Федеральное государственное бюджетное учреждение науки
«Институт системного программирования им. В.П. Иванникова
Российской академии наук»

На правах рукописи

Шуткин Василий Николаевич

Метод иерархических динамических уровней детализации для рендеринга больших трехмерных сцен с детерминированной динамикой

Специальность 2.3.5 — Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата физико-математических наук

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

Москва 2026

Оглавление

Введение

Актуальность исследования. Рендеринг сложных трехмерных сцен является одной из ключевых проблем компьютерной графики. Он возникает в таких предметных областях, как научная визуализация, автоматизация проектирования, инженерии и производства (CAD/CAM/CAE), цифровые двойники, визуальное моделирование зданий, сооружений и городской инфраструктуры, геоинформационные системы, компьютерные игры и анимация, виртуальная и дополненная реальность. Сложность сцен определяется множеством факторов: числом объектов с индивидуальными геометрическими моделями, типизацией и способами параметризации аналитических моделей, степенью детализации приближенных моделей (полигональных сеток, облаков точек, гауссовых сплаттов), реалистичностью применяемых материалов и текстур, композиционным представлением сцены и объектов, смешанным и разноскоростным характером динамики объектов. По мере роста производительности вычислительной техники, графических процессоров и кластеров, растут ожидания разработчиков прикладного программного обеспечения в отношении возможного увеличения сложности трехмерных сцен, которые могут быть отображены с требуемой частотой кадров и надлежащим визуальным качеством.

Для этих целей обычно применяют комбинации методов и техник оптимизации, среди которых следует выделить отсечение конусом видимости (англ. frustum culling), отсечение окклюзии (англ. occlusion culling), выделение значимых объектов, упрощение геометрии и применение уровней детализации (англ. level of detail, LOD), оптимизацию графического конвейера и балансировку вычислительной нагрузки между CPU и GPU. Иерархические уровни детализации (англ. hierarchical level of detail, HLOD) являются одним из наиболее перспективных методов оптимизации рендеринга сложных сцен благодаря масштабируемости по отношению к числу и сложности объектов (и возможности хранения представления сцены во внешней памяти). Однако данный метод не применим для динамических сцен, поскольку подготовка уровней детализации требует значительных вычислительных ресурсов, а их перманентное обновление в ходе событийного моделирования сцены неизбежно приводит к замедлению процессов рендеринга. Актуальным, в связи с этим, представляется создание методов, обладающих достоинствами иерархических уровней детализации HLOD, но применимых к динамическим сценам в практически важных классах графических приложений.

Целью работы является разработка и исследование методов рендеринга больших динамических трёхмерных сцен с поддержкой консервативного (гарантирующего заданную пространственную и временную точность) и интерактивного (стремящегося обеспечить

максимально возможную частоту генерации изображений при приемлемом уровне реализма) режимов отображения в индустриально значимых приложениях.

Основные задачи, решаемые в работе:

- Анализ существующих методов оптимизации рендеринга сложных сцен и исследование возможностей обобщения методов уровней детализации на динамический случай;
- Разработка и исследование метода уровней детализации для выбранного класса динамических полигональных сцен в основной и внешней памяти;
- Разработка и исследование алгоритмов консервативного и интерактивного рендеринга уровней детализации во внешней памяти;
- Программная реализация предложенного метода и разработанных алгоритмов для вычисления и рендеринга уровней детализации;
- Апробация разработанных метода и алгоритмов в результате проведения вычислительных экспериментов на синтетических и реальных индустриальных сценах и выработка практических рекомендаций.

Научная новизна:

- Выделен класс полигональных сцен с детерминированным дискретно-непрерывным характером динамики, допускающий конструктивное обобщение методов уровней детализации и имеющий важные индустриальные приложения;
- Предложен метод иерархических динамических уровней детализации HDLOD (сокращение от англ. Hierarchical Dynamic Level of Detail) для представления сцен с детерминированной динамикой.
- Предложен алгоритм вычисления уровней детализации HDLOD на основе классификации объектов динамической сцены, многоуровневой кластеризации и серий полигональных упрощений. Вычисленные уровни детализации позволяют оперативно формировать альтернативные упрощенные представления сцены с учетом текущего положения пространственной камеры и временного репера, разрешения устройства отображения и заданной визуальной точности;
- Предложены алгоритмы консервативного и интерактивного рендеринга уровней детализации HDLOD во внешней памяти, которые предусматривают раздельное хранение иерархии уровней и геометрических моделей, а также кэширование моделей в основной и видеопамяти с асинхронной загрузкой и вытеснением. Благодаря возможности переключения между консервативным и интерактивным режимами

непосредственно во время исполнения целевого приложения обеспечивается сбалансированное по производительности и качеству отображение сложных сцен;

– Получены теоретические оценки сложности основных алгоритмов, а также проведены серии вычислительных экспериментов, которые подтверждают эффективность разработанного метода HDLOD и возможность его конструктивного применения для широкого класса индустриальных приложений, включая визуальное моделирование строительных проектов и инфраструктурных программ.

Теоретическая важность проведенного исследования определяется выполненной формализацией практически значимого класса полигональных сцен с детерминированным дискретно-непрерывным характером динамики и введенными функциями близости объектов, необходимыми для пространственно-временной кластеризации сцен.

Теоретически важными представляются проведенное обобщение популярного метода иерархических уровней детализации на класс динамических полигональных сцен, разработанные и формально описанные алгоритмы вычисления и рендеринга уровней детализации HDLOD с введенной функцией визуального качества (погрешности), а также полученные теоретические оценки вычислительной сложности разработанных алгоритмов.

Практическая значимость полученных результатов заключается в возможности применения разработанного метода для рендеринга широкого класса сцен, не помещающихся в основной памяти, видеопамяти и требующих значительных вычислительных ресурсов CPU и GPU для отображения в консервативном и интерактивном режимах просмотра. В частности, метод демонстрирует высокую производительность и масштабируемость для больших полигональных сцен с доминирующими дискретными событиями, характерными для моделей сложных индустриальных проектов и масштабных инфраструктурных программ.

Благодаря достигнутым характеристикам метод нашел применение в ряде прикладных проектов, имеющих целью создание приложений визуального пространственно-временного моделирования индустриальных проектов, градостроительного планирования, управления требованиями и замечаниями в архитектурно-строительной отрасли.

Методология и методы исследования. В работе применены методы компьютерной графики, вычислительной геометрии, линейной алгебры и теории алгоритмов.

Достоверность и обоснованность результатов. Все разработанные методы и алгоритмы были реализованы и прошли экспериментальную апробацию.

Основные положения, выносимые на защиту:

1. Метод иерархических динамических уровней детализации HDLOD для представления и рендеринга сцен с детерминированным характером динамики.

2. Алгоритм вычисления уровней детализации HDLOD во внешней памяти на основе классификации динамических объектов, кластеризации объектов и полигональных упрощений.
3. Алгоритмы консервативного и интерактивного рендеринга HDLOD во внешней памяти.
4. Результаты вычислительных экспериментов и приложения.

Апробация работы. Основные результаты работы докладывались на:

1) SYNCHRO SOFTWARE & ISP RAS Regular seminars (2015-2017, ISP RAS, Moscow, Russia).
2) BENTLEY SYSTEMS Year in Infrastructure 2018 Conference (Hilton London Metropole, London, United Kingdom, October 15-18, 2018).
3) CGVCVIP 2019: 13th International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2019 (Porto, Portugal, 16-18 July 2019).
4) 26th ISTE International Conference on Transdisciplinary Engineering (Kashiwa Campus, The University of Tokyo, Japan, July 30 - August 1, 2019).
5) 29-я Международная конференция по компьютерной графике и машинному зрению ГрафиКон 2019 (23−26 сентября 2019, БГТУ, г. Брянск, Россия).
6) BIMAC 2021: IV Международная научно-практическая конференция «BIM-моделирование в задачах строительства и архитектуры» (21−23 апреля 2021, СПбГАСУ, г. Санкт-Петербург, Россия).
7) 13th European Conference on Product & Process Modelling ECPPM 2020-2021 (15 - 16 September 2021, Moscow, Russian Federation).
8) 31-я Международная конференция по компьютерной графике и машинному зрению ГрафиКон 2021 (27−30 сентября 2021, НГТУ им. Р.Е. Алексеева, г. Нижний Новгород, Россия).
9) HUAWEI Global Software Technology Summit – Lake Baikal Summit 2024 (Irkutsk, Russia, 19-21 August 2024)
10) HUAWEI Frontiers of Computer Graphics (17-18 октября 2024, МГУ Ломоносов Холл, г. Москва, Россия).
11) BIMAC 2025: VIII Международная научно-практическая конференция «Информационное моделирование в задачах строительства и архитектуры» (22−25 апреля 2025, СПбГАСУ, г. Санкт-Петербург, Россия).
12) 35-я Международная конференция ГрафиКон 2025 (30 сентября – 2 октября 2025 г., Поволжский государственный технологический университет, г. Йошкар-Ола, Россия).

Публикации. Основные результаты по теме диссертации изложены в 11 печатных изданиях [1-11], 2 из которых изданы в журналах, рекомендуемых ВАК [4, 10], 2 работы индексированы в Web of Science [1, 4] и 5 — в Scopus [1, 2, 4, 7, 8]. Получено свидетельство о государственной регистрации программы для ЭВМ [12].

Личный вклад автора. Основные результаты диссертации, в том числе положения, выносимые на защиту, получены лично автором. Научным руководителем постановлены задачи и осуществлены редакторские правки. Участие других соавторов ограничивалось помощью в программных реализациях, проведении вычислительных экспериментов и решении смежных задач.

Объём и структура работы. Диссертация состоит из введения, семи глав и заключения. Полный объём диссертации составляет 161 страницу, включая 35 рисунков и 20 таблиц. Список литературы содержит 135 наименований.

Глава 1. Обзор современных подходов к рендерингу больших сцен

Если трёхмерная сцена небольшая, а графического оборудование имеет достаточно высокую производительность, рендеринг сцены может быть осуществлён наивным способом, то есть последовательным отображением всех объектов сцены. Однако все графические библиотеки, нацеленные на эффективный рендеринг больших сцен, применяют множество оптимизаций для снижения вычислительных затрат. Существует два основных подхода к уменьшению вычислений при рендеринге: удаление невидимых поверхностей (англ. hidden surface removal) и упрощение геометрических представлений (англ. geometry simplification). Рассмотрим эти подходы в общих чертах.

1.1 Удаление невидимых поверхностей

Удаление невидимых поверхностей ставит перед собой цель как можно более раннего определения поверхностей объектов, которые не будут видны на экране, с целью их исключения из обработки в графическом конвейере. Эта идея строится на следующем очевидном утверждении: нет необходимости производить обработку и вычисления для поверхностей, которые в итоге не будут видны на экране. При этом поверхности могут быть не видны по двум причинам:

1) поверхность не попадает в область видимости виртуальной камеры (например, находится за спиной зрителя);
2) поверхность скрыта за другими поверхностями переднего плана.

Удаление поверхностей, не попадающих в область видимости, называется *отсечением по области видимости* (англ. view frustum culling). Удаление поверхностей, которые скрыты за другими поверхностями, называется *отсечением перекрытых поверхностей* или *отсечением окклюзии* (англ. occlusion culling).

Во всех современных графических API (таких как OpenGL, DirectX, и Vulkan) имеются реализации удаления невидимых поверхностей, однако только на стадии графического конвейера между вершинным и фрагментным шейдером. Так, после вершинного шейдера видеодрайверу становится известно положение вершин, что позволяет отбросить участки примитивов, которые не попадают в область видимости, и не запускать для них фрагментный шейдер. Можно сказать, что это своего рода отсечение по области видимости. Отсечение перекрытых поверхностей

реализовано в графических API как опциональный тест глубины (англ. depth test) с использованием Z-буфера (буфера глубины). Тест глубины для примитива производится после вершинного шейдера, и для тех фрагментов, которые провалили тест глубины (то есть находятся за примитивами, которые уже были растеризованы до этого), фрагментный шейдер не запускается. Таким образом, удаление невидимых поверхностей, реализованное в графических API, позволяют сократить число вызовов фрагментного шейдера. Однако, наибольшего сокращения вычисления можно добиться, если добавить удаление невидимых поверхностей на более ранних стадиях. Поэтому многие графические библиотеки стараются производить как можно более ранние проверки видимости. Например, отбраковка невидимых поверхностей может быть произведена на центральном процессоре, с тем чтобы вообще не передавать невидимые поверхности на графический процессор.

Примеры алгоритмов отсечения по области видимости можно найти в работе [22]. Обзор подходов к отсечению окклюзии представлен в работе [89].

1.2 Упрощение геометрических представлений

Упрощение геометрических представлений является вторым основным способом сокращения вычислений при рендеринге. Оно ставит перед собой цель сократить число примитивов (полигонов, треугольников) в геометрическом представлении объекта, при этом сохранив внешний вид объекта на экране. Во время рендеринга основные вычислительные ресурсы тратятся на обработку примитивов, поэтому сокращение числа примитивов может существенно снизить вычислительные затраты. Далее кратко рассмотрим процесс рендеринга, чтобы проиллюстрировать, какие вычисления производятся во время этого процесса. Можно выделить два способа отображения трёхмерных сцен [123]:

1) прямое отображение (англ. forward mapping), когда к каждому примитиву сцены применяются преобразования, проецирующие его в экранную плоскость;
2) обратное отображение (англ. backward mapping), когда для каждого пикселя экрана строится луч в трёхмерном пространстве, выходящий из виртуальной камеры и проходящий через этот пиксель, после чего осуществляется поиск примитивов, пересекающихся с этим лучом.

Прямое отображение на сегодняшний день является наиболее распространённым способом рендеринга, и все современные графические API, а также графическое оборудование и драйверы к нему, оптимизированы именно для такого способа. Очевидно, что количество

вычислений при прямом рендеринге пропорционально количеству примитивов, поэтому сокращение числа примитивов приведёт к сокращению вычислений.

Когда говорят об обратном отображении, как правило говорят о методах бросания лучей (англ. ray casting), трассировки лучей (англ. ray tracing) или трассировки путей (англ. path tracing). Трассировка путей уже достаточно давно используется для фотореалистичного рендеринга (в промышленном дизайне, дизайне интерьеров, в кинематографе и т.д.). В последнее время наблюдаются тенденции применения трассировки путей в интерактивных приложениях (таких как видеоигры), поддержка трассировки добавляется в графическое оборудование и графические API (к примеру, Nvidia RTX, DirectX Ray Tracing (DXR), Vulkan Ray Trancing Extensions). При обратном отображении необходимо иметь возможность быстро находить примитивы, пересекающиеся с лучом. Наивная реализация проверяла бы для каждого примитива факт его пересечения с лучом. Но как правило, для быстрого поиска пересечений используют ускоряющую структуру, такую как иерархия ограничивающих объёмов (англ. bounding volume hierarchy (BVH)). В любом случае, время поиска пропорционально числу примитивов, поэтому сокращение числа примитивов сократит время рендеринга.

В подходе с упрощением геометрических представлений есть несколько направлений:

1) одноразовое предварительное упрощение избыточно детализированной геометрии;
2) уровни детализации (англ. level of detail (LOD)), которые в свою очередь можно разделить на:
    a. дискретные уровни детализации (англ. discrete LOD);
    b. непрерывные уровни детализации (англ. continuous LOD);
    c. зависимые от камеры уровни детализации (англ. view-dependent LOD).

Одноразовое предварительное упрощение геометрии используется для геометрии, которая имеет избыточный уровень детализации. Например, это могут быть 3D-модели, полученные в результате сканирования объектов реального мира с применением таких технологий как лазерное сканирование или фотограмметрия. Как правило, лазерные сканеры продуцируют облако точек (англ. point cloud). Для быстрого рендеринга в целевых приложениях по этому облаку точек строится полигональная сетка (англ. polygon mesh), которая может иметь множество избыточных вершин и полигонов. Другим примером являются видеоигры. 3D-художник в ходе своей работы создаёт и редактирует очень детальную 3D-модель, но, когда модель готова, она оптимизируется и упрощается для использования в игре в контексте системных требований игры. Если игра затем портируется, например, на мобильные устройства, могут потребоваться дальнейшие упрощения 3D-моделей для того, чтобы она запускалась на более слабом оборудовании.

Для ряда приложений может быть достаточно однократного упрощения модели с последующим использованием только её упрощённого представления. Однако во многих случаях дальнейшего ускорения рендеринга можно добиться путём использования **уровней детализации**. Данный подход основан на следующем наблюдении: если объект находится далеко, нет смысла отображать детальную модель, поскольку детали всё равно не будут заметны. Вместо этого можно взять упрощённую модель и получить прирост производительности при сохранении качества изображения. Таким образом, необходимая степень детализации объекта выбирается в зависимости от разрешения экрана и размера проекции объекта на экран.

Впервые идея использования альтернативных представлений объектов при рендеринге изображений была высказана ещё в 1976 году в работе [31]. Однако термин «уровни детализации» (англ. level of detail) активно начал использоваться только в 1990-е годы, когда с развитием персональных компьютеров трёхмерная графика «шагнула в массы». Идея создания нескольких уровней детализации и выбора подходящего уровня в зависимости от расстояния до объекта (или размера объекта на экране) была высказана в работах [97, 57]. В работе [42] задача выбора уровня детализации рассматривается как оптимизационная задача (стоимость-выгода).

1.2.1 **Дискретные уровни детализации**

Рисунок 1.1 — Дискретные уровни детализации модели Стэнфордского кролика.

Дискретные уровни детализации подразумевают создание нескольких версий одной модели с различной степенью детализации (рис. 1.1). Во время визуализации, на основе положения объекта, положения камеры и разрешения экрана выбирается представление объекта с наиболее подходящей степенью детализации. Упрощённые версии объекта подготавливаются перед рендерингом, и используются во время рендеринга. Графический конвейер не требуется сильно изменять для поддержки дискретных уровней детализации, нужно лишь добавить имплементацию выбора подходящего уровня детализации для каждого объекта сцены. Данный метод является достаточно простым и эффективным в реализации, однако резкие переключения

между уровнями детализации могут быть заметны и могут ухудшить восприятие виртуальной сцены пользователем. Для сокрытия этого эффекта может применяться альфа-смешивание или морфинг [62].

Как правило, каждому уровню детализации ставится в соответствие значение геометрической погрешности (англ. geometric error), характеризующее отклонение упрощённого геометрического представления от оригинальной модели. Во время рендеринга, для оценки пригодности представления, его геометрическая погрешность переводится в так называемую экранную погрешность (англ. screen-space error). Для объекта выбирается наиболее упрощённое представление, удовлетворяющее требованию к экранной погрешности. Таким образом решается задача максимизации производительности при сохранении допустимого уровня качества.

1.2.2 **Непрерывные уровни детализации**

Непрерывные уровни детализации, в отличие от дискретных, хранят геометрию в некоторой структуре данных, из которой во время рендеринга можно извлечь геометрическое представление с требуемой степенью детализации [81]. Благодаря этому достигается большая гранулярность в выборе подходящей степени детализации, используется ровно столько полигонов, сколько необходимо для обеспечения целевого качества изображения. Другое достоинство непрерывных уровней детализации заключается в возможности более эффективной передачи моделей, поскольку полигональная модель хранится как некоторая простая базовая модель плюс набор операций уточнения. Таким образом становится возможным быстро переслать и отобразить базовую модель и затем досылать данные об операциях уточнения по мере необходимости.

Наиболее известной структурой данных для представления непрерывных уровней детализации является прогрессивная сетка (англ. progressive mesh) [60]. Концепция прогрессивной сетки основана на наблюдении, что для операции стягивания ребра существует обратная операция — разделение вершины (англ. vertex split) (рис. 1.3). Пусть некоторая полигональная сетка M была упрощена последовательностью n операций стягивания ребра до сетки M^0. Поскольку для операции стягивания ребра существует обратная операция, сетка M может быть представлена как сетка M^0 плюс последовательность из n записей об операциях разделения вершины $vsplit$. Таким образом, $(M^0, \{vsplit_0, \dots, vsplit_{n-1}\})$ — это представление M в формате прогрессивной сетки. Такой формат позволяет получить сетку с произвольным количеством треугольников путём применения необходимого числа операций разделения вершины к M^0. К достоинствам данного представления также относятся:

- удобство передачи полигональной сетки по сети: сетка M^0 может быть визуализирована быстро, затем она может уточняться по мере поступления записей о разделении вершин;
- эффективность хранения;
- естественная поддержка морфинга, поскольку операции стягивания ребра и разделения вершины могут визуализироваться плавно;
- поддержка выборочного уточнения (selective refinement): операции разделения вершины могут применяться только на тех участках сетки, где требуется повышенная детализация.

1.2.3 Зависимые от камеры уровни детализации

Зависимые от камеры уровни детализации являются расширением непрерывных уровней детализации и позволяют производить уточнение модели только в необходимых областях, а не равномерно по всей модели (англ. view-dependent refinement), что позволяет эффективно визуализировать большие или протяжённые объекты. Сторона объекта, которая находится близко к наблюдателю может быть представлена с высокой детализацией. В то же время, противоположная сторона, находящаяся вдали, может быть представлена в упрощённом виде (рис. 1.2). Также, больше полигонов может быть выделено в силуэтных участках, а меньше — во внутренних.

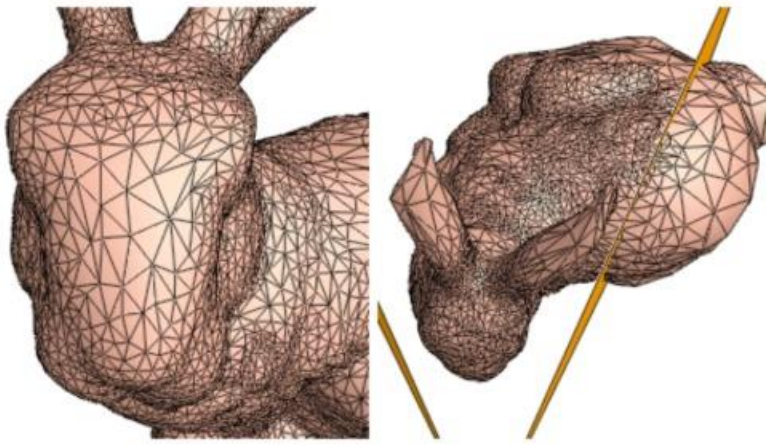


Рисунок 1.2 — Пример уточнения модели в зависимости от положения камеры (из работы [61]).

На практике такие уровни детализации могут применяться при визуализации ландшафтов, когда весь ландшафт представлен одной полигональной сеткой. Другим примером может послужить визуализация результатов магниторезонансной томографии в медицине, когда

получается цельная модель с множеством полигонов, а врач должен иметь возможность приблизиться и разглядеть в деталях разные её части.

1.2.4 **Иерархические уровни детализации**

Дальнейшим развитием концепции уровней детализации являются иерархические уровни детализации (англ. hierarchical levels of detail, HLOD) [40]. Необходимость в иерархических уровнях детализации возникает в сценах, состоящих из большого количества отдельных объектов, в особенности если объекты сцены уже сами по себе являются достаточно простыми. В таком случае существенное упрощение этих объектов невозможно, и традиционные уровни детализации оказываются неэффективными. Иерархические же уровни детализации обрабатывают группы объектов, создавая иерархию упрощённых представлений. Благодаря этому иерархические уровни детализации позволяют достичь высокой масштабируемости приложения по отношению к числу объектов в сцене.

Наибольшую эффективность иерархические уровни детализации показывают для статических сцен. Также, их применение ограниченно возможно для динамических сцен. При изменениях в сцене соответствующие представления в иерархии уровней детализации должны быть пересчитаны в параллельном потоке. Таким образом, иерархические уровни детализации могут демонстрировать хорошую эффективность в случаях, когда динамическое поведение имеют небольшое количество объектов сцены, а изменения происходят редко. Однако, как признают сами авторы, в сценах с большой степенью динамики иерархические уровни детализации оказываются неэффективны, поскольку время, требуемое для пересчётов, как правило больше времени, требуемого для рендеринга сцены [40].

1.3 Методы упрощения полигональных представлений

Одним из наиболее распространённых способов задания геометрии трёхмерных моделей является полигональная сетка (англ. polygon mesh). Такое представление является аппроксимацией формы объекта с помощью множества многоугольников (полигонов). Соответственно, чем больше полигонов в сетке, тем точней (детальней) задана форма объекта, однако тем выше вычислительные затраты при рендеринге.

Современные графические процессоры, драйверы и API к ним, оптимизированы именно под рендеринг полигональных сеток, а точнее даже под рендеринг триангулированных сеток

(англ. triangulated mesh), то есть полигональных сеток, в которых каждый полигон является треугольником. Поскольку любой планарный многоугольник (полигон) можно представить в виде набора треугольников (этот процесс называется триангуляцией), то любую полигональную сетку можно привести к триангулированной. Сетка треугольников является наиболее простым и универсальным способом представления геометрии любого объекта. Далее в работе будет рассматриваться геометрия только в виде сеток треугольников.

Множество исследований было посвящено задаче упрощения полигональных представлений (англ. mesh simplification). Целью упрощения является сокращение количества полигонов в модели при сохранении внешнего вида оригинальной модели насколько это возможно. Следует отметить, что большинство существующих алгоритмов упрощения работают с сетками треугольников. Однако это не ограничивает их применимость, поскольку полигональная сетка может быть приведена к сетке треугольников с помощью одного из множества алгоритмов триангуляции.

Как правило существуют два подхода к упрощению [81]:

- упрощение с целевым качеством, когда решается задача минимизации количества треугольников в сетке с учётом требований к её качеству;
- упрощение с заданным бюджетом треугольников, когда задается целевое количество треугольников и алгоритм пытается максимизировать качество представления.

Качество как правило оценивают по некоторой метрике разницы между упрощённой и оригинальной моделями, её также называют погрешностью упрощения (англ. simplification error). Эта погрешность может быть измерена множеством способов, например, при помощи метрики Хаусдорфа.

Среди наиболее популярных алгоритмов упрощения и их семейств следует выделить алгоритмы, основанные на таких операциях, как удаление вершины, стягивание ребра, исключение грани, кластеризация вершин, объединение вершин. Большинство алгоритмов основаны на итеративном применении операции при постоянном контроле погрешности получившейся модели. В некоторых приложениях важное значение играет топология полигональной сетки. Например, может требоваться, чтобы сетка была «водонепроницаемой» (представляла собой двумерное многообразие). В таком случае следует использовать только те методы упрощения, которые гарантируют сохранение топологии. Далее будут рассмотрены наиболее распространённые методы упрощения.

