes, numFragsForNode, alphaN)
 Float T1 ← CalcRenderingTime(x, s1, s2, o1, o2, v1, v2, f1, f2,
              t_cmd, t_ secondary, t_vert, t_frag,
              vertLimited, numNodes, numFragsForNode, alphaN)
 Float T2 ← CalcRenderingTime(numNodes, s1, s2, o1, o2, v1, v2, f1, f2,
              t_cmd, t_ secondary, t_vert, t_frag,
              vertLimited, numNodes, numFragsForNode, alphaN)
 Float minT ← Min(T0, Min(T1, T2))
 if minT = T0 then return 0
 if minT = T1 then return x
 if minT = T2 then return numNodes
end
```

3.3 Адаптивная стратегия рендеринга

Для эффективного рендеринга динамических сцен применяются следующие методы и техники:

- удаление невидимых объектов с использованием методов пространственной декомпозиции и индексирования;
- аппаратные проверки видимости с учетом временной когерентности статуса видимости;
- фрагментация и кэширование командных буферов с учетом локальных изменений в сцене.

Для выполнения этих методов требуются значительные вычислительные ресурсы, а объем работы на этапах графического конвейера зависит от их результатов. Поэтому важно сбалансированное использование ресурсов при конвейерной

обработке и передаче графических данных. Вместе с тем, вариативность в количестве и сложности индивидуальных объектов сцены, характере и интенсивности динамики делает подобную балансировку крайне сложной или даже невозможной при фиксировании конкретных методов и техник. Обеспечить высокую производительность вычислительной системы на широком классе задач рендеринга больших динамических сцен представляется возможным в результате адаптивного управления графическим конвейером, предусматривающего выбор и настройку альтернативных базовых методов и техник с учетом доступных ресурсов системы и особенностей отображаемой сцены на конкретном интервале модельного времени в конкретной пространственной области.

3.3.1 Рассматриваемые способы реализации процессов рендеринга

С помощью разработанной модели производительности графического конвейера проводится оценка эффективности базовых методов и техник для данного состояния динамической сцены. В зависимости от полученных результатов выбирается один из трех способов реализации процессов рендеринга в рамках однопроходной схемы:

1. Запись командного буфера без кэширования. Отсечение объектов, не попадающих в область видимости камеры.
2. Пространственное индексирование с использованием окто-дерева. Запись и кэширование командных буферов для каждого непустого узла. Отсечение узлов, не попадающих в область видимости камеры. Сбор данных для оценки количества невидимых узлов, объектов, вершин и фрагментов для данного состояния динамической сцены.
3. Пространственное индексирование с использованием окто-дерева. Запись и кэширование командных буферов для каждого непустого узла. Отсечение узлов, не попадающих в область видимости камеры. Выполнение аппаратных проверок видимости.

Эти способы выбраны, потому что наиболее значительное влияние на производительность рендеринга больших динамических сцен оказывают техники кэширования буферов, методы пространственного индексирования и выполнения проверок видимости. Предполагается, что аппаратные проверки видимости наиболее эффективны, когда количество объектов в сцене достаточно большое. Поэтому они не применяются в первом способе.

3.3.2 Критерии применения способов рендеринга

Сначала используется первый способ рендеринга. Для каждого кадра измеряется время записи команд рендеринга, время отсечения объектов и узлов окто-дерева. Оценивается время обновления командных буферов окто-дерева. Первый способ используется, пока выполняется условие:

$$T_{buf}(S_{vis}, HW) + T_{fc}(S, HW) < T_{update}(H, HW) + T_{buf}\left(H_{upd}, HW\right) + T_{\text{sec\_buf}}(H_{vis}, HW) + T_{fc}(H, HW). \quad (3.13)$$

Главным отличием первого и второго способов является использование окто-дерева. При применении первого способа окто-дерево еще может быть не создано. Запишем время составления командных буферов при условии использования предполагаемого окто-дерева H. Свойства предполагаемого окто-дерева можно получить, если задано количество объектов, приходящихся на один узел дерева. Значение $numInstPerNode$ устанавливается таким образом, чтобы хватило памяти для хранения вспомогательных буферов (см. формулы (3.1–3.3)).

$$T_{buf}(H, HW) = \alpha_n |H| c_n t_{buf}(HW) + |H| t_{\text{sec\_buf}}(c_n, HW), \quad (3.14)$$

$$c_n = numInstPerNode, \quad (3.15)$$

где h — предполагаемая высота дерева при заданном количестве объектов, приходящихся на один узел дерева (формула (3.1)),

$|H|$ — предполагаемое количество узлов дерева (формула (3.2)),

α_n — средняя доля перезаписываемых командных буферов,

c_n — среднее количество объектов в узле.

Для сцен с малым значением α_n и существенным временем записи ($T_{buf}(S_{vis}, HW) \gg T_{buf}\left(H_{upd}, HW\right) + T_{\text{sec\_buf}}(H_{vis}, HW)$) условие (3.13) перестает выполняться и происходит переход ко второму способу рендеринга. Производится создание окто-дерева (см. раздел 3.1) или выполняется обновление существующего окто-дерева. Также начинается накопление данных о количестве невидимых узлов, объектов, вершин и фрагментов. Для этого используется метод, описанный в разделе 3.2. Второй способ рендеринга используется до тех пор, пока выполняется условие:

$$T\left(S, S_{vis}, H, H_{vis}, H_{upd}, \emptyset, h = 1, g = 1, q = 0, HW\right) < T\left(S, S'_{vis}, H, H'_{vis}, H_{upd}, H_q, h = 1, g = 1, q = 1, HW\right), \quad (3.16)$$

где $T\left(S, S'_{vis}, H, H'_{vis}, H_{upd}, H_q, h = 1, g = 1, q = 1, HW\right)$ — время рендеринга объектов с выполнением проверок видимости (см. формулу (2.12)),

S_{vis} — видимые объекты сцены (видимость определяется с помощью отсечений),

S'_{vis} — видимые объекты сцены (видимость определяется с помощью отсечений и аппаратных проверок видимости),

H'_{vis} — видимые узлы окто-дерева (видимость определяется с помощью отсечений и аппаратных проверок видимости),

H_q — узлы окто-дерева, для которых выполняются аппаратные проверки видимости.

Множества S'_{vis}, H'_{vis} вычисляются с помощью накопленных данных о видимости элементов в течение многих кадров до тех пор, пока не будет произведена оценка видимости большинства объектов сцены (см. раздел 3.2). Далее выводится количество проверок видимости для эффективного рендеринга, которое используется для определения множества узлов H_q (узлы дерева при обходе в ширину). Условие (3.16) выполняется, когда аппаратные проверки видимости не являются эффективными. Переход к третьему способу рендеринга происходит, когда количество невидимых объектов достаточно высокое и, выполняя проверки видимости, можно сократить время рендеринга.

Во время выполнения третьего способа рендеринга также проводится оценка эффективности проверок видимости. Для каждого кадра измеряется время выполнения проверок видимости и выигрыш за счет проверок видимости. При неэффективном выполнении проверок видимости способ рендеринга изменяется в соответствии с условиями переходов (3.13, 3.16).

Рассмотрим псевдокод адаптивной стратегии. Процедура *CalcNumQueriesForScene* на основании данных о производительности вычислительной системы (*PerfMaps*) производит расчет количества проверок видимости для эффективного рендеринга.

```
procedure CalcNumQueriesForScene(Scene scene)
begin
 PerfMaps PerfMaps
 GetPerfMaps(PerfMaps)
 Bool isVertLimited ← IsVertLimited(PerfMaps)
 Integer numFragsForNode ← GetNumFragsForNode(scene.octree)
 Float alphaN ← GetAlphaN(scene)
 OcclStatArray occlStats ← GetOcclStats(scene.octree)
```

```
 Integer efficientNumQueries ←
  CalcNumQueries(PerfMaps.cmd,
   PerfMaps.secondary, PerfMaps.vert, PerfMaps.frag,
   isVertLimited, numFragsForNode, alphaN, occlStats)
 return efficientNumQueries
end
```

Процедура *AdaptiveRender* выполняет выбор одного из трех способов реализации рендеринга (*InliningFc*, *HierarchyFc*, *HierarchyFcOcclQuery*) и вызов соответствующей процедуры. Для выбора используются вспомогательные процедуры *IsInliningEfficient*, *IsOcclCullingEfficient*, реализующие вычисление условий (3.13), (3.16). Выбранный способ сохраняется в глобальной переменной с помощью процедур *GetRenderMethod*, *SetRenderMethod*.

```
procedure AdaptiveRender(Scene scene)
begin
 RenderingMethod method ← GetRenderMethod()
 if (method = InliningFc or method = HierarchyFc) and
      IsInliningEfficient(scene) then
   method ← InliningFc
 else if method = HierarchyFcOcclQuery and
      IsOcclCullingEfficient(scene) then
   method ← HierarchyFcOcclQuery
 else
  if GetRenderMethod() ≠ HierarchyFc then
   SetEfficientNumQueries(-1)
  endif
  Integer requiredNumOcclStats ←
          GetRequiredNumOcclStats(scene)
  if GetNumOcclStats() ≥ requiredNumOcclStats then
   Integer numQueries ← CalcNumQueriesForScene(scene)
   SetEfficientNumQueries(numQueries)
   ResetOcclStats()
   method ← HierarchyFcOcclQuery
  else
   method ← HierarchyFc
```

```
  endif
 endif
 SetRenderMethod(method)

 if method = InliningFc then
  RenderInlining(scene)
 else if method = HierarchyFc then
  Integer numOcclQueries ← GetTestNumQueries()
  RenderHierFcOcclQuery(scene, numOcclQueries, true)
  if GetOctreeAllocFailed() then
   SetRenderMethod(InliningFc)
   RenderInlining(scene)
  endif
 else
  Integer efficientNumQueries ← GetEfficientNumQueries()
  RenderHierFcOcclQuery(scene, efficientNumQueries, false)
  if GetOctreeAllocFailed() then
   SetRenderMethod(InliningFc)
   RenderInlining(scene)
  else
   Integer numQueries ← CalcNumQueriesForScene(scene)
   SetEfficientNumQueries(numQueries)
   ResetOcclStats()
  endif
 endif
end
```

Процедура *RenderInlining* выполняет рендеринг с записью командного буфера без кэширования. Значение переменной *periodicFrameIdx* задает индекс используемого командного буфера. Он может отличаться от индекса из предыдущего кадра, чтобы выполнять запись без ожидания готовности буфера. Процедура *GetMaxNumCommands* возвращает максимальное количество команд, которые отправляются на видеокарту. Когда количество объектов превышает значение *maxNumRecordedCmds*, выполняется отправка буфера и ожидание его готовности.

```
procedure RenderInlining(Scene scene)
```

```
begin
 Integer i
 Integer periodicFrameIdx ← GetPeriodicFrameIdx()
 CmdBuf cmdBuf
 GetCmdBuffer(cmdBuf, periodicFrameIdx)
 WaitForRenderingFinish(periodicFrameIdx)
 ResetCmdBuffer(cmdBuf)

 Frustum f
 BuildFrustum(f, scene)

 Integer maxNumRecordedCmds ← GetMaxNumCommands()
 Integer numRecordedCmds ← 0
 for i ← 0 to scene.numAllInstances-1 do
  MeshInstance inst ← scene.allInstances[i]
  if inst.visibility ≠ 0 and IsInsideFrustum(inst.bb, f) then
   RecordCmd(cmdBuf, inst, scene.buffersDescr)
   numRecordedCmds++
   if numRecordedCmds = maxNumRecordedCmds then
    numRecordedCmds ← 0
    SubmitCommands(cmdBuf)
    WaitForRenderingFinish(periodicFrameIdx)
    ResetCmdBuffer(cmdBuf)
   endif
  endif
 done

 if numRecordedCmds ≠ 0 then
  SubmitCommands(cmdBuf)
 endif
end
```

Процедура *RenderHierFcOcclQuery* выполняет второй или третий способ рендеринга в зависимости от значения *gradualCheck*. Параметр *numOcclQueries* определяет максимальное количество проверок видимости. При использовании второго

способа это значение поддерживается небольшим, чтобы снизить затраты на проверки видимости. Если окто-дерево не было создано, то производится создание и заполнение. Может оказаться так, что памяти не хватает на узлы дерева и буфера. В этом случае выставляет флаг *OctreeAllocFailed* и используется первый способ. Иначе выполняется получение результатов проверок видимости и составление главного командного буфера. Затем заполняется командный буфер *occlQueriesBuf* с проверками видимости. При значении флага *gradualCheck =true* выполняется обход дерева в ширину и отбор узлов, которые еще не были проверены. В конце процедуры записанные буфера отправляются на видеокарту.

```
procedure RenderHierFcOcclQuery(Scene scene,
      Integer numOcclQueries, Bool gradualCheck)
begin
 Integer i
 if not IsInit(scene.octree) and not GetOctreeAllocFailed() then
  OctreeNode root
  Integer numInstPerNode ←
       GetNumInstPerNode(scene.numAllInstances)
  BitArray instIntersectingSecants
  Integer usedCmdMem ← 0
  Bool allocFailed ← false
  for i ← 0 to scene.numAllInstances-1 do
   MeshInstance nextInst ← scene.allInstances[i]
   AddItemToOctree(root, nextInst,
    numInstPerNode, scene.allInstances, instIntersectingSecants
    usedCmdMem, allocFailed)
   if allocFailed break
  done
  if not allocFailed then scene.octree ← CreateOctree(root)
  else SetOctreeAllocFailed(true)
 endif

 if not IsInit(scene.octree) then return

 Integer periodicFrameIdx ← GetPeriodicFrameIdx()
```

```
CmdBuf cmdBuf
GetCmdBuffer(cmdBuf, periodicFrameIdx)
WaitForRenderingFinish(periodicFrameIdx)
ResetCmdBuffer(cmdBuf)

Frustum f
BuildFrustum(f, scene)

for node ∈ scene.octree do
 node.isOccluded[periodicFrameIdx] ← false
done

CheckQueryResults(scene.octree, scene, true)

CmdBufArray secondaryBuffers
for node ∈ scene.octree do
 if not node.isOccluded[periodicFrameIdx] and
    IsInsideFrustum(node.bb, f) then
    Push(secondaryBuffers, node.secondaryBuffer)
 endif
done

CmdBuf occlQueriesBuf
Integer recordedNumQueries ← 0
BoundingBox camBB
GetCameraBb(camBB, scene)
if numOcclQueries ≠ 0 then
 RecordHeavyOcclQueries(occlQueriesBuf,
   scene.octree, gradualCheck, f, camBB, numOcclQueries,
   recordedNumQueries)
endif