1.3.1 **Стягивание ребра**

Операция стягивания (схлопывания) ребра (англ. edge contraction, edge collapse) является одной из наиболее простых в реализации. Она подразумевает стягивание ребра (v_1, v_2) между вершинами v_1 и v_2 в единственную вершину $v_{\text{нов}}$ (рис. 1.3). При этом $v_{\text{нов}}$ может быть как одной из вершин v_1 или v_2, так и совершенно новой вершиной, вычисленной по некоторым правилам. При стягивании ребра происходит вырождение треугольников, опирающихся на это ребро, данные треугольники удаляются, что приводит к упрощению полигональной сетки. Преимущество этой операции заключается в том, что при соблюдении некоторых ограничений она может гарантировать сохранение топологии [38]. В работе [43] описывается алгоритм упрощения на основе операции стягивания ребра. На каждой итерации алгоритма выбирается ребро, стягивание которого привнесёт наименьшую погрешность. Для оценки погрешности авторы предлагают использовать метрику на основе квадрик, характеризующую сумму квадратов расстояний от одной вершины до набора плоскостей треугольников другой вершины. Данная метрика погрешности зарекомендовала себя как эффективная и простая в вычислении и реализации. С вершинами сетки могут быть ассоциированы цвета или текстурные координаты, их пересчёт при упрощениях является отдельной проблемой. Работа [44] является развитием работы [43] и адресована этой проблеме. Наконец, метрика погрешности на основе квадрик была обобщена на случай упрощения симплициальных комплексов произвольного типа в евклидовом пространстве произвольной размерности [46].

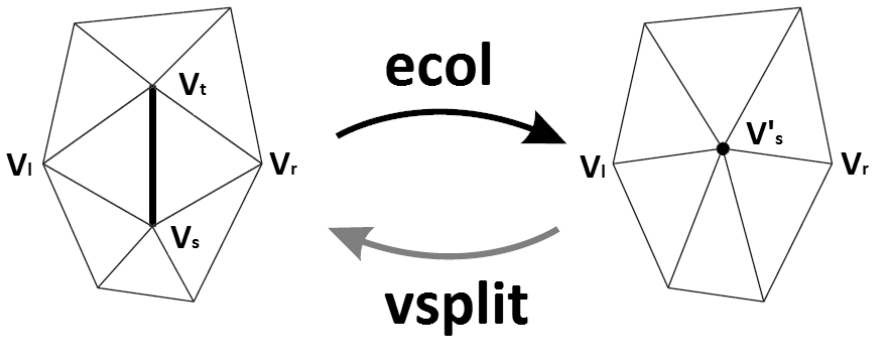


Рисунок 1.3 — Операция стягивания ребра (ecol) и обратная ей операция разделения вершины (vsplit).

Операция стягивания ребра применяется во множестве алгоритмов [124][51]. Она является ключевой в реализации прогрессивных сеток [60] и оптимизации полигональной сетки [59].

Работа [86] использует классическую схему с итеративным стягиванием рёбер, но предлагает новую метрику погрешности, оценивающую кривизну поверхности. Работа [48]

адресована вопросу упрощения нескольких сеток с алгоритмом стягивания рёбер с метрикой квадрик (когда упрощаются сетки, задающие форму составного объекта). В работе [73] предлагается дополнить метрику квадрик гауссовой метрикой кривизны, а также добавить операцию разделения ребра для лучшей обработки длинных и узких треугольников.

1.3.2 Стягивание пары вершин

Стягивание пары вершин (англ. vertex-pair collapse) похоже на стягивание ребра, с тем лишь отличием, что в данном случае не требуется наличия ребра между стягиваемыми вершинами. Стягивание несвязанных вершин позволяет соединять несвязные части и закрывать отверстия и туннели, что может позволить существенно улучшить качество упрощения в определённых случаях. Например, если полигональная сетка представляет собой набор простых несвязных примитивов (стена, состоящая из отдельных кирпичей). К недостаткам данного подхода можно отнести то, что в полигональной сетке, состоящей из n вершин, потенциально существует n^2 пар-кандидатов, что приведёт к сложности алгоритма $O(n^2)$. Поэтому алгоритмы, использующие операцию стягивания пары вершин, должны использовать некоторую эвристику для ограничения числа пар-кандидатов. Данная операция используется в работах [39][94][106], а также в работе [43] наряду со стягиванием ребра.

1.3.3 Стягивание треугольника

Операция стягивания треугольника (англ. triangle collapse) упрощает полигональную сетку путём стягивания треугольника (v_1, v_2, v_3) в единственную вершину $v_{\text{нов}}$ [49][54]. В качестве $v_{\text{нов}}$ может быть выбрана одна из вершин v_1, v_2 или v_3, или же вычислено новое положение вершины.

1.3.4 Удаление вершины

Операция удаления вершины (англ. vertex removal) была представлена в работе [105]. Она подразумевает удаление вершины и связанных с ней рёбер и треугольников с последующей триангуляцией образовавшегося отверстия. Следует отметить, что данная триангуляция не однозначна.

1.3.5 **Объединение полигонов**

Операция объединения полигонов (англ. polygon merging) объединяет соседние полигоны, которые являются почти планарными, в один, который затем триангулируется. Она была впервые представлена в работе [58] и в дальнейшем использовалась в «суперграних» (англ. superfaces) [64] и кластеризации граней [45].

1.3.6 **Кластеризация вершин**

Кластеризация вершин (англ. vertex clustering), используемая в работах [79][80][97], подразумевает формирование кластеров вершин и последующую замену всех вершин в одном кластере на одну вершину. При этом результирующая вершина может выбираться из множества вершин кластера или вычисляться как некоторая усреднённая. После подмены вершин вырожденные и дублированные рёбра и треугольники удаляются, что приводит к упрощению полигональной сетки. Для формирования кластеров могут использоваться различные способы. Например, в работе [97] для этого используется регулярная сетка. Итоговая степень упрощения полигональной сетки в таком случае зависит от разрешения регулярной сетки. Кластеризация вершин нарушает топологию.

В работе [75] представлен алгоритм упрощения полигональных сеток во внешней памяти (когда данные не помещаются в оперативную память). К тому же, данный алгоритм превосходит оригинальный алгоритм кластеризации вершин [97] по качеству упрощения за счёт использования метрики квадрик.

1.3.7 **Вспомогательные техники**

Для лучшего контроля отклонения упрощённой модели от оригинальной в ходе упрощения могут применяться оболочки (англ. simplification envelopes) [32]. Во многих приложениях используется создание карт нормалей (англ. normal maps) для упрощённых моделей по оригинальной модели. Это позволяет существенно упростить модель, но сохранить её облик и детали, поскольку для расчёта освещения будут сэмплироваться текстуры цвета и нормалей [33]. Например, этот подход широко применяется в игровой индустрии при подготовке 3D-художниками моделей для использования в видеоигре. Помимо метрик погрешности, основанных на анализе геометрического представления, можно использовать изображения

модели, снятые с положений, равномерно распределённых вокруг модели. Эта техника называется упрощением, управляемым изображением (англ. image-driven simplification) [76]. К её достоинствам можно отнести высокое качество упрощённых моделей, естественную обработку таких атрибутов как нормали, цвета и текстуры, хорошее сохранение силуэта и сильное упрощение скрытых частей модели. Однако она является одной из наиболее медленных. Работа [55] развивает данную идею с использованием дифференцируемого рендеринга и современных алгоритмов нелинейной оптимизации, позволяя поддержать любую реализацию расчёта освещения.

1.3.8 Создание уровней детализации при упрощении

Большинство алгоритмов, основанных на итеративном применении операций упрощения (таких как стягивание ребра, удаление и вершины и проч.) способны естественным образом генерировать серии уровней детализации путём сохранения промежуточных результатов упрощения.

Многие операции упрощения позволяют в качестве целевой вершины выбирать одну из существующих. В таком случае, множество вершин упрощённой полигональной сетки будет подмножеством вершин оригинальной. Таким образом, вся серия уровней детализации может опираться на одно множество вершин, а уровни детализации будут различаться наборами индексов, определяющими как связывать вершины в треугольники. Это позволяет эффективней хранить уровни детализации, а также избежать необходимости пересчёта вершинных цветов и текстурных координат при упрощении.

1.3.9 Оценка погрешности упрощения

Неотъемлемым аспектом задачи упрощения геометрических представлений является вопрос оценки качества упрощения. Как правило, оценивается *погрешность упрощения*, то есть то, насколько упрощённая модель «хуже» оригинальной. Оценка погрешности бывает необходима в следующих случаях:

- для оценки качества результатов упрощения;
- для сравнения различных алгоритмов упрощения (лучший алгоритм упростит модель до меньшего числа полигонов при меньшей погрешности);

- в процессе упрощения, чтобы в первую очередь производить операции, которые вносят наименьшую погрешность, или вовремя остановить процесс упрощения;
- во время рендеринга для выбора наиболее подходящего уровня детализации.

При этом нет однозначного ответа на вопрос как оценить погрешность упрощения. С одной стороны, можно оценить чисто *геометрическую погрешность*, то есть сравнить исходную и упрощённую геометрию в терминах расстояния:

- между вершинами;
- от вершины до плоскости;
- от вершины до поверхности;
- между поверхностями.

С другой стороны, часто этого может оказаться недостаточно, ведь у геометрии могут быть различные вершинные атрибуты: цвета, нормали, текстурные координате и т.д. И если в результате упрощения сильно искажены значения этих атрибутов, то визуально модель может выглядеть неприемлемо, при том, что геометрическая погрешность находится в разумных пределах. Поэтому может потребоваться оценка *атрибутивной погрешности*.

Работа [29] предлагает инструмент для сравнения поверхностей с использованием сэмплирования. Результат сравнения может выдаваться в виде площадей, объёмов, максимальной и средней погрешности и т.д. В работе [21] предлагается использовать метрику Хаусдорфа с некоторыми оптимизациями вычислений. Также, работы [116, 98] посвящены эффективному вычислению метрики Хаусдорфа. Работа [69] вводит понятие «заметность сетки» (англ. mesh saliency), как метрику важности различных областей сетки с точки зрения визуального восприятия человеком. Работа [114] предлагает улучшенный способ вычисления заметности сетки. Работа [74] содержит обзор способов вычисления заметности для сеток и облаков точек.

1.4 Рендеринг больших сцен в условиях ограниченной памяти

Когда речь заходит о визуализации больших сцен, ключевой проблемой, определяющей скорость рендеринга, может стать нехватка основной или видеопамяти. Если некоторое программное приложение должно работать с большими сценами, минимальным требованием является его стабильная работа в условиях нехватки памяти. Это означает, что приложение должно контролировать объём доступной ему основной и видеопамяти, осуществляя загрузку и выгрузку данных по мере необходимости. Если для рендеринга сцены не хватает памяти, сцена может быть разделена на части. Сначала может быть загружена и растеризована первая часть

сцены, затем она должна быть выгружена и загружена другая часть. Когда все части сцены растеризованы, результирующее изображение выводится на экран. Такой подход, конечно, позволяет получить корректное изображение, но ценой больших временных затрат. Однако, в большинстве случаев большая сцена может быть визуализирована с достаточной частотой кадров, если учесть следующие наблюдения:

- если пользователь рассматривает всю сцену целиком, в силу большого размера сцены это скорее всего означает, что объекты сцены находятся достаточно далеко, поэтому для них могут быть использованы уровни детализации, что снизит требования к объёмам памяти; также в таком случае может помочь исключение маленьких объектов;
- если пользователь находится «внутри сцены» и рассматривает какой-либо объект, как правило это означает, что большинство объектов могут быть исключены техниками отсечения по пирамиде видимости и отсечения невидимых (закрытых) объектов, а для объектов на заднем плане могут использоваться упрощённые представления.

Таким образом, вышеупомянутые техники упрощения и отсечения не только повышают скорость рендеринга за счёт сокращения растеризуемых элементов, но и способны существенно сократить требования по памяти для визуализации сцены. Тем не менее, простое применение этих техник для произвольной сцены, вообще говоря, не гарантирует достижения заданного уровня производительности. Поэтому, в зависимости от конкретного приложения как правило рассматриваются два режима визуализации:

- визуализация с заданным уровнем качества, гарантирующая определённый уровень детализации представлений объектов, а также отображение всех видимых объектов, при этом производительность может быть неудовлетворительной (консервативный режим);
- визуализация с целевым уровнем производительности, когда средства визуализации стремятся поддерживать определённую скорость рендеринга, как правило заданную в виде целевой частоты кадров; в таком случае качество полученных изображений может ухудшаться из-за чересчур упрощённых представлений или отсутствия части объектов (интерактивный режим или режим постоянной частоты кадров (англ. constant frame rate mode)).

Эти режимы могут быть применены как для случая визуализации сложных сцен без ограничений памяти (когда основным ограничением выступает производительность центрального и графического процессоров), так и для визуализации в условиях ограниченной памяти. В таком случае, режим может определять ещё и политику загрузки данных.

Для решения задачи визуализации в условиях ограниченной памяти было разработано множество внешних алгоритмов (англ. external algorithm, out-of-core algorithm). Далее условимся назвать задачу рендеринга в условиях ограниченной памяти внешним рендерингом (по аналогии с англоязычным термином out-of-core rendering).

1.4.1 Способы представления сцены для внешнего рендеринга

Существующие подходы к внешнему рендерингу предлагают различные способы организации и хранения геометрического представления сцены, а также политики загрузки и выгрузки данных. Также, многие подходы дополняются спекулятивной опережающей загрузкой (англ. prefetching). Задача опережающей загрузки состоит в том, чтобы предугадать какие данные понадобятся в скором времени, и обеспечить их присутствие в основной или видеопамяти к тому моменту, как они понадобятся.

В работе [118] описывается алгоритм, представляющий сцену в виде графа иерархических уровней детализации. Каждый узел графа содержит упрощённое представление поддерева (если точнее, массив упрощённых представлений с различной степенью детализации). Для целей внешнего рендеринга в основную память изначально загружается только скелет графа. Скелет содержит минимально необходимый набор данных для обхода графа и принятия решения о рендеринге тех или иных узлов: данные об узлах и связях между ними, ограничивающие объёмы и значения геометрический погрешности узлов. Все геометрические представления хранятся на диске. На каждом кадре осуществляется обход графа, начинающийся с корневого узла. Во время обхода осуществляются проверки отсечения и точности. На их основе строится так называемый фронт — набор представлений для рендеринга на текущем кадре, соответствующий подмножеству разреза графа. Алгоритм использует два параллельных процесса: процесс P_R осуществляет обход графа и рендеринг, в то время как процесс P_I занимается загрузкой представлений с диска. P_R обходит скелет графа сцены и вычисляет фронт с учётом текущего положения камеры. В это время P_I не простаивает, а занимается опережающей загрузкой для предыдущего кадра. Когда P_R завершает вычисление фронта, он отправляет сигнал P_I, и P_I начинает загрузку тех представлений из фронта, которых нет в основной памяти. При этом P_R не ждёт пока все представления будут загружены, а осуществляет рендеринг загруженных представлений. Когда P_I загрузил все представления из фронта, он переходит к опережающей загрузке.

Опережающая загрузка пытается предсказать изменения фронта двух типов:

– в результате движения камеры могут стать видны новые узлы;

– в результате приближения или отдаления от объектов могут потребоваться соответственно более детальные или более упрощённые представления.

Опережающая загрузка в работе [118] использует расширенную пирамиду видимости с приоритезацией на основе угла между объектом и линией зрения для предсказания появления объектов, а также опережающую загрузку детей и родителей узлов фронта для предсказания ситуаций переключения уровней детализации.

Любой внешний алгоритм должен определять условия удаления или замещения данных в основной памяти. В работе [118] используется вариация политики «наиболее давно использовавшийся» (англ. least recently used, LRU), которая гарантирует, что не будут удалены недавно загруженные представления или представления из фронта.

В работе [35] представлена система iWalk для интерактивного внешнего рендеринга больших моделей. Данная система использует три потока: поток рендеринга отвечает за определение видимых узлов иерархии модели и отдаёт запросы на загрузку геометрическому кэшу, второй поток определяет узлы для опережающей загрузки и отправляет команды на опережающую загрузку геометрическому кэшу, а третий поток управляет геометрическим кэшем. Система поддерживает приближённый и консервативный режимы. В работе также описывается инкрементальный и быстрый внешний алгоритм построения иерархического представления большой модели на устройстве хранения данных, который подразумевает использование одного файла для хранения скелета и хранение геометрического представления каждого узла в отдельном файле. Иерархия представляет собой окто-дерево, листовые узлы которого содержат геометрию модели. Во время рендеринга для определения видимых узлов используется алгоритм приоритезированной послойной проекции (англ. prioritized-layered projection, PLP) для приближённого режима и его консервативное расширение cPLP для консервативного режима. Следует отметить, что система iWalk не использует уровни детализации.

В работе [37] представлен внешний алгоритм рендеринга на основе HLOD, который не требует хранения полного скелета в основной памяти, а для того, чтобы уточнить представление какого-либо узла не требуется загрузка всех его детей. В отличие, например, от [118], где скелет используется для обхода графа и для определения фронта, данный алгоритм загружает узлы последовательно, начиная с корня дерева HLOD. При этом должно выполняться следующее правило: если узел загружен, то загружены и все его предки. Таким образом, загруженные узлы образуют поддерево. В частности, это позволяет быстро переключаться на более упрощённые представления, поскольку они всегда загружены. Неудобной особенностью HLOD для внешнего рендеринга может оказаться тот факт, что узел содержит представление всего поддерева. Это означает, что родительский и дочерний узлы не могут быть растеризованы одновременно,

поскольку родительское представление уже включает дочернее. При работе внешнего алгоритма может возникнуть ситуация, когда узел имеет высокую погрешность, но при этом лишь часть детей этого узла загружены в основную память. В таком случае приходится либо растеризовать узел, имеющий недостаточное качество, (в случае интерактивного режима) или же ждать загрузку всех детей (в случае консервативного) для обеспечения целевого уровня качества. Одним из основных результатов работы [37] является подход, когда отсутствующее в памяти представление узла заменяется соответствующей частью представления его родителя при помощи плоскостей отсечения OpenGL.

1.4.2 **Рендеринг больших сцен с использованием уровней детализации**

В одной из ранних работ [20] описывается система для рендеринга больших сцен, в которой ближняя геометрия отображается с уровнями детализации, а дальняя — в виде текстурированной сетки глубины. Система автоматически балансирует экранные погрешности упрощённой геометрии и искажений текстурированной сетки глубины. Приводятся схемы опережающей загрузки и управления данными. Работа [70] предлагает алгоритм рендеринга ландшафтов используя кластеры треугольников, которые кэшируются на графическом процессоре и позволяют разгрузить центральный процессор. В работе [120] для рендеринга используется трассировка лучей, кэширование данных и стратегия загрузки данных по запросу, а также иерархическая гибридная схема аппроксимации наподобие воксельного представления для ещё не загруженной геометрии. Работа [68] предлагает сочетать подход стоимость-выгода из работы [42] с иерархическими уровнями детализации [40] при рендеринге во внешней памяти, а также улучшает функцию гистерезиса, которая используется для минимизации мерцаний при частой смене уровней детализации. Работа [127] предлагает алгоритм внешнего рендеринга, в котором используются не только упрощения геометрии, но и отсечения окклюзии. В нём модель представляется как иерархия кластеров прогрессивных сеток. Иерархия кластеров используется для грубого выборочного уточнения, а прогрессивные сетки — для мелкозернистого локального уточнения. Работа [53] посвящена внешнему рендерингу с тенями, используя гибридное представление из точек и полигонов. Предлагается метод выбора и опережающей загрузки уровней детализации для рендеринга с тенями. Работа [27] предлагает алгоритмы вычисления видимости и организацию данных для сочетания уровней детализации с отсечениями окклюзии для областей сцены.

Классическая работа [36] посвящена эффективному предсказанию того, какая геометрия вскоре станет видимой, с целью опережающей загрузки. Работа [135] описывает схему, которая изучает поведение каждого пользователя и строит предсказания для опережающей загрузки.

Работа [52] адресована проблеме возникновения трещин между представлениями из иерархии уровней детализации, когда требуется либо существенно увеличивать число треугольников на границах, либо производить дорогостоящую сшивку во время рендеринга. Такая проблема возникает при построении иерархических уровней детализации сверху-вниз, когда модель представляется единым набором треугольников и «нарезается» на части. Предлагается алгоритм внешнего рендеринга, закрывающий трещины так называемыми «толстыми границами». Аналогичная идея для геометрии в виде NURBS поверхностей представлена в работе [23].

Говоря о задаче кластеризации сверху-вниз, невозможно не упомянуть о технологии Unreal Engine Nanite [88] и о подобных ей реализациях [34]. В последние годы графические API претерпевают значительное развитие в сторону передачи работы с центрального процессора на графический. Становится возможным реализовывать многие операции сразу на графическом процессоре, минуя центральный. Появился так называемый сеточный шейдер (англ. mesh shader), как более эффективная альтернатива вершинному шейдеру (англ. vertex shader). Технология Unreal Engine Nanite использует нововведения графических API и обеспечивает рендеринг больших полигональных сеток, как правило, представляющих собой топологически замкнутые двумерные поверхности, на основе подготовленных деревьев кластеров. Кластеризация сеток осуществляется сверху-вниз с последующим упрощением фрагментов. Большое внимание уделяется вопросу устранения трещин на границах кластеров. Обход загруженного в видеопамять дерева кластеров и рендеринг отобранных кластеров осуществляются непосредственно на графическом процессоре, причем загрузка данных кластеров происходит напрямую с диска в видеопамять.

В статье [128] предлагается схема управления памятью графического процессора при рендеринге больших полигональных сеток, статья [56] адресована выборочному уточнению большой сетки при трассировке лучей.

В работах [100, 99] описывается представление сцены и эффективная передача данных при клиент-серверном взаимодействии, причём во второй работе рассматриваются динамические сцены. Работа [132] посвящена интерактивному рендерингу по сети, в особенности вопросам минимизации вычислительной нагрузки на клиенте, а также сетевой задержке при передаче данных с сервера на клиент. Работа [129] посвящена веб-визуализации больших моделей городов, в том числе предлагая использовать картинки с изображениями зданий, генерируемые по запросу на сервере.

Работа [104] предлагает систему рендеринга с уровнями детализации, учитывающую архитектуру современных графических процессоров. В ней используются морфинги геометрии для плавного перехода между уровнями детализации, а также имеется поддержка текстур и уровни детализации для текстур. В работе [91] предлагается алгоритм упрощения сетки на графическом процессоре, а также подход к внешнему рендерингу, минимизирующий передачу данных между центральным и графическим процессором, а в следующей работе от тех же авторов [92] приводится подход, учитывающий количество видеопамяти. Диссертация [93] посвящена разработке параллельных алгоритмов для рендеринга больших моделей с учётом архитектуры современных графических процессоров. В работе [122] рассматривается задача, когда не только геометрия сцены, но и данные об источниках света не помещаются в видеопамять. Предлагается эффективная схема управления данными с учётом ограничений по видеопамяти и оптимизации пересылок данных между центральным и графическим процессором. Однако в данном случае речь идёт не об интерактивном рендеринге, а о длительном фотореалистичном рендеринге.

В работе [101] предлагается способ страничной организации данных на диске для быстрого доступа к данным во время навигации по сцене. Работа [26] предлагает схему размещения данных на диске для быстрого поиска, использующую оптимальную избыточность данных.

Оригинальные работы по уровням детализации подразумевают использование полигональных сеток в качестве геометрических представлений. Полигональная сетка была и остаётся удобным и по сей день широко используемым типом геометрии, однако заметим, что концепция уровней детализации (а также иерархических уровней детализации) успешно обобщается и на другие типы геометрии: NeRF [70], 3D Gaussian [65], облако точек [90], воксельное представление [85, 134], NURBS [23]. В работе [78] предлагается использовать гибридное представление полигональная сетка + воксель.

1.5 Уровни детализации для моделей зданий и городов

Одними из главных прикладных областей, в которых возникают задачи рендеринга больших моделей, являются моделирование зданий и городов. Приведём обзор литературы, в которой задачи генерации и использования уровней детализации рассматриваются в контексте визуализации зданий и городов.

Работы [102, 71] предлагают алгоритмы упрощения полигональных сеток, нацеленные на упрощение моделей зданий. Используются особенности моделей зданий, связанные с наличием

больших планарных областей. Ставится цель сохранить основные особенности здания, чтобы его форма оставалась узнаваемой, но убрать несущественные детали. Работа [131] учитывает информацию об особенностях зданий в текстурах во время упрощения. Работа [119] посвящена вопросу классификации с учётом семантики при упрощении моделей зданий. На вход подаётся детальная полигональная сетка, алгоритм способен классифицировать планарные структуры зданий, выделять деревья, крыши и фасады и фильтровать ненужные детали. На выходе алгоритма для зданий получаются водонепроницаемые сетки простой формы. Алгоритм способен генерировать уровни детализации как уровни проработки (англ. level of development).

Работа [125] посвящена упрощению моделей зданий и интерактивному рендерингу моделей городов. Для близлежащих зданий используются индивидуальные уровни детализации, а для удалённых зданий — кластеры, полученные путём объединения 2D отпечатков зданий с экструдированием по высоте. Работа [47] предлагает улучшения качества упрощения, размера текстур и производительности рендеринга, решая проблемы неэффективного использования памяти под текстуры и задержек с просадками качества при интерактивной навигации. В работе [133] предлагается использовать гибридные представления уровней детализации (точки, линии, сплэты) для моделей городов. Работа [129] посвящена веб-рендерингу больших моделей городов.

При создании иерархических уровней детализации происходит объединение объектов. Для текстурированных моделей естественным подходом при этом является объединение их текстур в текстурный атлас. В работах [83, 130] рассматриваются задачи построения оптимальных текстурных атласов для урбанистических моделей.

1.6 Уровни детализации для динамических сцен

Ряд работ посвящён уровням детализации для деформирующихся полигональных сеток. Причём динамика имеет детерминированный характер, то есть сетка может быть представлена как функция от времени, или как последовательность сеток. Реализация, описанная в работе [112], поддерживает динамическое изменение позиций и атрибутов вершин, связности и топологии. Сетка представляется в виде графа T-DAG. Также предлагается алгоритм инкрементального построения графа, позволяющий начать проигрывать анимацию пока новые шаги по времени ещё вычисляются. Эта идея получила дальнейшее развитие в работе [113]. Теперь авторы предлагают производить упрощение не только в пространственном аспекте, но и во временном. Временная информация разделяется на высоко и низкочастотную, что позволяет быстро производить грубые обновления во временном пространстве, которые могут быть адаптивно уточнены для получения более высокой детализации. Работа [121] предлагает

использовать алгоритм на основе сэмплирования точек для рендеринга анимаций. Для представления анимированной сетки с несколькими уровнями детализации создается многоуровневое представление, состоящее из предварительно отфильтрованных сэмплов точек и треугольников. Предлагается алгоритм выборки и стратификации для эффективного создания подходящих наборов сэмплов точек для движущихся полигональных сеток. Работа [66] посвящена расширению концепции «прогрессивной сетки» на временной фактор. Предлагается структура, которая позволяет извлекать желаемое упрощённое представление для любого кадра анимации. Упрощение сетки происходит итеративным схлопыванием рёбер, в результате чего многоуровневая сетка представляется как иерархия операций схлопывания рёбер, что позволяет извлекать из неё представление с произвольной степенью упрощения. Во время анимации происходят обновления этой иерархии, чтобы она оставалась актуальной. В работе [95] рассматривается задача рендеринга деформирующихся изоповерхностей, возникающая в задачах симуляции жидкости. Вся анимация представляется как четырёхмерная тетраэдрическая сетка. Изоповерхность для желаемого момента времени может быть восстановлена путём пересечения тетраэдрической сетки с трёхмерной гиперплоскостью. Предлагается способ интерактивного рендеринга таких четырёхмерных тетраэдрических сеток на основе построения внешней многоуровневой структуры данных, основанной на упрощении с метрикой квадрик. Недавняя работа [25] адресована проблемам эффективного рендеринга на графическом процессоре при четырёхмерных пространственно-временных числовых симуляциях. Сначала производится тесселяция геометрии для предопределённых временных шагов, а затем эти тесселяции соединяются в параметрическом пространстве каждой геометрической сущности и формируют тетраэдры. Приводятся алгоритмы интерактивной визуализации четырёхмерных сеток, включающие тетраэдры и пентатопы, с использованием геометрического шейдера и без него.

Работа [117] посвящена созданию и использованию иерархических уровней детализации для рендеринга толпы анимированных персонажей. Используются две комплементарных структуры данных. Первая — это скелет с окто-деревьями, ассоциированными с каждой конечностью, который используется для вычисления уровней детализации геометрии и анимации. Вторая структура — это разбиение сцены по тайлам, позволяющее выбирать уровни детализации для геометрии, анимации и поведения персонажей. Поверх этого строится квад-дерево, позволяющее производить дальнейшие оптимизации рендеринга. Например, объединять геометрию разных персонажей в частях сцены, сильно удалённых от камеры.

В работе [96] также рассматривается задача рендеринга больших толп анимированных персонажей. Предлагается реализация непрерывных уровней детализации (позволяющая избежать артефактов резкого переключения, возникающих у дискретных уровней детализации)

с использованием технологии инстанцирования геометрии (как раз появившейся в то время в графических API) для эффективного рендеринга.

Таким образом, исследования в области применения уровней детализации для динамических сцен в основном посвящены деформирующейся геометрии, возникающей при анимировании персонажей или при научных симуляциях, таких как симуляция жидкости. Можно констатировать, что исследованию динамических сцен, в которых объекты появляются и исчезают, не уделялось должного внимания. Тем не менее такие сцены возникают во множестве прикладных областей, в особенности в задачах 4D-моделирования зданий (визуализация плана строительства здания) и в информационном моделировании городов (визуализация плана застройки города, или же наоборот, истории развития города).

Глава 2. Метод HDLOD для динамических сцен с детерминированным характером событий

2.1 Сцена

Пусть сцена $S(t)$ определена в трёхмерном евклидовом пространстве E^3 на моделируемом временном периоде $t \in [0,T]$ и представлена как линейный список объектов $s(g_s, p_s(t), m_s(t)) \in S$. Каждый объект имеет неизменное геометрическое представление $g_s \subseteq E^3$. Статус присутствия объектов в сцене определяется их функциями присутствия $p_s(t)\colon [0,T] \to \{0,1\}$ таким образом, что функция $p_s(t)$ принимает единичное значение, если объект s присутствует в сцене в момент времени t, и нулевое значение — если отсутствует. Положение объекта определяется функцией положения $m_s(t)\colon [0,T] \to M$, где M — множество матриц размерности 4×4. Следует упомянуть, что движение объектов может быть задано разными способами, например, как набор ключевых точек, в которых заданы позиция и ориентация объекта, с указанием времени, когда объект находится в этой ключевой точке. Положение объекта в каждый момент времени может определяться как результат интерполяции положений в ближайших ключевых точках. Однако, для целей рендеринга, как правило, используется задание положения объекта с помощью модельной матрицы преобразования, более того, любое представление положения объекта может быть выражено при помощи матрицы. Поэтому без ограничения общности можно утверждать, что в каждый момент времени $t \in [0,T]$ положение объекта может быть представлено матрицей преобразования.

На рис. 2.1 показан пример динамической сцены, моделирующей строительство бизнес-центра в соответствии с предварительно подготовленным планом проекта. По мере изменения модельного времени элементы конструкций и оборудование устанавливаются на строительной площадке или удаляются. Такое поведение можно воспроизвести при помощи функций присутствия. Примеры типичных функций присутствия представлены на рис. 2.2. Элементы ландшафта остаются неизменными на протяжении всего моделируемого периода. Также в ходе строительства по строительной площадке перемещается различная техника (рис. 2.3).

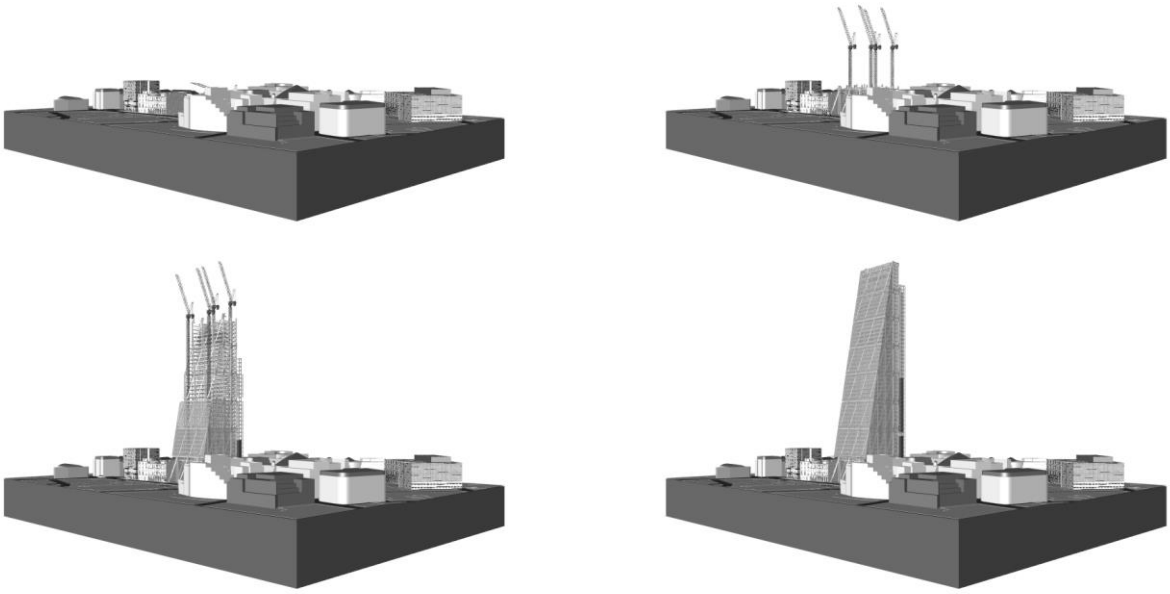


Рисунок 2.1 — Пример сцены с детерминированной динамикой, моделирующей строительство бизнес-центра.

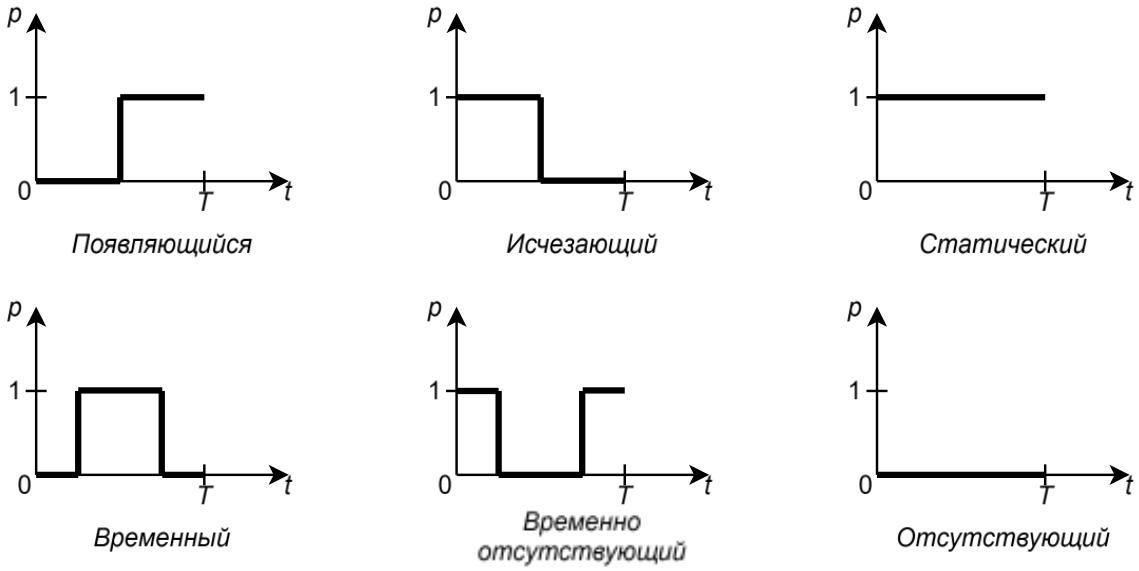


Рисунок 2.2 — Примеры некоторых типичных функций присутствия объекта.

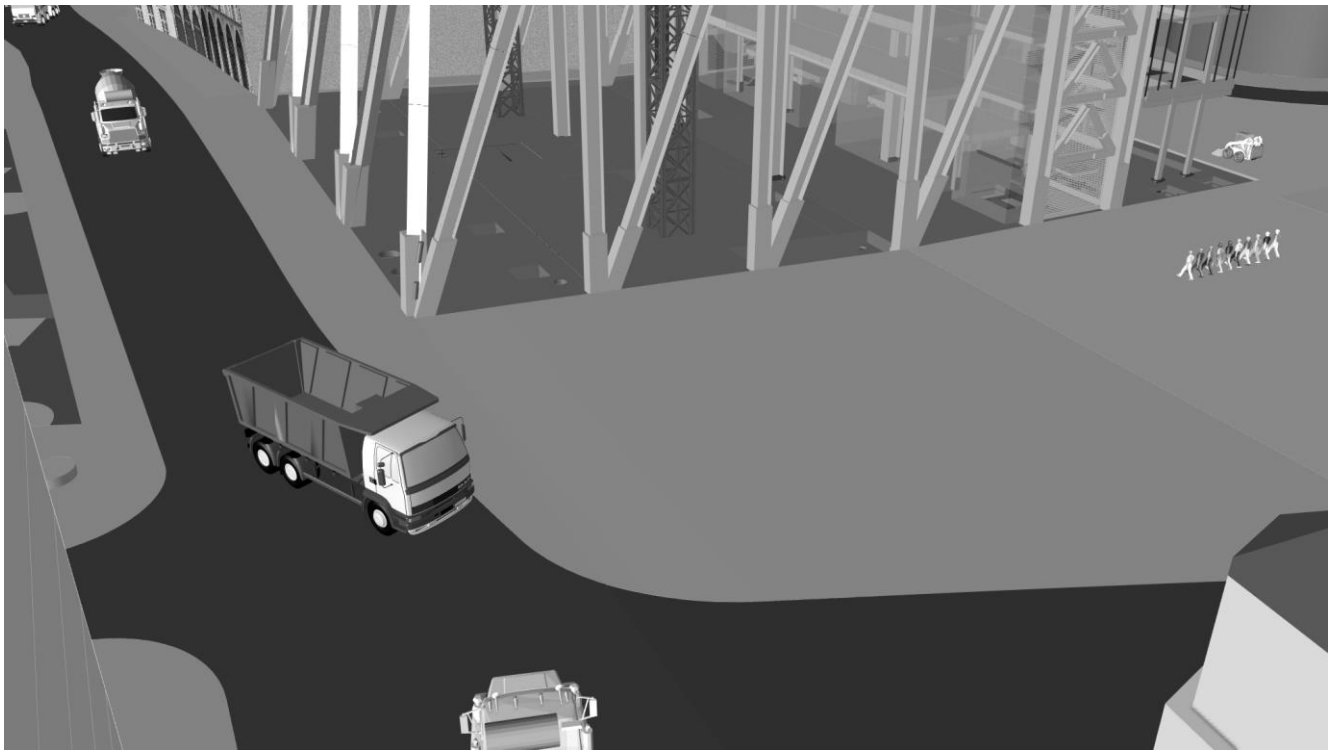


Рисунок 2.3 — Примеры динамических объектов в индустриальных сценах: автомобили, техника (в том числе и строительные краны) и люди.

В дальнейшем будем предполагать, что геометрическое представление объектов сцены задано в виде сеток треугольников. Не будем делать никаких предположений о топологии граничных представлений объектов и не будем требовать, чтобы граничное представление было связным или являлось многообразием, поскольку для целей рендеринга эти требования являются избыточными. Более того, любое геометрическое представление может быть приведено к сетке треугольников.

По динамическому поведению все объекты сцены могут быть поделены на три класса.

В класс *статических* объектов попадают объекты, присутствие и положение которых не изменяется на протяжении всего моделируемого периода, то есть $\forall t \in [0, T]\ p_s(t) = 1, m_s(t) = I$, где I — единичная матрица.

Класс *псевдо-динамических* объектов состоит из объектов, которые появляются и исчезают, однако их положение остаётся неизменным: $\forall t \in [0, T]\ m_s(t) = I$, но $\exists t_1, t_2 \in [0, T]$, такие что $p_s(t_1) \neq p_s(t_2)$.

Наконец, класс *динамических* объектов состоит из объектов, положение которых меняется с течением времени: $\exists t_1, t_2 \in [0, T]$, такие что $m_s(t_1) \neq m_s(t_2)$. Они могут появляться/исчезать или постоянно присутствовать.

2.2 Метод иерархических динамических уровней детализации HDLOD

Будем называть иерархическими динамическими уровнями детализации HDLOD (от англ. hierarchical dynamic levels of detail) дерево кластеров $C = \{c(g_c, p_c(t), m_c(t), b_c(t), \delta_c(t))\} \cup \{\prec\}$, представленное множеством кластеров c и отношений агломерации $\prec$. Отношение агломерации $c' \prec c$ соответствует отношению ребёнок-родитель в дереве, а также отражает тот факт, что родительский кластер $c \in C$ был сформирован из упрощённых геометрического и поведенческого представлений дочернего кластера $c' \in C$. Представление кластера является упрощённым агрегированным представлением его детей. Таким образом, кластеры у корня дерева имеют представления с наименьшим уровнем детализации, а листовые кластеры — с наибольшим.

Каждый кластер c имеет геометрическое представление g_c, а также функции, описывающие его динамическое поведение, определённые на всём интервале моделирования $[0, T]$. Это функция присутствия $p_c(t): [0, T] \to \{0,1\}$, функция положения $m_c(t): [0, T] \to M$, функция ограничивающего объёма $b_c: [0, T] \to B$, где B — множество объёмов в E^3, и функция геометрической погрешности $\delta_c: [0, T] \to [0, \Delta_c]$, где Δ_c — максимальное локальное отклонение геометрии кластера от геометрии соответствующих ему оригинальных объектов на всём временном отрезке $[0, T]$.

Задача упрощения геометрических представлений является хорошо изученной. В то же время, упрощению динамического поведения объектов уделялось мало внимания. Поскольку объекты, агрегированные в кластеры, могут иметь различное динамическое поведение, присутствие кластера неизбежно становится связанным с его погрешностью. Так, в некоторый момент времени часть объектов, соответствующих некоторому кластеру, могут присутствовать, а часть отсутствовать. При этом необходимо принять решение о необходимости отображения всего кластера. Если отобразить кластер, это повлечёт погрешность, связанную с тем, что часть объектов отсутствует, однако они были отображены в качестве представления кластера. И наоборот, если проигнорировать кластер, это повлечёт погрешность, связанную с тем, что присутствующие объекты не были показаны. Для описания такого рода погрешности вводится функция геометрической погрешности кластера $\delta_c(t)$, которая описывает пространственную погрешность упрощённого представления кластера c по сравнению с оригинальными объектами с учётом их присутствия в момент времени $t \in [0, T]$.

Функция ограничивающего объёма $b_c(t)$ позволяет определить ограничивающий объём кластера c в любой момент времени $t \in [0, T]$. Ограничивающий объём может быть задан как

ограничивающий параллелепипед, ориентированный произвольно (oriented bounding box, OBB) или по осям координат (axis-aligned bounding box, AABB), или же как любая другая структура, в том числе иерархия ограничивающих объёмов, которая может изменяться с течением времени. Главным требованием к ограничивающим объёмам является эффективность тестов их пересечения с пирамидой видимости, для быстрого определения видимости кластеров.

По аналогии с классификацией объектов сцены, кластеры могут быть поделены на статические, псевдо-динамические и динамические. Статические кластеры агрегируют только статические объекты. Их положение, ограничивающий объём, присутствие и геометрическая погрешность не зависят от времени, поэтому такие кластеры могут быть представлены как $c(g_c, p_c, m_c, b_c, \delta_c)$. Псевдо-динамические кластеры агрегируют псевдо-динамические и, возможно, статические объекты. Их положение и ограничивающий объём остаются неизменными с течением времени моделирования, в то время как присутствие и геометрическая погрешность меняются $c\big(g_c, p_c(t), m_c, b_c, \delta_c(t)\big)$. Динамические кластеры агрегируют динамические объекты. Все их атрибуты являются функциями от времени $c\big(g_c, p_c(t), m_c(t), b_c(t), \delta_c(t)\big)$. Пример дерева HDLOD содержащего статические, псевдо-динамические и динамические кластеры показан на рисунке 2.4.

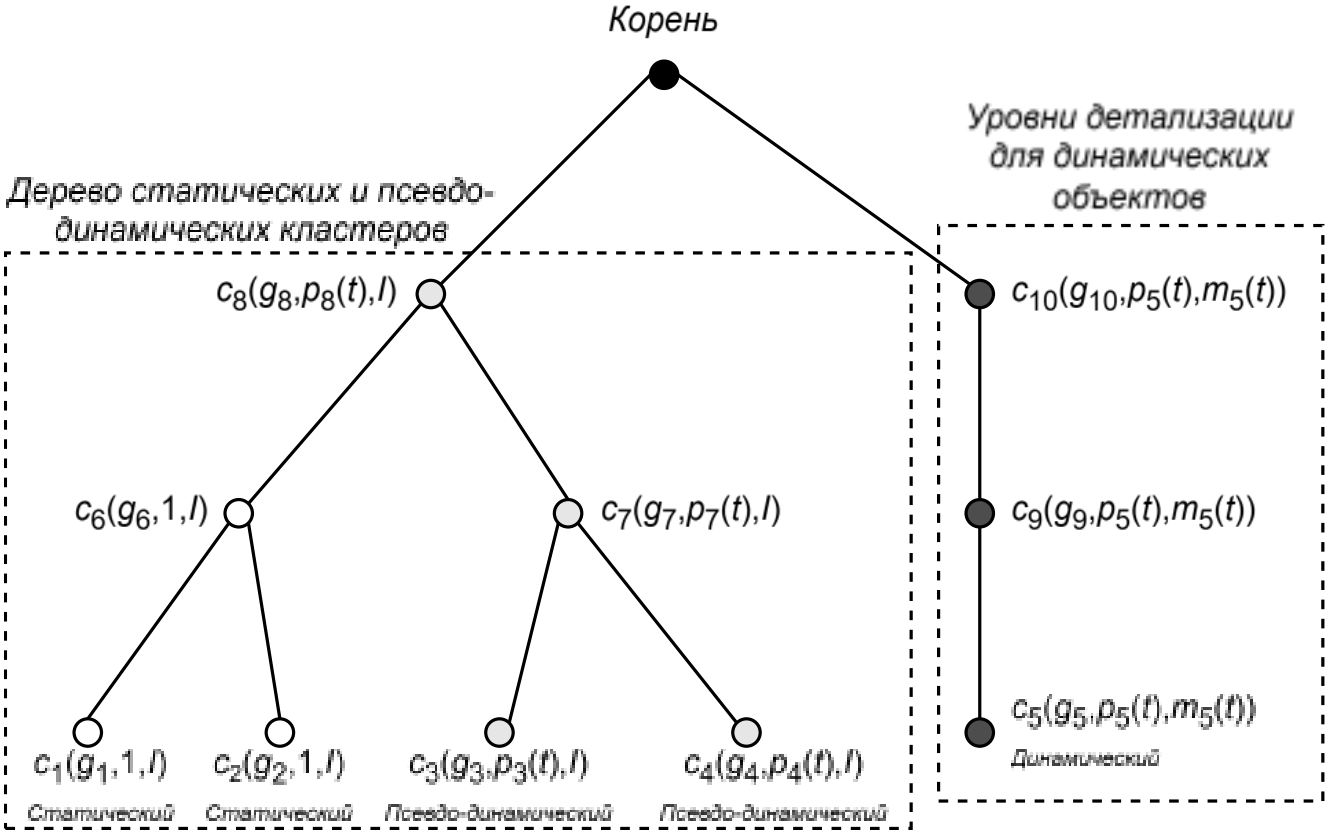


Рисунок 2.4 — Пример дерева HDLOD, содержащего статические, псевдо-динамические и динамические кластеры.

Вне зависимости от конкретного алгоритма кластеризации, для целей эффективного рендеринга кластеры дерева HDLOD должны удовлетворять следующим свойствам:

1. Ограничивающий объём родительского кластера $b_c(t)$ покрывает ограничивающий объём $b_{c'}(t)$ любого дочернего кластера на всём временном периоде: $\forall c' \prec c \; \forall t \in [0,T] \;\; b_{c'}(t) \subseteq b_c(t)$.
2. Геометрическая погрешность $\delta_{c'}(t)$ дочернего кластера не превышает погрешность $\delta_c(t)$ родительского кластера на всём временном периоде: $\forall c' \prec c \; \forall t \in [0,T] \;\; \delta_{c'}(t) \leq \delta_c(t)$.

Геометрическая погрешность листовых кластеров равняется нулю: $\forall t \in [0,T], \forall c \in C, \text{такого что } \nexists c' \in C \; c' \prec c \;\; \delta_c(t) = 0$. Это означает, что их геометрическое и поведенческое представление является точным и может удовлетворить любым требованием к погрешности. Как правило, этого можно добиться путём использования оригинальных объектов сцены в качестве листовых кластеров.

2.3 Рендеринг HDLOD

Главная цель построения дерева кластеров HDLOD для сцены — это эффективный рендеринг сцены. Рендеринг осуществляется на некоторый (текущий) момент модельного времени $t_0 \in [0,T]$. При этом осуществляется обход дерева кластеров, начиная с корневого узла и распространяясь к листовым узлам. При посещении каждого кластера необходимо принять решение о том отображать ли его, пропустить, или посетить более детальные дочерние представления. Для этого происходит вычисление значений его функций поведения для текущего момента модельного времени.

Первым делом выполняется проверка видимости кластера. Для этого используется его ограничивающий объём $b_c(t_0)$ и производится проверка отсечения по области видимости. Может также выполняться отсечение окклюзии. Если ограничивающий объём кластера признан невидимым, кластер и его поддерево исключаются из дальнейшего анализа.

В противном случае выполняется проверка качества кластера. Для этого, по значению функции погрешности $\delta_c(t)$ вычисляется экранная погрешность SSE в пикселях. Для стандартной перспективной проекции SSE может быть вычислена по формуле:

$$SSE = \frac{\delta_c(t) \cdot R}{2 \cdot d \cdot \tan\frac{\theta}{2}}$$

где R — разрешение экрана в пикселях по высоте, d — расстояние от камеры до кластера, а θ — вертикальный угол раствора пирамиды видимости (рисунок 2.5).

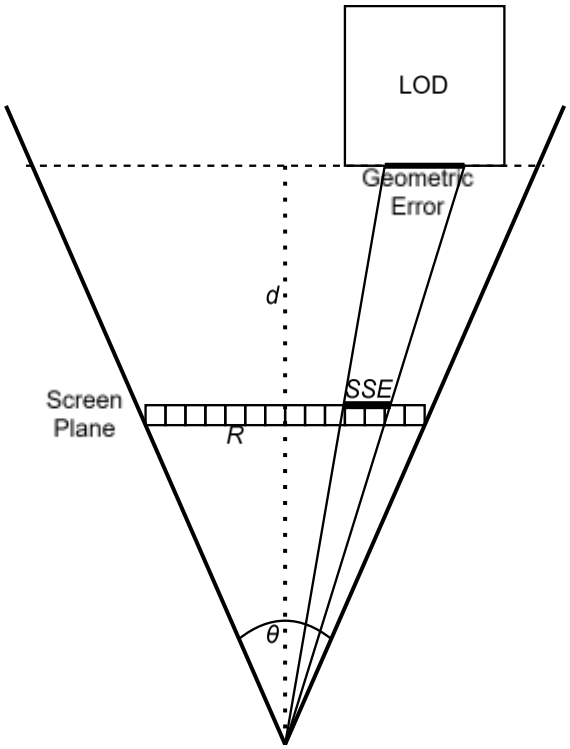


Рисунок 2.5 — Иллюстрация формулы расчёта экранной погрешности.

Если экранная погрешность кластера не превышает порогового значения, этот кластер выбирается для отображения. В противном случае, осуществляется обход дочерних кластеров.