if recordedNumQueries ≠ 0 then
 Push(secondaryBuffers, occlQueriesBuf)
```

```
  endif

  RecordExecBuffers(cmdBuf, secondaryBuffers)
  SubmitCommands(cmdBuf)
end
```

3.4 Метод генерации сцен

В данной работе рассматривается рендеринг больших динамических сцен. Для наиболее полной апробации предлагаемых в данной работе методов требуется провести тестирование на большом количестве разнородных сцен. Предлагается метод генерации динамических сцен с большим количеством параметров, которые позволяют получить различные тестовые данные. В результате работы метода создается файл в формате FBX с полигональным представлением сцены и информацией о видимости объектов. Рассмотрим параметры предлагаемого метода:

- Входной файл со зданием в формате FBX.
- Количество зданий по осям X, Y.
- Расстояние между зданиями.
- Размеры кирпича.
- Ширина и длина здания в кирпичах.
- Высота этажа здания в кирпичах.
- Количество этажей в здании.
- Наличие перегородок и окон.
- Ширина и высота окна.
- Наличие дополнительных объектов внутри зданий.
- Количество дополнительных объектов на этаже.
- Входной файл с дополнительным объектом в формате FBX.
- Назначение видимости объектов вдоль кривой, заполняющей пространство.
- Количество ячеек в сетке по осям X,Y,Z.
- Временной интервал между появлением соседних ячеек.
- Порядок обхода трехмерной сетки.
- Коэффициент пространственно-временной когерентности событий.

При задании входного файла, он используется в качестве модели здания. Если входной файл не задан, то производится создание модели здания по заданным па-

раметрам. Допускается генерация зданий различных размеров с окнами и перегородками (рисунок 3.4). Итоговое здание расставляется в горизонтальной плоскости на заданном расстоянии. Для повышения вычислительной нагрузки возможно размещение дополнительных объектов на каждом этаже зданий (рисунок 3.5). Если входной файл с дополнительным объектом не указан, то в качестве объектов используются модели кирпичей.

Полученный квартал зданий является статическим. Следующие параметры позволяют назначить информацию о видимости в разные моменты времени. Для этого используются кривые, заполняющие пространство: Гильберта, Мортона, построчного порядка Row-prime (рисунок 3.6). В качестве системы координат используется правая тройка XYZ (ось Z направлена вверх). Порядок обхода позволяет изменить направление обхода по каждой оси (рисунок 3.5). На каждую ячейку сетки приходится множество объектов сцены, которое становится видимым в момент времени $n * t$, где n — номер ячейки, t — временной интервал. Номер n вычисляется по выбранной кривой, заполняющей пространство, согласно порядку обхода.

Рисунок 3.4 — Одно высокое здание (слева). Группа маленьких зданий (справа).

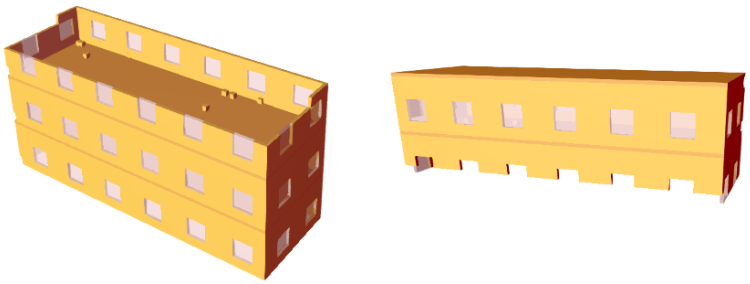

Рисунок 3.5 — Проход снизу вверх (слева), проход сверху вниз (справа).

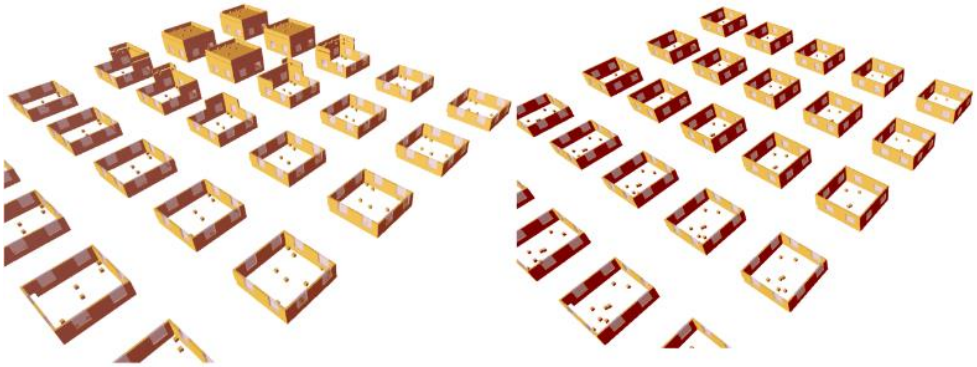

Рисунок 3.6 — Объекты выводятся вдоль пространственной кривой Гильберта (слева), объекты выводятся вдоль кривой с построчным порядком. Показано для одного момента времени.

Важным параметром является коэффициент пространственно-временной когерентности событий C. Пусть сцена содержит N объектов. При значении коэффициента $C = 1$ время вывода объектов соответствует номеру ячейки вдоль кривой, заполняющей пространство. При других значениях C вычисляется количество объектов с нарушением времени вывода: $V = N(1 - C)$. Затем выполняется отбор объектов с индексами $0, \frac{N}{V}, \frac{2N}{V}, \frac{3N}{V}, \ldots$. Для отобранных объектов случайным образом определяется новое время появления в сцене. Таким образом, при уменьшении значения C вывод объектов отклоняется от порядка, задаваемого кривой, заполняющей пространство (рисунок 3.7).

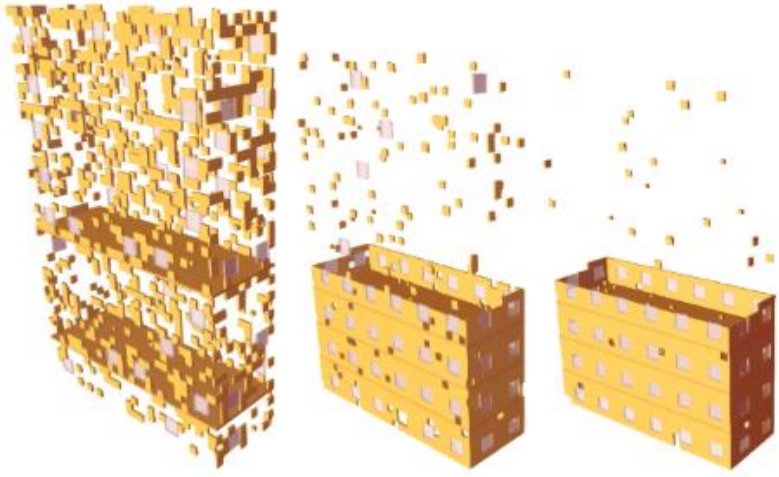

Рисунок 3.7 — Вывод объектов сцен с разными значениями коэффициента пространственно-временной когерентности (0.1, 0.95, 0.99) вдоль кривой Row-prime.

Предлагаемый метод также можно использовать для преобразования статической сцены в динамическую. Для этого в качестве входных данных задается файл в формате FBX. Выставляется флаг о назначении видимости вдоль кривой, заполняющей пространство. В результате работы метода создается файл в формате FBX с информацией о видимости объектов. На рисунке 3.8 приведен пример, показывающий результат такого преобразования.

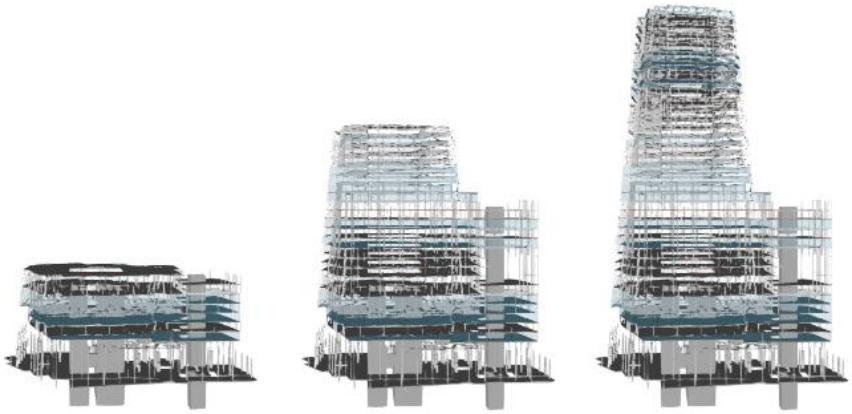

Рисунок 3.8 — Генерация динамической сцены из статической. Время появления рассчитано вдоль кривой Row-prime.

ГЛАВА 4. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МОДЕЛИ ПРОИЗВОДИТЕЛЬНОСТИ И СТРАТЕГИИ РЕНДЕРИНГА

4.1 Принципы и организация библиотеки программ для рендеринга динамических сцен

Предложенная модель и методы были реализованы в рамках графической библиотеки VulkanRenderer.dll с использованием программного интерфейса рендеринга Vulkan. Произведено внедрение графической библиотеки в программу для выполнения пространственно-временного моделирования и планирования индустриальных проектов (рисунок 4.1). Для использования разработанных методов необходимо переключить способ рендеринга в окне для вывода кадрового буфера на новый способ “Vulkan”. После этого проводится инициализация графических подсистем Vulkan и преобразование объектов сцены:

- Преобразование системы координат корневых объектов и камеры.
- Создание треугольных сеток по аналитическим, твердотельным моделям и полигональным сеткам.
- Загрузка геометрии в видеопамять.
- Загрузка текстур в память.
- Создание ключевых кадров с информацией об изменении свойств объектов сцены (видимость, положение, материал) по диаграмме Ганта.
- Инициализация поверхности VkSurfaceKHR для вывода кадрового буфера по дескриптору окна Windows.
- Поиск подходящего GPU процессора, который имеет хотя бы одну очередь для выполнения команд рендеринга.
- Создание логического устройства со своим контекстом рендеринга по найденному GPU процессору.
- Создание пулов для записи команд рендеринга.
- Создание пулов для записи результатов проверок видимости.
- Создание буферов UBO для задания положений и свойств материалов объектов.
- Создание цепочки кадровых буферов (Swapchain), которая поддерживает способ вывода кадрового буфера на экран Mailbox. Этот способ уменьшает задержку при получении свободного кадрового буфера.
- Создание семафоров для синхронизации доступа к командному буферу.
- Создание конфигураций конвейеров для всех материалов сцены.

Для проведения оценок с помощью модели производительности графического конвейера, описанной в главе 2, проводится вычисление времени выполнения этапов рендеринга. Для этого производится генерация, рендеринг тестовых сцен, запись результатов в файл. При дальнейшем использовании разработанной графической библиотеки выполняется чтение файла с результатами тестирования.

Рассмотрим функции графической библиотеки. Поддерживается навигация по сцене с использованием мыши. При нажатии левой кнопки мыши создается луч, выходящий из камеры в трехмерное пространство. Вычисляется пересечение луча с ограничивающими параллелепипедами AABB объектов сцены. Для ускорения поиска пересечений используется окто-дерево, если оно было создано. Ближайшая точка пересечения луча и сцены используется, как точка опоры при вращении камеры. При зажатой левой кнопке мыши осуществляется вращение вокруг точки опоры. Колесо прокрутки используется для приближения и удаления камеры в направлении просмотра.

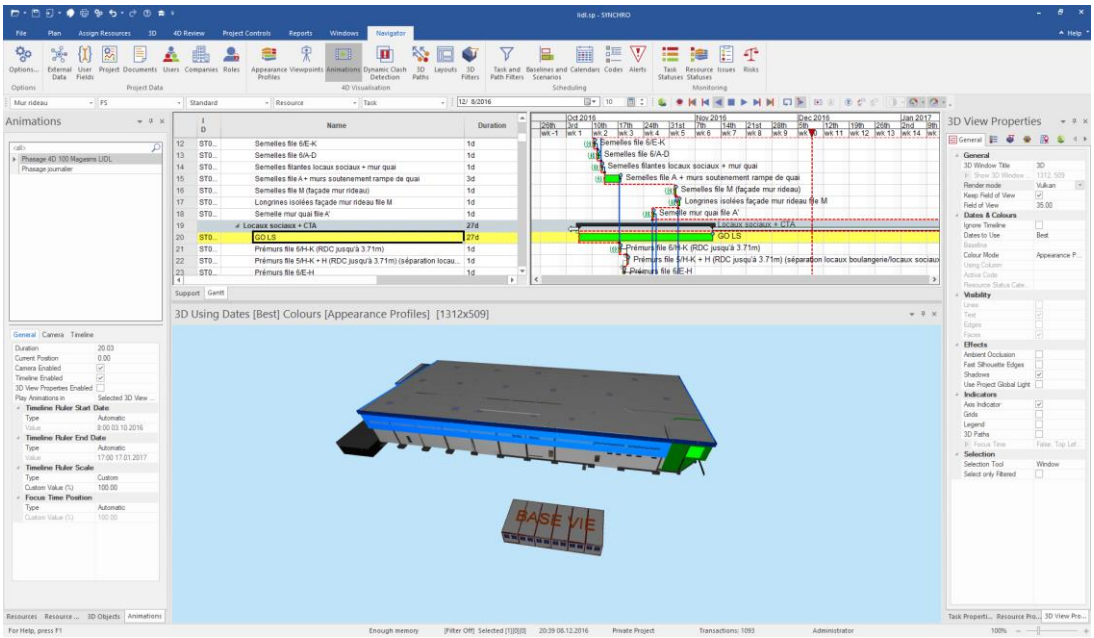

Рисунок 4.1 — Главное окно программы для пространственно-временного моделирования и планирования индустриальных проектов.

Поддерживается вывод на экран динамической сцены. После задания текущего календарного времени проводится перерасчет свойств объектов сцены для данного времени. Для этого осуществляется поиск ключевых событий со свойствами сцены на данное время. Если значение видимости отличается от текущего, то выполняется запись нового значения и обновление окто-дерева. Если изменяется положение, то проводится удаление объекта из текущего узла дерева и вставка в

новый узел. Также в случае изменения положения или материала в соответствующий буфер UBO записывается положение и свойства нового материала. Осуществляется рендеринг видимых объектов сцены с помощью методов, описанных в разделе 3.3.

Поддерживается запись проектной анимации в видеофайл. В диалоге по выбору анимации в контекстном меню есть опция по записи кадров проектной анимации в файл. Выбирается разрешения файла и количество кадров в секунду, путь к выходному файлу. Поддерживаются выходные форматы данных: jpg, png, bmp, avi. Допускается добавление в кадровый буфер описаний ресурсов, диаграммы Ганта, календарного времени, различных текстовых описаний. После запуска процесса проигрывания проектной анимации производится рендеринг сцены и запись результатов в видеофайл или запись каждого кадра в отдельный файл. Время выполнения этого процесса зависит от времени рендеринга сцены для получения кадрового буфера.

4.2 Программные модули, реализующие модель производительности графического конвейера

В реализации модели производительности графического конвейера выделяются процедуры по тестированию производительности для получения времени выполнения этапов рендеринга на оборудовании пользователя и вычислению итогового времени рендеринга. На рисунке 4.2 показана схема проведения измерений производительности. Необходимо измерить время выполнения всех рассматриваемых в работе этапов рендеринга. При этом в цикле проводится несколько измерений с различными сценами для каждого этапа. Это делается для получения данных, в которых учитывается зависимость времени обработки элементов на GPU от количества элементов (вершин, фрагментов, вспомогательных командных буферов). В конце работы результаты записываются в файл для повторного использования в процессе отображения других сцен.

Реализация выполнялась для операционных систем семейства Windows на языке C++ с использованием программного интерфейса Vulkan. Рассмотрим наиболее важные программные модули, которые использовались в реализации. В листинге 4.1 приведены используемые далее определения типов. В них явно указано наличие знака, количество бит в числе. Это используется для возможности переноса кода на другие платформы.

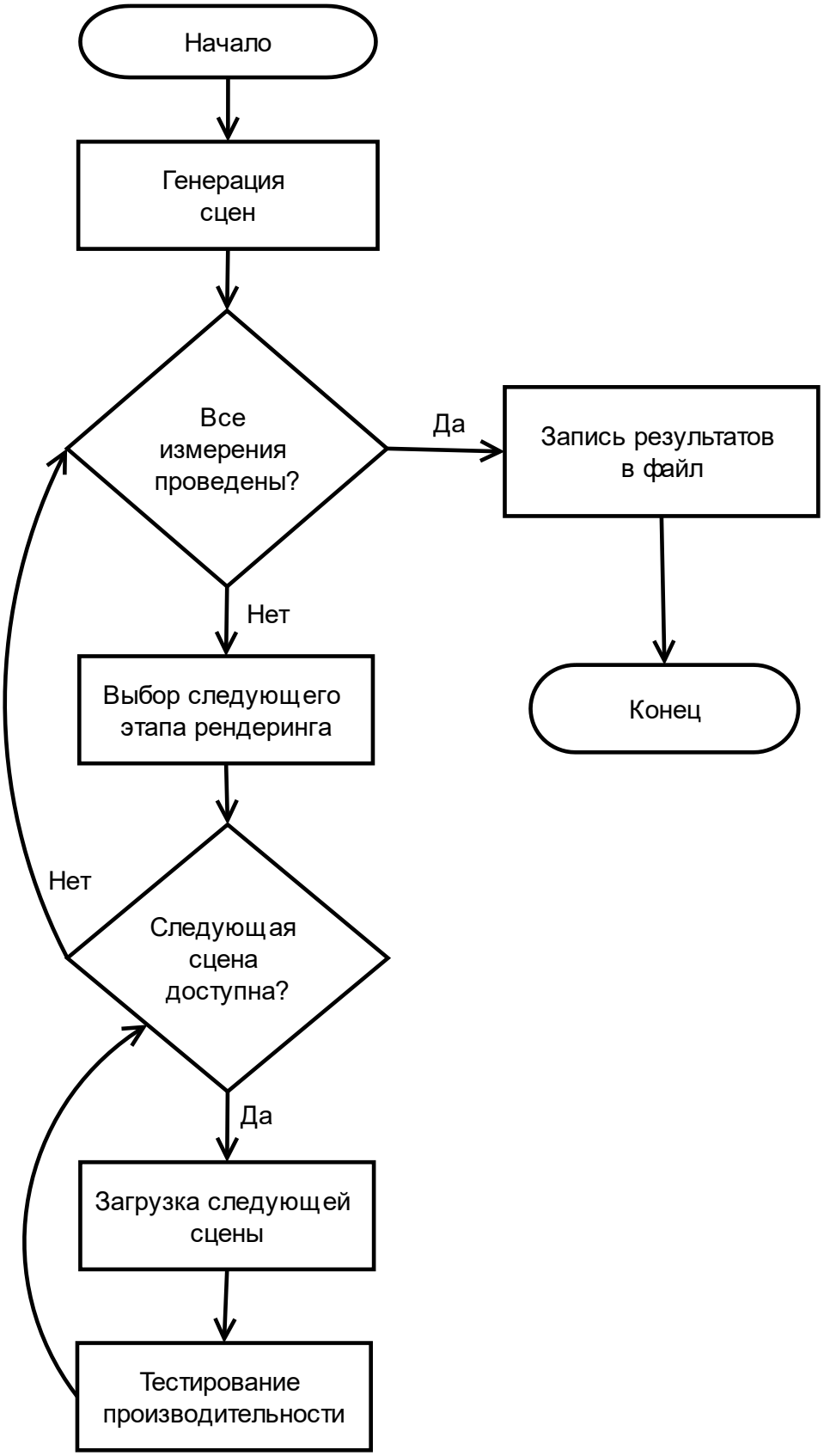

Рисунок 4.2 — Схема проведения измерений производительности для получения времени выполнения этапов рендеринга.

Листинг 4.1 — Определения типов

```
typedef unsigned char u8;
typedef short s16;
typedef int s32;
typedef long long s64;
typedef unsigned int u32;
typedef unsigned long long u64;
typedef float r32;
typedef double r64;
```

В листинге 4.2 приведены объявления классов, которые используются для записи времени рендеринга. Класс *CmdRecordTime* предназначен для хранения времени записи команд на центральном процессоре для заданного количества объектов (команд). Время записи зависит от методики по расширению буфера при добавлении новых элементов. В случае, когда памяти недостаточно, можно увеличить размер буфера на константу или в несколько раз. Поэтому важно также учитывать количество записываемых команд.

Класс *VertexRenderingTime* предназначен для хранения времени обработки вершин для заданного количества объектов и вершин в объекте. Отметим, что в вершинном шейдере также производится загрузка матрицы преобразования в кэш память. Если загрузка происходит часто, то это может стать узким местом. Параметры, задающие количество вершин и объектов, позволяют это учесть.

Класс *FragRenderingTime* предназначен для хранения времени обработки фрагментов для заданного количества фрагментов. Современные GPU процессоры имеют этап для выполнения композиции (render output unit) [93]. При увеличении количества фрагментов этот этап может стать узким местом. Задание количества обрабатываемых фрагментов позволяет это учесть.

Класс *OcclQueryTime* предназначен для хранения времени выполнения аппаратных проверок видимости, которое включает в себя отправку, выполнение на GPU и получение результатов проверок видимости, для заданного количества объектов. Выделение трех стадий выполнения проверок видимости важно, потому что часть из них выполняется на CPU, часть — на GPU. Это позволяет отдельно рассчитать время рендеринга на CPU и GPU.

Класс *SecondaryBufSubmitTime* предназначен для хранения времени записи и отправки вспомогательных буферов команд для заданного количества буферов и

количества объектов, приходящихся на один буфер. При увеличении количества используемых вспомогательных буферов может увеличиться время отправки. Это связано с тем, что вспомогательный буфер занимает, по крайней мере, одну страницу памяти. Поэтому использование большого количества буферов может увеличить расходуемый объем памяти. Также при отправке может возникнуть задержка, связанная с зачитыванием буферов, расположенных в разных областях памяти. Используемые параметры позволяют это учесть.

Листинг 4.2 — Объявления классов для записи времени рендеринга