Если кластер выбран для отображения, осуществляется проверка его присутствия. Если $p_c(t_0) = 1$, то считается, что кластер присутствует в текущий момент времени и осуществляется рендеринг его геометрического представления g_c в положении $m_c(t_0)$. В противном случае кластер считается отсутствующим и не отображается.

Обход гарантированно завершится в листовых кластерах, поскольку у них $\delta_c(t) = 0$ и их качество всегда является достаточным.

Ниже представлен псевдокод алгоритма:

функция «посетить кластер»

если проверка видимости (область видимости, окклюзия) говорит, что кластер является видимым

если качество кластера является достаточным

если кластер присутствует ($p_c(t_{current}) = 1$)

отобразить кластер

иначе посетить детей

иначе пропустить кластер и его детей

Следует отметить следующие особенности дерева HDLOD.

Утверждение. Дерево HDLOD является иерархией ограничивающих объёмов. Это следует из того, что ограничивающие объёмы всех детей лежат внутри ограничивающего объёма родителя $\forall c \in C\ \forall c' \prec c\ \forall t \in [0,T]\ \ b_{c'}(t) \subseteq b_c(t)$.

У этого утверждения есть два следствия, которые позволяют производить эффективные проверки видимости.

Следствие 1. Если кластер невидим, то можно с уверенностью утверждать, что невидимы все его дети.

Следствие 2. Если ограничивающий объём кластера полностью попадает в область видимости, то можно отключить проверку попадания в область видимости при посещении дочерних кластеров.

2.4 Многовариантная модификация метода HDLOD

В ряде приложений возникает задача визуализации одной сцены в нескольких 3D окнах с разной динамикой в каждом окне. Поскольку динамика является частью HDLOD кластеров, в таких приложениях потребуется генерировать отдельные HDLOD представления для каждого окна. Проводить кластеризацию несколько раз — это вычислительно затратно, а хранить несколько деревьев HDLOD затратно по памяти. Другим обременительным ограничением является необходимость пересчёта всего дерева HDLOD, в случае, когда поменялась только динамика. Чтобы преодолеть эти ограничения, предлагается модификация HDLOD метода, названная многовариантным методом HDLOD. Она предусматривает раздельное хранение данных о динамике и геометрических данных кластеров, позволяя, тем самым, применять различные динамические сценарии для одних и тех же сцен, и обновляя только данные о динамике в случае изменений в поведении объектов.

В многовариантной версии метода, иерархические динамические уровни детализации HDLOD определяются как дерево кластеров $C = \{c(g_c, b_c, \delta_c)\} \cup \{\prec\}$ и множество динамик $D = \{d_s(p_s(t), m_s(t), \sigma_s(t), \tau_s(t))\}$. Теперь параметры кластеров более не зависят от времени, что делает их похожими на традиционные иерархические уровни детализации HLOD. Геометрическое представление кластера $g_c = \{g_{prim}(\iota_s)\}$ — это теперь набор примитивов. Каждый примитив соответствует некоторому объекту s и хранит его глобально-уникальный идентификатор ι_s. Это похоже на концепцию пакетной 3D модели (batched 3D model) из стандарта 3D Tiles [13]. Пакетная 3D модель — это полигональная сетка, в которой каждая

вершина может иметь ассоциированную с ней информацию (хранящуюся в таблице пакетов batch table). Метод HDLOD не конкретизирует из каких примитивов состоит геометрическое представление кластера, они могут быть элементами конструктивной сплошной геометрии, аналитически заданной поверхностью или тоже полигональной сеткой. Также, кластер может иметь и единое представление, в котором идентификаторы ассоциированы с каждой вершиной или полигоном. Последний случай является особенно важным с точки зрения эффективного рендеринга, поскольку позволяет оперировать кластером как единой сущностью, и тем самым уменьшить накладные расходы на обработку и число вызовов отрисовки. Здесь важно то, что для каждой части представления кластера должен быть известен идентификатор оригинального объекта. Это позволяет стилизовать части кластера в соответствии с внешним видом оригинальных объектов в различные моменты времени в ходе рендеринга.

Все параметры, которые зависят от времени, переходят в множество динамик $D = \{d_s(p_s(t), m_s(t), \sigma_s(t), \tau_s(t))\}$. Динамика d_s хранит информацию о динамическом поведении оригинального объекта s. Наряду с уже знакомой функцией присутствия $p_s(t)$ и функцией движения $m_s(t)$ здесь также вводится функция цвета $\sigma_s(t)$ и функция прозрачности $\tau_s(t)$, поскольку для динамических сцен зачастую требуется не только моделирования присутствия и движения объектов, но и изменения цвета и прозрачности с течением времени. Тот факт, что динамика выносится в отдельную структуру и не учитывается при кластеризации, позволяет моделировать больше динамических параметров, не усложняя кластеризацию.

Алгоритм рендеринга похож на тот, что используется в оригинальном методе, но есть некоторые отличия. Осуществляется обход дерева кластеров и при посещении узла первым делом производится проверка на попадание в область видимости. Для этого необходимо получить ограничивающий объём кластера. Для статических и псевдо-динамических кластеров ограничивающий объём доступен как неизменный параметр b_c. Динамические кластеры не допускают дальнейшее объединение, что означает, что всё геометрическое представление кластера соответствует одному объекту. Идентификатор этого объекта ι_s может быть получен из геометрического представления кластера, затем в множестве D осуществляется поиск динамики d_s этого объекта по идентификатору, берётся его функция движения $m_s(t)$ и находится преобразование системы координат $m_s(t_0)$. Это преобразоавние может быть применено к b_c, чтобы получить ограничивающий объём на текущий момент модельного времени $b_c(t_0)$.

Присутствие, цвет и прозрачность определяются для каждого примитива кластера с использованием идентификатора ι_s, путём поиска соответствующей динамики d_s и вычисления $p_s(t_0), \sigma_s(t_0), \tau_s(t_0)$.

Во время рендеринга требуется определять присутствие, цвет и прозрачность разных частей кластера по идентификатору ι_s, который ассоциирован с частью геометрического представления кластера g_c, путём нахождения соответствующей динамики d_s и вычисления $p_s(t_0), \sigma_s(t_0), \tau_s(t_0)$. Для эффективного рендеринга кластера в виде единой полигональной сетки в вершинный шейдер передаётся RGBA цвет. RGB значения вычисляются для каждой части кластера по ι_s этой части и вычислению $\sigma_s(t_0)$. Присутствие и прозрачность объединены в альфа-канале цвета. Если $p_s(t_0) = 0$, альфа выставляется 0, чтобы спрятать сооветствующую часть сетки путём отбрасывания фрагментов в шейдере. Если $p_s(t_0) = 1$, то альфа устанавливается в значение функции прозрачности $\tau_s(t_0)$. В программной реализации метода для рендеринга использовался графический API OpenGL. RGBA цвета передаются в вершинный шейдер через shader storage buffer object (SSBO). У каждой вершины сетки есть целочисленный вершинный атрибут, который является индексом в SSBO. Перед рендерингом каждой сетки осуществляется вычисление цветов для вершин сетки и их отправка на графический процессор через SSBO. Они достаются из SSBO в вершинном шейдере и передаются во фрагментный шейдер для расчёта затенения по Фонгу. Фрагментный шейдер отбрасывает фрагменты с нулевой альфой и вычисляет затенение для остальных фрагментов. Для рендеринга прозрачных поверхностей используется алгоритм порядко-независимой прозрачности на основе взвешенного смешивания [84]. Рендеринг сцен с прозрачными объектами осуществляется в два прохода. В первом проходе отображаются только непрозрачные объекты с тестом глубины. Во втором проходе отображаются только прозрачные объекты с тестом глубины относительно непрозрачных, но без записи в буфер глубины. Это означает, что шейдер для первого прохода должен отбрасывать фрагменты, у которых $\alpha \neq 1$ (есть прозрачность), а шейдер для второго прохода должен отбрасывать непрозрачные фрагменты ($\alpha = 1$). Следует также отметить, что, используя ι_s, можно сопоставить с вершинами и другие свойства материала, такие как цвет зеркального отражения, испускаемый цвет, или даже текстуры. Также, ι_s позволяет осуществить выделение объектов по клику мыши, поскольку идентификатор объекта может быть получен из вершин грани, по которой был произведён клик. По сравнению с оригинальным методом, многовариантный метод HDLOD имеет несколько большую вычислительную стоимость рендеринга, поскольку он требует обрабатывать части кластера и искать их динамические данные по идентификатору на CPU и имеет несколько более сложный шейдер на GPU. Однако, это представляется разумной платой за гибкость, которую предоставляет многовариантный метод.

Алгоритм генерации в многовариантной модификации HDLOD метода отличается только тем, что в нём нет необходимости учитывать временную близость кластеров во время кластеризации. Поиск соседей осуществляется только в пространстве, что может выливается в лучшую кластеризацию в терминах плотности кластеров по сравнению с оригинальным методом.

Далее в работе будут рассматриваться алгоритмы генерации и рендеринга только оригинального метода HDLOD.

Глава 3. Генерация HDLOD

Первым этапом генерации HDLOD является классификация объектов сцены на статические, псевдо-динамические и динамические. Для каждого класса используется свой метод формирования кластеров.

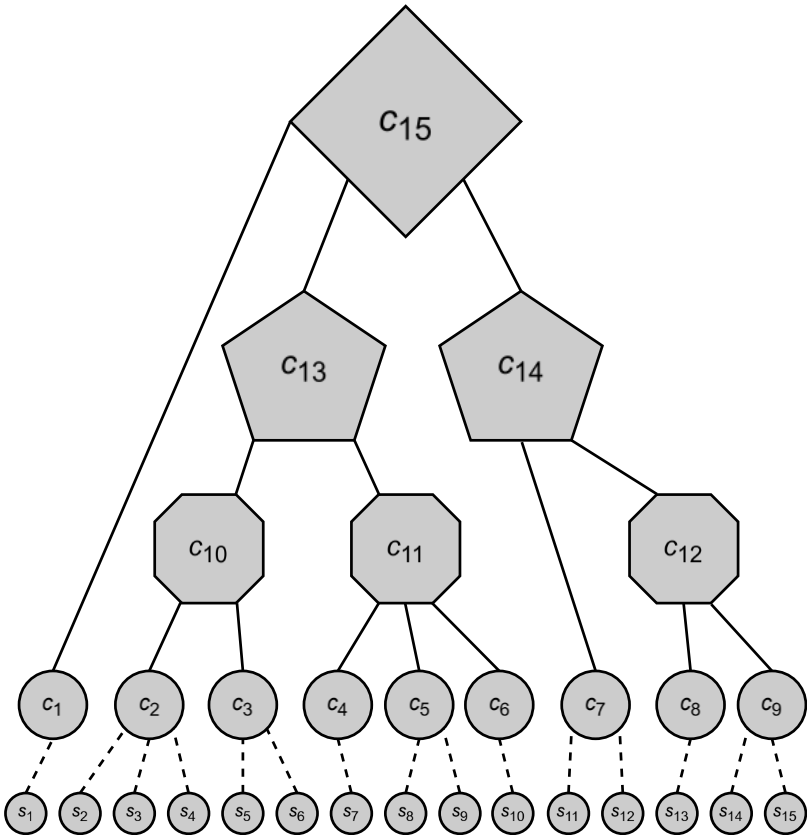


Рисунок 3.1 — Пример генерации HDLOD из набора объектов сцены $s_1 - s_{15}$, которые объединяются в кластеры $c_1 - c_9$ без упрощения, которые в свою очередь объединяются в кластеры $c_{10} - c_{12}$ с упрощением, которые объединяются в кластеры c_{13}, c_{14} с большим упрощением, и в итоге всё объединяется в один кластер c_{15} с наибольшим упрощением.

Статические объекты можно рассматривать как частный случай псевдо-динамических, поэтому статические и псевдо-динамические объекты могут быть объединены в единое дерево кластеров. В этом дереве листовые узлы — это кластеры содержащие точные представления групп объектов. Внутренние узлы — это кластеры, агрегирующие представления соответствующих поддеревьев. При этом с ростом уровня в дереве геометрия и динамическое поведение кластеров становятся всё более упрощёнными, а корень представляет собой наиболее упрощённое представление всей сцены (рисунок 3.1).

Динамические объекты могут иметь различные траектории движения и различные моменты появления и исчезновения. Это существенно осложняет анализ и препятствует их

эффективному объединению в кластеры. Поэтому для каждого динамического объекта создаётся отдельный кластер. В типичных сценах различные объекты могут использовать общую модель. Например, по строительной площадке могут перемещаться несколько экскаваторов, которые имеют одинаковое геометрическое представление, то есть являются экземплярами одной общей модели экскаватора. Это учитывается при формировании кластеров для динамических объектов: они могут ссылаться на общее геометрическое представление. Следует отметить, что для кластеров динамических объектов возможно дополнительно создать родительские кластеры, которые будут содержать упрощённые геометрию, функцию видимости и функцию положения.

В общих чертах метод генерации дерева кластеров HDLOD может быть описан следующим образом:

1. Построение дерева кластеров.
    a. Получение ограничивающих объёмов объектов.
    b. Кластеризация на каждом уровне дерева.
2. Генерация кластеров.
    a. Объединение полигональных сеток дочерних кластеров для формирования полигональной сетки нового кластера.
    b. Упрощение полигональной сетки кластера.
3. Экспорт.
    a. Генерация текстурных атласов.
    b. Экспорт данных в формат для рендеринга.

Первым этапом является *построение дерева кластеров* для статических и псевдо-динамических объектов. На данном этапе используется только информация об ограничивающих объёмах объектов сцены. Поскольку ограничивающий объём представляет собой достаточно компактную структуру данных, можно не беспокоиться об ограничениях по оперативной памяти и загрузить ограничивающие объёмы всех объектов сцены в оперативную память даже для очень больших сцен. Дерево кластеров строится снизу вверх по шагам. На первом шаге запускается *процедура кластеризации*, которая группирует объекты сцены в кластеры. Эти кластеры становятся листовыми узлами дерева HDLOD. На втором шаге та же процедура кластеризации используется уже для группировки этих кластеров в новые (родительские) кластеры. И так далее, пока процесс не сойдётся к единому корневому кластеру.

После того, как дерево построено, запускается *генерация кластеров*, а именно, формирование их геометрических представлений. При этом происходит *объединение полигональных сеток* дочерних кластеров и *упрощение* полученной сетки до целевой геометрической погрешности. Данный этап является самым вычислительно затратным. Поскольку полигональные сетки занимают много места в памяти, необходимо подгружать их в

оперативную память только по мере необходимости, и освобождать память, когда они становятся не нужны. Также, этап генерации кластеров может быть распараллелен, так как кластеры одного уровня не зависят друг от друга, а зависят только от дочерних кластеров. *Алгоритм упрощения полигональной сетки* также заслуживает отдельного внимания, поскольку качество и время генерации в основном зависят именно от него.

Наконец, последний этап — это *экспорт*, при котором данные экспортируются в формат, который будет наиболее удобен при рендеринге. Для совместимости с различным программным обеспечением предпочтительней использовать открытые стандарты. Для представления иерархических уровней детализации существует стандарт Cesium 3D Tiles. HDLOD данные совместимы со стандартом 3D Tiles и могут быть экспортированы в этот формат, что будет показано в главе 5. Самая вычислительно затратная часть этапа экспорта — это *генерация текстурных атласов*. Она необходима по соображениям эффективности и простоты реализации рендеринга. А именно, чтобы на одну полигональную сетку приходилось не более одной текстуры. Поскольку полигональная сетка кластера является результатом объединения сеток множества объектов сцены, то и текстура кластера должна быть получена объединением множества текстур в единый атлас.

В дальнейших разделах будут подробно описаны все эти этапы и алгоритмы.

3.1 Построение дерева кластеров

Для начала рассмотрим общий алгоритм построения дерева HDLOD для статических и псевдо-динамических объектов. Данный алгоритм начинает с индивидуальных объектов сцены и последовательно группирует их во всё большие и большие кластеры. Алгоритм выстраивает дерево кластеров снизу-вверх по шагам. На каждом шаге формируется новый уровень дерева. На первом шаге формируется первый уровень дерева (листовые узлы дерева), на втором шаге — второй уровень дерева (родители листовых узлов). На третьем шаге формируется третий уровень дерева, в котором узел может являться родителем узла со второго или первого уровня. И так процесс продолжается по шагам, пока на последнем шаге не будет сформирован корень дерева. Следует заметить, что уровни дерева в данной терминологии не соответствуют порядку достижения вершины при обходе поиском в глубину из корня (рис. 3.2). Такая нумерация уровней нужна для упрощения распараллеливания генерации кластеров.

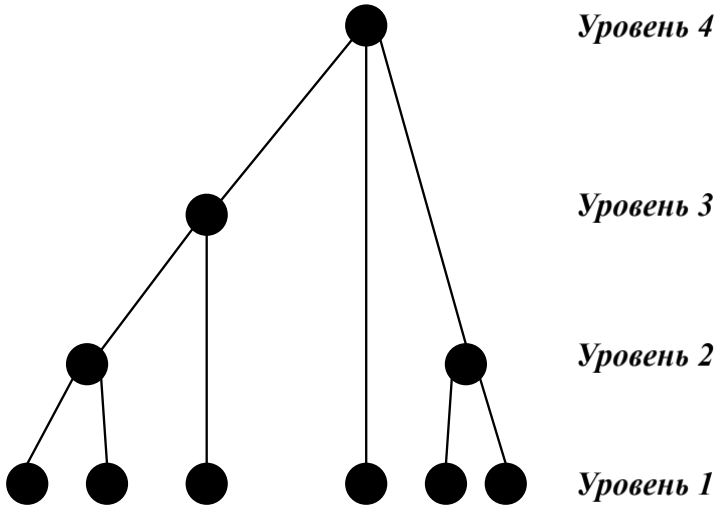


Рисунок 3.2 — Уровни при построении дерева HDLOD.

С ростом уровня растут размеры кластеров и степень их упрощения. Напомним, что на этапе построения кластеров для экономии памяти доступна только информация об ограничивающих объёмах. Обозначим w — линейный размер ограничивающего объёма. Например, если в качестве ограничивающих объёмов используются прямоугольные параллелепипеды, то в качестве w разумно взять диагональ прямоугольного параллелепипеда.

Пусть W — размер всей сцены (например, диагональ ограничивающего параллелепипеда сцены), а L — целевое число уровней дерева. Процесс построения дерева кластеров организован таким образом, что на каждом уровне l (от 1 до L) производится попытка сформировать кластеры размером не более чем некоторое пороговое значение $w(l)$. При этом должны выполняться соотношения: $w(1) < w(2) < \cdots < w(L)$, чтобы обеспечить рост размеров кластеров с ростом уровня, и $w(L) \geq W$, чтобы процесс завершился на уровне L формированием единого кластера для всей сцены. Ограничение по размеру $w(l)$ обеспечивает пространственную близость объектов при кластеризации, то есть соседние объекты объединяются в один кластер пока не будет превышен размер $w(l)$.

Алгоритм учитывает не только пространственный, но и временной аспект. Чтобы объекты попали в один кластер, они должны быть близки не только в пространстве, но и во времени. Введём метрику *временного расстояния* между объектами. Пусть сцена моделируется на временном отрезке $[0, T]$. Пусть два объекта a и b имеют функции присутствия $p_a(t)$ и $p_b(t)$. Тогда временное расстояние между объектами (или расстояние между их функциями присутствия) можно определить как:

$$d_T(a, b) = \int_0^T |p_a(t) - p_b(t)|\, dt$$

Если функции присутствия полностью совпадают, то временное расстояние будет равно нулю. Наоборот, если они полностью не совпадают, то есть там, где одна функция присутствия принимает значение 0, другая принимает значение 1, и наоборот, то временное расстояние будет равно T (рисунок 3.3).

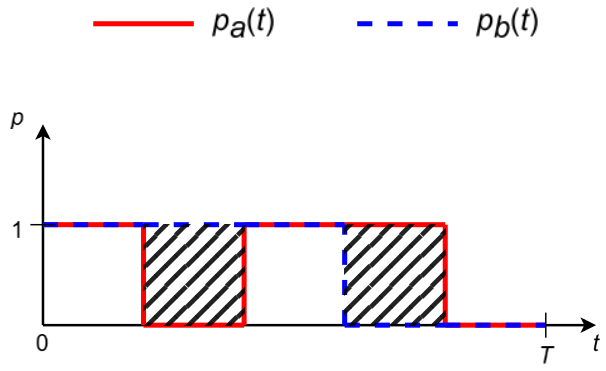


Рисунок 3.3 — Пример вычисления временного расстояния между объектами a и b (значение интеграла показано заштрихованными областями).

Однако, поскольку в один кластер могут попасть много объектов, требуется метрика, которая позволила бы определить степень расхождения множества функций присутствия. Пусть имеется множество объектов O, у каждого из которых есть своя функция присутствия. Тогда метрику расхождения между ними можно определить как:

$$d_T(O) = \int_0^T \left(\max_{o \in O} p_o(t) - \min_{o \in O} p_o(t) \right) dt$$

Значение данной метрики далее будем называть *временным расхождением* и обозначать как γ.

По аналогии с размерами кластеров учитывается и временное расхождение, таким образом, что пороговое значение временного расхождения растёт с ростом уровня. Пусть Γ — временное расхождение всей сцены. Процесс построения дерева кластеров организован таким образом, что на каждом уровне l (от 1 до L) производится попытка сформировать кластеры с временным расхождением не более чем некоторое пороговое значение $\gamma(l)$. При этом должны выполняться соотношения: $\gamma(1) < \gamma(2) < \cdots < \gamma(L)$, чтобы обеспечить рост временного расхождения кластеров с ростом уровня, и $\gamma(L) \geq \Gamma$, чтобы процесс завершился на уровне L формированием единого кластера для всей сцены. Также важно, чтобы выполнялось соотношение $\gamma(1) = 0$. Это необходимо потому, что листовые кластеры (кластеры первого уровня) должны содержать точные представления, то есть их геометрическая погрешность $\delta(t)$ должна равняться нулю на всём временном отрезке, поэтому объединять в листовые кластеры допустимо только те объекты, функции присутствия которых полностью совпадают (временное

расхождение равно нулю). Ограничение по размеру $\gamma(l)$ обеспечивает временную близость объектов при кластеризации, то есть соседние объекты объединяются в один кластер пока не будет превышено временное расхождение $\gamma(l)$.

Теперь перейдём к описанию алгоритма построения дерева кластеров.

Входные данные. Множество объектов сцены S, у которых известны их ограничивающие объёмы и функции присутствия; целевое число уровней дерева L.

Выходные данные. Дерево кластеров.

Описание алгоритма. Сначала производится вычисление размера всей сцены W и временного расхождение для всей сцены Γ, используя данные объектов из множества S. Для вычисления W строится ограничивающий объём, в который входят ограничивающие объёмы всех объектов из S. Γ вычисляется как $\Gamma = d_T(S)$.

Далее алгоритм работает по шагам. На каждом шаге l от 1 до L алгоритм формирует l-ый уровень дерева. При этом происходит вычисление пороговых значений $w(l)$ и $\gamma(l)$. Для формирования кластеров запускается *процедура кластеризации*, которой передаются пороговые значения $w(l)$ и $\gamma(l)$. Она формирует кластеры, которые должны удовлетворять следующим ограничениям:

1) размер кластера w_c не должен превышать пороговое значение $w_c \leq w(l)$;
2) временное расхождение кластера γ_c не должно превышать пороговое значение $\gamma_c \leq \gamma(l)$.

Псевдокод алгоритма:

вычислить размер сцены W
вычислить временное расхождение сцены $\Gamma = d_T(S)$
инициализировать множество объектов для кластеризации N*:* $N = S$
для каждого уровня l *от* 1 *до* L
    вычислить $w(l)$
    вычислить $\gamma(l)$
    запустить процедуру кластеризации с параметрами N, $w(l)$, $\gamma(l)$ *и получить множество кластеров* C
    для каждого кластера $c \in C$
        если $|c| > 1$ *или* $l = 1$
            для каждого объекта в кластере $o \in c$
                удалить o *из* N
            добавить c *в* N
            добавить c *в результирующее дерево на уровень* l

Алгоритм оперирует множеством объектов N (объекты для следующей кластеризации). В данном контексте термин «объект» может означать как объект сцены, так и кластер. Для того, чтобы сформировать кластеры на каждом уровне, запускается процедура кластеризации, в которую передаётся множество N. Она объединяет объекты из N в новые кластеры (множество C). Далее происходит обработка множества C. Для каждого кластера $c \in C$, если размер кластера больше 1 ($|c| > 1$) или текущий уровень — первый, то все объекты, вошедшие в этот кластер, изымаются из N, а в N добавляется сам кластер c, после чего c добавляется в дерево на уровень l.

Если размер кластера равен 1, то есть если в кластер вошёл только один объект (иными словами, для этого объекта не удалось найти соседей с учётом ограничений $w(l)$ и $\gamma(l)$), этот объект оставляется в множестве N и будет участвовать в кластеризации на следующих шагах. Поскольку $w(l+1) > w(l)$ и $\gamma(l+1) > \gamma(l)$, рано или поздно каждый объект будет объединён в кластер с какими-то другими. А так как $w(L) = W$ и $\gamma(L) = \Gamma$, на последнем уровне все объекты гарантированно объединятся в один кластер, который будет корнем получившегося дерева.

В качестве $w(l)$ можно выбрать значение, равное $w(l) = W/2^{L-l}$. При этом $w(l)$ будет расти нелинейно с ростом уровня, что является предпочтительным, поскольку пространство является трёхмерным, а размер $w(l)$ — линейный. Для $\gamma(l)$ же можно выбрать линейный рост: $\gamma(l) = \frac{l-1}{L-1}\Gamma$.

Теорема. Корректность алгоритма построения дерева кластеров. При выборе пороговых значений $w(l) = W/2^{L-l}$ и $\gamma(l) = \frac{l-1}{L-1}\Gamma$, $l = 1, \dots L$ алгоритм построения дерева кластеров корректно завершается за L шагов.

Доказательство.

$$w(L) = \frac{W}{2^{L-L}} = \frac{W}{2^0} = \frac{W}{1} = W,$$

$$\gamma(L) = \frac{L-1}{L-1}\Gamma = \Gamma,$$

поэтому на последнем шаге L все объекты гарантированно попадут в один кластер и алгоритм завершится.

$$\forall l = 1, \dots L \;\; w(l+1) = \frac{W}{2^{L-l-1}} > \frac{W}{2^{L-1}} = w(l),$$

$$\forall l = 1, \dots L \;\; \gamma(l+1) = \frac{l}{L-1}\Gamma > \frac{l-1}{L-1}\Gamma = \gamma(l),$$

что обеспечивает ослабление ограничений с ростом уровня и увеличение числа объектов, входящих в кластеры.

$$\gamma(1) = \frac{1-1}{L-1}\Gamma = 0,$$

что обеспечивает формирование точных листовых кластеров.

3.2 Кластеризация

Алгоритм построения дерева кластеров вызывает процедуру кластеризации для формирования кластеров на шаге. Задача процедуры кластеризации: используя информацию об ограничивающих объёмах объектов и функциях присутствия объектов, сформировать кластеры из объектов, близких в пространстве и во времени с учётом предельных значений размера w и временного расхождения γ.

Входные данные. Множество объектов O, у которых известны их ограничивающие объёмы и функции присутствия; предельный размер w; предельное временное расхождение γ. Данный алгоритм поддерживает ограничивающие объёмы только в виде прямоугольных параллелепипедов, ориентированных по осям координат (англ. axis-aligned bounding box, AABB).

Выходные данные. Множество кластеров C, каждый кластер содержит объекты из O. Вхождение объекта $o \in O$ в кластер c будем обозначать как $o \in c$. Для кластеров должны выполняться следующие ограничения: $\forall c \in C$, такого что $|c| > 1$ должно выполняться: $w_c \leq w, \gamma_c \leq \gamma$ (кластер должен удовлетворять предельным значениям), $\forall o \in O\ \exists! c \in C: o \in c$ (каждый объект должен войти в какой-либо кластер, притом только в один), $\forall c \in C\ c \neq \emptyset$ (ни один кластер не должен быть пуст). Допускаются кластеры, состоящие только из одного объекта (что интерпретируется как то, что для объекта не удалось найти соседей с учётом ограничений w и γ, и такой кластер может не удовлетворять ограничению w). Также допустима ситуация, когда все кластеры состоят только из одного объекта (ничего не скластеризовалось с учётом ограничений w и γ).

Для быстрого поиска соседей в алгоритме применяется шесть индексов: по x, y, z координатам минимальной точки AABB и по x, y, z координатам максимальной. Также, инициализируется структура данных для отслеживания того, какие объекты уже были кластеризованы (это может быть просто массив булевых переменных), чтобы исключить из поиска соседей объекты, которые уже вошли в какой-либо кластер. Основной цикл алгоритма — это обход всех объектов в детерминированном порядке (например, по возрастанию x координаты минимальной точки ограничивающего объёма) с попыткой сформировать кластер вокруг объекта. Даже если для объекта не получилось найти соседей (получился кластер с одним объектом), он всё равно добавляется в результирующее множество кластеров. В программной реализации множества O и C для эффективности хранятся в виде массивов.

Псевдокод алгоритма:

инициализировать структуру данных для отслеживания того, какие объекты были кластеризованы

инициализировать 6 индексов (объекты, упорядоченные по x, y, z координатам минимальной и максимальной точки их AABB)

для каждого $o \in O$ *в детерминированном порядке*

если o *ещё не кластеризован*

инициализировать кластер c *как пустое множество*

добавить o *в* c

пометить o *как кластеризованный*

если размер c *не превышает предельного значения* w

найти соседей по 3 координатам в диапазоне $[x_{max} - w, x_{min} + w]$, $[y_{max} - w, y_{min} + w]$, $[z_{max} - w, z_{min} + w]$

для каждого соседа n

если добавление n *в* c *не приведёт к превышению кластером* c *предельных значений* w *и* γ

добавить n *в* c

пометить n *как кластеризованный*

добавить c *в результирующее множество кластеров* C

Для каждого объекта o, который ещё не кластеризован, первым делом создаётся кластер, содержащий один этот объект. Если кластер уже не удовлетворяет предельному значению w, он так и будет состоять из одного этого объекта и помещается в результирующее множество кластеров C. В противном случае осуществляется поиск соседей. Если x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max} — координаты минимальной и максимальной точек AABB объекта соответственно, то при поиске соседей ищутся объекты, AABB которых входит в область $[x_{max} - w, x_{min} + w]$, $[y_{max} - w, y_{min} + w]$, $[z_{max} - w, z_{min} + w]$. Эта область определена с запасом. Объекты, лежащие за пределами этой области, уже гарантированно приведут к увеличению размера кластера больше w.

Далее, соседи обрабатываются по очереди. Для каждого соседа производится попытка добавить его в кластер и вычисляются значения w_c и γ_c кластера. Если $w_c \leq w, \gamma_c \leq \gamma$, то сосед добавляется в кластер. В противном случае сосед игнорируется. По мере добавления соседей, значения w_c и γ_c у кластера растут, и вероятность добавить следующего соседа уменьшается. Тем не менее алгоритм обрабатывает всех потенциальных соседей. После того, как все соседи обработаны, кластер добавляется в результирующее множество кластеров C.

Процедура поиска соседей осуществляет поиск соседей отдельно по трём координатам и формирует три множества соседей. Затем осуществляется пересечение множеств, поскольку, например, некоторый сосед может быть очень близко по x координате, но очень далеко по y координате. Псевдокод:

составить множество соседей по x координате

составить множество соседей по y координате

составить множество соседей по z координате

взять пересечение множеств соседей по координатам

Процедура поиска соседей по одной координате устроена следующим образом. Сначала вычисляются минимальное и максимальное значения координаты, в рамках которых следует искать соседей, чтобы не превысить размер w. Например, в случае x координаты это будет отрезок $[x_{max} - w, x_{min} + w]$. Для быстрого поиска используются индексы по координате минимальной и максимальной точки ограничивающего объёма. Проверяются все объекты, которые хотя бы краем попадают в этот отрезок, но в результат добавляются только те объекты, которые полностью попадают в этот отрезок. Поскольку w — это диагональ AABB, а здесь поиск идёт по осям координат, получается, что соседи выбираются с запасом. Псевдокод:

минимум *= координата максимальной точки AABB* ***объекта*** *– w*

максимум *= координата минимальной точки AABB* ***объекта*** *+ w*

соседи *= пустое множество*

следующий объект *= следующий объект после* ***объекта*** *согласно* ***индексу по минимальной точке***

пока координата минимальной точки ограничивающего объёма ***следующего объекта*** *≤* ***максимум***

> *если* ***следующий объект*** *не кластеризован и координата максимальной точки ограничивающего объёма* ***следующего объекта*** *≤* ***максимум***
>
> > *добавить* ***следующий объект*** *в множество* ***соседей***
> >
> > ***следующий объект*** *= следующий объект после* ***следующего объекта*** *согласно* ***индексу по минимальной точке***

предыдущий объект *= предыдущий объект перед* ***объектом*** *согласно* ***индексу по максимальной точке***

пока координата максимальной точки ограничивающего объёма ***предыдущего объекта*** *≥* ***минимум***

> *если* ***предыдущий объект*** *не кластеризован и координата минимальной точки ограничивающего объёма* ***предыдущего объекта*** *≥* ***минимум***
>
> > *добавить* ***предыдущий объект*** *в множество* ***соседей***
> >
> > ***предыдущий объект*** *= предыдущий объект перед* ***предыдущим объектом*** *согласно* ***индексу по максимальной точке***

вернуть множество ***соседей***

Теорема. Сложность алгоритма кластеризации. Пусть n — число входных объектов и k — среднее число соседей, удовлетворяющих пороговому значению w. Тогда пространственно-временная кластеризация объектов на уровне может быть выполнена за $O\big(n(\log n + k)\big)$ операций.

Доказательство. Основной цикл алгоритма — это обработка всех n объектов из множества O, что уже даёт оценку сложности не менее $O(n)$ операций. Хотя по мере прохождения цикла будет расти число кластеризованных объектов (которые игнорируются в цикле), так что цикл будет значительно ускоряться к концу.

При обработке каждого объекта самой вычислительно тяжёлой операцией является поиск соседей. Поиск объекта осуществляется по трём координатам и по каждой координате в двух индексных структурах, итого в шести индексных структурах. В качестве индексной структуры можно использовать упорядоченный массив с поиском за $O(\log n)$, так что поиск в шести массивах имеет сложность $O(6\log n) = O(\log n)$. После того, как объект найден в массиве, осуществляется последовательный обход массива для составления множества соседей по координате. k — среднее число соседей, поэтому сбор соседей осуществляется за $O(k)$. При поиске соседей по координатам просматривается число элементов порядка k (пока координата лежит в пределе w). Множества соседей по трём координатам пересекаются для получения финального множества соседей. Для хранения множеств могут использоваться структуры данных на основе хэширования (такие как *unordered_set* из стандартной библиотеки C++), для которых поиск, вставка и удаление выполняются за $O(1)$. Поэтому можно утверждать, что пересечение множеств соседей по координатам может быть выполнено за $O(k)$, а значит весь поиск соседей — за $O(\log n + k)$ операций.

Обработка k соседей выполняется за $O(k)$. Это вычисление таких параметров, как размер ограничивающего объёма и временное расхождение, при попытке добавить соседа в кластер.

Операции добавления объекта в кластер и пометки объекта как кластеризованного могут быть выполнены за $O(1)$, если соответствующие структуры данных представляют собой массивы.

Таким образом, сложность обработки одного объекта не превышает $O(\log n + k)$ операций, а значит сложность всего алгоритма не превышает $O\big(n(\log n + k)\big)$ операций.

Заметим, что на самом деле величины n и k уменьшаются по мере прохождения основного цикла алгоритма, так как кластеризованные объекты исключаются из анализа. Более того, на практике число k будет являться небольшой конечной величиной, так как это соседи, вошедшие в небольшой фиксированный объём пространства вокруг объекта.

3.3 Генерация кластеров: формирование геометрических представлений кластеров

После того, как дерево кластеров построено, начинается процесс генерации кластеров, который заключается в создании и упрощении геометрических представлений. Поскольку геометрические представления занимают много памяти, они хранятся во внешней памяти и загружаются в оперативную память в процессе генерации по мере необходимости. Дерево кластеров хранится в оперативной памяти, поскольку оно содержит лишь информацию об ограничивающих объёмах кластеров, их идентификаторах и связях родитель-ребёнок, и поэтому не занимает много места.

Генерация кластеров осуществляется по уровням дерева. Сначала обрабатывается первый уровень, при этом происходит генерация геометрических представлений кластеров из представлений оригинальных объектов. Затем обрабатывается второй уровень, при этом генерируются геометрические представления кластеров второго уровня из представлений кластеров первого уровня. Затем обрабатывается третий уровень, при этом генерируются геометрические представления кластеров третьего уровня из представлений кластеров второго и, возможно, первого уровней (рисунок 3.2). И так далее, пока не будет обработан последний уровень, содержащий корневой кластер.

Обработка кластеров происходит по уровням, поскольку это позволяет произвести распараллеливание. А именно, кластеры одного уровня не зависят друг от друга, а только от кластеров нижних уровней, поэтому обработка кластеров одного уровня может осуществляться в нескольких параллельных потоках.

Будем считать, что у каждого кластера имеется свой уникальный идентификатор. Это может быть целочисленное число, GUID или URI. Эти идентификаторы присваиваются кластерам ещё при построении дерева, также каждый кластер хранит список идентификаторов дочерних кластеров. Будем считать, что по этому идентификатору можно однозначным образом найти и получить геометрию из внешней памяти.

Генерация одного кластера заключается в следующем:

- полигональные сетки дочерних кластеров загружаются в оперативную память (используется список идентификаторов дочерних кластеров);
- полигональные сетки объединяются в одну новую сетку;
- освобождается оперативная память, выделенная под полигональные сетки дочерних кластеров;
- новая сетка упрощается до целевой геометрической погрешности ε_c;

- полученная полигональная сетка кластера сохраняется во внешней памяти, так, чтобы её можно было найти по идентификатору кластера;
- освобождается оперативная память, выделенная под полигональную сетку;

Геометрическая погрешность кластеров должна расти с ростом уровня в дереве (поскольку родительские кластеры представляют собой более крупные и более упрощённые представления групп объектов). Так как размер кластеров растёт с ростом уровня, то целевую геометрическая погрешность ε_c для кластера разумно выбирать относительно его фактического размера w_c ($w_c \le w(l)$) по формуле $\varepsilon_c = rw_c$, где r — задаваемая пользователем константа, параметр генерации кластеров, который будем называть *относительной погрешностью*, $0 < r < 1$. Исключение составляет первый уровень, который должен содержать точные представления. Поэтому для кластеров первого уровня упрощение не производится.

Входные данные алгоритма. Дерево кластеров; правило, по которому для идентификатора кластера можно найти путь к его геометрическому представлению во внешней памяти; значение относительной погрешности r. Дерево загружено в оперативную память, а геометрия находится во внешней памяти (в виде файлов на диске, в базе данных или в удалённом сетевом расположении).

Выходные данные. Геометрические представления всех кластеров, сохранённые на диске.

Псевдокод:

для каждого уровня l *от* 1 *до* L
    для каждого кластера c *на уровне* l *(допускается параллельное исполнение)*
        получить список идентификаторов дочерних кластеров кластера c
        для каждого идентификатора id *из списка идентификаторов*
            зачитать полигональную сетку с диска по id
        объединить полигональные сетки в одну
        освободить память, выделенную под дочерние сетки
        если $l > 1$
            вычислить размер кластера w_c *(например, как диагональ его ограничивающего объёма)*
            упростить сетку до целевой геометрической погрешности $\varepsilon_c = rw_c$
        сохранить сетку на диске
        освободить память, выделенную под сетку

Наиболее вычислительно затратной процедурой в данном алгоритме является упрощение полигональной сетки. Алгоритм упрощения сетки будет рассмотрен в следующем разделе.

3.4 Упрощение геометрии

Наиболее вычислительно затратной частью процесса генерации кластеров является упрощение полигональных сеток. Как было упомянуто в обзорной части работы, существует два подхода к упрощению полигональных сеток: упрощение до целевой погрешности и упрощение до целевого числа треугольников. В данном случае используется упрощение до целевой погрешности.

Пусть ε — значение целевой геометрической погрешности. Алгоритм упрощения производит итеративное объединение вершин в радиусе ε в одну вершину. В основном цикле алгоритма осуществляется последовательный обход всех вершин. Для каждой вершины v ищутся соседние вершины в радиусе ε. При поиске соседей учитываются вершинные атрибуты: выбираются только вершины с таким же цветом, идентификатором текстуры и идентификатором объекта. Найденные соседние вершины «объединяются» в одну вершину v: они помечаются как удалённые, а все треугольники, ссылающиеся на них, теперь будут ссылаться на вершину v, после чего вершина v помечается как обработанная. Обработанные и удалённые вершины исключаются из дальнейшего анализа и будут игнорироваться при поиске соседей для остальных вершин.

Важный свойством данного алгоритма упрощения является сохранение вершинных атрибутов. Вершины с разными цветами не объединяются, что позволяет сохранить правильные цвета граней и гарантировать цветовое сходство между исходной и упрощённой моделью. Идентификатор текстуры будет необходим при экспорте HDLOD и генерации текстурных атласов, так как в один кластер объединяются различные объекты с различными текстурами; идентификатор текстуры позволяет «помнить» о том, какая текстура использовалась для вершины, после упрощения сетки. Идентификатор объекта необходимо сохранять для каждой вершины для реализации выделения по клику мыши в 3D окне во время рендеринга. Это позволит отследить, к какому из начальных объектов сцены относится треугольник в упрощённом представлении HDLOD, и таким образом реализовать пользовательскую бизнес-логику.

Вторым достойным упоминания свойством данного алгоритма является то, что множество вершин результирующей упрощённой полигональной сетки является подмножеством вершин исходной. Это может позволить эффективно хранить уровни детализации, в виде единого массива вершин и нескольких массивов индексов, определяющих треугольники каждого уровня детализации.

Для быстрого поиска соседей в радиусе ε данный алгоритм использует регулярную сетку. Размер ячейки регулярной сетки выбирается большим, чем ε. Поэтому, для поиска соседей

достаточно проверить 27 ячеек вокруг вершины (ячейку, в которой находится вершина, и все соседние ячейки). Регулярная сетка может требовать много памяти. Увеличение размера ячейки будет снижать расход памяти, но увеличивать время поиска соседей. Правильный подбор размера ячейки может обеспечить быструю работу алгоритма при разумном расходе памяти.

Входные данные алгоритма. Полигональная сетка, представленная как массивы вершин и треугольников; целевая геометрическая погрешность ε.

Массив вершин (переменная ***вершины*** далее в псевдокоде) представляет собой массив, в котором хранятся данные о вершинах, такие как (x, y, z) координаты вершины, (r, g, b, a) цвет, (u, v) текстурные координаты, а также идентификатор текстуры и идентификатор объекта.

Массив треугольников (переменная ***треугольники*** в псевдокоде) представляет собой массив, в котором хранятся индексы вершин треугольников. Каждый треугольник в массиве треугольников — это три индекса в массив вершин, определяющие треугольник.

Выходные данные. Массивы вершин и треугольников упрощённой полигональной сетки (переменные ***выход вершины*** и ***выход треугольники***).

Псевдокод:

число вершин *= размер массива (****вершины****)*

обработано *= новый массив булевых переменных размера* ***число вершин*** *инициализированный значением «ложь»*

соответствие индексов *= новый массив целочисленных переменных размера* ***число вершин***

для каждого i *от 1 до* ***числа вершин***

    соответствие индексов [i] = i

для каждого ***индекса вершины*** *от 1 до* ***числа вершин***

    если ***обработано*** [***индекс вершины***] == *ложь*

        вершина = ***вершины*** [***индекс вершины***]

        индексы соседних вершин = *собрать соседей* (***вершина***, ε)

        для каждого ***индекса соседней вершины*** *в* ***индексах соседних вершин***

            соседняя вершина = ***вершины*** [***индекс соседней вершины***]

            если ***обработано*** [***индекс соседней вершины***] == *ложь и цвет* ***соседней вершины*** *совпадает с цветом* ***вершины*** *и идентификатор объекта* ***соседней вершины*** *совпадает с идентификатором объекта* ***вершины*** *и идентификатор текстуры* ***соседней вершины*** *совпадает с идентификатором текстуры* ***вершины*** *и расстояние от* ***соседней вершины*** *до* ***вершины*** $\leq \varepsilon$

                обработано [***индекс соседней вершины***] = *истина*

                соответствие индексов [***индекс соседней вершины***] = ***индекс вершины***

        обработано [***индекс вершины***] = *истина*

для каждого ***треугольника*** *из* ***треугольников***

старый индекс 1 = *индекс первой вершины* ***треугольника***

старый индекс 2 = *индекс второй вершины* ***треугольника***

старый индекс 3 = *индекс третьей вершины* ***треугольника***

новый индекс 1 = ***соответствие индексов*** [***старый индекс 1***]

новый индекс 2 = ***соответствие индексов*** [***старый индекс 2***]

новый индекс 3 = ***соответствие индексов*** [***старый индекс 3***]

вершина 1 = ***вершины*** [***новый индекс 1***]

вершина 2 = ***вершины*** [***новый индекс 2***]

вершина 3 = ***вершины*** [***новый индекс 3***]

если треугольник ***вершина 1, вершина 2, вершина 3*** *не вырожденный*

добавить ***вершину 1*** *в* ***выход вершины***

добавить ***вершину 2*** *в* ***выход вершины***

добавить ***вершину 3*** *в* ***выход вершины***

выход индекс 1 = *индекс* ***вершины1*** *в* ***выход вершинах***

выход индекс 2 = *индекс* ***вершины2*** *в* ***выход вершинах***

выход индекс 3 = *индекс* ***вершины3*** *в* ***выход вершинах***

добавить новый треугольник (***выход индекс 1, выход индекс 2, выход индекс 3***) *в* ***выход треугольники***

Для каждой вершины в массиве ***обработано*** отслеживается, была ли она уже обработана. Основной цикл алгоритма — это обход всех вершин. Для каждой вершины сначала выполняется проверка, была ли она уже обработана, и если да, то вершина игнорируется. Для вершины осуществляется поиск соседей в радиусе ε. Пока что будем предполагать, что есть некоторая эффективно реализованная процедура, возвращающая ближайших соседей в радиусе ε. Тогда для каждой *соседней вершины* необходимо проверить ряд условий, при удовлетворении которых *соседняя вершина* может быть объединена с *текущей обрабатываемой вершиной*. Объединение заключается в том, что *соседняя вершина* помечается как обработанная и происходит подмена индекса *соседней вершины* в треугольниках на индекс *текущей вершины* (треугольники, которые раньше ссылались на *соседнюю вершину*, теперь будут ссылаться на *текущую вершину*). Для подмены индексов используется массив ***соответствие индексов***, сами треугольники модифицируются только после завершения цикла обработки всех вершин. После того, как все соседи вершины обработаны, вершина помечается как обработанная (и поэтому в дальнейших итерациях цикла будет игнорироваться).

После завершения цикла обработки вершин начинается цикл обработки треугольников. Для каждого треугольника производится подмена его вершин, используя ***соответствие индексов***. Далее осуществляется проверка вырожденности треугольника в линию или в точку. Вырожденные треугольники игнорируются, невырожденные добавляются в выходной массив

треугольников, а их вершины — в выходной массив вершин. За счёт объединения вершин происходит вырождение части треугольников, что и обеспечивает сокращение числа треугольников и упрощение полигональной сетки. Выходной массив может содержать дублирующиеся вершины, поэтому после упрощения полигональной сетки может понадобиться ещё один обход всех вершин с целью удаления дубликатов и пересчёта индексов.

Для эффективного поиска ближайших соседей необходимо использовать структуру пространственной индексации. Это может быть любая из известных структур, но текущей реализации использовалась регулярная сетка. Её достоинства заключаются в простоте реализации и быстром поиске ячейки для вершины. Пусть g — шаг регулярной сетки, а (x, y, z) — координаты вершины, тогда ячейку регулярной сетки, в которую попадает вершина, можно найти по формуле $\left(i_x, i_y, i_z\right) = \left(\frac{x}{g}, \frac{y}{g}, \frac{z}{g}\right)$. Если $g \geq \varepsilon$, то для обнаружения всех соседей вершины в радиусе ε достаточно рассмотреть $3 \times 3 \times 3 = 27$ ячеек вокруг ячейки вершины, то есть в области $[i_x - 1, i_x + 1], \left[i_y - 1, i_y + 1\right], [i_z - 1, i_z + 1]$. Благодаря этому, поиск соседей может быть осуществлён за $O(1)$. Недостатком регулярной сетки является большое потребление памяти, а точнее, если выбран маленький шаг сетки, то число соседей будет небольшим, что сократит число вычислений, но расход памяти может быть неприемлемо большим. Увеличение шага сетки ведёт к уменьшению расхода памяти, но и к увеличению числа ложных соседей, которых придётся выбраковывать вычислением точного расстояния до целевой вершины. Размер сетки следует выбирать исходя из-за объема доступной памяти, локализации поиска соседей и заполненности ячеек.

Теорема. Сложность алгоритма полигонального упрощения. Пусть n — число вершин полигональной сетки, ε — заданная геометрическая погрешность упрощения, M — ограничение по памяти на предельное число ячеек сетки, K — целевая заполненность ячеек, V — объём сетки. Тогда вычислительная сложность алгоритма составляет $O(n + nK)$ операций, если $n \leq K \min(M, V/\varepsilon^3)$ и $O\left(n + n^2/\min(M, V/\varepsilon^3)\right)$ операций в противном случае.

Доказательство. Развёртывание структуры пространственной индексации (регулярная сетка) выполняется путём обхода всех вершин ($O(n)$) , определения ячейки для вершины по формуле $\left(i_x, i_y, i_z\right) = \left(\frac{x}{g}, \frac{y}{g}, \frac{z}{g}\right)$ за $O(1)$ и добавления индекса вершины в массив индексов вершин ячейки ($O(1)$), что в итоге имеет асимптотическую сложность $O(n)$ операций.

Рассмотрим цикл обработки вершин. Операции проверки, обработана ли вершина, пометка вершины как обработанной, и правка соответствия индексов выполняются за $O(1)$, поскольку они требуют лишь доступ к элементу массива по индексу. Поиск соседей в радиусе ε с использованием регулярной сетки выполняется за $O(1)$, поскольку для этого достаточно

рассмотреть $3 \times 3 \times 3 = 27$ ячеек (шаг регулярной сетки не меньше ε). Пусть в среднем в одной ячейке находится k вершин. Тогда для каждой вершины в среднем находится $27k$ соседей, и их обработка выполняется за $O(27k) = O(k)$ операций, а обработка n вершин — за $O(nk)$ операций.

Обработка треугольников тоже выполняется за $O(n)$ операций. Обработка каждого треугольника выполняется за $O(1)$, поскольку состоит из таких операций как доступ к элементу массива по индексу ($O(1)$), вставка в конец массива ($O(1)$) и проверка треугольника на вырожденность. Вырожденность треугольника определяется анализом трёх вершин, например путём вычисления векторного произведения или косинуса угла треугольника, что не зависит от общего числа вершин или треугольников и выполняется за $O(1)$. Общее число треугольников сопоставимо с числом вершин n, поэтому обработка всех треугольников осуществляется за $O(n)$ операций.

Таким образом получается оценка $O(n + nk + n) = O(n + nk)$ операций. Теперь попробуем оценить число k. Пусть m — число ячеек в сетке. Во-первых, имеет место ограничение по памяти $m \leq M$. Во-вторых, шаг сетки не может быть меньше ε, поэтому $m \leq V/\varepsilon^3$. Таким образом, $m \leq \min(M, V/\varepsilon^3)$. С другой стороны, имеется целевая заполненность ячеек K. Среднее число вершин в ячейке k связано с числом ячеек следующим образом: $k = \frac{n}{m}$.

Рассмотрим случай, когда $k \leq K$, то есть удаётся не превзойти целевую заполненность ячеек. Тогда $\frac{n}{m} \leq K$, то есть $n \leq Km \leq K \min(M, V/\varepsilon^3)$. Таким образом доказана первая часть теоремы: при $n \leq K \min(M, V/\varepsilon^3)$ выполняется соотношение $k \leq K$, и сложность алгоритма вместо $O(n + nk)$ может быть оценена как $O(n + nK)$ операций.