```
class CmdRecordTime {
public:
  u32 numInstances;
  r64 recordingTime;
  CmdRecordTime();
  CmdRecordTime(u32 numInst, r64 recTime);
};

class VertexRenderingTime {
public:
  u32 numVertices;
  u32 numInstances;
  r64 time;
  VertexRenderingTime();
  VertexRenderingTime(u32 numVerts, u32 numInst, r64 time);
};

class FragRenderingTime {
public:
  u32 numFrags;
  r64 time;
  FragRenderingTime();
  FragRenderingTime(u32 numFrag, r64 time);
};

class OcclQueryTime {
```

```
public:
   u32 numInstances;
   r64 querySendTime;
   r64 queryExecTime;
   r64 queryRecvTime;
   OcclQueryTime();
   OcclQueryTime(u32 numInst, r64 querySendTime,
                        r64 queryExecTime, r64 queryRecvTime);
};

class SecondaryBufSubmitTime {
public:
   u32 numSecondaries;
   u32 numInstPerSecondary;
   r64 timePerSecondary;
   SecondaryBufSubmitTime() ;
   SecondaryBufSubmitTime(u32 numSecondaryBufs,
      u32 numInstPerSecondaryBuf, r64 timePerSecondaryBuf);
};
```

В листинге 4.3 приведены объявления классов, которые используются для записи результатов с измерениями производительности рендеринга. Класс *MeasurementResults* содержит результаты измерений:

- Время выполнения отсечения объектов, не попадающих в область видимости камеры (Frustum Culling).
- Время записи команды рендеринга.
- Время обработки одной вершины.
- Время обработки одного фрагмента.
- Время отправки, выполнения, получения результата проверки видимости.
- Время записи и отправки вспомогательного буфера команд.
- Время подготовки к передаче кадрового буфера для вывода на экран (presentation). В Vulkan для этого производится вызов процедуры vkQueuePresentKHR.
- Время установки конфигурации графического конвейера. Новая конфигурация задается при изменении материала.

- Объем памяти для хранения буфера с матрицей преобразования и цветом объекта. Размер буфера может быть больше, чем размер структуры *sizeof(shdubo_m_diff)*. Это связано с тем, что минимальный размер определяется на основе значения *minUniformBufferOffsetAlignment*, которое можно получить с помощью вызова процедуры *vkGetPhysicalDeviceProperties*.
- Объем памяти для хранения команды рендеринга. Зачастую командные буфера хранятся в основной памяти. Для определения прироста памяти используется процедура *GlobalMemoryStatusEx*.
- Объем памяти для хранения вспомогательного буфера.

Класс *MeasurementArrays* содержит массивы с результатами проведенных измерений времени записи команд, обработки фрагментов, обработки вершин, выполнения проверок видимости, записи и отправке вспомогательных буферов команд. Этот класс содержит только те результаты измерений, которые зависят от объема вычислений.

Класс *EnabledMeasurements* содержит флаги, которые позволяют задать выполнение необходимых измерений. Это позволяет написать одну процедуру для выполнения измерений и проводить только необходимые измерения.

Класс *MeasurementParams* содержит входные параметры и результаты измерений для одной сцены. Входные параметры задают имя файла с полигональным представлением сцены, направление обхода объектов сцены, расстояние от камеры до ограничивающего параллелепипеда AABB сцены. Результаты вычислений содержат все необходимые характеристики сцены и измеренные времена различных этапов рендеринга.

Листинг 4.3 — Объявления классов для записи результатов с измерениями производительности рендеринга

```
#include <vector>
class MeasurementResults {
public:
  r64 fc;
  r64 cmd;
  r64 vert;
  r64 frag;
  r64 querySend;
```

```
    r64 queryExec;
    r64 queryRecv;
    r64 secondary;
    r64 present;
    r64 setPipeState;
    u64 bUbo;
    u64 bCmd;
    u64 bMinSecondary;
    MeasurementResults();
};

class MeasurementArrays {
public:
    std::vector<CmdRecordTime> cmd;
    std::vector<VertexRenderingTime> vert;
    std::vector<FragRenderingTime> frag;
    std::vector<OcclQueryTime> query;
    std::vector<SecondaryBufSubmitTime> secondary;
};

class EnabledMeasurements {
public:
    bool fc;
    bool cmd;
    bool vert;
    bool frag;
    bool query;
    bool secondary;
    bool present;
    bool pipeState;
    EnabledMeasurements();
};

class MeasurementParams {
```

```
public:
  //входные данные
  const char *filename;
  u32 numRenderedFrames;
  r32 distanceFromCamera;
  bool isBackToFront;
  EnabledMeasurements enabled;

  //выходные данные
  MeasurementResults results;
  r64 oneInstRenderTime;
  r64 totalInstRenderTime;
  u32 numInstances;
  u32 numNodes;
  u32 numVerticesInInst;
  u32 numVerticesTotal;
  u32 numFrags;
  MeasurementParams();
};
```

В листинге 4.4 приведены объявления процедур, которые используются для проведения тестирования производительности вычислительной системы. Процедура *measure_generateScenes* выполняет генерацию сцен с различными значениями параметров *batchSize*, *numObjects*. В реализации использовались сцены из кубов и сцены из сфер с большим количеством полигонов. Сцены из кубов позволяют измерить производительность с увеличивающимся количеством объектов. Сцены из сфер позволяют измерить производительность при увеличении количества вершин в объекте.

Процедура *measure_generateScenesForFragRendering* выполняет генерацию сцен, которые в дальнейшем будут использоваться для проведения измерений по обработке фрагментов для данного материала. Отдельная процедура необходима, потому что требуется выводить большое количество фрагментов, а количество вершин должно быть маленьким. Для этого подходят сцены, в которых объекты, включающие два треугольника, расположены близко к камере.

Процедура *measure_makeForScene* предназначена для проведения измерений производительности при выполнении рендеринга переданной сцены. Сначала производится загрузка сцены. Затем в цикле выполняется запись буфера команд. При этом проводятся вычисления, которые заданы в объекте *enabled*. Результаты измерений записываются в объект *results*.

Процедура *measure_mem* производит вычисление объема памяти, который требуется для хранения буфера с матрицей преобразования и цветом объекта, одной команды рендеринга, одного вспомогательного буфера с одной командой рендеринга. Для этого проводится измерение затрат по памяти при выполнении рендеринга сцен с некоторым количеством объектов. Используется процедура *GlobalMemoryStatusEx*.

Процедура *measure_fc* производит измерение времени отсечения объектов, не попадающих в область видимости камеры. Для этого проводится проверка положения точек AABB относительно плоскостей усеченной пирамиды камеры. Нормали плоскостей смотрят внутрь пирамиды. Вычисляется расстояние со знаком от вершины до плоскости. Если все вершины AABB находятся за плоскостью, то объект считается невидимым.

Процедура *measure_hierarchyTraversal* производит измерение времени обхода дерева. Во время обхода узлов дерева выполняется большое количество обращений в память. Если узлы дерева находятся не в одной области памяти, то может возникнуть кэш-промах (cache miss). После промаха происходит обращение к основной памяти, что связано с дополнительными временными затратами. Это может снизить производительность рендеринга на CPU, когда дерево достаточно большое.

Процедура *measure_command* производит измерение времени записи команд рендеринга для данных файлов с описанием сцен и данного материала.

Процедура *measure_query* производит измерение времени выполнения проверок видимости для данных файлов с описанием сцен и данного материала.

Процедура *measure_vert* производит измерение времени обработки вершин для данных файлов с описанием сцен и данного материала.

Процедура *measure_frag* производит измерение времени обработки фрагментов для данных файлов с описанием сцен и данного материала.

Процедура *measure_secondary* производит измерение времени записи и отправки вспомогательных буферов для данных файлов с описанием сцен и данного материала.

Процедура *measure_makeForMaterial* предназначена для измерения времени выполнения всех этапов рендеринга для данного материала.

Процедура *measure_makeCache* производит измерение времени выполнения этапов рендеринга для данных материалов и кэширование результатов.

Процедура *measure_get* предназначена для получения усредненных времен выполнения этапов рендеринга для данной сцены и материала. Эта процедура вызывается в процессе отображения динамической сцены и возвращает актуальные значения для времен выполнения этапов рендеринга.

Процедура *measure_getUboSizeBytes* возвращает количество байт, которое требуется для хранения в памяти матрицы преобразования и цвета объекта.

Процедура *measure_getCmdUsageBytes* возвращает количество байт, которое требуется для хранения в памяти команды рендеринга.

Процедура *measure_getMinSecondaryBytes* возвращает количество байт, которое требуется для хранения в памяти вспомогательного буфера с одной командой рендеринга.

Процедура *measure_readSerializedData* производит чтение записанных на жесткий диск данных с информацией о проведенных измерениях.

Процедура *measure_writeSerializedData* производит запись на жесткий диск данных с информацией о проведенных измерениях.

Листинг 4.4 — Процедуры для проведения измерений производительности вычислительной системы

```
#include <vector>
#include <string>

//Процедуры генерации сцен
void measure_generateScenes(
  std::vector<std::string> *outFilenames, Material *mat);
void measure_generateScenesForFragRendering(
  std::vector<std::string> *outFilenames, Material *mat);

//Измерение производительности
void measure_makeForScene(MeasurementParams *p);
void measure_mem(u64 *bUbo, u64 *bCmd, u64 *bMinSecondary);
void measure_fc(r64 *fc);
```

```
void measure_hierarchyTraversal(r64 *hierTrav);
void measure_command(std::vector<CmdRecordTime> *cmd,
   std::vector<std::string> *filenames, Material *mat);
void measure_query(std::vector<OcclQueryTime> *query,
   std::vector<std::string> *filenames);
void measure_vert(std::vector<VertexRenderingTime> *vert,
   r64 *efficientVertexRenderingTime, Material *mat,
         std::vector<std::string> *filenames);
void measure_frag(std::vector<FragRenderingTime> *frag,
   Material *mat, std::vector<std::string> *filenames);

void measure_secondary(
  std::vector<SecondaryBufSubmitTime> *secondarySubmit,
  r64 *present, u32 numSecondaries,
        std::vector<std::string> *filenames);

//Кэширование результатов
void measure_makeForMaterial(MeasurementResults *res,
  MeasurementArrays *arrays, Material *mat);
void measure_makeCache(u32 numMaterials, Material *matArray);
void measure_get(MeasurementResults *res, Scene *scene,
  Material *mat);

//Объем памяти
u64 measure_getUboSizeBytes();
u64 measure_getCmdUsageBytes();
u64 measure_getMinSecondaryBytes();

//Сериализация
void measure_readSerializedData();
void measure_writeSerializedData();
```

4.3 Особенности программной реализации адаптивной стратегии рендеринга

Рассмотрим классы и процедуры, которые использовались при реализации адаптивной стратегии рендеринга, предложенной в данной работе, на языке C++. В листинге 4.5 приведены объявления классов, задающих окто-дерево. Класс *BoundingBox* задает параллелепипед AABB через центр и размеры по осям X,Y,Z.

Класс *OctreeNode* задает узел окто-дерева: уникальный индекс, ограничивающий параллелепипед AABB, параметры рендеринга AABB, идентификаторы объектов данного узла, указатель на родительский узел, указатель на дочерние узлы, оценка площади содержащихся в узле объектов.

Класс *Octree* задает параметры окто-дерева: флаги невидимых узлов, флаги узлов, попавших в область видимости камеры, количество уровней в окто-дереве, корневой узел окто-дерева, дескрипторы для хранения в видеопамяти параллелепипедов AABB для узлов окто-дерева.

Листинг 4.5 — Объявления классов для задания окто-дерева

```
class BoundingBox {
public:
  r32 center[3];
  r32 hsize[3];
};

class OctreeNode {
public:
  u32 idx;
  BoundingBox bv;
  MeshInstance *bvInst;
  U32HashSet *instances;
  OctreeNode *parent;
  OctreeNode *subnodes[8];
  r32 occupiedArea;
};

class Octree {
public:
```

```
    BitArray occlusionCulledFlags;
    BitArray frustumCulledFlags;
    u8 numLevels;
    OctreeNode *root;
    VgBuffersDescr buffersDescr;
};
```

В листинге 4.6 приведены объявления классов, задающих элементы сцены. Класс *Camera* задает положение, ориентацию камеры, матрицу перспективной проекции, матрицу для задания положения камеры.

Класс *Material* задает цвет материала и текстуру, если она используется. Значение *diff* определяет интенсивность рассеиваемого света. Значения *texW, texH* используются при расчете объема памяти, который требуется для хранения текстуры.

Класс *Mesh* задает смещение в памяти и количество вершин треугольной сетки. Все треугольные сетки хранятся в одном буфере для геометрии сцены, который создается в видеопамяти (device local).

Класс *MeshInstance* задает объект сцены с данной треугольной сеткой, ограничивающим параллелепипедом AABB, матрицей преобразования, материалом. Также кэшируется видимость объекта в данный момент времени и указатель на узел дерева, который его содержит.

Класс *BitArray* предназначен для описания битовых флагов объектов сцены. Хранение флагов в отдельном массиве используется для повышенной когерентности при обращении в кэш память центрального процессора (cache coherency).

Класс *Scene* задает описание трехмерной сцены:

- дескрипторы геометрии сцены,
- дескрипторы ограничивающих параллелепипедов узлов дерева,
- объекты сцены,
- материалы объектов сцены,
- ограничивающий параллелепипед AABB сцены,
- камера,
- окто-дерево,
- ключевые кадры анимации положения и видимости объектов сцены,
- флаги для отметки наполненных узлов дерева,
- флаги для отметки пересекающихся с параллелепипедом камеры узлов дерева,

- дескрипторы буферов для хранения матриц преобразования и цвета всех объектов сцены,
- массив отложенных обновлений матриц преобразования объектов сцены.

Листинг 4.6 — Объявления классов для задания элементов сцены

```
class Camera {
public:
  Vertex3 pos;
  Vertex3 forward;
  Vertex3 up;
  Matrix viewMatrix;
  Matrix projMatrix;
};

class Material {
public:
  u32 idx;
  char *textureFilename;
  u32 texW, texH;
  float diff[3];
};

class Mesh {
public:
  s32 drawStart;
  s32 drawCount;
};

class MeshInstance {
public:
  u32 idx;
  Mesh* mesh;
  Material *material;
  BoundingBox bb;
  Matrix transform;
```

```
    u8 visibility;
    OctreeNode *node;
};

Class BitArray {
public:
    u32 numObjects;
    char *bits;
};

class Scene {
public:
    VgBuffersDescr buffersDescr;
    VgBuffersDescr bbBuffersDescr;
    MeshInstance *allInstances;
    u32 numAllInstances;
    Material *materials;
    u32 numMaterials;
    BoundingBox sceneBB;
    Camera *cam;
    Octree *octree;
    TrnTl *trntl;
    VisTl *vistl;
    MatTl *mattl;
    u64 animStartTime;
    u64 animEndTime;
    BitArray *heavyNodes;
    BitArray *cameraInsideNodeFlags;
    VkDescriptorSetLayout descriptorSetLayout;
    u32 effectiveUboOffset;
    u32 numMatricesInUBO;
    u32 numUBOs;
    VkDescriptorPool descriptorPool[NumPrimBufs];
    VkDescriptorSet *sets[NumPrimBufs];
```

```
  VkBuffer       *ubos[NumPrimBufs];
  VkDeviceMemory *uboMems[NumPrimBufs];
  std::unordered_map<u32, DelayedTransformUpload> delayedUploads;
};
```

В листинге 4.7 приведены объявления классов, задающих деревья с ключевыми кадрами для анимации свойств объектов сцены. Класс *TrnKfr* задает параметры ключевого кадра с положением объекта: момент времени, матрица преобразования, метод интерполяции.

Класс *TrnCv* используется для хранения всех ключевых кадров с положением для данного объекта. Массив *m_kfrArray* поддерживается в сортированном состоянии для ускоренного поиска ключевых кадров по времени с помощью алгоритма двоичного поиска.

Класс *TrnTl* используется для хранения всех объектов *TrnCv* для данной сцены. Во время отображения динамической сцены изменяется время анимации. Процедура *getChangedInstances* используется для вычисления объектов, которые изменились при установке нового анимационного времени. Для этого выполняется обход массива *m_instToCvMap* и определение объектов, у которых изменилась матрица преобразования.

Аналогичным образом объявлены классы для обработки временных событий, связанных с изменением видимости и свойств материалов объектов. Класс *VisKfr* задает параметры ключевого кадра преобразования видимости объекта: момент времени, видимость, метод интерполяции. Класс *VisCv* используется для хранения всех ключевых кадров видимости для данного объекта. Класс *VisTl* используется для хранения всех объектов *VisCv* для данной сцены.

Класс *MatKfr* задает параметры ключевого кадра с материалом: момент времени, цвет, метод интерполяции. Класс *MatCv* используется для хранения всех ключевых кадров с материалами для данного объекта. Класс *MatTl* используется для хранения всех объектов *MatCv* для данной сцены.

Листинг 4.7 — Объявления классов с ключевыми кадрами для анимации свойств объектов сцены

```
class TrnKfr {
public:
  u64 m_time;
```

```
   enum InterpolType m_interpolType;
   r32 m_item[16];
};

class TrnCv {
public:
   TrnKfrArray *m_kfrArray;
   u32 m_instIdx;
   TrnKfr *m_prev;
   TrnKfr *m_next;
   void findClosest(TrnKfr **res, u32 *idx, u32 middleIdx,
   bool isRightPart, u64 needle, u32 numItems, TrnKfr *keys);
   void insertKey(TrnKfr *key);
   s32 getItem(Matrix *itemRes, u64 time, TrnKfr **outPrev,
      TrnKfr **outNext);
};

class TrnTl {
public:
   TrnCvHashSet *m_instToCvMap;
   void init();
   TrnCv *getCv(u32 instIdx);
   void insertKey( u32 instIdx, TrnKfr *key );
   s32 getTrn(Matrix *res, u32 instIdx, u64 pTime);
   s16 getChangedInstances(U32Array *instances, MatrixArray *items,
      u64 prevTime, u64 time);
   void getBoundaryTimes(u64 *min, u64 *nextToMin, u64 *max);
   u32 numCurves();
   u32 numKeyframes();
   s16 free();
};
```

В листинге 4.8 приведены объявления классов для задания буферов Vulkan. Константа *NumPrimBufs* используется для установки количества главных буферов

рендеринга. Класс *VgBuffersDescr* содержит дескрипторы программного интерфейса Vulkan: вершинный буфер, буфер нормалей, буфер с текстурными координатами, буфер с индексами.

Класс *shdubo_m_diff* задает матрицу преобразования и цвет объекта сцены.

Класс *DelayedTransformUpload* используется для отложенной загрузки матрицы преобразования (цвета) объекта при движении объекта (назначении нового цвета).

Листинг 4.8 — Объявления классов для задания буферов Vulkan