Если ограничение по памяти M или значение ε не позволяют задать достаточно мелкий шаг сетки и $m = \min(M, V/\varepsilon^3)$, а $k > K$, то сложность алгоритма оценивается как $O(n + nk) = O\left(n + n \cdot \frac{n}{m}\right) = O\left(n + n^2/\min(M, V/\varepsilon^3)\right)$ операций. Это завершает доказательство.

Заметим, что по мере прохождения цикла, число обработанных вершин будет расти, и всё больше вершин будет игнорироваться, поэтому k будет уменьшаться в процессе работы алгоритма.

Поскольку при обработке треугольников вершины всегда добавляются в конец выходного массива вершин, это может привести к наличию дублирующихся вершин этом массиве. От дубликатов можно избавится добавлением ассоциативного массива, ключом которого является вершина, а значением — её индекс в выходном массиве вершин. Сравнение вершин необходимо производить не только по координатам, но вообще по всем атрибутам: цвету, текстурным координатам, индексу текстуры и индексу объекта. Перед добавлением вершины в выходной

массив можно осуществить её поиск в ассоциативном массиве. Если она не найдена, значит это новая вершина, её следует добавить как в выходной массив, так и в ассоциативный массив. В противном случае, такая вершина уже встречалась ранее и следует взять её индекс из ассоциативного массива. Поиск и вставка в ассоциативный массив осуществляются за $O(\log n)$. Поэтому удаление дубликатов для n вершин можно произвести за $O(n \log n)$.

Теорема. Корректность алгоритма полигонального упрощения. Алгоритм гарантирует, что геометрическая погрешность полигонального упрощения по метрике Хаусдорфа не превысит заданного параметра ε.

Доказательство. Приведём определение метрики Хаусдорфа. Пусть (M, d) — метрическое пространство, A и B — непустые компактные подмножества пространства M. Метрика Хаусдорфа определяется как:

$$H(A,B) = \max\big(h(A,B), h(B,A)\big), \text{ где } h(A,B) = \sup_{a \in A} \inf_{b \in B} d(a,b)$$

Односторонняя метрика Хаусдорфа $h(A,B)$ из A в B представляет собой супремум из расстояний между каждой точкой $a \in A$ и её ближайшим соседом $b \in B$ (рисунок 3.4).

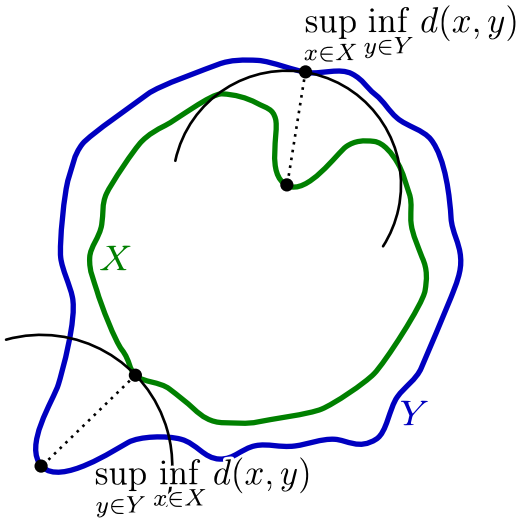


Рисунок 3.4 — Расстояние Хаусдорфа между двумя кривыми X и Y.

Докажем, что расстояние Хаусдорфа между исходной полигональной сеткой и сеткой, упрощённой описанным алгоритмом, не будет превышать ε. В нашем случае пространство — трёхмерное евклидово. Пусть A — множество точек исходной сетки, а B — множество точек упрощённой. Множества A и B — дискретные. Очевидно, что $B \subseteq A$, так как алгоритм лишь удаляет точки из исходной сетки, но не добавляет новых. Оценим $h(A,B)$. Основной цикл алгоритма обходит все точки и для каждой (ещё не обработанной) точки b находит (ещё не обработанных) соседей на расстоянии, не превышающем ε. Пусть a — один из таких соседей точки b. Точка a был в A, но её уже не будет в B. Покажем, что расстояние от a до ближайшей

точки из B будет не больше ε. У a могут быть соседи ближе, чем b, но в худшем случае b оказывается ближайшим соседом a, но $\|b - a\| \leq \varepsilon$. В дальнейшем ходе работы алгоритма точка b уже никуда не сдвинется, так как она помечена как обработанная и не будет участвовать в качестве соседа для других точек. Поэтому $h(A, B) \leq \varepsilon$. Теперь оценим $h(B, A)$. Поскольку $B \subseteq A$, то из этого немедленно следует, что $h(B, A) = 0$. Поэтому геометрическая погрешность упрощения по Хаусдорфу $H(A, B) = \max\big(h(A, B), h(B, A)\big) = \max(h(A, B), 0) = h(A, B) \leq \varepsilon$. Что и требовалось доказать.

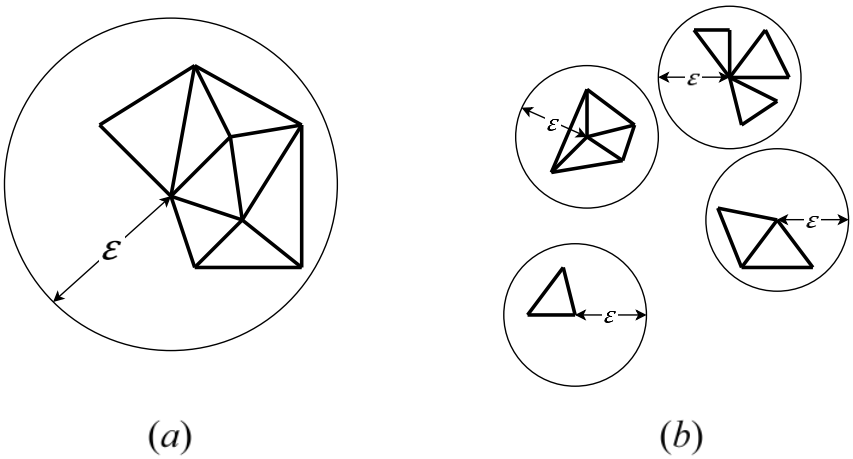


Рисунок 3.5 — Случаи вырождения полигональной сетки при упрощении: (a) размер сетки меньше ε; (b) сетка представляет собой «острова» треугольников размера меньше ε.

Также следует заметить, что множество B не может оказаться пустым. В худшем случае, когда ε превышает размер всей полигональной сетки, все вершины объединятся в одну (рисунок 3.5a). Таким образом корректность алгоритма сохраняется даже в таком предельном случае, но полигональная сетка выродится в одну точку и уже перестанет быть полигональной сеткой. Однако такой случай не представляет практической ценности — как правило ε выбирается сильно меньшим, чем размер сетки. Но возможен другой случай, когда в результате упрощения полигональной сетки все треугольники выродятся (в точки или линии) даже при $|B| > 3$. Это возможно, если полигональная сетка не является связной и представляет собой «острова» треугольников в «пустоте», и при этом размер каждого острова не превышает ε (рисунок 3.5b). Тогда все «острова» треугольников выродятся в точки и треугольников не останется. На практике такое возникает редко, тем не менее в процесс генерации HDLOD можно добавить проверки на пустоту полигональных сеток после упрощения и удалять из дерева HDLOD кластеры, которые выродились. Либо же принудительно оставлять один треугольник из исходной полигональной сетки в случае её полного вырождения.

3.5 Вычисление параметров кластеров

При формировании нового кластера из дочерних кластеров необходимо вычислить его функцию присутствия и функцию геометрической погрешности. Функция присутствия $p_c(t)$ кластера c может быть вычислена путём усреднения функций присутствия дочерних кластеров $c' \prec c$ по следующей формуле:

$$p_c(t) = \frac{\sum_{c' \prec c} w_{c'} p_{c'}(t)}{\sum_{c' \prec c} w_{c'}}.$$

где w — линейный размер кластера (например, длина диагонали ограничивающего объёма AABB).

Использование взвешенных сумм позволяет в большей степени учитывать присутствие более значимых или крупных дочерних кластеров. Таким образом, функция $p_c(t)$ выражает взвешенную долю видимых объектов кластера в момент времени t. Если она принимает единичное значение, это означает, что все объекты кластера присутствуют в сцене в момент времени t, если нулевое — отсутствуют в момент времени t.

Для каждого кластера необходимо принять решение о том, стоит ли отображать его или использовать более точные дочерние представления. Для этого, согласно следующей формуле, вычисляется функция геометрической погрешности $\delta_c(t)$:

$$\delta_c(t) = \begin{cases} \varepsilon_c(2p_c(t) - 1) + \big(1 - p_c(t)\big) \cdot 2w_c \text{ при } \frac{1}{2} \le p_c(t) \le 1 \\ p_c(t) \cdot 2w_c \text{ при } 0 \le p_c(t) < \frac{1}{2} \end{cases}$$

где ε_c — это геометрическая погрешность кластера после упрощения его представления.

Если функция присутствия $p_c(t)$ кластера равняется нулю момент времени t, это означает что все оригинальные объекты сцены, соответствующие этому кластеру, отсутствуют в момент времени t, поэтому геометрическое представление кластера не должно отображаться, и геометрическая погрешность в данном случае равняется нулю. Если $p_c(t)$ принимает единичное значение в момент времени t, то все оригинальные объекты сцены, соответствующие этому кластеру, присутствуют в этот момент времени. Отображение кластера повлечёт погрешность ε_c, связанную с отклонением геометрического представления кластера от совокупного представления соответствующих оригинальных объектов. При $\frac{1}{2} < p_c(t) < 1$ отображение кластера помимо ε_c повлечёт погрешность ещё и связанную с тем, что часть объектов отсутствуют в этот момент времени, однако были отображены в качестве представления кластера. При $0 < p_c(t) < \frac{1}{2}$ кластер не будет отображаться, что повлечёт погрешность, связанную с тем, что часть присутствующих объектов не будут отображены. При формальном расчёте $\delta_c(t)$ должна представлять собой расстояние между геометрическим представлением

кластера и совокупным представлением соответствующих ему оригинальных объектов, например, по метрике Хаусдорфа, в различные моменты времени. Поскольку вычисление точного расстояния имеет высокую вычислительную сложность, предлагается грубо аппроксимировать погрешность размером кластера w_c . Таким образом, при $p_c(t) = \frac{1}{2}$ погрешность $\delta_c(t)$ достигает максимального значения равного размеру кластера w_c. В этом случае кластер будет выбран для отображения только если его размер на экране очень мал.

После того, как функции $\delta_c(t)$ вычислены для всех кластеров дерева HDLOD, функции присутствия $p_c(t)$ могут быть округлены до значений 0 или 1: $p_c(t)$: $[0, T] \to \{0,1\}$. Это необходимо для экономии памяти, поскольку функция присутствия в памяти представляется как ассоциативный массив, ключом которого является момент модельного времени, а значением — значение функции. Округление до значений 0 и 1 и удаление последовательно идущих одинаковых значений позволяют значительно сократить размер массива. Также, это приводит определение функции присутствия кластера $p_c(t)$ в соответствие с определением функции присутствия объекта $p_s(t)$.

3.6 Удаление внутренних граней

Радикальным способом упрощения, не зависящем от выбора алгоритма упрощения, может стать удаление внутренних невидимых граней. При кластеризации происходит объединение геометрии, в результате чего в геометрическом представлении кластера может оказываться большое число внутренних треугольников, невидимых снаружи кластера. Удаление таких треугольников может существенно снизить сложность кластера при сохранении его внешнего вида. Это особенно важно при подготовке HLOD для цифровых индустриальных моделей, содержащих большое число конструктивных и инженерных элементов, перекрывающих друг друга и не видимых снаружи генерируемых кластеров. При этом важно заметить, что упрощённые представления используются только тогда, когда камера располагается достаточно далеко от объекта, то есть кластеры HLOD всегда просматриваются снаружи. Исключение может составлять только случай интерактивного внешнего рендеринга, когда необходимое детальное представление ещё не успело загрузиться из внешней памяти, и поэтому используется упрощённое представление. Данный случай не является столь критичным, если навигация по сцене осуществляется плавно, и геометрия успевает загружаться.

Удаление внутренних граней способно не только уменьшить объем данных, но и существенно повысить скорость генерации и рендеринга HLOD за счет сокращения общего числа

обрабатываемых и растеризуемых полигонов, а также ускорения времени подгрузки данных из внешней памяти.

Недостатком удаления внутренних граней является невозможность его применения в приложениях, где требуется интроспекция внутренностей модели путём изменения прозрачности частей модели или проведением секущих плоскостей.

Удаление внутренних граней может быть осуществлено одним из нескольких способов:

– алгоритм наподобие затенения фонового освещения (англ. ambient occlusion) [67], когда из грани выпускаются лучи, и если хотя бы один луч достиг «неба» (не встретил на пути геометрию), то эту грань можно считать видимой снаружи. Различие с затенением заключается в том, что там вычисляется доля лучей, достигших неба, для расчёта степени освещённости, а в нашем случае для вынесения вердикта о видимости грани достаточно одного луча, попавшего в небо (рис. 3.6а);
– выпускание лучей снаружи к центру геометрии (равномерно по сфере) с целью обнаружения внешних граней (рис. 3.6б);
– рендеринг геометрии с разных ракурсов, когда в качестве цвета пикселя выводится идентификатор грани; идентификаторы, попавшие во фрейм буфер — это обнаруженные внешние грани (рис. 3.6в).

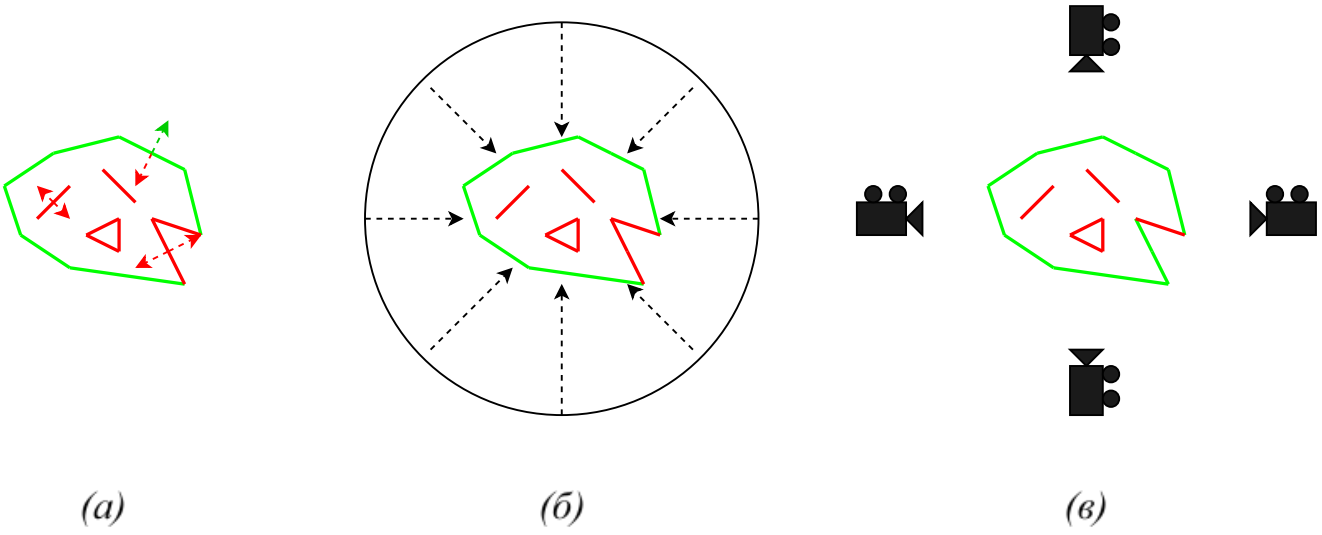


Рисунок 3.6 — Способы реализации удаления внутренних граней.

Препятствием к применению удаления внутренних граней может послужить наличие прозрачных объектов в сцене. В таком случае можно производить операцию поиска внешних граней по слоям (аналогично методу depth peeling [82, 41]): сначала обнаруживаются все внешние грани без учёта прозрачности, затем из числа обнаруженных граней исключаются прозрачные, и процедура запускается вновь. Процесс завершается на той итерации, когда в числе

обнаруженных граней будут только непрозрачные. Грани, которые не были обнаружены ни на одной из итераций считаются внутренними и могут быть удалены. Для ускорения процесса можно ограничить число итераций, тем самым ограничив число обнаруживаемых слоёв прозрачности.

3.7 Текстурные атласы

Кластеры HDLOD могут агрегировать большое число объектов. В случае текстурированных сцен эти объекты будут иметь разные текстуры. С точки зрения рендеринга было бы неэффективно хранить и использовать всё множество текстур для кластера, так как это повлекло бы большие затраты памяти, а также низкую производительность рендеринга из-за постоянных переключений между текстурами в ходе растеризации полигональной сетки кластера. Наиболее целесообразным решением является объединение всех текстур, используемых кластером, в единую текстуру — текстурный атлас (рис. 3.7). Таким образом каждый кластер может представляться одной полигональной сеткой с одной текстурой.

Кластеризация проводится в иерархической манере: кластеры объединяются в другие кластеры. Однако не следует применять тот же принцип для текстурных атласов. Не стоит создавать атласы из атласов, поскольку паковка текстур в атлас осуществляется не оптимально и в атласах всегда остаются пустые места. Наиболее оптимальным решением будет отслеживать, какие текстуры используются кластером, и готовить текстурные атласы для каждого кластера из исходных текстур. Для этого с каждой вершиной сетки, помимо текстурных координат, можно ассоциировать ещё и идентификатор текстуры.

Отдельной проблемой является выбор разрешения текстуры атласа. Как правило, оно выбирается исходя из возможностей оборудования, системных требований программного обеспечения, сложности сцены или как пользовательский параметр. Разумно использовать текстуры одинакового размера для всех кластеров. Более высокоуровневые (и более упрощённые кластеры) агрегируют больше объектов, что при фиксированном размере атласа будет приводить к снижению размера текстур для их паковки в атлас. Благодаря этому снижение разрешения текстур будет осуществляться естественным способом наряду с упрощением геометрии.

Входные данные алгоритма:

- массив вершин полигональной сетки, где каждой вершине приписаны текстурные координаты и идентификатор текстуры;
- массив текстур (картинки могут храниться во внешней памяти);
- размер (разрешение) атласа R.

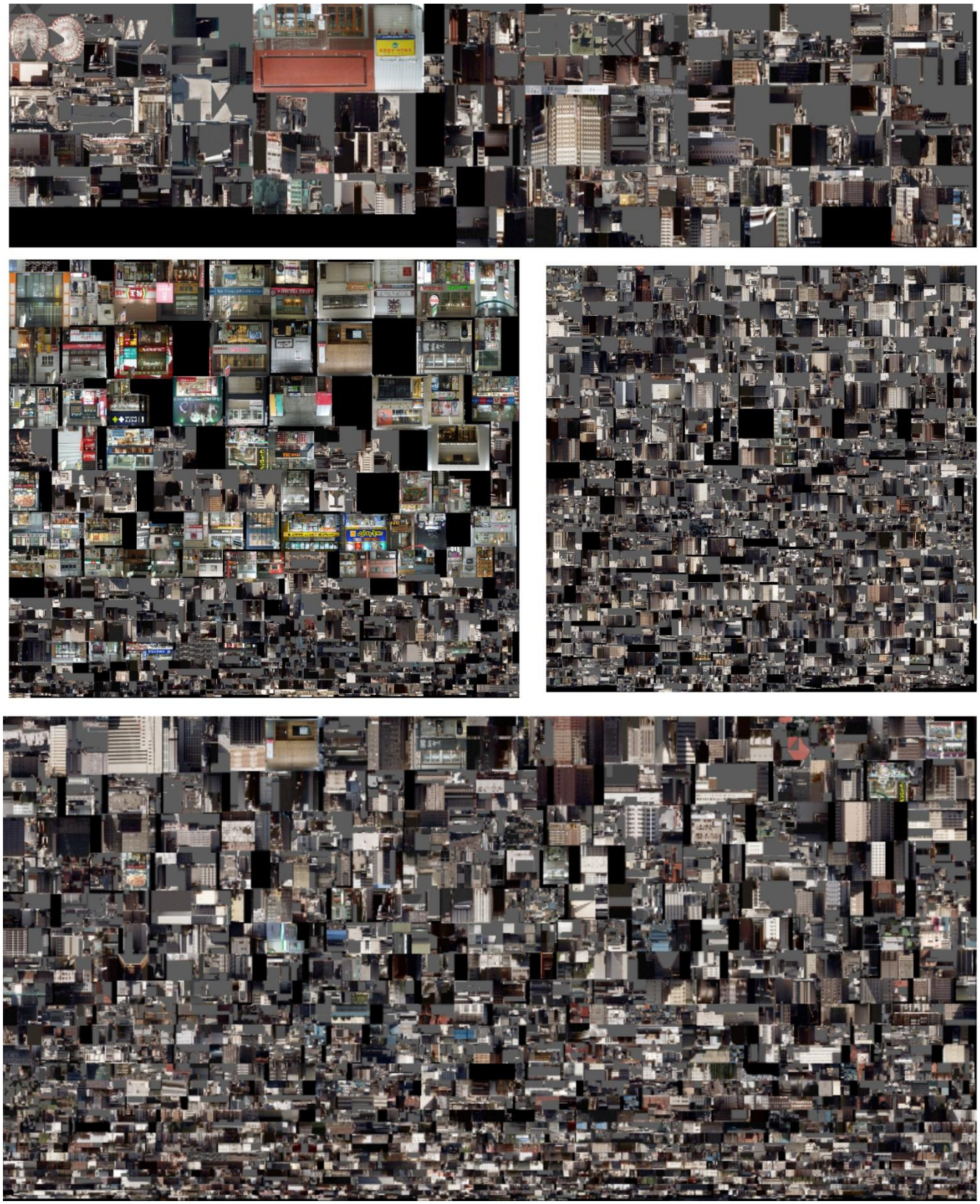


Рисунок 3.7 — Примеры текстурных атласов.

Выходные данные:

– массив вершин, где каждой вершине приписаны текстурные координаты в атласе и идентификатор текстуры атласа;

– текстура атласа размера не более $R \times R$.

Псевдокод:

для каждой текстуры

вычислить количество повторений по направлениям $+S, -S, +T, -T$ *на основе текстурных координат*

вычислить ширину w *и высоту* h *текстуры с учётом повторений*

уменьшить текстуру, так чтобы $\max(w, h) < R$

осуществить попытку паковки текстур в атлас

пока паковка не получается

уменьшить размеры всех текстур в одинаковой степени

повторить попытку паковки

инициализировать картинку атласа размера $R \times R$ *чёрными пикселями*

для каждой текстуры

получить координаты текстуры в атласе (позицию s, t *и размер* w, h*), которые определила процедура паковки*

зачитать картинку текстуры в оперативную память

преобразовать картинку: уменьшить размер и выполнить повторение, получить картинку размера (w, h)

скопировать картинку в атлас в позицию (s, t)

освободить картинку из памяти

вычислить преобразование текстурных координат (зная позицию и размер в атласе и с учётом повторений)

для каждой вершины

получить идентификатор текстуры

по идентификатору текстуры получить преобразование текстурных координат

применить преобразование к текстурным координатам вершины

подменить идентификатор текстуры в вершине на идентификатор текстуры атласа

обрезать неиспользуемые края картинки атласа

привести размер картинки атласа к размерам, поддерживаемым графическим API (например, чтобы ширина и высота делились на 4, либо были степенями 2)

Первая часть алгоритма — это вычисление числа повторений текстур. Для каждой текстуры следует найти максимальное и минимальное значение текстурных координат. Если текстурная координата меньше 0, или больше 1 (то есть указывает за пределы картинки), картинка должна быть повторена несколько раз и пиксель должен сэмплироваться из копии

картинки. У текстур, как правило, есть несколько вариантов параметра заворачивания (англ. wrapping):

- повторение (англ. repeat) — картинка должна быть повторена (скопирована);
- зеркальное повторение (англ. mirrored repeat) — картинка должна быть повторена каждый раз с зеркальным отражением относительно предыдущего раза;
- прижать к краю (англ. clamp to edge) — при повторении берётся крайний пиксель из картинки;
- прижать к границе (англ. clamp to border) — при повторении берётся заданный цвет границы.

По максимальному и минимальному значениям текстурных координат определяется число повторений по направлениям $+S, -S, +T, -T$. Затем вычисляется размер с учётом повторений. Если этот размер превышает размер атласа, он уменьшается. Отметим, что на данном этапе нет необходимости загружать картинку из внешней памяти: вычисления производятся только с размерами картинки.

Вторым этапом алгоритма является непосредственно паковка. Как известно, 2D-паковка является NP-трудной задачей [77]. Существует множество алгоритмов и программных реализаций для её решения. Не будем останавливаться на них, а предположим, что имеется готовая процедура паковки, которая выдаёт булев результат: получилось или не получилось упаковать. В данном алгоритме происходит итеративное уменьшение размеров всех текстур, пока процедура паковки не завершится успехом. Размеры всех текстур следует уменьшать в равной степени, чтобы качество отображения объектов кластеров было одинаковым.

Третий этап — непосредственно построение самой картинки атласа. Исходные картинки загружаются в память по одной. Для каждой из них происходит предварительное уменьшение, а затем повторение (копирование, копирование с отзеркаливанием или заполнение крайними или граничными пикселями), после чего она копируется в картинку атласа. На данном этапе также вычисляются преобразования текстурных координат с учётом положения текстуры в атласе и числа её повторений.

Четвёртый этап — это преобразование вершин: обновление их текстурных координат и индекса текстуры.

Наконец, в самом конце выполняется обрезка неиспользованных краёв итоговой картинки, поскольку в результате паковки может задействоваться не вся площадь картинки атласа. Другой технически важной частью является приведение разрешения картинки к стандартам современных графических API.

Следует упомянуть два особых случая, которые опущены в псевдокоде алгоритма, но требуют корректной обработки.

Во-первых, в кластер могут попадать объекты не только с текстурами, но и без текстур. Объектам без текстур будут соответствовать вершины, в которых идентификатор текстуры имеет неустановленное значение. Если в сетке имеются такие вершины, то в атлас следует добавить небольшой квадратик белого цвета, а таким вершинам приписать текстурные координаты, указывающие в центр этого квадратика. При рендеринге сетки цвет фрагмента следует представлять как произведение цвета текстуры и вершинного цвета. Размер квадратика определяется размером атласа и точностью числа с плавающей точкой, задающего текстурные координаты.