```
#include <vulkan/vulkan.h>
#include <unordered_map>
#define NumPrimBufs 1

class VgBuffersDescr {
public:
  VkBuffer       vertexBuffer;
  VkDeviceMemory vertexBufferMemory;
  VkBuffer       normalBuffer;
  VkDeviceMemory normalBufferMemory;
  VkBuffer       texCoordBuffer;
  VkDeviceMemory texCoordBufferMemory;
  VkBuffer       indexBuffer;
  VkDeviceMemory indexBufferMemory;
};

typedef struct shdubo_m_diff {
  Matrix model;
  r32 diffuseColor[4];
} shdubo_m_diff;

class DelayedTransformUpload {
public:
  u32 flagIdx;
  u32 numTimesUploaded;
  shdubo_m_diff instMDiff;
```

};

В листинге 4.9 приведены объявления вспомогательных классов и процедур для реализации предлагаемого метода рендеринга. Класс *Frustum* задает плоскости усеченной пирамиды камеры. Класс *RenderingParams* задает параметры рендеринга сцены.

Тип-перечисление *StratMethod* задает способ рендеринга, который используется в данный момент времени.

Процедура *strat_init* производит инициализацию внутренних объектов адаптивного метода рендеринга, выполняет тестирование производительности оборудования пользователя.

Процедура *strat_getCurMethod* возвращает используемый в данный момент способ рендеринга.

Процедура *strat_isInliningFeasible* производит оценку того, является ли запись командного буфера для каждого кадра целесообразной для данного состояния динамической сцены.

Процедура *strat_isOcclusionCullingFeasible* производит оценку того, является ли использование аппаратных проверок видимости целесообразным для данного состояния динамической сцены.

Процедура *strat_getNumOcclQueriesForDuration* возвращает количество аппаратных проверок видимости, которые можно выполнить за данный промежуток времени.

Процедура *strat_recordNodesRecursive* производит рекурсивную запись командных буферов для узлов дерева.

Процедура *strat_renderHierarchyFC* выполняет рендеринг сцены с отсечением узлов, которые не попадают в область видимости камеры.

Процедура *strat_updateForNewAnimTime* выполняет обновление командных буферов и подсчет количества видимых объектов (вершин) при изменении времени анимации.

Процедура *strat_render* выполняет рендеринг сцены с данными параметрами рендеринга. Рендеринг выполняется согласно описанию, приведенному в главе про стратегию рендеринга.

Листинг 4.9 — Объявления вспомогательных классов и процедур предлагаемой стратегии рендеринга

```
class Plane {
```

```
public:
   float p[4];
};

class Frustum {
public:
   Plane left;
   Plane right;
   Plane top;
   Plane bottom;
   Plane nearp;
   Plane farp;
};

class RenderingParams {
public:
   Scene *scene;
   Octree *h;
   Frustum *f;
   CmdOpts *cmdOpts;
};

typedef enum StratMethod {
   SA_Inlining, SA_Octree, SA_Octree_Occl
} StratMethod;

void strat_init(Scene *scene);
StratMethod strat_getCurMethod();

bool strat_isInliningFeasible(Scene *scene);
bool strat_isOcclusionCullingFeasible(Scene *scene);
u32 strat_getNumOcclQueriesForDuration(r64 durationMillis);

void strat_recordNodesRecursive(OctreeNode *node, Scene *scene);
```

```
void strat_renderHierarchyFC( Scene *scene, Octree *h,
   Frustum *frustum, OctreeNode *node );
void strat_render(RenderingParams *params);

void strat_updateForNewAnimTime(Scene *scene, u64 newTime,
   U32Array *changedInstIndices, U8Array *visibilities );
```

ГЛАВА 5. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

В данной главе приводятся результаты вычислительных экспериментов для подтверждения эффективности разработанных методов и модели производительности графического конвейера. Характеристики тестовой вычислительной системы: Intel Core i7-7700 3.6GHz, 16GB RAM, NVIDIA Geforce GTX 1070.

5.1 Тестовые наборы сцен

Для тестирования производительности методов используются реальные и синтетические сцены, которые показаны на рисунках 5.1–5.9. Их характеристики приведены в таблице 1. Тестовые наборы 1–4 содержат индустриальные сцены. Тестовые наборы 5–48 содержат синтетические сцены с различным количеством графических элементов. Краткое описание рассматриваемых сцен:

1. Тестовая сцена 1 содержит одно здание, занимающее достаточно большую площадь. Она содержит самое маленькое количество треугольников из рассматриваемых сцен.
2. Тестовая сцена 2 содержит здание на строительной площадке, а также несколько больших плоскостей с планами строительства. Половину объема сцены занимают плоскости и строительная площадка.
3. Тестовая сцена 3 содержит одно здание с большим количеством объектов и окружение, занимающее большую площадь, но содержащее маленькое количество объектов.
4. Тестовая сцена 4 содержит здание с самым большим количеством треугольников и объектов. Она является наиболее требовательной к вычислительным ресурсам CPU и GPU процессоров.
5. Тестовые наборы 5–13 содержат сцены с различным количеством параллелепипедов. Они расположены на виду, при выполнении рендеринга обрабатывается большое количество фрагментов. Это позволяет провести эксперименты с увеличением количества объектов и обрабатываемых фрагментов.
6. Тестовые наборы 14–22 содержат сцены, которые включают 256 сфер с различным количеством полигонов. Это позволяет провести эксперименты с увеличением количества вершин и полигонов при одинаковом количестве объектов.
7. Тестовые наборы 23–32 содержат сцены, которые получены с помощью предложенного метода генерации тестовых сцен. Каждая сцена содержит здания, расположенные в узлах двумерной сетки. Каждое здание заполнено

большим количеством кубов. Это позволяет провести эксперименты на эффективность проверок видимости при увеличении количества объектов.

8. Тестовые наборы 33–40 содержат сцены, которые получены с помощью предложенного метода генерации тестовых сцен. Каждая сцена содержит здания, расположенные в узлах двумерной сетки. Каждое здание заполнено большим количеством высокополигональных сфер. Это позволяет провести эксперименты на эффективность проверок видимости при увеличении количества полигонов.
9. Тестовые наборы 41–48 содержат объекты с текстурами, которые расположены в несколько рядов вблизи от камеры.

Таблица 1 — Характеристики тестовых наборов сцен

№	Количество вершин	Количество треугольников	Количество объектов
1	15037746	5012582	50521
2	32483139	10827713	71961
3	34609623	11536541	109991
4	94388454	31462818	270431
5 – 13	36000 – 6480000	12000 – 2160000	1000 – 180000
14 – 22	41472 – 30412800	13824 – 10137600	256
23 – 32	4356000 – 7596000	1452000 – 2532000	121000 – 211000
33 – 40	2613600 – 131133600	871200 – 43711200	72600 – 76800
41 – 48	144 – 13824	48 – 4608	4 – 384

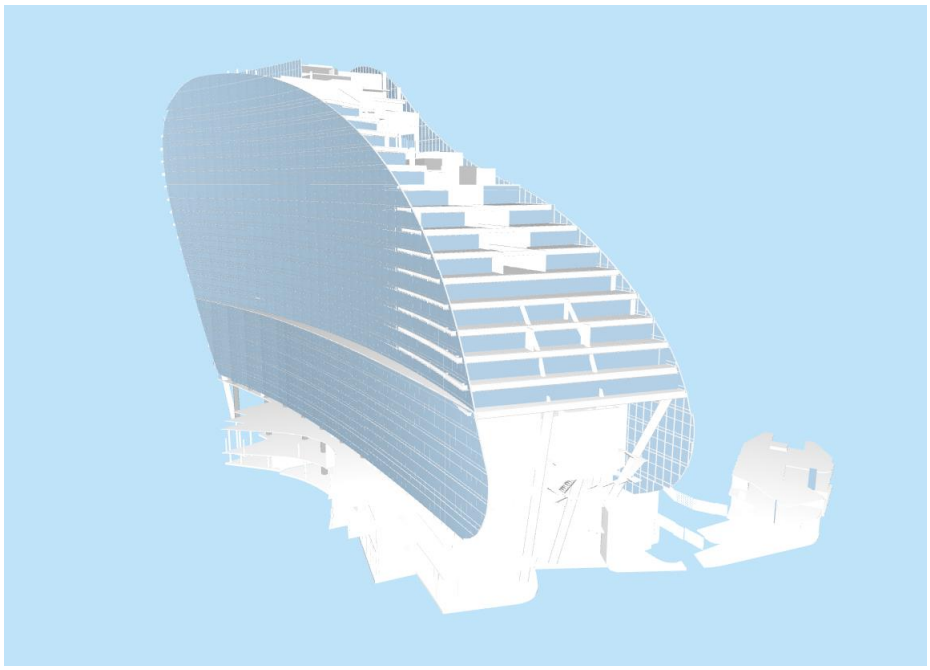

Рисунок 5.1 — Тестовая сцена 1, которая состоит из 5 миллионов треугольников.

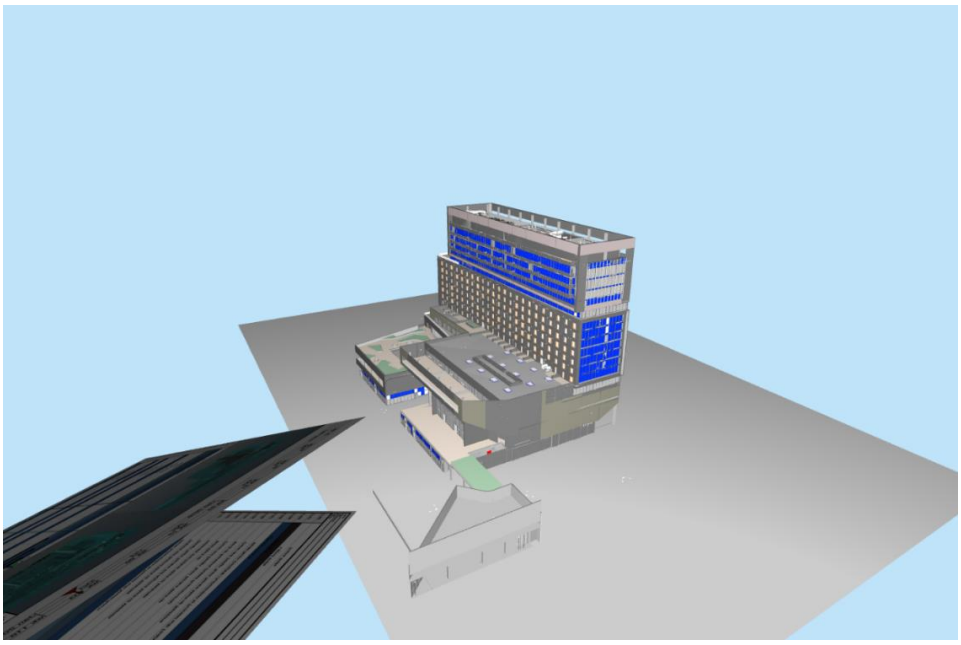

Рисунок 5.2 — Тестовая сцена 2, которая состоит из 10.8 миллионов треугольников.

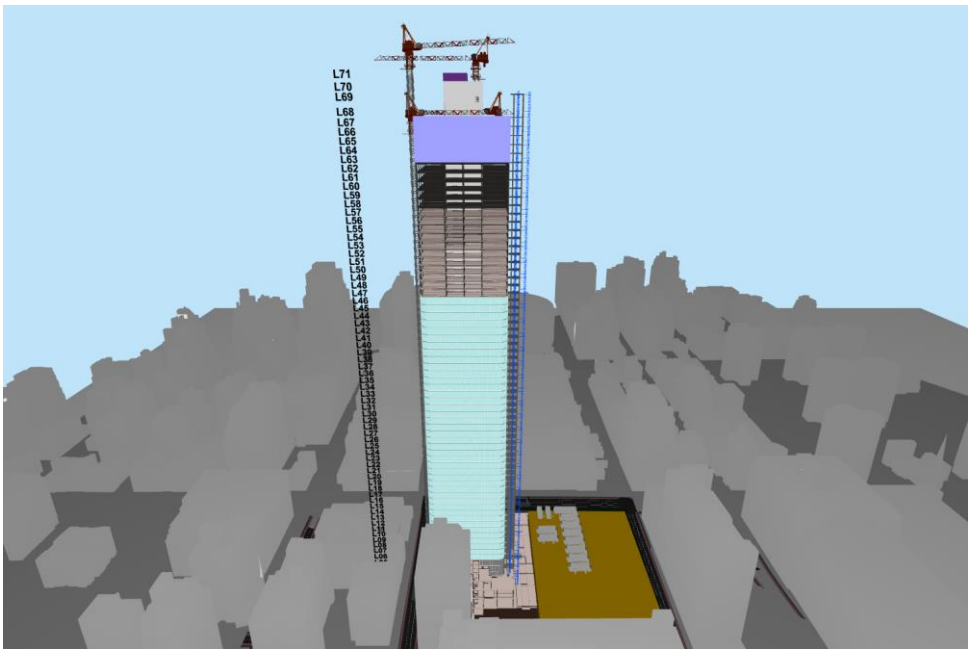

Рисунок 5.3 — Тестовая сцена 3, которая состоит из 11.5 миллионов треугольников.

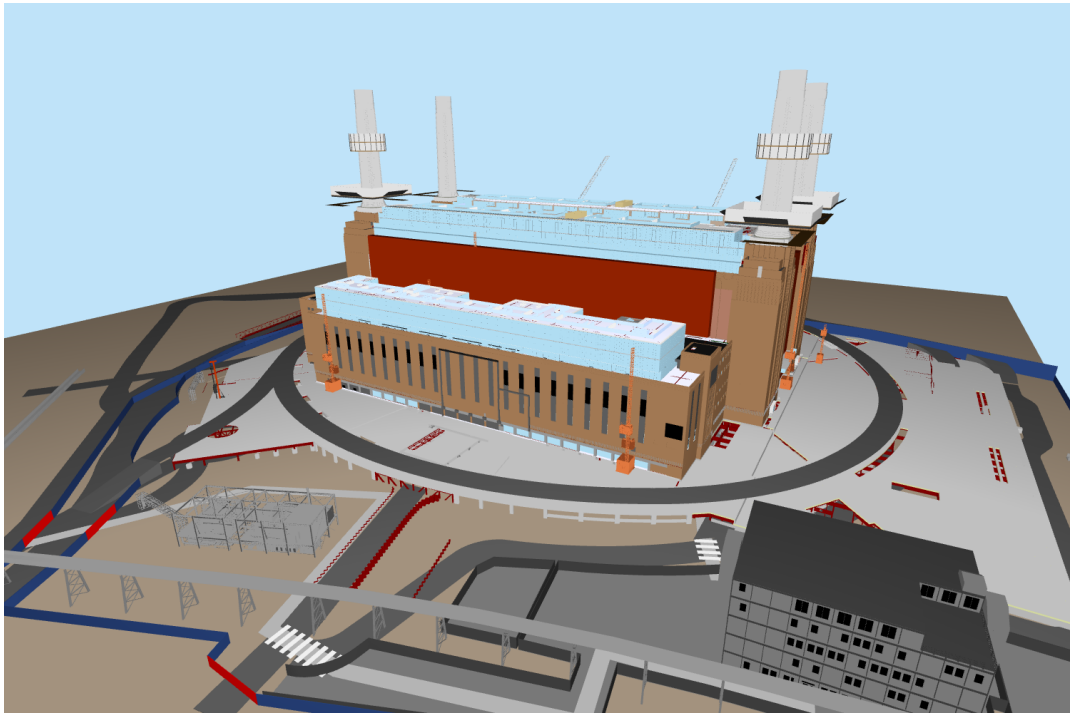

Рисунок 5.4 — Тестовая сцена 4, которая состоит из 31 миллиона треугольников.

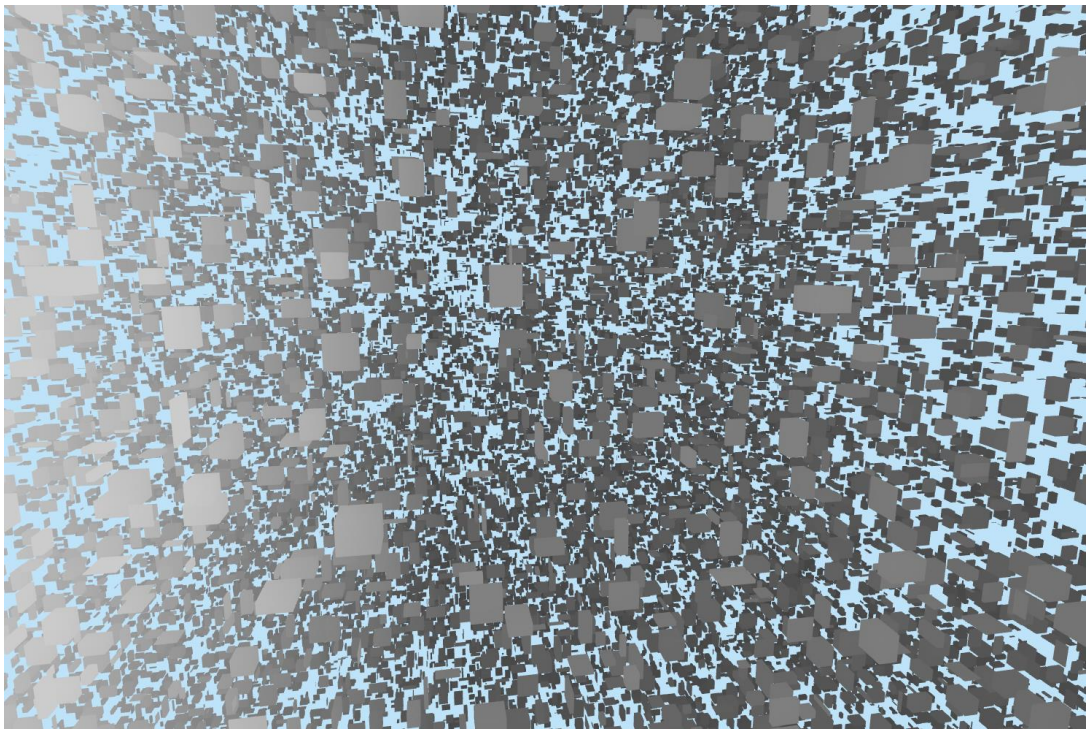

Рисунок 5.5 — Одна из тестовых сцен (тестовые наборы 5–13), содержащих большое количество кубов различного размера.

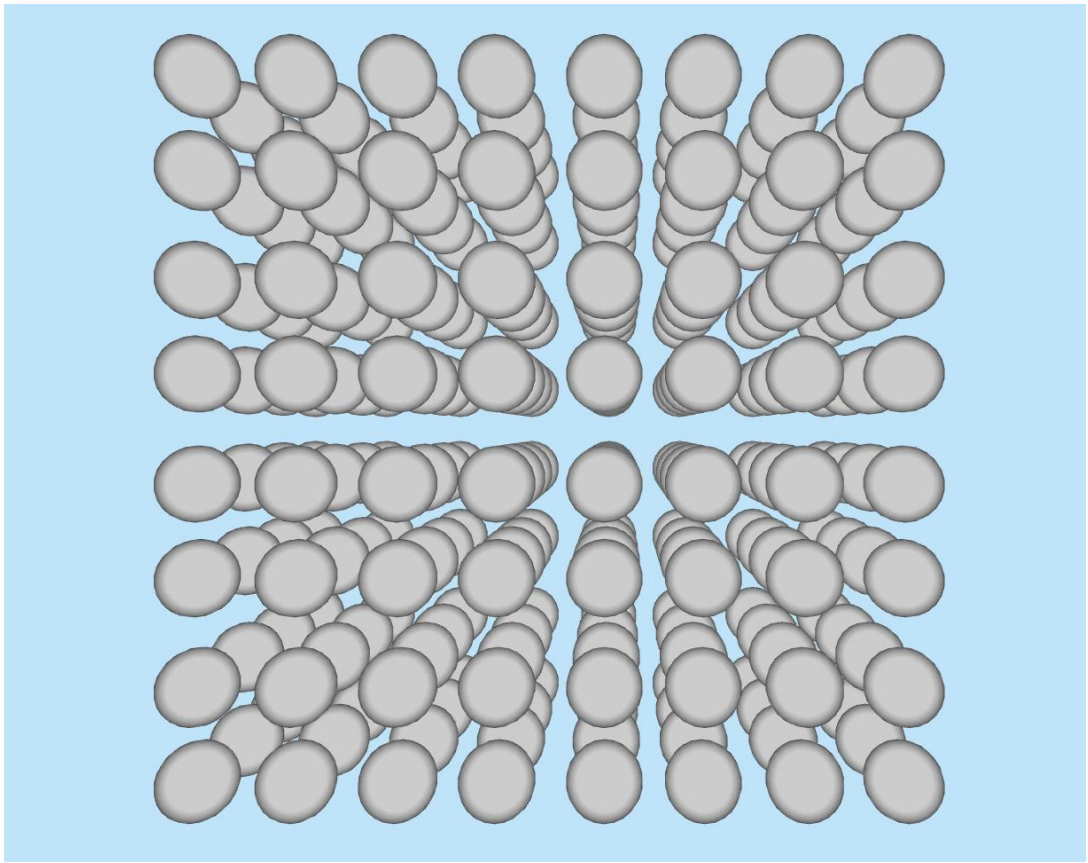

Рисунок 5.6 — Одна из тестовых сцен (тестовые наборы 14–22), содержащих сферы с большим количеством полигонов.

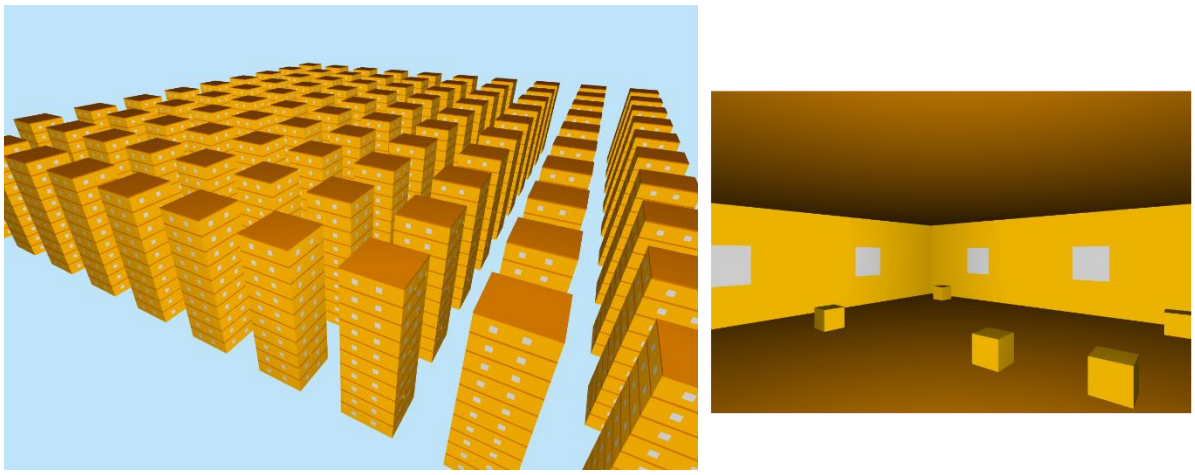

Рисунок 5.7 — Одна из тестовых сцен из зданий (тестовые наборы 23–32), содержащих большое количество кубов.

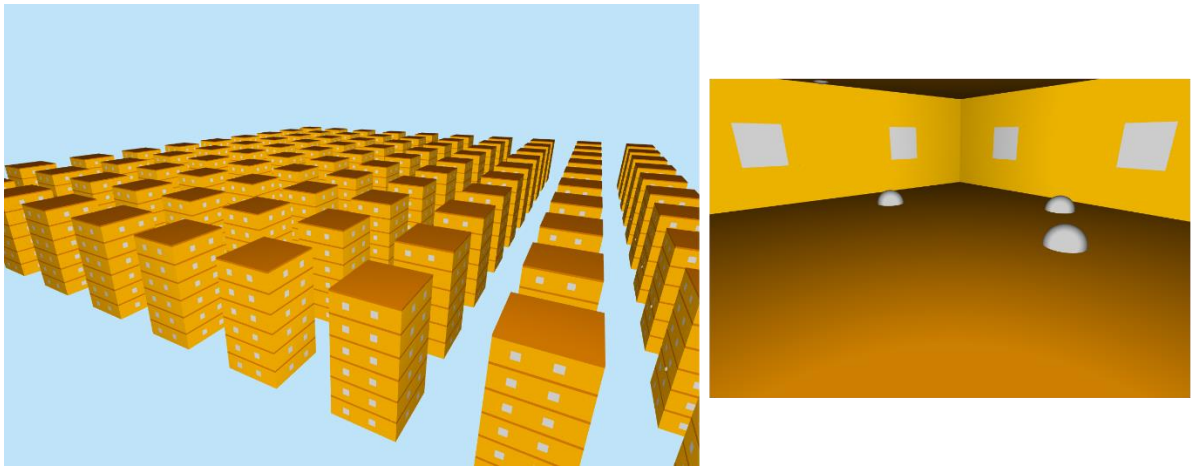

Рисунок 5.8 — Одна из тестовых сцен из зданий (тестовые наборы 33–40), содержащих большое количество высокополигональных сфер.

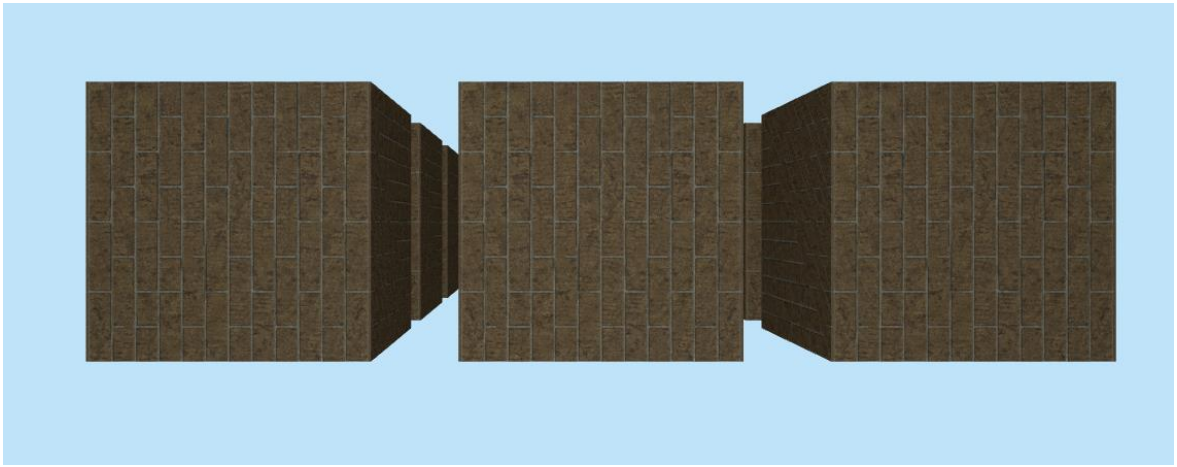

Рисунок 5.9 — Одна из тестовых сцен (тестовые наборы 41–48), содержащих объекты с текстурами.

5.2 Применение методов пространственного индексирования при удалении невидимых поверхностей

Было проведено тестирование эффективности методов пространственного индексирования при выполнении рендеринга с аппаратными проверками видимости узлов при использовании следующих методов: BVH SAH, LBVH с применением кривой Мортона, окто-дерево. Во время тестирования производилось измерение времени составления пространственного индекса, перемещение камеры по сцене, измерение среднего количества видимых объектов. Тестирование проводилось для конечного состояния тестовых сцен, когда все строительные работы завершены, и большинство объектов присутствует. Использовался следующий алгоритм рендеринга:

1. Отсечение узлов, не попадающих в область видимости камеры.
2. Загрузка результатов проверок видимости с GPU процессора.
3. Запись и отправка команд рендеринга для объектов видимых узлов.
4. Запись и отправка проверок видимости для узлов дерева.

Результаты тестирования показаны на рисунках 5.10, 5.11.

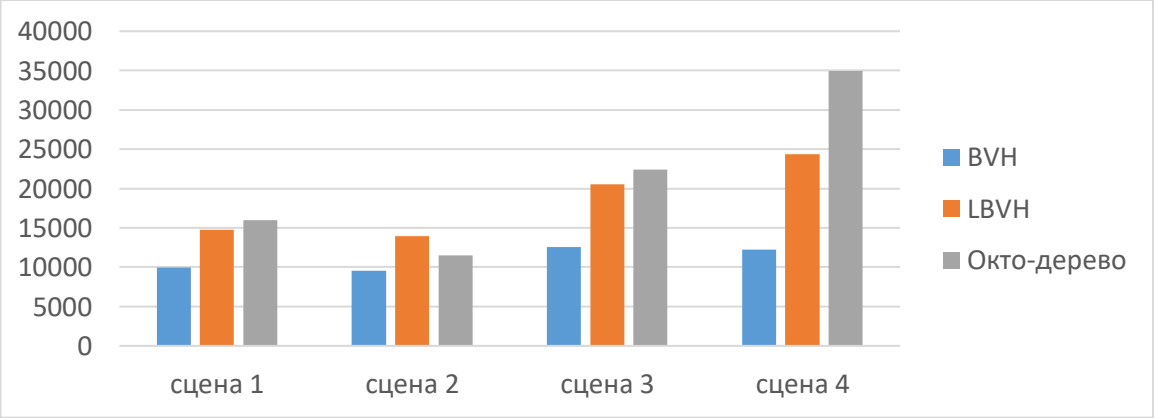

Рисунок 5.10 — Среднее количество отображаемых объектов при выполнении рендеринга с использованием BVH, LBVH, окто-дерева.

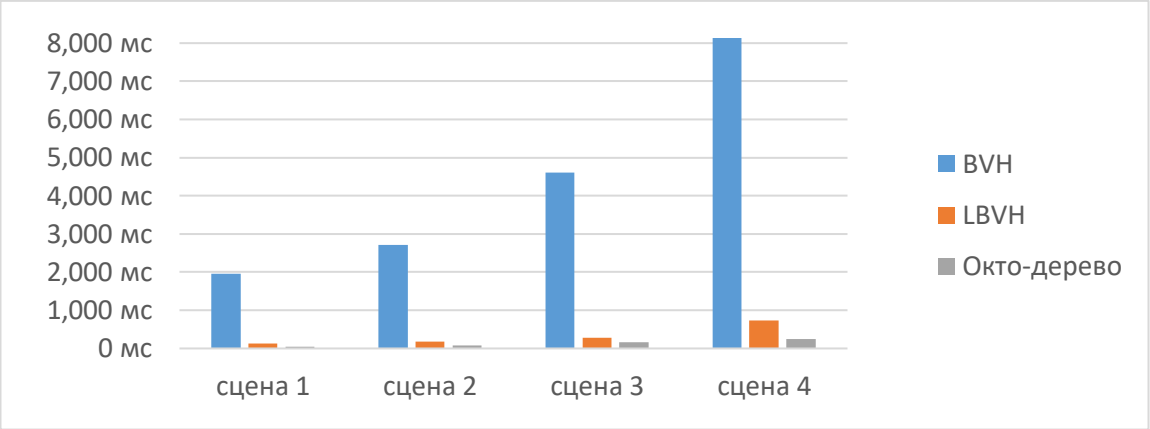

Рисунок 5.11 — Среднее время создания пространственных индексов BVH, LBVH, окто-дерева при подготовке к выполнению рендеринга сцен.

Структуры пространственного разбиения BVH, LBVH в среднем дают наименьшее количество видимых объектов благодаря наилучшей кластеризации объектов (рисунок 5.10). Необходимо учесть, что в данной работе рассматриваются динамические сцены, состояние которых изменяется со временем. В процессе отображения нужно выполнять обновление пространственных индексов. Это вызывает трудности при проведении отложенных проверок видимости. При получении результатов проверок видимости дерево может уже не содержать проверяемые узлы. Можно отложить обновление дерева до тех пор, пока не накопится достаточное количество новых объектов, но это ухудшает качество кластеризации и понижает частоту кадров в момент составления структуры пространственного разбиения.

Создание пространственного индекса оказывается в разы быстрее с использованием окто-дерева (рисунок 5.11). Также этот метод индексирования позволяет выполнять отложенные проверки видимости благодаря индексированию областей пространства, а не множества объектов. Таким образом, для пространственного индексирования объектов динамических сцен наиболее целесообразным является использование окто-дерева.

При построении окто-дерева можно использовать различные способы сохранения ссылок на объекты: хранить ссылку на объект в первом пересекающемся узле (single reference octree), хранить множество ссылок на объекты в листовых узлах (multiple reference octree). Первый способ повышает производительность рендеринга за счет сокращения количества команд при записывании буферов команд для каждого узла дерева. Второй способ повышает производительность рендеринга благодаря наилучшей кластеризации. Однако в данной работе предлагается выполнять запись командного буфера для объектов в узле дерева. При использовании

множественных ссылок происходит повторная запись одних и тех же команд в разные буфера. Из-за этого выполняется отображение большого количества лишних объектов. Было проведено тестирование для определения влияния этого эффекта на производительность при отображении индустриальных сцен (сцены 1–4). Создается окто-дерево с заданной высотой. Выполняется обход дерева с отсечением узлов, не попадающих в область видимости камеры. В главный буфер записываются ссылки на командные буфера видимых узлов. Измеряется количество команд для отображения объектов. Затем этот буфер отправляется на GPU для выполнения рендеринга. Результаты показали, что использование множественных ссылок значительно ухудшает производительность рендеринга при увеличении высоты окто-дерева (рисунки 5.12 – 5.15). Рендеринг такого количества лишних объектов является неоправданным. Таким образом, использование окто-дерева без оптимизаций (single reference octree) является более целесообразным в данной работе.

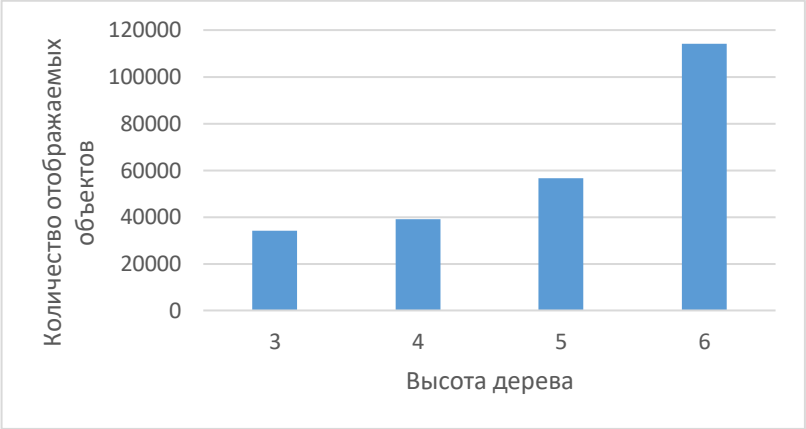

Рисунок 5.12 — Среднее количество отображаемых объектов в зависимости от высоты дерева при выполнении рендеринга сцены 1 с использованием окто-дерева с множественными ссылками.

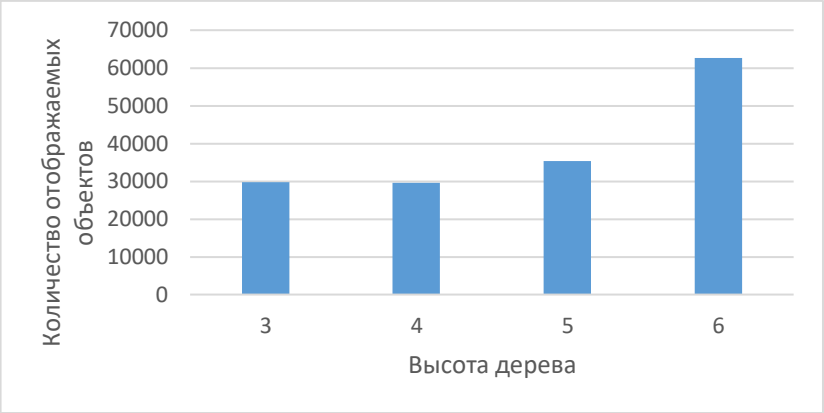

Рисунок 5.13 — Среднее количество отображаемых объектов в зависимости от высоты дерева при выполнении рендеринга сцены 2 с использованием октодерева с множественными ссылками.

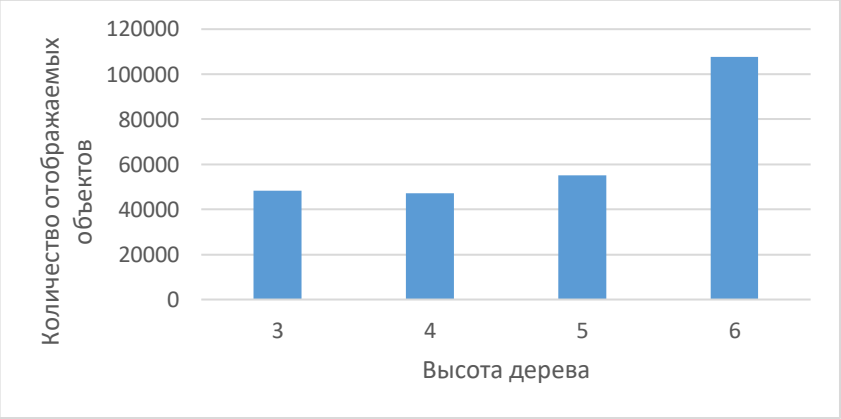

Рисунок 5.14 — Среднее количество отображаемых объектов в зависимости от высоты дерева при выполнении рендеринга сцены 3 с использованием октодерева с множественными ссылками.

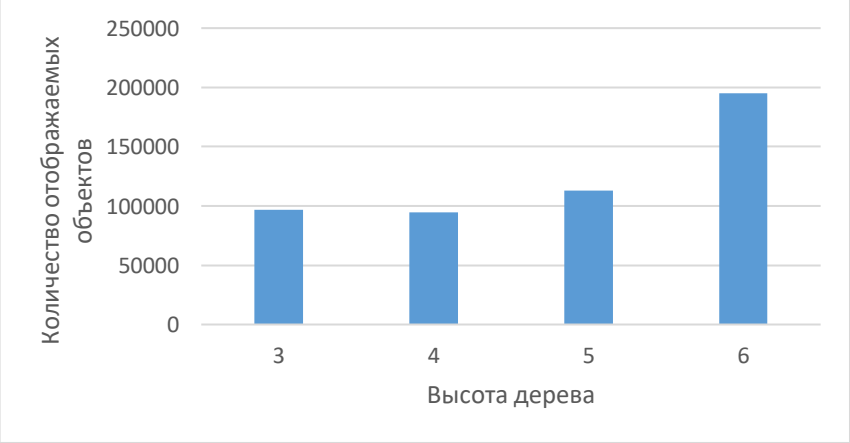

Рисунок 5.15 — Среднее количество отображаемых объектов в зависимости от высоты дерева при выполнении рендеринга сцены 4 с использованием октодерева с множественными ссылками.

5.3 Анализ эффективности аппаратных проверок видимости

Для определения эффективности проверок видимости проводится накопление данных о видимости объектов, а также других элементов сцены. Для этого в разделе 3.2 предлагается выполнять аппаратные проверки видимости для затратных узлов дерева. В этом разделе показана эффективность выполнения проверок видимости при увеличении количества проверок. Проводится генерация сцены, содержащей 300000 случайно расположенных параллелепипедов. Выполняется заданное количество проверок видимости при обходе окто-дерева в ширину. Измеряется количество невидимых объектов и временные затраты на проверки видимости. В таблице 2 приведены результаты тестирования при увеличении количества проверок видимости. Показано, что большинство невидимых объектов можно вычислить с помощью ограниченного количества проверок видимости. Это достигается благодаря иерархическому выполнению проверок видимости при обходе дерева в ширину. Таким образом, за ограниченное время можно получить консервативную оценку количества невидимых объектов в сцене.

Таблица 2 — Количество невидимых объектов в зависимости от количества проверяемых узлов

Количество проверяемых узлов	Количество невидимых объектов	Время выполнения проверок видимости (миллисек.)
10	497	0.0017
20	497	0.0033
30	9731	0.005
100	45276	0.017
200	54644	0.033
300	77209	0.050
1000	121419	0.166
2000	127399	0.33
3000	142498	0.49
10000	148955	1.65
50000	163287	8.23

5.4 Анализ производительности составления и отправки командных буферов

В этом разделе выполняется тестирование производительности составления и отправки командных буферов с использованием программного интерфейса рендеринга Vulkan. Создается сцена с заданным количеством случайно расположенных кубов. Выполняется запись главного командного буфера одним из трех способов и отправка на GPU. Используются следующие способы записи буферов:

- Запись команд для каждого объекта (без кэширования). При этом большая часть времени тратится на запись команд на CPU, в то время как GPU процессор простаивает (таблица 3).
- Кэширование вспомогательных буферов. Выполняется запись одной команды во вспомогательный буфер. При этом большая часть времени тратится на отправку буферов. Это связано с тем, что минимальный размер буфера в памяти может быть намного больше размера команды. Большая часть буфера остается свободной, но, тем не менее, отправляется на GPU (таблица 4).
- Кэширование вспомогательных буферов. Выполняется запись 1000 команд во вспомогательный буфер. Увеличение количества команд во вспомогательном буфере улучшает производительность — сокращается время записи, расход памяти уменьшается (таблица 5).

Результаты показаны для постепенно увеличивающегося количества объектов в сцене. Таким образом, для сокращения затрат при записи буферов можно кэшировать вспомогательные буфера с командами рендеринга. Определение количества команд в буфере для наиболее эффективного рендеринга выполняется в предлагаемом методе построения окто-дерева (раздел 3.1).

Таблица 3 — Затраты при записи команд без кэширования

Количество объектов	Время записи буфера, мс	Время отправки буфера, мс	Потребление памяти, МБ
10000	1.8	0.03	183
50000	9.6	0.05	239
100000	19.6	0.06	310
300000	55.5	0.07	641

Таблица 4 — Затраты при использовании одного вспомогательного буфера на объект

Количество объектов	Время записи буфера, мс	Время отправки буфера, мс	Потребление памяти, МБ
10000	0.15	1.4	324
50000	0.9	7.0	1008
100000	1.9	14.5	1868
300000	5.8	43.0	5313

Таблица 5 — Затраты при использовании одного вспомогательного буфера на 1000 объектов

Количество объектов	Время записи буфера, мс	Время отправки буфера, мс	Потребление памяти, МБ
10000	0.02	0.08	160
50000	0.02	0.10	189
100000	0.03	0.12	228
300000	0.03	0.18	400

Отметим, что высокое потребление памяти связано с командными буферами, а не геометрией или текстурами. Для нашей тестовой системы размер командного буфера равен 16КБ, а 16ГБ оперативной памяти может вместить примерно 1000000 командных буферов. Однако другие конфигурации со встроенной графикой допускают только 60000 командных буферов (1ГБ памяти).