Во-вторых, некоторая текстура может повторяться так много раз, что для паковки в атлас её придётся уменьшить настолько, что она выродится в точку. Для таких текстур имеет смысл проводить некоторое усреднение цвета (или вычисление доминантного цвета) и создавать отдельные небольшие квадратики этого цвета. Текстурные координаты для вершин, использующих такую текстуру, будут указывать в центр соответствующего квадратика.

3.8 Обновление HDLOD

Если в геометрии сцены или в динамическом поведении объектов произошли изменения, необходимо осуществить перегенерацию HDLOD. Процесс генерации HDLOD может быть достаточно длительным, что может стать существенным ограничением применимости HDLOD, в частности в приложениях, в которых требуется поддержка интерактивного редактирования 4D моделей. Однако, если изменения локальные или частичные, есть возможность произвести быстрое частичное обновление HDLOD. В данном разделе будет обсуждаться вопрос обновления HDLOD в случае изменений в сцене.

Отдельно следует рассмотреть случай, когда меняется только информация о динамике, а геометрия сцены остаётся прежней. Для данного случая предлагается использовать многовариантную модификацию метода HDLOD (раздел 2.4), которая заключается в том, что информация о динамике хранится отдельно от геометрии. Это позволяет производить быстрые обновления в случае, когда поменялась только динамика (а также использовать одно дерево кластеров для визуализации сцены с разными динамическими поведениями).

Наибольшую сложность представляет случай, когда происходят изменения геометрии. Далее будут приведены способы реакции на такие изменения и способы частичного обновления HDLOD данных.

3.8.1 **Инкрементальное обновление HDLOD**

Для широкого ряда приложений, например, для приложений просмотра 4D моделей, может быть достаточно однократной генерации HDLOD с последующим многократным использованием одного набора HDLOD данных. В таких приложениях сцена может быть подготовлена один раз (в момент публикации модели) и затем она может многократно просматриваться множеством пользователей на различных клиентских устройствах. Однако, в приложениях для редактирования 4D моделей возникает необходимость быстро обновлять HDLOD представления в случае изменения геометрических представлений или динамического расписания, чтобы не заставлять пользователя проводить время в ожидании увидеть результат внесённых им изменений.

Многовариантный метод HDLOD (раздел 2.4) уже частично решает эту проблему. В нём динамика хранится отдельно от геометрии, поэтому в случае, когда динамика поменялась, не требуется осуществлять долгий процесс перегенерации кластеров.

Однако для случаев, когда изменяется геометрия, необходим способ инкрементального обновления HDLOD, при котором бы пересчитывались только те кластеры, которые потенциально затронуты изменениями.

Напомним, что процесс генерации HDLOD можно разделить на две большие части:

1) кластеризация объектов сцены и построение дерева кластеров;
2) генерация геометрических представлений кластеров (путём объединения и упрощения представлений дочерних кластеров).

Если изменения в сцене настолько серьёзные, что меняется кластеризация (дерево кластеров), то осуществить инкрементальные обновления не представляется возможным и требуется заново проводить кластеризацию и построение дерева кластеров, после чего генерировать представления всех кластеров. Если же изменения в сцене локальные (например, изменились геометрические представления некоторых объектов сцены, но их положения и ограничивающие объёмы остались близки к прежним), то становится возможным не проводить кластеризацию, а использовать прежнее дерево кластеров. В нём необходимо найти те кластеры, в которые попали изменившиеся объекты, и произвести перегенерацию представлений только для этих кластеров.

Таким образом, для реализации инкрементальных обновлений HDLOD требуется имплементировать следующие аспекты:

1) механизм отслеживания изменений в сцене (как найти изменившиеся объекты и понять какого рода изменения произошли);

2) политика оценки степени изменений (слишком значительные изменения — полная перегенерация, локальные изменения — перегенерация представлений только затронутых кластеров);
3) алгоритм поиска изменившихся кластеров по набору изменившихся объектов сцены;
4) реализация перегенерации только изменившихся кластеров с использованием прежних представлений для неизменившихся дочерних кластеров.

3.8.2 Отслеживание изменений, обнаружение и перегенерация «устаревших» кластеров

Метод HDLOD предполагает, что исходная сцена задаётся как плоский список объектов. Для отслеживания изменений необходимо иметь возможность отслеживания изменений каждого объекта. Более того, каждый объект должен быть идентифицируемым, а каждый HDLOD кластер должен хранить идентификаторы вошедших в него объектов. Это позволит обнаруживать изменившиеся объекты и находить изменившиеся кластеры.

Для отслеживания изменений объектов сцены будем предполагать, что каждый объект s снабжён временной меткой t_s. Временная метка, например, может представлять собой 64-битное целочисленное значение, которое задаёт число секунд с момента времени начала эпохи (как правило, с 00:00:00 01.01.1970) до определяемого момента времени. Временная метка хранит дату и время последней модификации объекта.

Для того, чтобы производить инкрементальные обновления HDLOD, необходимо хранить временные метки и в самих HDLOD кластерах. Это нужно, чтобы понять, что кластер «устарел» и требуется его обновление. Напомним, что в дереве HDLOD кластеров листовые кластеры формируются из оригинальных объектов сцены, а внутренние кластеры формируются из дочерних кластеров. На самом деле, временные метки нужны только в листовых кластерах, чтобы путём их сравнения с временными метками объектов, вошедших в кластер, определить, «устарел» ли кластер, или нет. Для внутренних кластеров их признак «устарелости» можно определить следующим образом: если хотя бы один и дочерних кластеров является «устаревшим», то этот кластер тоже является «устаревшим».

При первой генерации HDLOD, все объекты сцены объединяются в некоторое число листовых кластеров. Вхождение объекта s в кластер c обозначается как $s \prec c$. Временная метка кластера может быть определена как максимум (наиболее поздняя) из временных меток вошедших в него объектов: $t_c = \max\limits_{s \prec c} t_s$.

В ходе обновления HDLOD, кластер считается «устаревшим», и требует обновления, если:

1. $\forall s \prec c \;\; t_s > t_c$ (для листовых кластеров)
2. Или $\exists c' \prec c$, такой что c' является «устаревшим» (для внутренних кластеров)

Используя эти правила, признак «устарелости» может быть вычислен для кластеров итеративно по уровням дерева. Сначала он вычисляется для листовых кластеров, затем для их родителей, затем для родителей этих родителей и т.д.

Перегенерация «устаревших» кластеров также осуществляется итеративно по уровням дерева. На первой итерации запускается генерация листовых кластеров. При этом те кластеры, у которых признак «устарелости» не выставлен, игнорируются. На следующей итерации обрабатывается следующий уровень дерева (родители листовых кластеров) и так далее. В целом процесс отличается от обычной генерации кластеров только наличием проверки признака «устарелости» кластера, поэтому в программной реализации первичная генерация кластеров и перегенерация «устаревших» кластеров могут быть реализованы одной процедурой. Если кластеры хранятся на диске, каждый кластер в отдельном файле, то обновление кластеров реализуется естественным образом. «Устаревшие» кластеры генерируются и их файлы перезаписываются, в то время как остальные файлы не затрагиваются, но читаются при генерации родительских кластеров.

3.8.3 **Оценка степени изменений**

В начале процесса обновления HDLOD необходимо оценить степень и характер изменений, произошедших с момента предыдущего обновления (или генерации) HDLOD. Возможны два сценария:

1. изменения *глобальные* и запускается полная перегенерация HDLOD;
2. изменения *локальные* и осуществляется перегенерация лишь изменившихся кластеров.

Изменения можно считать глобальными, если требуется перестроение дерева кластеров, то есть если требуется менять кластеризацию объектов. Разумеется, об этом можно говорить лишь в контексте конкретных ограничений на кластеры HDLOD. При генерации HDLOD могут задаваться следующие ограничения (требования) к дереву HDLOD:

- ограничение на размер кластера w в зависимости от уровня, то есть размер ограничивающего объёма кластера на уровне l не должен превышать $w(l)$;
- ограничение на число детей кластера (любой кластер может иметь не более X детей);
- ограничение на число треугольников в кластере (геометрическое представление кластера может содержать не более X треугольников);

– в случае оригинального метода HDLOD (когда каждый кластер имеет своё описание динамического поведения в виде функции присутствия) накладывается ограничение на степень расхождения функций присутствия (объекты, вошедшие в кластер, должны иметь схожие функции присутствия), а именно: временное расхождение кластера не должно превышать значение $\gamma(l)$.

Рассмотрим возможные реакции на следующие типы изменений:

– добавление новых объектов;
– удаление объектов;
– изменение геометрии существующих объектов;
– изменение динамического поведения существующих объектов.

Добавление объектов. Пусть ранее имелось N объектов, которые вошли в K листовых кластеров. Пусть теперь добавилось M объектов. Реакция на изменения: если эти M объектов можно добавить в K кластеров без нарушения ограничений, такие изменения можно считать локальными. В противном случае изменения считаются глобальными и требуется полная перегенерация HDLOD.

Удаление объектов. Как правило, удаление объектов не способно нарушить ограничений, поскольку оно ослабляет, а не усиливает характеристики кластеров (уменьшается ограничивающий объём, число треугольников и т.д.). Какие-то кластеры могут опустеть и потребуется их удаление, однако это не является жёстким требованием для проведения полной перегенерации (зависит от реализации). Тем не менее, кластеризация может стать менее оптимальной в терминах плотности кластеров (минимизация ограничивающего объёма при максимизации числа детей), числа треугольников в кластере (минимизация общего числа кластеров) и т.д. Число детей кластера может сократиться до одного, что является нежелательным, так как кластер перестаёт быть агрегированным представлением и иерархические уровни детализации вырождаются до обычных уровней детализации. Реакция на изменения: удалить вырожденные кластеры, вычислить новые параметры кластеров (ограничивающий объём, число треугольников, число детей и т.д.) и на основе них вычислить показатели качества HDLOD. Если показатели качества ниже допустимых значений, изменения считаются глобальными, в противном случае локальными.

Изменение геометрии существующих объектов. Если изменения не нарушают ограничения, то их можно считать локальными. В противном случае изменения считаются глобальными. Например, объект передвинулся, что влечёт увеличение ограничивающего объёма кластера и нарушение ограничения на размер кластера. Другой пример: в геометрическом представлении объекта увеличилось число треугольников, что приводит к нарушению

ограничения на число треугольников в кластере. Также, можно учитывать показатели качества. Например, объект передвинулся, что влечёт существенное увеличение ограничивающего объёма кластера, и теперь было бы оптимальней включить этот объект в другой кластер.

Изменение динамического поведения существующих объектов. В случае многовариантного метода HDLOD данный вид изменений не затрагивает кластеризацию и требуется лишь перегенерация вспомогательных структур данных, которые хранят динамику, поэтому такие изменения обрабатываются независимо. В случае оригинального метода HDLOD необходимо выполнить проверку на удовлетворение ограничению по временному расхождению кластеров.

Конечно, также могут произойти изменения одновременно всех типов. Какие-то объекты добавились, какие-то удалились, у каких-то изменилась геометрия или динамика. Описанные выше реакции на изменения можно обобщить следующим образом:

1. необходимо произвести попытку обновить дерево кластеров (пока без пересчёта геометрии);
2. произвести проверку ограничений;
3. вычислить показатели качества дерева HDLOD.
4. если обновлённое дерево кластеров удовлетворяет ограничениям и требованиям к качеству, изменения можно считать локальными, в противном случае изменения считаются глобальными.

При локальных изменениях осуществляется поиск «устаревших» кластеров. После чего запускается процедура, которая генерирует геометрию (и динамику) только для этих кластеров, не затрагивая остальные. При глобальных изменениях осуществляется удаление всех прежних HDLOD данных и запускается новая генерация HDLOD.

Глава 4. Рендеринг HDLOD во внешней памяти

В данной главе покажем, что метод HDLOD может быть применим в условиях ограниченной памяти, в частности, для рендеринга больших сцен, для которых вся геометрия не может быть одномоментно размещена в оперативной или видеопамяти. В отличие от традиционных алгоритмов, работающих полностью в основной памяти (англ. in-core algorithm), внешние алгоритмы (англ. out-of-core algorithm) предполагают, что данные находятся во внешней памяти. В случае задачи рендеринга больших сцен, данные могут находиться на дисковом накопителе в базе данных или в виде файлов, или же загружаться по сети с удалённой машины. Внешний алгоритм рендеринга должен управлять загрузкой/выгрузкой данных, так чтобы в оперативной и видеопамяти находилась только релевантная часть данных.

4.1 Представление HDLOD для рендеринга во внешней памяти

Для рендеринга во внешней памяти дерево кластеров разделяется на так называемый скелет (который находится в оперативной памяти) и содержимое кластеров (которое находится во внешней памяти и загружается в видеопамять память по мере необходимости). Формально дерево кластеров можно представить как объединение множеств узлов скелета и содержимого кластеров $C = C' \cup C''$, где $C' = \{c'(b_c(t), \delta_c(t), p_c(t))\} \cup \{\prec\}$, $C'' = \{c''(g_c, m_c(t))\}$. C' — это скелет дерева, в нём хранятся заголовки кластеров c' и информация о структуре дерева $\prec$. Таким образом, скелет содержит минимально необходимый набор информации для обхода дерева и принятия решения об отображении кластеров. Ограничивающий объём $b_c(t)$ кластера используется для определения видимости кластера, функция геометрической погрешности $\delta_c(t)$ — для выбора кластеров с подходящим уровнем детализации, а функция $p_c(t)$ — для определения присутствия кластера. Следует отметить, что эти функции могут хранится в компактном виде, поэтому скелет не будет занимать много места в основной памяти. «Тяжёлые» атрибуты кластера, такие как его геометрическое представление g_c в виде полигональной сетки и функция положения $m_c(t)$, которая определяет траекторию движения кластера и может состоять из большого количества ключевых точек с заданными преобразованиями системы координат, вынесены в отдельную структуру данных во внешней памяти. Будем предполагать, что каждый заголовок кластера c' ссылается на некоторое содержимое c''. Ссылки могут быть реализованы различными способами: в виде указателя, уникального идентификатора, URL или с помощью какого-либо другого ассоциативного механизма. В общем случае скелет также может

быть разбит на части, которые могут обрабатываться отдельно, а кластеры, имеющие простое представление, могут быть частью скелета. Единственным требованием является то, что скелет должен потреблять небольшое количество памяти по сравнению со всем представлением.

4.2 Консервативный рендеринг HDLOD

Алгоритм консервативного рендеринга HDLOD ставит перед собой цель гарантировать заданный уровень качества полученных изображений. По большому счёту, это базовый алгоритм рендеринга HDLOD, описанный в разделе 2.3. Однако, при внешнем рендеринге необходимо осуществлять загрузку содержимого кластеров в оперативную/видеопамять. Разумеется, если содержимое всех кластеров помещается в оперативной/видеопамяти, подойдёт и наивная реализация, просто загружающая все данные перед началом рендеринга. Тем не менее, в случае больших сцен, данные которых не помещаются одномоментно в оперативной/видеопамяти, необходимо грамотно управлять загрузкой/выгрузкой данных таким образом, чтобы в памяти находилась только действительно необходимая порция данных.

Загрузку содержимого кластеров из внешней памяти в видеопамять будем называть *кэшированием*, а структуру данных, в которой хранятся загруженные данные, будем называть *кэшем*.

Для того, чтобы избежать простоев во время обхода дерева кластеров при рендеринге, алгоритм должен использовать несколько потоков. Так, обход дерева может осуществляться в главном потоке, а загрузка содержимого кластеров должна осуществляться асинхронно во втором потоке. Главный поток осуществляет обход дерева, согласно алгоритму из раздела 2.3 и принимает решение о том, какие кластеры должны быть отображены. Проверки ограничивающих объёмов, погрешностей и присутствия кластеров могут быть произведены немедленно, поскольку они обращаются к атрибутам скелета, который находится в оперативной памяти. Меняется только логика процедуры «отобразить кластер». Теперь обходчик должен выполнить проверку, загружено ли содержимое этого кластера в кэш. Если да, то необходимо осуществить блокировку этого содержимого, чтобы оно не было выгружено в ходе рендеринга, и произвести отрисовку. Если рендеринг осуществляется на графическом процессоре, проще всего оставлять содержимое кластера заблокированным до начала рендеринга следующего кадра, так как, в зависимости от возможностей графического API, не всегда может представляться возможным узнать в какой момент был закончен рендеринг этого содержимого кластера.

Если содержимое кластера не загружено на момент его посещения, необходимо поместить его в *очередь загрузки* (для асинхронной загрузки вторым потоком) и продолжить обход.

Второй поток достаёт элементы из очереди загрузки, производит загрузку, после чего помещает элемент в *очередь результатов*.

Главный поток, в какой-то момент, должен произвести обработку очереди результатов: достать содержимое кластера из очереди результатов, поместить его в кэш, заблокировать его и произвести отрисовку, и так пока очередь результатов не опустеет. Такую обработку можно производить каждый раз, когда содержимое кластера не найдено в кэше (во время обхода). Также её необходимо произвести после обхода, причём необходимо дождаться, пока из очереди результатов не будут получены все элементы, которые были помещены в очередь загрузки. Это можно отследить с помощью целочисленной переменной-счётчика, которая будет инкрементироваться каждый раз при помещении элемента в очередь загрузки, и декрементироваться при извлечении элемента из очереди результатов. В начале обхода эта переменная должна иметь нулевое значение, и в конце обхода необходимо продолжать попытки извлечь элементы из очереди результатов (в ожидании что второй поток их загрузит), пока значение этой переменной не обнулится.

Псевдокод алгоритма консервативного рендеринга HDLOD:

```
функция «обход дерева»
    очистить набор заблокированных кластеров
    инициализировать счётчик запрошенных кластеров значением 0
    посетить корневой кластер
    пока счётчик запрошенных кластеров не равен 0
        обработать очередь результатов

функция «отобразить кластер»
    переместить кластер в начало списка использовавшихся кластеров
    если содержимое кластера находится в кэше
        поместить кластер в набор заблокированных
        произвести отрисовку содержимого кластера
    иначе
        поместить кластер в очередь загрузки
        инкрементировать счётчик запрошенных кластеров
        обработать очередь результатов
```

функция «обработать очередь результатов»

пока очередь результатов не пуста

достать содержимое кластера из очереди результатов

декрементировать счётчик запрошенных кластеров

поместить кластер в набор заблокированных

поместить содержимое кластера в кэш

произвести отрисовку содержимого кластера

функция «поместить в кэш»

если в кэше есть место

добавить содержимое кластера в кэш

иначе

для каждого кластера от конца к началу списка использовавшихся кластеров

если кластер не заблокирован

удалить содержимое кластера из кэша

если в кэше есть место

выйти из цикла

удалить кластеры, которые были удалены из кэша, из списка использовавшихся кластеров

добавить содержимое кластера в кэш

При работе в условиях ограниченной памяти, для кэша устанавливается ограничение по потребляемой видеопамяти. Когда кэш заполняется, необходимо выгрузить некоторые кластеры, чтобы позволить загрузить новые. Для принятия решения о том, какие кластеры выгружать, используется политика замещения наиболее давно использовавшихся (англ. least recently used, LRU). Для этой цели алгоритм поддерживает *список использовавшихся кластеров*. Каждый раз, когда кластер выбирается для отображения, он помещается в начало списка. Таким образом, в конце списка всегда оказываются наиболее давно использовавшиеся кластеры, которые и будут являться первыми кандидатами на выгрузку. Однако, при реализации важно позаботиться о том, чтобы не были выгружены кластеры, которые всё ещё используются для рендеринга. Для этих целей используется *набор заблокированных кластеров*. Когда содержимое кластера запрашивается на загрузку, он помещается в набор заблокированных кластеров. Когда производится попытка выгрузить наиболее давно использовавшиеся кластеры, осуществляется обход списка использовавшихся кластеров от конца к началу, при этом те кластеры, которые не находятся в наборе заблокированных, выгружаются, пока не освободится достаточно видеопамяти для загрузки нового кластера. При этом возможна ситуация, когда все

кэшированные кластеры являются заблокированными, то есть нельзя выгрузить ни один кластер. Тогда загрузка нового кластера всё равно осуществляется с нарушением ограничения по видеопамяти. Таким образом, *ограничение видеопамяти является нестрогим в случае консервативного рендеринга*, и может нарушаться, когда минимально необходимый для обеспечения целевого качества набор кластеров превышает ограничение по видеопамяти. Это происходит потому, что консервативный рендеринг обязан гарантировать целевое качество.

Рассмотрим структуры данных, которые использует алгоритм.

Кэш — ассоциативный массив, в котором ключом является строка URI кластера, а значением — содержимое кластера в видеопамяти (например, идентификатор буфера в видеопамяти). Требуется быстрый поиск, вставка и удаление. На C++ для этого могут использоваться классы стандартной библиотеки *unordered_map* (поиск, вставка и удаление элементов за $O(1)$) или *map* (поиск, вставка и удаление элементов за $O(\log n)$). Также кэш должен отслеживать **текущий размер** данных в видеопамяти и **предельный размер** (задаваемый пользователем).

Набор заблокированных кластеров — набор (множество) строк URI кластеров. Требуется быстрый поиск и вставка. На C++ для этого могут использоваться классы стандартной библиотеки *unordered_set* (поиск и вставка элементов за $O(1)$) или *set* (поиск и вставка элементов за $O(\log n)$).

Список использовавшихся кластеров — это список строк URI кластеров. В начале списка находится URI наиболее недавно использовавшегося кластера, в конце — наиболее давно использовавшегося (англ. least recently used, LRU). Требуется быстрый произвольный доступ, удаление, вставка в начало и обход от конца к началу. Традиционный список поддерживает быстрый обход, вставку и удаление, однако произвольный доступ не поддерживается. Поэтому, необходимо либо в самом кластере хранить указатель на элемент списка, либо поддерживать дополнительный ассоциативный массив, где ключом будет строка URI кластера, а значением — указатель на элемент списка. Таким образом, когда потребуется переместить кластер в начало списка, можно будет быстро найти его в списке, удалить и добавить в начало. На C++ для реализации списка может использоваться класс стандартной библиотеки *list* (вставка и удаление за $O(1)$), дополненный *unordered_map* (поиск, вставка и удаление элементов за $O(1)$) или *map* (поиск, вставка и удаление элементов за $O(\log n)$). Например:

```
list<string> list;
unordered_map<string, typename list<string>::iterator> map;
```

Очередь загрузки — очередь строк URI кластеров. **Очередь результатов** — очередь структур данных с загруженным содержимым кластеров. Необходимо обеспечить безопасность при доступе к очереди нескольких потоков. На C++ для реализации этих очередей может

использоваться класс стандартной библиотеки *deque* (вставка в конец и извлечение из начала за $O(1)$), операции с которым необходимо защитить мьютексом *mutex*. Для того, чтобы второй поток не вращался в горячем цикле, он может спать и просыпаться при помещении элемента в очередь загрузки по нотификации от *condition_variable*.

Теорема. Сложность алгоритма выбора кластеров. Пусть n — общее число кластеров в представлении иерархии уровней детализации HDLOD, k — число кластеров, размещённых в кэше, а m — число кластеров, которое требуется загрузить для отображения текущего кадра. Тогда выбор кластеров для консервативного рендеринга в наихудшем случае выполняется за $O(n \log k + mk \log k)$ операций.

Доказательство. Будем считать, что для быстрого поиска используются структуры данных на основе сортировки, а не хэширования (худший случай).

В худшем случае алгоритм обойдёт всё дерево, то есть посетит все n кластеров. Это уже даёт сложность не менее $O(n)$ операций. При посещении каждого кластера потребуется выполнить следующие операции:

- поиск кластера в списке использовавшихся кластеров: $O(\log k)$, так как число кластеров в списке использовавшихся кластеров совпадает с числом кластеров в кэше k;
- поиск кластера в кэше: $O(\log k)$;
- перед отрисовкой кластера он помещается в набор заблокированных за $O(\log k)$.

Таким образом, сложность обработки всех кластеров дерева: $O(n \log k)$ операций.

Обработка очереди результатов производится после обхода и во время обхода. Очередь результатов содержит не более m элементов. При обработке каждого из них:

- в худшем случае потребуется освободить место в кэше, для чего потребуется обойти список использовавшихся кластеров от конца к началу для выбора кандидатов на выгрузку. В худшем случае, когда задано совсем низкое ограничение по видеопамяти, может возникнуть ситуация, что совершён обход всего списка, то есть k элементов;
- при обходе списка использовавшихся кластеров требуется осуществить поиск каждого кластера в наборе заблокированных $O(\log k)$ и удалить из кэша $O(\log k)$, а также удалить из самого списка $O(\log k)$. Таким образом сложность обработки одного элемента списка использовавшихся кластеров не превысит $O(3 \log k) = O(\log k)$, а значит сложность обработки одного элемента очереди результатов не превысит $O(k \log k)$;

Таким образом, обработка очереди результатов в самом худшем случае осуществляется за $O(mk \log k)$ операций. В итоге, общая оценка сложности выбора кластеров для консервативного рендеринга составляет $O(n \log k + mk \log k)$ операций. Что и требовалось доказать.

Реальное число посещаемых кластеров зависит от положения камеры и точности, предъявляемой к отображению сцены, и будет меньше n. В большинстве случаев оно соотносится с высотой дерева кластеров. Значения k и m существенно зависят от размера кэша, а также от способа навигации по сцене и характера динамики. Причём $k \leq n$ и $m \leq n$.

Для (редкого) предельного случая нехватки памяти, когда ни один из кластеров не может быть выгружен, есть возможность предусмотреть оптимизацию в виде булева флага, который бы говорил о том, что бессмысленно повторно производить попытку освободить место, для предотвращения обхода всего списка использовавшихся кластеров.

4.3 Интерактивный рендеринг HDLOD

Основная идея интерактивного рендеринга заключается в том, чтобы не ждать загрузки кластеров, а использовать те, которые уже загружены. При этом качество изображения будет ниже, но сократится число простоев при рендеринге. Алгоритм интерактивного рендеринга использует те же структуры данных, что и алгоритм консервативного рендеринга: кэш, очередь загрузки, очередь результатов, набор заблокированных кластеров, список использовавшихся кластеров.