Рассмотрим технику записи командного буфера на GPU процессоре в вычислительном шейдере (compute shader). Для этого создается DrawIndirect буфер в видеопамяти, который содержит команду для каждого объекта: начальный индекс, количество индексов, количество экземпляров. Индексы задают геометрию объекта в текущем вершинном буфере. Если количество экземпляров устанавливается равным нулю, то объект не отображается. В вычислительном шейдере выполняется обход всех объектов, проверка попадания в область видимости камеры и установка количества экземпляров (видимости объекта). За счет параллелизации проверок видимости и обращений в память достигается высокая скорость выполнения.

Проведем сравнение времени рендеринга при выполнении записи буфера на CPU и GPU процессорах с использованием техник:

- запись DrawIndirect буфера на GPU,

- запись главного командного буфера на CPU,
- кэширование на CPU и отправка подготовленных буферов.

Программная реализация с использованием DrawIndirect буфера была взята с сайта [94]. Создается сцена, в которой объекты расставлены в узлах сетки. Тестирование проводится при увеличении размеров сетки. Результаты тестирования показаны в таблице 6. Выполнение записи буфера на GPU оказывается быстрее благодаря параллелизации. Однако эта техника может столкнуться с трудностями при обработке объектов больших динамических сцен. В данной работе предлагается выполнять проверки видимости и запись команд для узлов дерева. Выделение кластеров близких объектов позволяет сократить объем вычислений при возрастании количества объектов. Результаты с применением техники записи для узлов октодерева (четвертая колонка таблицы) показывают, что за счет кэширования буферов для узлов дерева повышается эффективность рендеринга с ростом количества объектов.

Таблица 6 — Сравнение времени рендеринга при использовании техник записи командного буфера на GPU и CPU

Количество объектов	Время рендеринга с использованием буфера DrawIndirect, мс	Время рендеринга с выполнением записи на CPU, мс	Время рендеринга с выполнением записи на CPU с применением кэширования, мс
8000	0.8	2.16	3.01
27000	1.85	3.13	4.17
64000	3.95	6.81	5.6
125000	7.35	13.14	8.95
216000	12.5	21.6	13.93

5.5 Тестирование модели производительности графического конвейера

Для тестирования предложенной модели производительности использовались синтетические и реальные наборы данных (сцены 1–48). Синтетические наборы данных содержат несколько видов объектов, которые расположены случайным образом либо составляют каркас зданий. Реальные сцены содержат простые и сложные объекты в разных пространственных конфигурациях. Тестирование проводилось для конечного состояния динамических сцен. Эти сцены позволяют оценить точность модели производительности в условиях разнородной нагрузки на

CPU и GPU процессоры. В рамках тестирования проводилось измерение среднего времени рендеринга за несколько кадров и вычисление времени по формуле (2.12), посчитано отклонение от точного времени.

Проведем тестирование с использованием способа рендеринга с записью команд без кэширования. Проводилось отсечение объектов, не попадающих в область видимости камеры, а затем выполнялась запись командного буфера. Время работы рассчитывалось по формуле $T(S, S_{vis}, \emptyset, \emptyset, \emptyset, \emptyset, h = 0, g = 0, q = 0, HW)$. Ниже приведены результаты тестирования с использованием тестовых сцен 1–48.

Таблица 7 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования (тестовые сцены 1–4)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
50521	15037746	15.16	12.92	15
71961	32483139	20.75	20.28	2
109991	34609623	19.81	26.76	35
270431	94388454	69.87	65.75	6

Таблица 8 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования (тестовые сцены 5–13)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
1000	36000	1.47	1.68	14
10000	360000	2.82	3.20	14
40000	1440000	6.67	8.67	30
60000	2160000	9.91	12.26	24
80000	2880000	12.45	15.84	27
100001	3600036	14.97	19.40	30

120000	4320000	16.99	23.02	35
150000	5400000	21.99	28.28	29
180000	6480000	24.58	34.31	40

Таблица 9 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования (тестовые сцены 14–22)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
256	41472	1.79	1.96	9
256	172032	1.83	2.00	9
256	337920	1.92	2.15	12
256	860160	2.01	2.13	6
256	1554432	2.14	2.24	5
256	4611072	2.84	2.93	3
256	15813120	5.05	5.15	2
256	9461760	3.70	3.82	3
256	30412800	8.30	8.50	2

Таблица 10 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования (тестовые сцены 23–32)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
121000	4356000	27.91	24.12	14
131000	4716000	30.32	25.95	14
141000	5076000	30.24	27.74	8
151000	5436000	31.76	29.78	6
161000	5796000	39.79	31.91	20
171000	6156000	35.61	34.07	4

181000	6516000	44.58	36.27	19
191000	6876000	46.55	38.27	18
201000	7236000	48.84	40.19	18
211000	7596000	50.34	42.12	16

Таблица 11 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования (тестовые сцены 33–40)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
72600	2613600	16.67	15.07	10
73200	20973600	19.17	17.67	8
73800	39333600	21.68	21.92	1
74400	57693600	24.40	26.20	7
75000	76053600	27.61	30.39	10
75600	94413600	30.31	34.66	14
76200	112773600	33.34	38.90	17
76800	131133600	35.97	43.15	20

Таблица 12 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования (тестовые сцены 41–48)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
4	144	2.23	1.81	19
8	288	1.95	2.08	6
16	576	1.93	2.37	23
32	1152	1.90	2.38	25
64	2304	2.16	2.23	3
96	3456	2.13	2.75	29

192	6912	2.20	2.78	27
384	13824	2.14	2.82	32

Проведем тестирование с использованием способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта. Проводилось получение результатов проверок видимости, отсечение объектов, не попадающих в область видимости камеры, а затем выполнялась запись командного буфера и отправка новых проверок видимости для каждого объекта. Время работы рассчитывалось по формуле $T(S, S_{vis}, \emptyset, \emptyset, \emptyset, S_{vis}, h = 0, g = 0, q = 1, HW)$. Ниже приведены результаты тестирования с использованием тестовых сцен 1–48.

Таблица 13 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта (тестовые сцены 1–4)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
50521	15037746	19.53	19.31	1
71961	32483139	24.09	24.61	2
109991	34609623	41.01	45.58	11
270431	94388454	83.14	92.75	12

Таблица 14 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта (тестовые сцены 5–13)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
1000	36000	1.65	1.91	16
10000	360000	4.88	5.29	8
40000	1440000	16.68	17.29	4
60000	2160000	23.19	25.03	8
80000	2880000	30.03	32.63	9

100001	3600036	36.70	39.88	9
120000	4320000	47.98	46.94	2
150000	5400000	51.93	56.64	9
180000	6480000	61.99	65.70	6

Таблица 15 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта (тестовые сцены 14–22)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
256	41472	2.15	2.03	6
256	172032	2.17	2.07	5
256	337920	2.32	2.21	5
256	860160	2.48	2.19	12
256	1554432	2.55	2.30	10
256	4611072	3.26	3.00	8
256	15813120	5.41	5.21	4
256	9461760	4.19	3.88	7
256	30412800	8.83	8.56	3

Таблица 16 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта (тестовые сцены 23–32)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
121000	4356000	35.74	38.70	8
131000	4716000	37.51	41.63	11
141000	5076000	43.18	44.57	3
151000	5436000	42.94	47.47	11
161000	5796000	44.96	50.42	12

171000	6156000	47.36	53.30	13
181000	6516000	57.89	56.22	3
191000	6876000	53.44	59.16	11
201000	7236000	54.85	62.07	13
211000	7596000	66.86	65.02	3

Таблица 17 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта (тестовые сцены 33–40)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
72600	2613600	22.49	23.83	6
73200	20973600	22.47	24.35	8
73800	39333600	22.49	24.58	9
74400	57693600	24.50	25.36	4
75000	76053600	24.06	25.78	7
75600	94413600	24.06	26.11	9
76200	112773600	25.14	26.30	5
76800	131133600	23.83	26.37	11

Таблица 18 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с записью команд без кэширования и аппаратными проверками видимости каждого объекта (тестовые сцены 41–48)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
4	144	1.87	1.65	12
8	288	1.87	2.05	9
16	576	1.91	2.05	7
32	1152	1.92	1.72	10
64	2304	2.26	2.24	1

96	3456	2.08	1.99	5
192	6912	2.12	2.18	3
384	13824	2.04	1.81	11

Проведем тестирование с использованием способа рендеринга с кэшированием команд для узлов окто-дерева. Проводилось отсечение узлов, не попадающих в область видимости камеры, а затем выполнялась отправка вспомогательных командных буферов для видимых узлов дерева. Время работы рассчитывалось по формуле $T(S, S_{vis}, H, H_{vis}, H_{upd}, \emptyset, h = 1, g = 1, q = 0, HW)$. Ниже приведены результаты тестирования с использованием тестовых сцен 1–48.

Таблица 19 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд для узлов окто-дерева (тестовые сцены 1–4)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
50521	15037746	9.83	9.17	7
109991	34609623	37.82	31.26	17
270431	94388454	60.14	55.76	7
221796	30462912	16.97	14.54	14

Таблица 20 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд для узлов окто-дерева (тестовые сцены 5–13)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
1000	36000	1.90	1.99	5
10000	360000	4.76	4.14	13
40000	1440000	18.27	13.77	25
60000	2160000	23.67	17.52	26
80000	2880000	29.33	21.82	26

100001	3600036	35.10	26.41	25
120000	4320000	40.92	31.14	24
150000	5400000	50.04	38.48	23
180000	6480000	59.31	46.67	21

Таблица 21 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд для узлов окто-дерева (тестовые сцены 14–22)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
256	41472	1.98	2.03	3
256	172032	2.10	2.08	1
256	337920	2.32	2.22	4
256	860160	2.34	2.20	6
256	1554432	2.45	2.31	6
256	4611072	3.16	3.00	5
256	15813120	5.02	5.21	4
256	9461760	3.79	3.89	3
256	30412800	8.31	8.57	3

Таблица 22 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд для узлов окто-дерева (тестовые сцены 23–32)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
121000	4356000	7.17	7.86	10
131000	4716000	10.10	10.36	3
141000	5076000	11.11	11.26	1
151000	5436000	11.77	12.15	3
161000	5796000	12.29	12.93	5

171000	6156000	12.64	13.60	8
181000	6516000	13.04	14.36	10
191000	6876000	13.20	14.75	12
201000	7236000	13.55	15.14	12
211000	7596000	13.95	15.61	12

Таблица 23 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд для узлов окто-дерева (тестовые сцены 33–40)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
72600	2613600	4.09	4.70	15
73200	20973600	6.67	7.39	11
73800	39333600	9.64	11.65	21
74400	57693600	12.66	15.88	25
75000	76053600	15.59	20.09	29
75600	94413600	18.46	24.31	32
76200	112773600	21.24	28.52	34
76800	131133600	24.18	32.74	35

Таблица 24 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд для узлов окто-дерева (тестовые сцены 41–48)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
4	144	2.12	1.71	19
8	288	1.89	1.71	10
16	576	1.86	1.71	8
32	1152	1.90	1.71	10
64	2304	2.10	1.64	22

96	3456	2.08	1.62	22
192	6912	2.23	1.62	27
384	13824	1.96	1.62	18

Проведем тестирование с использованием способа рендеринга с фрагментацией и кэшированием команд для узлов окто-дерева и с аппаратными проверками видимости узлов. Проводилось получение результатов проверок видимости, отсечение узлов, не попадающих в область видимости камеры, а затем выполнялась отправка вспомогательных командных буферов для видимых узлов дерева и отправка новых проверок видимости для узлов дерева. Время работы рассчитывалось по формуле $T\left(S, S_{vis}, H, H_{vis}, H_{upd}, H_{vis}, h = 1, g = 1, q = 1, HW\right)$. Ниже приведены результаты тестирования с использованием тестовых сцен 1–48.

Таблица 25 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд и с аппаратными проверками видимости узлов окто-дерева (тестовые сцены 1–4)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
50521	15037746	11.64	11.20	4
71961	32483139	40.80	31.40	23
109991	34609623	55.49	50.12	10
270431	94388454	87.75	76.86	12

Таблица 26 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд и с аппаратными проверками видимости узлов окто-дерева (тестовые сцены 5–13)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
1000	36000	2.38	2.35	1
10000	360000	8.10	5.98	26
40000	1440000	30.81	23.14	25

60000	2160000	39.02	29.15	25
80000	2880000	48.36	36.70	24
100001	3600036	57.48	45.02	22
120000	4320000	67.35	53.55	20
150000	5400000	81.00	66.36	18
180000	6480000	93.86	78.65	16

Таблица 27 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд и с аппаратными проверками видимости узлов окто-дерева (тестовые сцены 14–22)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
256	41472	2.06	2.05	0
256	172032	2.13	2.09	2
256	337920	2.41	2.24	7
256	860160	2.42	2.22	8
256	1554432	2.57	2.33	9
256	4611072	3.28	3.02	8
256	15813120	5.14	5.23	2
256	9461760	3.94	3.91	1
256	30412800	8.53	8.59	1

Таблица 28 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд и с аппаратными проверками видимости узлов окто-дерева (тестовые сцены 23–32)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
121000	4356000	6.23	6.79	9
131000	4716000	11.12	11.28	1
141000	5076000	12.34	12.51	1

151000	5436000	12.97	13.19	2
161000	5796000	13.24	13.51	2
171000	6156000	13.39	13.56	1
181000	6516000	13.36	13.78	3
191000	6876000	13.45	13.87	3
201000	7236000	13.77	14.01	2
211000	7596000	13.99	14.22	2

Таблица 29 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд и с аппаратными проверками видимости узлов окто-дерева (тестовые сцены 33–40)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
72600	2613600	3.50	4.00	14
73200	20973600	5.04	5.59	11
73800	39333600	6.55	7.54	15
74400	57693600	7.97	9.44	19
75000	76053600	9.44	11.73	24
75600	94413600	10.72	13.53	26
76200	112773600	11.93	15.34	29
76800	131133600	13.41	17.41	30

Таблица 30 — Сравнение измеренного и рассчитанного времени рендеринга при использовании способа рендеринга с кэшированием команд и с аппаратными проверками видимости узлов окто-дерева (тестовые сцены 41–48)

Количество объектов	Количество вершин	Измеренное время рендеринга (миллисек.)	Рассчитанное время рендеринга (миллисек.)	Отклонение (%)
4	144	2.15	1.71	20
8	288	1.87	1.71	9
16	576	1.90	1.71	10

32	1152	1.85	1.71	8
64	2304	2.09	1.64	22
96	3456	2.09	1.62	23
192	6912	2.22	1.62	27
384	13824	2.00	1.62	19

Таким образом, результаты тестирования предложенной модели производительности рендеринга показали, что ее можно применять для получения достаточно точных оценок времени рендеринга при использовании разнообразных сцен и способов рендеринга.

5.6 Анализ производительности адаптивной стратегии рендеринга

Тестирование проводится на четырех динамических сценах (сцены 1–4), характеристики которых приведены в таблице 1. В процессе анимации сцен происходит добавление новых объектов, удаление временных объектов, изменение цвета объектов. Количество объектов в тестовых сценах растет со временем.

Тестирование заключается в отображении сцены с фиксированным положением камеры, проигрывании анимации и измерении времени рендеринга с применением одного из трех способов реализации рендеринга:

- Frustum Culling. Выполняется обход окто-дерева с отсечением узлов, не попадающих в камеру. Проводится запись команд в главный командный буфер без кэширования. Аппаратные проверки видимости не используются.
- Occlusion queries and FC. Выполняются аппаратные проверки видимости относительно буфера глубины на GPU процессоре для всех узлов окто-дерева. Получение результатов проверок видимости происходит с задержкой в несколько кадров для избежания простоя центрального процессора. Также применяется разработанная техника записи команд с использованием кэширования и фрагментации буферов. Таким образом, по сравнению с предыдущим способом добавляется кэширование буферов и аппаратные проверки видимости.
- Предложенная адаптивная стратегия.

При проведении рендеринга использовались два кадровых буфера (double buffering), один главный командный буфер, выполнялся вывод на экран последнего

записанного кадрового буфера (mailbox), применялось сглаживание, запись командных буферов осуществлялась в однопоточном режиме. Программная реализация написана с использованием графической библиотеки Vulkan.

Методы отбраковки объектов с использованием усеченной пирамиды видимости (view frustum) и аппаратных проверок видимости (occlusion queries) широко используются в индустрии. Применение первого способа имеет смысл, когда нет необходимости в проверках видимости. Применение второго способа имеет смысл, когда сцена является достаточно сложной и проверки видимости могут дать прирост производительности. Таким образом, сравнение позволяет показать целесообразность применения различных способов рендеринга в рамках стратегии в зависимости от состояния сцены и производительности вычислительной системы.

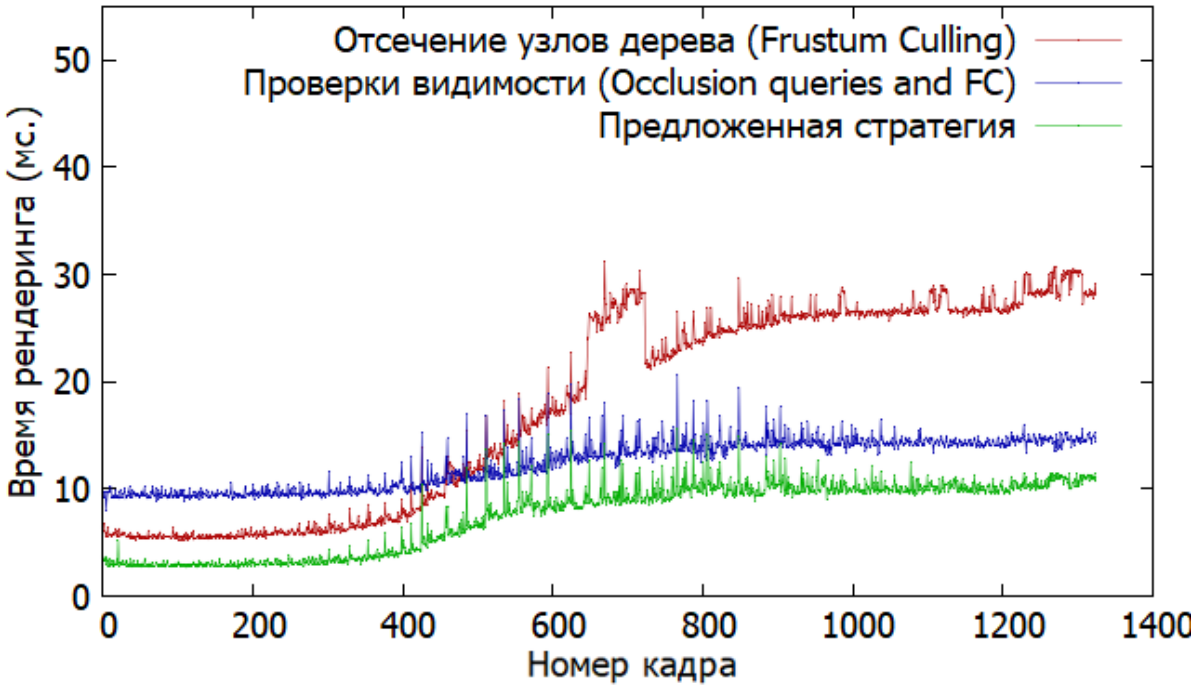

Рисунок 5.16 — Сравнение времени рендеринга с использованием рассматриваемых способов при отображении сцены 1 и одновременном проигрывании анимации.

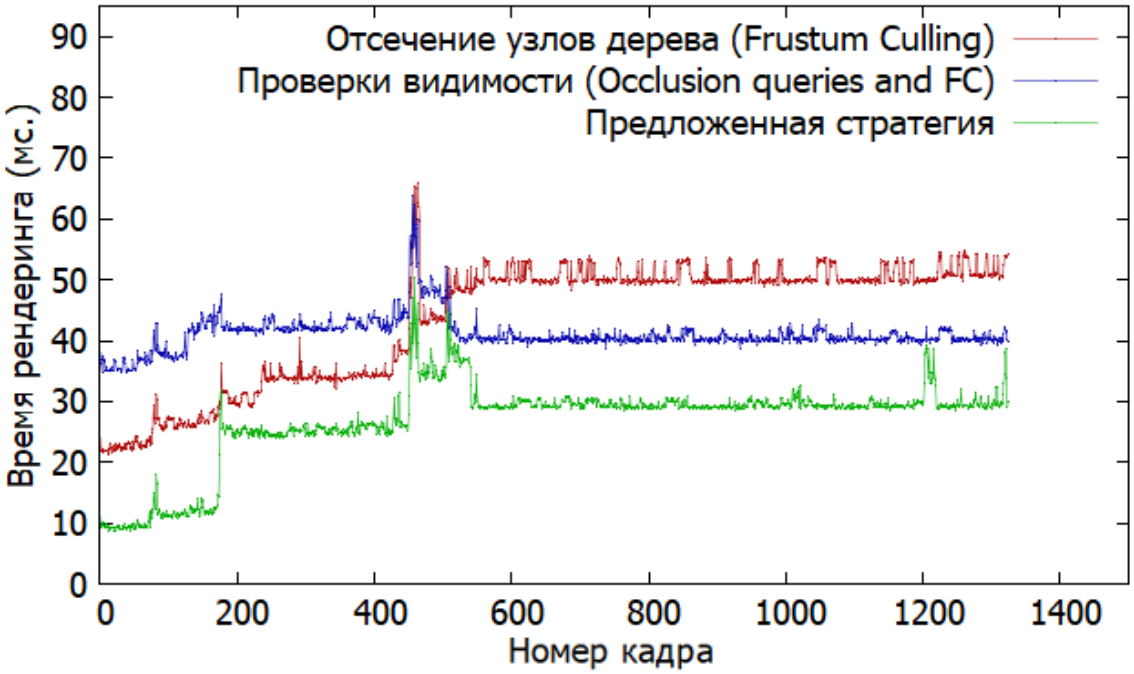

Рисунок 5.17 — Сравнение времени рендеринга с использованием рассматриваемых способов при отображении сцены 2 и одновременном проигрывании анимации.

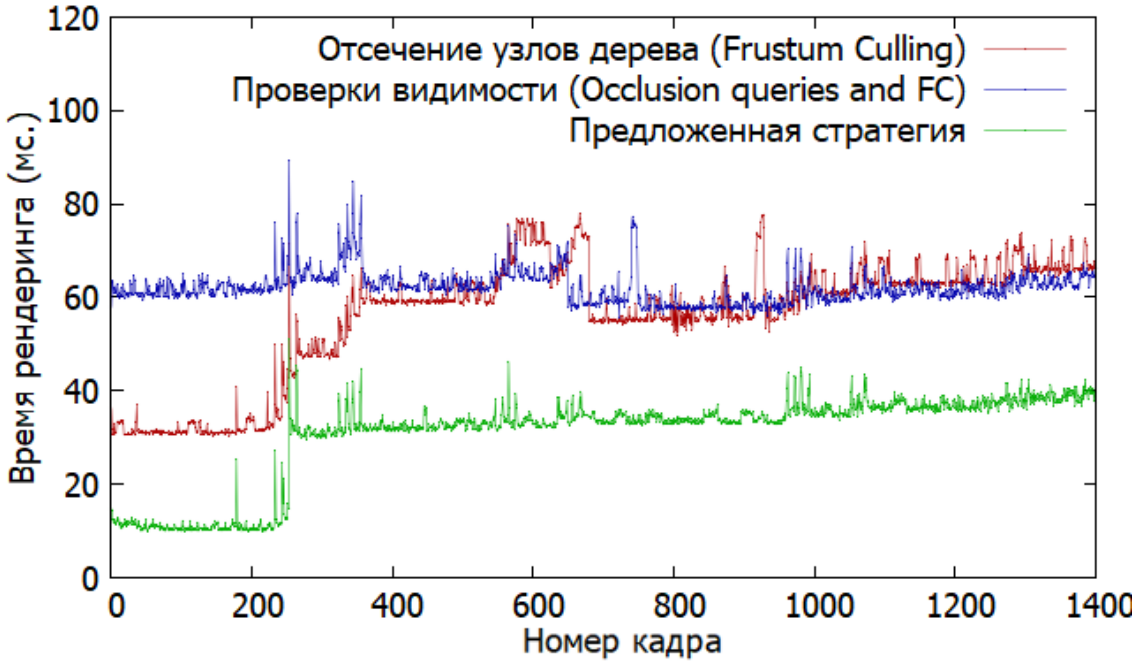

Рисунок 5.18 — Сравнение времени рендеринга с использованием рассматриваемых способов при отображении сцены 3 и одновременном проигрывании анимации.

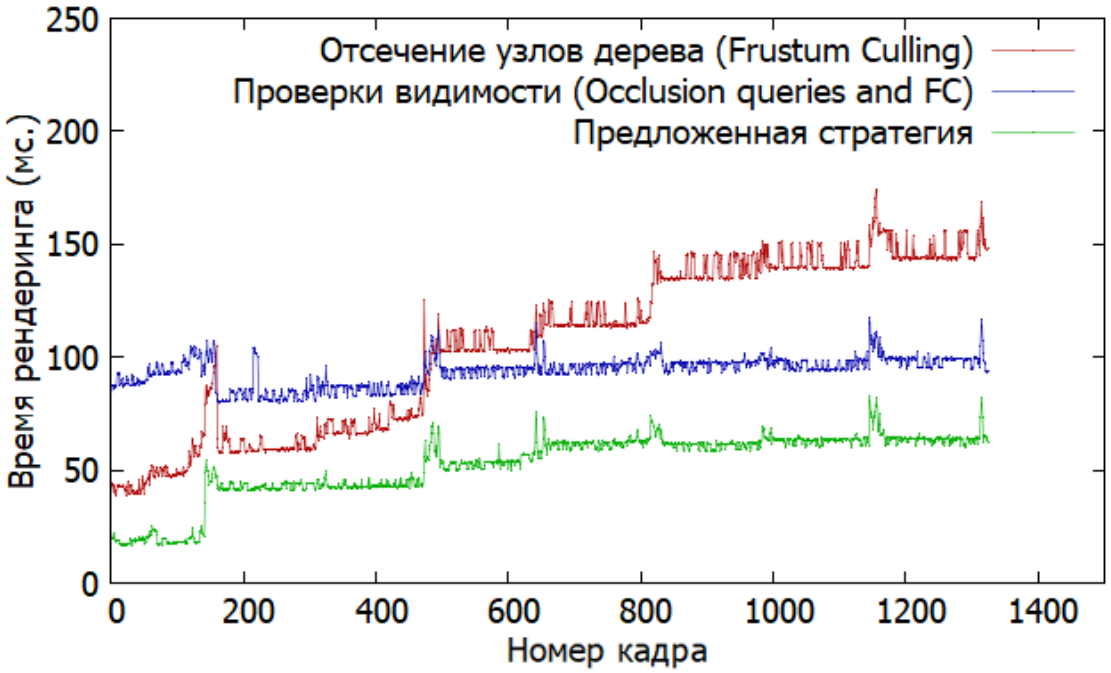

Рисунок 5.19 — Сравнение времени рендеринга с использованием рассматриваемых способов при отображении сцены 4 и одновременном проигрывании анимации.

Таблица 31 — Время выполнения основных этапов рендеринга сцены 4 на момент окончания анимации

Способ	Время записи и отправки буфера команд, мс.	Время преобразования вершин, мс.	Время выполнения и получения результатов проверок видимости, мс.
Frustum Culling	47.1	22.9	-
Occlusion queries and FC	28.3	13.1	8.6
Предложенная стратегия	2.8	13.8	1.1

Результаты тестирования производительности рендеринга сцен 1–4 приведены на графиках 5.16–5.19. В таблице 31 приведено время выполнения этапов рендеринга сцены 4, когда анимация завершилась.

Способ “Frustum Culling” работает эффективно при малом количестве объектов, а при увеличении количества объектов время записи командного буфера отнимает значительное время.

Способ “Occlusion queries and FC” работает эффективно при отображении сцен 1, 2, 4. На графиках есть переломный момент, когда проверки видимости начинают давать прирост производительности. Проверки видимости оказываются неэффективными при отображении сцены 3, потому что она содержит одно здание, большинство объектов которого всегда видны.

В рамках адаптивной стратегии применяются различные способы рендеринга по мере обработки событий об изменении видимости объектов. В начале анимации сцен используется способ с отсечением объектов и записью командных буферов без кэширования. Когда запись команд отнимает значительное время, происходит смена способа рендеринга. Затем проводится накопление данных о количестве видимых графических элементов и объектов, вычисляется количество проверок видимости для эффективного рендеринга.

Использование стратегии позволяет повысить производительность в начале анимации. Это связано с тем, что не тратятся лишние ресурсы на поддержку структуры пространственного разбиения и применяется отсечение на уровне объектов. Также стратегия эффективна в середине и конце анимации. Это связано с тем, что проводится вычисление и адаптивное обновление количества проверок видимости, необходимого для эффективного рендеринга текущего состояния сцены. Ухудшение производительности методов (скачки на графике) в основном связано с индексированием объектов и перезаписыванием командных буферов узлов дерева при появлении новых объектов.

В среднем за счет адаптивного управления графическим конвейером, предусматривающего выбор и настройку альтернативных базовых методов и техник, достигается прирост производительности в 1.3–2.5 раза на рассмотренных тестовых сценах.

Рассмотрим влияние коэффициента пространственно-временной когерентности на производительность рендеринга. Для тестов берется сцена 32, содержащая 211000 объектов. С помощью метода, описанного в разделе 3.4, производится добавление событий об изменении видимости объектов с разными значениями пространственно-временной когерентности. В результате получаем 15 сцен с постепенно возрастающими значениями коэффициента. В процессе тестирования выполняется проигрывание анимации, отображение сцены и измерение среднего времени

рендеринга. В таблице 32 приведены результаты тестирования с использованием рассматриваемых методов рендеринга. Значение коэффициента незначительно влияет на способ “Frustum Culling”, потому что выполняется запись командного буфера для каждого кадра без кэширования. Другие рассматриваемые способы используют технику записи команд с применением кэширования и фрагментации буферов. Начиная с некоторого значения коэффициента, производительность рендеринга улучшается. Это связано с тем, что при высоких значениях коэффициента за один шаг анимации обрабатываются события о выводе объектов, расположенных относительно близко друг к другу. Благодаря этому сокращается время записи буферов в процессе составления кадрового буфера.

Таблица 32 — Время рендеринга в зависимости от коэффициента пространственно-временной когерентности

Коэффициент пространственно-временной когерентности	Время рендеринга способом Frustum Culling, мс.	Время рендеринга способом Occlusion queries and FC, мс.	Время рендеринга с использованием предлагаемой стратегии, мс.
0.001	86.64	57.9	52.97
0.01	85.92	57.84	52.64
0.03	86.01	57.91	52.82
0.1	85.84	57.44	52.8
0.2	85.53	57.46	53.1
0.3	85.62	57.41	52.83
0.4	85.77	57.49	52.97
0.5	84.12	53.69	41.47
0.6	84.55	53.43	41.68
0.7	83.11	43.35	37.48
0.8	82.56	41.14	34.45
0.9	80.92	36.93	30.33
0.95	79.5	32.85	27.28
0.999	70.63	29.45	22.73
1	70.51	27.14	20.31

5.7 Сравнение производительности разработанной и текущей графических библиотек

Разработанная стратегия и методы программно реализованы в составе системы визуального пространственно-временного (4D) моделирования и планирования индустриальных проектов. В этой системе также используется коммерческая графическая библиотека, позволяющая выполнять рендеринг трехмерных сцен. В данном разделе сравнивается производительность текущей графической библиотеки и графической библиотеки с разработанными методами.

Сначала рассмотрим отображение статических сцен (взяты сцены 1–4 на конечный момент времени). Текущая библиотека выполняет следующие графические оптимизации: объединение полигональных сеток, кэширование командного буфера. За счет этого достигается наилучшая производительность при отображении статических сцен (таблица 33). Если эти оптимизации не используются, то производительность падает значительно.

Таблица 33 — Время рендеринга статических сцен с использованием графических библиотек

Способ / Сцена	Текущая графич. библиотека без оптимизаций, мс.	Текущая графич. библиотека, мс.	Разработанная графич. библиотека, мс.
1	294.8	5.3	8.5
2	367.4	13.6	28.7
3	277.15	12.8	38.5
4	1090.0	17.3	49.1

Рассмотрим отображение динамических сцен. В таблице 34 приведено сравнение времен проигрывания проектной анимации. В процессе анимации камера движется по заданному пути, выполняется рендеринг с заданным разрешением изображения и определенной частотой кадров, измеряется суммарное время работы. Отметим, что зачастую падение производительности происходит при изменении анимационного времени, потому что при этом выполняется обновление структуры пространственного разбиения и запись новых буферов. Текущая графическая библиотека при этом испытывает трудности, потому что она выполняет описанные оптимизации при изменении времени. Разработанные методы позволяют получить

прирост производительности в 2–9 раз при работе с относительно большими динамическими сценами за счет фрагментации и обновления только затронутых узлов дерева и их командных буферов.

Таблица 34 — Время работы при проигрывании проектных анимаций тестовых сцен с использованием графических библиотек

Способ / Сцена	Текущая графич. библиотека, сек.	Разработанная графич. библиотека, сек.
1	108	69
2	119	30
3	381	160
4	735	84

ЗАКЛЮЧЕНИЕ

Основные результаты диссертационной работы состоят в следующем:

1. Предложена модель производительности графического конвейера для однопроходной схемы рендеринга динамических трехмерных сцен. Модель позволяет оценивать требуемые ресурсы (время обработки и передачи графических данных, объем основной и графической памяти) в зависимости от применяемых базовых методов и характеристик отображаемой сцены. Для получения релевантных оценок на оборудовании пользователя параметры модели калибруются с использованием тестовых наборов.
2. Предложена адаптивная стратегия рендеринга динамических трехмерных сцен, реализующая выбор и настройку базовых методов удаления невидимых объектов, техник отложенных аппаратных проверок видимости и кэширования командных буферов на основе модели производительности графического конвейера непосредственно в процессе отображения сцены. Для ускорения вычислений стратегия предусматривает использование регулярных окто-деревьев в качестве структуры пространственной декомпозиции и индексирования динамической сцены.
3. Предложен метод генерации динамических трехмерных сцен, который позволяет синтезировать семейства сцен с разными характеристиками, определяемыми количеством и сложностью индивидуальных объектов, пространственной разреженностью сцены, интенсивностью событий и их пространственно-временной когерентностью. Сгенерированные синтетические сцены могут использоваться для организации репрезентативных наборов тестов и тестирования методов и программных средств компьютерной графики.
4. Проведено исследование и обоснование разработанной адаптивной стратегии рендеринга псевдо-динамических трехмерных сцен. В частности, показана ее высокая эффективность на разнообразных синтетических и реальных индустриальных сценах.

СПИСОК ЛИТЕРАТУРЫ

1. Гонахчян В.И. Модель производительности графического конвейера для однопроходной схемы рендеринга динамических трехмерных сцен // Труды института системного программирования РАН. 2020. №4 (32). стр. 53–72.
2. Гонахчян В.И. Алгоритм удаления невидимых поверхностей на основе программных проверок видимости // Труды института системного программирования РАН. 2018. №2 (30). стр. 81–98.
3. Semenov V. A., Shutkin V. N., Zolotov V. A., Morozov S. V., Gonakhchyan V. I. Visualization of Large Scenes with Deterministic Dynamics // Programming and Computer Software. 2020. Vol. 46. pp. 223–232.
4. Gonakhchyan V., Tarlapan O., Semenov V. Generating dynamic 3D scenes for rendering benchmarks // MCCSIS 2019: 13th Multi Conference on Computer Science and Information Systems. 2019. pp. 485–488.
5. Gonakhchyan V.I. Efficient command buffer recording for accelerated rendering of large 3d scenes // MCCSIS 2018: 12th Multi Conference on Computer Science and Information Systems. 2018. pp. 397–402.
6. Gonakhchyan V.I. Comparison of hierarchies for occlusion culling based on occlusion queries // Proceedings of the GraphiCon 2017 conference on Computer Graphics and Vision. 2017. pp. 32–36.
7. Гонахчян В.И. Адаптивный метод рендеринга динамических трехмерных сцен // Международная конференция ГрафиКон-2019 по компьютерной графике и машинному зрению. 2019. стр. 32–36.
8. Баяковский Ю. М., Игнатенко А. В., Фролов А. И. Графическая библиотека OpenGL. Учебно-методическое пособие, МГУ ВМиК. 2003.
9. Боресков А. Программирование компьютерной графики. Современный OpenGL. ДМК Пресс. 2019. 372 с.
10. Vulkan 1.0.26 — A Specification // URL: https://vulkan.lunarg.com/doc/view/1.0.26.0/linux/vkspec.chunked/index.html (accessed 01.03.2018).
11. Алейникова А.В., Павлов А.С., Пачев А.Н., Манучарян Л. Х. HLSL-шейдеры основы и практические шаги по созданию вертексных и пиксельных шейдеров // Труды Ростовского государственного университета путей сообщения. 2017. № 4. С. 5–8.

12. Михайлюк М.В., Тимохин П.Ю., Мальцев А.В. Метод тесселяции на GPU рельефа земли для космических видеотренажеров // Программирование. 2017. №4. С. 39–47.
13. Marschner S. et al. Fundamentals of Computer Graphics / S. Marschner, P. Shirley, M. Ashikhmin, M. Gleicher, N. Hoffman, et al., 4 ed., Fourth edition. | Boca Raton: CRC Press, Taylor & Francis Group, [2016]: A K Peters/CRC Press, 2018.
14. Akenine-Möller T., Haines E., Hoffman N. Real-time rendering. CRC Press, 2019.
15. Перевалов, С.С., Данилов, Е.А. Исследование отложенного рендеринга на основе техники multiple render targets // Международный студенческий научный вестник. 2018. № 3-2. С. 323–326.
16. Legacy OpenGL. URL: https://www.khronos.org/opengl/wiki/Legacy_OpenGL (accessed 01.03.2018).
17. Lorach T. Approaching Zero Driver Overhead // 2014. URL: https://on-demand.gputechconf.com/siggraph/2015/presentation/SIG1512-Tristan-Lorach.pdf (accessed 01.03.2018).
18. Blackert A. Evaluation of multi-threading in Vulkan. 2016.
19. Shiraef J.A. An exploratory study of high performance graphics application programming interfaces. 2016.
20. A survey of visibility for walkthrough applications / Cohen-Or D. et al. // IEEE Transactions on Visualization and Computer Graphics. 2003. No. 3 (9). pp. 412–431.
21. Боресков А. В. О некоторых методах удаления невидимых поверхностей в задачах визуализации трехмерных сцен большой сложности: дис. канд. ф.-м.н. наук: 05.13.11. М., 2003.
22. Airey J. Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations. 1991. PhD thesis, University of North Carolina, Chappel Hill.
23. Airey J., Rohlf J. H., Brooks F. P. Towards image realism with interactive update rates in complex virtual building environments // Computer Graphics (1990 Symposium on Interactive 3D Graphics). 1990. Vol. 24 (2). pp. 41–50.
24. Teller S. J. Visibility computations in densely occluded polyhedral environments. PhD diss., University of California at Berkeley, 1992.
25. Teller S.J., Séquin C.H. Visibility preprocessing for interactive walkthroughs // ACM SIGGRAPH Computer Graphics. 1991. No. 4 (25). pp. 61–70.

26. Luebke D., Georges C. Portals and mirrors: Simple, fast evaluation of potentially visible sets // In Proceedings of the 1995 symposium on Interactive 3D graphics. 1995. P. 105.
27. Funkhouser T. A., Séquin C. H., Teller S. J. Management of large amounts of data in interactive building walkthroughs // 1992 Symposium on Interactive 3D Graphics. 1992. No. 25 (2). pp. 11–20.
28. Durand F. et al. Conservative visibility preprocessing using extended projections // Proceedings of SIGGRAPH 2000. pp. 239–248.
29. Fuchs H., Kedem Z. M., Naylor B. F. On visible surface generation by a priori tree structures // In Proceedings of the 7th annual conference on Computer graphics and interactive techniques. 1980. pp. 124–133.
30. Sudarsky O., Gotsman C. Dynamic scene occlusion culling // IEEE Transactions on Visualization and Computer Graphics 5. 1999. no. 1. pp. 13–29.
31. Garlick B., Baum D.R., Winget J.M. Interactive viewing of large geometric databases using multiprocessor graphics workstations // Siggraph ’90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation). 1990.
32. Kumar S., Manocha D., Garrett B., Lin M. Hierarchical back-face culling // 7th Eurographics Workshop on Rendering. 1996. pp. 231–240.
33. Pharr M., Humphreys G. Physically based rendering: From theory to implementation / M. Pharr, G. Humphreys, Elsevier, 2010.
34. Greene N., Kass M., Miller G. Hierarchical Z-buffer visibility. ACM Press, 1993. pp. 231–238.
35. Greene N. Hierarchical polygon tiling with coverage masks. ACM Press, 1996. pp. 65–74.
36. Warnock J. E. A Hidden Surface Algorithm for Computer Generated Halftone Pictures. PhD Thesis, Computer Science Dept., University of Utah, TR 4-15, 1969.
37. Visibility culling using hierarchical occlusion maps / Zhang H. et al. ACM Press, 1997. pp. 77–88.
38. Coorg S., Teller S. Temporally coherent conservative visibility // Computational Geometry 12. 1999. No. 1-2. pp. 105–124.
39. Coorg S., Teller S. Real-time occlusion culling for models with large occluders // In Proceedings of the 1997 symposium on Interactive 3D graphics. 1997. pp. 83–90.
40. Hudson T. et al. Accelerated occlusion culling using shadow frustra // In Proceedings 13th Annual ACM Symposium on Computational Geometry. 1997. pp 1–10.
41. Software Occlusion Culling / Chandrasekaran C. et al. // Intel Developer Zone. 2013.

42. Bartz D, Meißner M, Hüttner T. Extending Graphics Hardware For Occlusion Queries In OpenGL // InWorkshop on Graphics Hardware. 1998. pp. 97–103.
43. NV_occlusion_query. URL: https://www.khronos.org/registry/OpenGL/extesions/NV/NV_occlusion_query.txt (accessed 01.03.2018).
44. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful / Bittner J. et al. // Computer Graphics Forum. 2004. No. 3 (23). pp. 615–624.
45. Mattausch O., Bittner J., Wimmer M. CHC++: Coherent Hierarchical Culling Revisited // Computer Graphics Forum. 2008. No. 2 (27). pp. 221–230.
46. Guthe M., Balázs Á., Klein R. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. 2006. pp. 207–214.
47. Bucci J., Doghramachi H. Deferred+: next-gen culling and rendering for dawn engine // URL: https://eidosmontreal.com/en/news/deferred-next-gen-culling-and-rendering-for-dawn-engine (accessed 01.03.2018).
48. Funkhouser T.A., Séquin C.H. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. ACM Press, 1993. pp. 247–254.
49. Garey M.R., Johnson D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York, 1979.
50. Ibaraki T., Hasegawa T., Teranaka K., Iwase J. The Multiple-Choice Knapsack Problem // J. Oper. Res. Soc. Japan 21, 1978, pp. 59–94.
51. Wimmer M., Wonka P. Rendering time estimation for real-time rendering. 2003. pp. 118–129.
52. Egenhofer M.J. Spatial SQL: A query and presentation language // IEEE Transactions on knowledge and data engineering. 1994. No. 1 (6). pp. 86–95.
53. Güting R.H. GRAL: An extensible relational database system for geometric applications // VLDB. Citeseer, 1989. pp. 33–44.
54. Scholl M., Voisard A. Thematic map modeling // Symposium on Large Spatial Databases. Springer, 1989. pp. 167–190.
55. Güting R.H., Schneider M. Realms: A foundation for spatial data types in database systems // International Symposium on Spatial Databases. Springer, 1993. pp. 14–35.
56. Bayer R. The universal B-Tree for multidimensional Indexing. 1996.
57. Extendible hashing — a fast access method for dynamic files / Fagin R. et al. // ACM Transactions on Database Systems (TODS). 1979. No. 3 (4). pp. 315–344.
58. Kriegel H.P. Performance comparison of index structures for multi-key retrieval // ACM SIGMOD Record. ACM, 1984. pp. 186–196.

59. Litwin W. Linear hashing: a new tool for file and table addressing // VLDB. 1980. pp. 1–3.
60. Larson P.Å. Linear hashing with partial expansions // VLDB. 1980. pp. 224–232.
61. Bayer R., McCreight E. Organization and maintenance of large ordered indexes // Software pioneers. Springer, 2002. pp. 245–262.
62. Bentley J.L. Multidimensional binary search trees used for associative searching // Communications of the ACM. 1975. No. 9 (18). pp. 509–517.
63. Bentley J.L., Friedman J.H. Data structures for range searching // ACM Computing Surveys (CSUR). 1979. No. 4 (11). pp. 397–409.
64. Fuchs H., Abram G.D., Grant E.D. Near real-time shaded display of rigid objects // ACM SIGGRAPH Computer Graphics. ACM, 1983. pp. 65–72.
65. Finkel R.A., Bentley J.L. Quad trees a data structure for retrieval on composite keys // Acta Informatica. 1974. No. 1 (4). pp. 1–9.
66. Samet H. The design and analysis of spatial data structures / H. Samet, Addison-Wesley.
67. Nievergelt J., Hinterberger H., Sevcik K.C. The Grid File: An Adaptable, Symmetric Multikey File Structure // ACM Transactions on Database Systems. 1984. No. 1 (9). pp. 38–71.
68. Tamminen M. The extendible cell method for closest point problems // BIT. 1982. No. 1 (22). pp. 27–41.
69. Kriegel H.P., Seeger B. Multidimensional order preserving linear hashing with partial expansions // International Conference on Database Theory. Edited by G. Ausiello, P. Atzeni. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986. pp. 203–220.
70. Kriegel H.P., Seeger B. Multidimensional dynamic quantile hashing is very efficient for non-uniform record distributions // IEEE Third International Conference on Data Engineering. Los Angeles, CA, USA: IEEE, 1987. pp. 10–17.
71. Ouksel M.A. The interpolation-based grid file // Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems. Portland, Oregon, United States: ACM Press, 1985. pp. 20–27.
72. Robinson J.T. The K-D-B-tree: a search structure for large multidimensional dynamic indexes // Proceedings of the 1981 ACM SIGMOD international conference on Management of data. Ann Arbor, Michigan: ACM Press, 1981. pp. 10–18.
73. Lomet D.B., Salzberg B. The hB-tree: a multiattribute indexing method with good guaranteed performance // ACM Transactions on Database Systems. 1990. No. 4 (15). pp. 625–658.

74. Seeger B., Kriegel H.P. Techniques for design and implementation of spatial access methods // Proceedings of the 14th VLDB Conference. 1988. pp. 360–371.
75. Hinrichs K. Implementation of the grid file: Design concepts and experience // BIT. 1985. No. 4 (25). pp. 569–592.
76. Nievergelt J., Hinrichs K. Storage and Access Structures for Geometric Data Bases // Foundations of Data Organization. Edited by S.P. Ghosh, Y. Kambayashi, K. Tanaka. Boston, MA: Springer US, 1987. pp. 441–455.
77. Six H.W., Widmayer P. Spatial searching in geometric databases // Proceedings of the Fourth International Conference on Data Engineering. Los Angeles, CA, USA: IEEE Comput. Soc. Press, 1988. pp. 496–503.
78. Hutflesz A., Six H.W., Widmayer P. The R-file: an efficient access structure for proximity queries // Proceedings of the Sixth International Conference on Data Engineering. Los Angeles, CA, USA: IEEE Comput. Soc, 1990. pp. 372–379.
79. Guttman A. R-trees: a dynamic index structure for spatial searching // ACM SIGMOD Record. 1984. No. 2 (14). pp. 47.
80. Greene D. An implementation and performance analysis of spatial data access methods // Proceedings of the Fifth International Conference on Data Engineering. Los Angeles, CA, USA: IEEE Comput. Soc. Press, 1989. pp. 606–615.
81. Roussopoulos N., Leifker D. Direct spatial search on pictorial databases using packed R-trees // ACM SIGMOD Record. 1985. No. 4 (14). pp. 17–31.
82. Clark, J. H. Hierarchical geometric models for visible surface algorithms // Communications of the ACM. 1976. No. 19 (10). pp. 547–54.
83. Weghorst H., Hooper G., Greenberg D. Improved computational methods for ray tracing. 1984.
84. Wald I. On fast construction of SAH-based bounding volume hierarchies // In IEEE Symposium on Interactive Ray Tracing. 2007. pp. 33–40.
85. Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D. Fast BVH construction on GPUs // In Computer Graphics Forum. 2009. Vol. 28, No. 2. pp. 375–384.
86. Glassner A. S. Space subdivision for fast ray tracing // IEEE Computer Graphics and applications. 1984. Vol. 4, No. 10. pp. 15–24.
87. Morozov S., Semenov V., Tarlapan O., Zolotov V. Indexing of Hierarchically Organized Spatial-Temporal Data Using Dynamic Regular Octrees // In: Petrenko A., Voronkov A. (eds) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science. Vol. 10742. pp. 276–290.

88. Золотов В.А., Петрищев К.С., Семенов В.А. Исследование методов пространственного индексирования динамических сцен на основе регулярных октодеревьев // Программирование. Компьютерная графика. 2016. № 6. стр. 59–66.
89. Золотов В.А. Перспективные методы индексирования пространственно-временных данных: дис. канд. ф.-м.н. наук: 05.13.11. М., 2015.
90. Weisstein, Eric W. Least Squares Fitting—Logarithmic // From MathWorld—A Wolfram Web Resource. URL: https://mathworld.wolfram.com/LeastSquaresFittingLogarithmic.html (accessed 08.04.2019).
91. Черняк А.А. и др. Методы оптимизации: теория и алгоритмы / А.А. Черняк, Ж.А. Черняк, Ю.М. Метельский, С.А. Богданович, 2 изд., М: Юрайт, 2017. 357 с.
92. Weisstein, Eric W. Bisection // From MathWorld—A Wolfram Web Resource. URL: https://mathworld.wolfram.com/Bisection.html (accessed 08.04.2019)
93. Render output unit. URL: https://en.wikipedia.org/wiki/Render_output_unit (accessed 01.03.2018).
94. Vulkan/examples/computecullandlod. URL: https://github.com/SaschaWillems/Vulkan/tree/master/examples/computecullandlod (accessed 03.09.2020)