Основным отличием является то, что поиск кластера в кэше осуществляется перед его посещением. При посещении кластера он обязан быть в кэше и в списке заблокированных. В начале обхода проверяется, загружен ли корневой кластер. Если нет, то он помещается в очередь на загрузку и производится ожидание его загрузки. Надо заметить, что это будет происходить только один раз — при первом рендеринге HDLOD. После чего корневой кластер уже не будет удаляться из кэша, поскольку всегда будет посещаться и будет находиться в списке заблокированных.

При посещении кластера первым делом производится проверка качества кластера. Если качество является недостаточным, вычисляется набор видимых детей (с использованием проверок попадания в область видимости и, возможно, окклюзии), после чего проверяется, загружены ли видимые дети. Только в случае, если все видимые дети загружены, они посещаются. Если хотя бы один видимый ребёнок не загружен, то обход продолжить невозможно, так как это приведёт к тому, что некоторые видимые объекты сцены не будут отображены. Поэтому отображается текущий кластер, пусть даже он и не удовлетворяет

требованиям по качеству. В отличие от консервативного рендеринга здесь не происходит ожидания загрузки требуемых кластеров, а используются те кластеры, которые уже загружены. Однако видимые дети запрашиваются на загрузку, чтобы можно было отобразить их на следующих кадрах.

По аналогии с алгоритмом консервативного рендеринга запрос кластера на загрузку осуществляется путём помещения его URI в очередь загрузки. Второй поток осуществляет асинхронную загрузку и помещает содержимое кластера в очередь результатов. Главный поток осуществляет обработку очереди результатов в начале и в конце обхода. В начале обхода обрабатываются кластеры, запрошенные на предыдущем кадре, а в конце обхода те, которые были запрошены на текущем кадре и уже успели загрузиться. Обработка очереди результатов заключается в том, что содержимое кластера достаётся из очереди результатов и помещается в кэш (загружается в видеопамять).

Добавление в кэш осуществляется следующим образом: если в кэше есть место, то кластер добавляется в кэш (содержимое кластера загружается в видеопамять, URI кластера сохраняется в ассоциативном массиве). Если места нет, то производится попытка освободить место путём удаления из кэша наиболее давно использовавшихся кластеров. Для этого осуществляется обход списка использовавшихся кластеров, но игнорируются заблокированные кластеры (то есть те, которые посещались на этом или предыдущем кадре). Может так получиться, что освободить достаточно места не удалось. В таком случае, в отличие от консервативного рендеринга, добавления нового кластера в кэш не происходит, ограничение на размер кэша не нарушается.

Псевдокод алгоритма интерактивного рендеринга HDLOD:

```
функция «обход дерева»
    очистить очередь загрузки
    обработать очередь результатов
    очистить набор заблокированных кластеров
    инициализировать счётчик запрошенных кластеров значением 0
    если корневой кластер попадает в область видимости
        если корневой кластер не в кэше
            поместить корневой кластер в очередь загрузки
        пока корневой кластер не в кэше
            обработать очередь результатов
        посетить корневой кластер
        обработать очередь результатов
```

функция «посетить кластер»

поместить кластер в набор заблокированных

если качество является достаточным

если кластер присутствует ($p_c(t_{current}) = 1$*)*

отобразить кластер

иначе

выполнить проверки видимости (попадание в область видимости, окклюзия) для детей и получить множество видимых детей

если все видимые дети загружены

посетить видимых детей

иначе

для каждого видимого ребёнка

если счётчик запрошенных кластеров меньше лимита запрошенных кластеров

поместить ребёнка в очередь загрузки

инкрементировать счётчик запрошенных кластеров

иначе

выйти из цикла

если кластер присутствует ($p_c(t_{current}) = 1$*)*

отобразить кластер

функция «отобразить кластер»

переместить кластер в начало списка использовавшихся кластеров

произвести отрисовку содержимого кластера

функция «обработать очередь результатов»

пока очередь результатов не пуста

достать содержимое кластера из очереди результатов

поместить содержимое кластера в кэш

функция «поместить в кэш»

если в кэше есть место

добавить содержимое кластера в кэш

иначе

для каждого кластера от конца к началу списка использовавшихся кластеров

если кластер не заблокирован

удалить содержимое кластера из кэша

если в кэше есть место

выйти из цикла

удалить кластеры, которые были удалены из кэша, из списка использовавшихся кластеров

если в кэше есть место

добавить содержимое кластера в кэш

Поскольку во время обхода каждый посещаемый кластер помещается в набор заблокированных, это гарантирует, что родители отображённых кластеров не будут удалены из кэша. То есть, *если кластер загружен, то его родитель тоже загружен*. Благодаря этому свойству всегда будет иметься резервный кластер на случай, если в область видимости попали новые (не загруженные) кластеры, и не произойдёт ситуации, когда часть объектов сцены не отображается из-за того, что кластеры не успели загрузиться. Это позволяет данному алгоритму жертвовать точностью в случае, когда кластеры не успели загрузиться, или когда в кэше нет места, но не пропускать объекты сцены.

На практике, загрузка содержимого кластеров в видеопамять может требовать много времени. Для того, чтобы не загружать большие объёмы данных одномоментно и избежать резких падений частоты кадров при рендеринге, имеет смысл ограничить число загружаемых кластеров на кадр. Для этого алгоритм снабжается двумя переменными: *лимит запрошенных кластеров* и *счётчик запрошенных кластеров*. Счётчик запрошенных кластеров инициализируется нулевым значением в начале обхода и инкрементируется каждый раз, когда кластер запрашивается за загрузку (за исключением корневого кластера). Кластеры запрашиваются на загрузку только пока счётчик запрошенных кластеров не превышает лимита запрошенных кластеров. При визуализации с целевой частотой кадров в размере 60 кадров в секунду может быть достаточно установить лимит даже равным единице, то есть загружать не более одного кластера за кадр. Однако, вообще говоря, всё зависит от размера кластеров и общего числа кластеров в дереве, поэтому лимит запрошенных кластеров выносится как параметр алгоритма.

Теорема. Сложность алгоритма выбора кластеров. Пусть n — общее число кластеров в представлении иерархии уровней детализации HDLOD, k — число кластеров, размещённых в кэше, а m — число кластеров, которое требуется загрузить для отображения текущего кадра. Тогда выбор кластеров для интерактивного рендеринга в наихудшем случае выполняется за $O(n \log k + mk \log k)$ операций.

Доказательство. Будем считать, что для быстрого поиска используются структуры данных на основе сортировки, а не хэширования (худший случай).

В худшем случае алгоритм обойдёт всё дерево, то есть посетит все n кластеров. Это уже даёт сложность не менее $O(n)$ операций. При посещении каждого кластера в худшем случае потребуется выполнить все следующие операции:

- поиск кластера в кэше: $O(\log k)$;
- поиск кластера в списке использовавшихся кластеров при отображении: $O(\log k)$. Причём отображаться будут не все кластеры в дереве, а только подмножество листовых кластеров некоторого поддерева (так как не могут быть одновременно отображены предок и потомок), поэтому поиск будет осуществлён не для всех n кластеров;

Таким образом, асимптотическая сложность обхода всего дерева составляет: $O(n \log k)$ операций;

Перед обходом и после него осуществляется обработка очереди результатов. Очередь результатов может содержать не более m кластеров. При обработке каждого из m кластеров в худшем случае нужно будет обойти весь список использовавшихся кластеров $O(k)$ с осуществлением поиска каждого из использовавшихся кластеров в наборе заблокированных $O(\log k)$, что приводит к сложности обработки очереди результатов $O(mk \log k)$ операций.

Итого, асимптотическая сложность всего алгоритма, то есть обхода дерева и обработки очереди результатов составляет $O(n \log k + mk \log k)$ операций. Что и требовалось доказать.

Реальное число посещаемых кластеров зависит от положения камеры и точности, предъявляемой к отображению сцены, и будет меньше n. В большинстве случаев оно соотносится с высотой дерева кластеров. Значения k и m существенно зависят от размера кэша, а также от способа навигации по сцене и характера динамики. Причём $k \leq n$ и $m \leq n$.

Сложность алгоритма выбора кластеров получается такая же, как для консервативного режима. Но большим отличием является то, что в интерактивном режиме значение m можно ограничить максимальным числом догружаемых кластеров на кадр, что позволяет обеспечить сокращение времени рендеринга кадра по сравнению с консервативным режимом.

4.4 Оценка времени рендеринга

В предположении, что обход и анализ дерева кластеров, загрузка и рендеринг моделей кластеров реализуются в параллельных потоках, и время выполнения процессов пропорционально количеству обрабатываемых кластеров, имеет место следующая оценка затрат на интерактивный и консервативный рендеринг.

Оценка времени рендеринга кадра. Пусть n — число кластеров в представлении иерархии уровней детализации HDLOD, которые подлежат обходу и анализу для определения кластеров, отображаемых на текущем кадре. Причем k кластеров, необходимых для отображения, находятся в кэше и m кластеров требуется загрузить. Тогда время рендеринга кадра выражается следующим образом:

$$t_{frame}(n,k,m) = \max(t_{trav}, t_{load}, t_{GPU})$$

$$t_{trav} = nt_{visit} + mt_{cache}$$

$$t_{load} = mt_{read}$$

$$t_{GPU} = \begin{cases} (k+m)t_{rend} + mt_{vload} & if\ conservative\ mode \\ kt_{rend} + mt_{vload} & if\ interactive\ mode \end{cases}$$

где t_{visit} — среднее время анализа кластера при обходе дерева (включая проверки видимости, погрешности, присутствия и наличия в кэше), t_{cache} — среднее время загрузки одного кластера в кэш (включая время записи заголовка кластера и затраты на вытеснение), t_{read} — среднее время чтения модели одного кластера из внешней памяти в основную, t_{vload} — среднее время пересылки модели кластера из основной памяти в видеопамять и t_{rend} — среднее время рендеринга модели одного кластера на графическом процессоре.

Значение n зависит от положения камеры и точности, предъявляемой к отображению сцены. В большинстве случаев оно соотносится с высотой дерева кластеров. Значения k и m существенно зависят от размера кэша, а также от способа навигации по сцене и характера динамики. При интерактивном рендеринге значение m можно ограничить максимальным числом догружаемых кластеров на кадр, что позволяет обеспечить сокращение времени рендеринга кадра по сравнению с консервативным режимом.

Глава 5. Программная реализация

Программная реализация метода HDLOD и описанных алгоритмов выполнена на языке C++, для рендеринга с использованием графического процессора используется API OpenGL. Программное решение состоит из 8 модулей и насчитывает 19 621 строк кода. Список модулей с кратким описанием представлен в таблице 1. Описание используемых сторонних библиотек приведено в таблице 2.

Целевая платформа: Microsoft Windows. Сборка и запуск на других платформах не тестировались, однако решение содержит только кроссплатформенные компоненты, поэтому должно быть портируемым под другие платформы.

Таблица 1 — Модули в программной реализации.

Название модуля	Описание	Сторонние зависимости	Строк кода
HDLODGenerator.dll	Библиотека для генерации HDLOD	stb, nlohmann	6710
HDLODRenderer.dll	Реализация рендеринга HDLOD (консервативного и интерактивного)	stb, nlohmann	3182
HDLODApp.exe	Консольное приложение для генерации и рендеринга HDLOD	Нет	419
HDLODFrom CityGML.dll	Импорт геометрии из CityGML файлов в HDLODGenerator	libcitygml, glm	359
HDLODFromIFC.dll	Импорт геометрии из IFC файлов в HDLODGenerator	glm, IfcOpenShell и его зависимости (boost, OpenCASCADE)	419
HDLODFrom3D.dll	Импорт геометрии из файлов с 3D моделями в HDLODGenerator	ASSIMP, stb	299
OpenGLRenderer.dll	Графический движок для рендеринга полигональных сеток	OpenGL, GLAD, GLFW, glm	5581
ExteriorFinder.dll	Библиотека для определения внешних/внутренних граней полигональной сетки	OpenGL, GLAD, GLFW, glm	2652

Таблица 2 — Сторонние библиотеки, используемые в программной реализации.

Название	Назначение
ASSIMP	Импорт различных 3D форматов в универсальное представление ASSIMP для унифицированной обработки различных 3D моделей
boost	Зависимость IfcOpenShell
GLAD	Получение указателей на OpenGL функции для работы с OpenGL
GLFW	Кроссплатформенная библиотека для создания окон, в том числе для целей 3D рендеринга
glm	Математические операции над векторами, матрицами, кватернионами
IfcOpenShell	Парсинг файлов в формате IFC и импорт данных
libcitygml	Парсинг файлов в формате CityGML и импорт данных
nlohmann	Чтение и запись JSON файлов (используется для представления HDLOD данных в формате 3D Tiles)
OpenCASCADE	Зависимость IfcOpenShell
stb	Библиотека для работы с картинками: чтение и запись JPEG, PNG файлов, изменение размера картинок, паковка картинок в текстурный атлас

Для сохранения HDLOD данных на диске используется файловый формат Cesium 3D Tiles [13]. Использование открытого стандарта позволяет обеспечить совместимость со сторонними средствами просмотра, такими как веб-движки CesiumJS и Three.js. Однако, стандарт 3D Tiles не предусматривает хранение информации о динамике. Тем не менее, он допускает хранение дополнительных метаданных, что используется при сохранении HDLOD для записи информации о динамике.

5.1 Расширение формата Cesium 3D Tiles

Формат 3D Tiles был разработан компанией Cesium для потоковой передачи и рендеринга большого объема трехмерного геопространственного контента, такого как фотограмметрия, 3D-здания, BIM/CAD, экземпляры объектов и облака точек. В данном формате сцена представляется как иерархические уровни детализации: дерево тайлов, также называемое тайлсетом. Каждый тайл имеет своё геометрическое представление (в виде полигональной сетки или облака точек), а также дополнительные данные (геометрическая погрешность, ограничивающий объём и различные метаданные).

Концепция тайла в 3D Tiles совпадает с концепцией кластера в HDLOD. Поэтому HDLOD кластеры могут быть сохранены в виде тайлов. Однако, в отличие от тайлов, у кластеров имеется ещё и динамическое поведение. Например, геометрическая погрешность зависит от времени, а также имеется функция присутствия. Тем не менее динамика также может быть сохранена в формат 3D Tiles как дополнительные данные, о чём и пойдёт речь в данном разделе.

В формате 3D Tiles данные представляются следующим образом: имеется один JSON файл для всего тайлсета, описывающий дерево тайлов (скелет дерева) и множество файлов с геометрией тайлов. Файлы тайлов могут иметь следующие форматы (в версии 3D Tiles 1.0):

- Batched 3D Model (.b3dm) используется для представления группы объектов в виде одной полигональной сетки с возможностью идентификации каждого объекта.
- Instanced 3D Model (.i3dm) хранит набор инстансов одной полигональной сетки с разными трансформациями, позволяя идентифицировать отдельные инстансы.
- Point Clouds (.pnts) хранит облако точек, позволяя идентифицировать группы точек как отдельные объекты.
- Composite Tiles (.cmpt) позволяет хранить разнородную геометрию в одном тайле (например, часть геометрии тайла задаётся как b3dm, а другая часть — как i3dm).

Для сохранения HDLOD в формате 3D Tiles скелет дерева кластеров следует записать в JSON файл тайсета (дерево кластеров — это дерево тайлов), а содержимое каждого кластера — в B3DM файл.

5.1.1 **Пакетный формат Cesium Batched 3D Model**

Геометрия внутри B3DM фактически хранится в формате бинарного glTF в виде единого буфера с набором вершинных атрибутов. Для того, чтобы понимать, к каком объекту относится вершина, для каждой вершины хранится дополнительный целочисленный атрибут *batchId*. В B3DM файле также хранится таблица Batch Table, в которой хранятся метаданные для каждого *batchId*. Это позволяет ассоциировать метаданные с вершинами (и гранями) полигональной сетки и производить стилизацию частей сетки или получать идентификатор объекта для части сетки. К примеру, пусть тайл представляет собой несколько моделей домов, объединённых в одну полигональную сетку. Реализация выделения дома по клику пользователя в 3D окне будет выглядеть следующим образом: по клику в окне обнаруживается треугольник, из его вершины получается *batchId*, по данному *batchId* в Batch Table ищется идентификатор дома. Реализация стилизации дома будет выглядеть следующим образом: по идентификатору дома в Bath Table

ищется соответствующий ему *batchId*, после чего все вершины с данным *batchId* перекрашиваются в нужный цвет.

5.1.2 **Представление оригинального HDLOD в формате 3D Tiles**

Для сохранения HDLOD в формат 3D Tiles предлагается использовать следующие соглашения:

- в JSON файл тайлсета для каждого тайла записываются дополнительные атрибуты в JSON объект *extras*: объект *HDLODPresenceFunction* и объект *HDLODErrorFunction*;
- *HDLODPresenceFunction* описывает функцию присутствия;
- *HDLODErrorFunction* описывает функцию геометрической погрешности;
- напомним, что функции присутствия и геометрической погрешности являются кусочно-постоянными;
- каждая функция описывается как два массива: массив ключей *keys* и массив значений *values*;
- ключи *keys* представляют собой моменты модельного времени, в которые меняются значения функции (это целочисленное значение, задающее число секунд с 00:00:00 01.01.1970);
- значения *values* — это значения функции в соответствующий момент времени и до следующего момента (или до конца времён, если это последнее значение);
- до первого момента времени значение функции следует полагать равным нулю;
- в обязательный атрибут *geometricError* каждого тайла записывается значение геометрической погрешности, полученное в результате упрощения полигональной сетки;
- в корневой JSON объект (соответствующий всему тайлсету) в атрибуты *extensionsRequired* и *extensionsUsed* записываются строки *HDLODPresenceFunction* и *HDLODErrorFunction*

Стандарт 3D Tiles определяет опциональный атрибут *extras* для каждого тайла, позволяющий задавать дополнительные данные. Эта возможность по расширяемости формата используется для сохранения информации о динамике при записи HDLOD данных. Пример определения тайла приведён в листинге 5.1. Альтернативным вариантом реализации может быть запись функций присутствия и погрешности в опциональный атрибут тайла *extensions*, вместо *extras*.

Листинг 5.1 — пример записи тайла в JSON файле тайлсета.

"boundingVolume": {"box": [-148.7778326230914, -48.657615196962674, -0.6617522251357535, 33.450789419596305, 0.0, 0.0, 0.0, 29.999995644150705, 0.0, 0.0, 0.0, 16.6699989236032]},
"children": […],
"content": {"uri": "Cluster58797.b3dm"},
"extras": {
"HDLODErrorFunction": {
"keys": [0, 1291698000, 1305003600, 1305176580, 1307422800, 1309237200, 1310965200, 1313488800, 1317186000, 1318395600, 1320037200, 1320210000, 1321246800, 1323666000, 1324270800, 1324443600, 1328072400, 1329973200, 1331701200, 1331874000, 1334811600, 1339477200, 1371804780],
"values": [0.0, 5.800341963132183, 19.828478108804184, 30.880682224530567, 33.84032962649621, 34.56667988047225, 34.835552278663336, 50.19455523417696, 53.293894935835986, 71.07588313939318, 85.02961903827386, 89.23565950367315, 68.35619002993896, 64.66577095360067, 51.67932526861153, 50.180848397890514, 47.486503107800736, 31.793167180325113, 29.369639902620545, 13.532990165008094, 2.0340973112098055, 1.105736589217111, 0.09585079896497826]
},
"HDLODPresenceFunction": {
"keys": [0, 1320210000],
"values": [0.0, 1.0]
}
},
"geometricError": 0.09585079896497826

5.1.3 **Представление многовариантного HDLOD в формате 3D Tiles**

В многовариантной версии HDLOD динамика хранится отдельно от кластеров. Для обеспечения такого разделения предлагается в Batch Table хранить идентификаторы оригинальных объектов сцены, а информацию о динамике хранить в отдельном JSON файле, с перечислением функций динамического поведения для каждого объекта по его идентификатору. Пример записи Batch Table с идентификаторами объектов приведён в листинге 5.2, идентификаторы перечислены в массиве *globalID*. Пример комплементарного JSON файла с информацией о динамике представлен в листинге 5.3. В файле записан массив объектов *objects*,

для каждого указаны его функции присутствия и цвета. Напомним, что функция геометрической погрешности в многовариантном методе не применяется. Используется фиксированная геометрическая погрешность, записанная для каждого тайла (кластера) в JSON файле тайлсета. Части полигональной сетки стилизуются (скрываются или показываются в соответствии с функцией присутствия и раскрашиваются в соответствии с функцией цвета) в шейдере. Для этого сопоставляется *batchId* вершины, идентификатор объекта и значения его функций в текущий момент модельного времени. В данной реализации стили для каждой полигональной сетки вычисляются на текущий момент модельного времени, передаются на графический процессор в SSBO (Shader Storage Buffer Object), после чего в шейдере *batchId* вершины используется как индекс в этом SSBO для чтения стиля вершины.

Листинг 5.2 — Пример записи Batch Table в B3DM файле.

{"BATCH_LENGTH":11}{"globalID":["1BVmourPDBHeyheIeFbb6n","1OkTh_LnL7dvjdtCBnG6Ml","2jOxh6kEr9HuACy58dQDAy","236m8NbcD2DvJbRCOxdR8y","1NTTyNVcX8zeQGcU2vvOZo","0s2O2hB2TEwA9bPAexyQX$","10cDy1vQv9_fKRsMV$axPW","0FoLLEe6PFsRxvesDxvbV3","3SOUbYQGn7jxGa1aF8q0Sx","3t26gE2ev8uBIz$878Ws9e","1X3GEID4b0F8i5CIvGcXM9"]}

Листинг 5.3 — Пример JSON файла с информацией о динамике.

"objects": [
{
"colorKeys": [1429520400,1429722000],
"colorValues": [24, 255, 24, 255, -1, -1],
"globalID": "1BVmourPDBHeyheIeFbb6n",
"presenceKeys": [0, 1429520400],
"presenceValues": [false, true]
},
...
{
"colorKeys": [1454576400,1455728400],
"colorValues": [24, 255, 24, 255, -1, -1],
"globalID": "1OkTh_LnL7dvjdtCBnG6Ml",
"presenceKeys": [0, 1454576400],
"presenceValues": [false, true]
}

Функция цвета записывается в JSON следующим образом: массив *colorKeys* содержит моменты модельного времени, в которые функция цвета меняет своё значение. Массив *colorValues* содержит значения функции цвета в виде пар цвет-прозрачность. Цвет записывается в виде трёх целочисленных значений RGB в диапазоне [0, 255]. Прозрачность записывается в виде целочисленного значения в диапазоне [0, 255], где 0 означает полную прозрачность (невидимость), а 255 означает полную непрозрачность.

Если в некоторый момент времени отсутствует переопределение цвета (то есть должен использоваться цвет объекта, заданный его материалом), то вместо трёх значений цветовых компонент записывается -1. Аналогичным образом происходит для прозрачности. До первого момента времени считается, что переопределение цвета и прозрачности не заданы, то есть используется цвет и прозрачность материала объекта.

Таким образом, определение функции цвета объекта с идентификатором *1BVmourPDBHeyheIeFbb6n* в листинге 5.3 означает, что до момента *1429520400* объект следует отображать с его исходным цветом и прозрачностью, начиная с момента *1429520400* и до *1429722000* следует использовать зелёный цвет (24, 255, 24), а начиная с момента *1429722000* и до конца времён объект снова должен отображаться в исходном виде (цвет -1 и прозрачность -1).

5.2 Генератор HDLOD

Генерация HDLOD реализована в библиотеке *HDLODGenerator*. Библиотека имеет C-интерфейс, находящийся в одном файле *HDLODGenerator.h* (приложение А.1). В интерфейсе используется язык C, чтобы позволить задействовать библиотеку в других языках программирования. Реализация выполнена на C++. Для представления HDLOD кластера используется структура *Cluster*. У каждого кластера есть уникальный идентификатор *id*, который представляется целочисленным значением. Кластер также хранит идентификаторы своих детей. Полигональная сетка представлена структурой *Mesh*. Кластер не хранит свою полигональную сетку напрямую, она ассоциируется с кластером по его идентификатору, хранится на диске и загружается в оперативную память только когда потребуется. Дерево кластеров (класс *ClusterTree*) хранит кластеры по уровням для удобства параллельной обработки. Основные типы данных приведены в листинге 5.4.

Листинг 5.4 — основные структуры данных генератора HDLOD.

```
typedef std::map<time_t, float> PresenceFunction;
typedef std::map<time_t, double> ErrorFunction;
```

```
struct Bounds {
        glm::dvec3 min;
        glm::dvec3 max;
};
struct Cluster {
        double error = 0.0;
        long id = 0;
        long parentID = 0;
        int triangleCount = 0;
        Bounds bounds;
        PresenceFunction presenceFunction;
        ErrorFunction errorFunction;
        std::vector<long> childrenIDs;
};
typedef std::vector<std::shared_ptr<Cluster>> ClusterArray;

struct Mesh {
        std::vector<glm::dvec3> positions;
        std::vector<glm::vec2> texCoords;
        std::vector<glm::vec3> normals;
        std::vector<glm::u8vec4> colors;
        std::vector<long> textureIDs;
        std::vector<long> objectIDs;
        std::vector<unsigned int> indices;
};
class ClusterTree {
private:
        struct Index {
                int level;
                int index;
        };
        std::vector<ClusterArray> m_levels;
        std::unordered_map<long, Index> m_map;
        glm::dvec3 m_translation = glm::dvec3(0);
};
```

Основным классом реализации является класс *Generator*, в котором сохраняются параметры генерации и принятые на вход данные, также запускаются основные этапы процесса генерации.

Первым этапом является построение дерева кластеров в методе *Generator::generateClusterTree*. Само построение дерева реализовано в классе *TreeBuilder*, который на вход получает список исходных полигональных сеток в виде *ClusterArray*, производит кластеризацию, выстраивая дерево кластеров итеративно по уровням и возвращает *ClusterTree*.

Непосредственно алгоритм кластеризации, описанный в разделе 3.2, реализован в классе *Clusterizer*. Класс *KMeansClusterizer* содержит дополнительную оптимизацию кластеризации методом k-средних: объекты перемещаются между кластерами с целью минимизации ограничивающих объёмов кластеров.

После того, как дерево кластеров построено, начинается генерация кластеров в методе *Generator::generateClusters*. Она происходит по уровням, кластеры одного уровня обрабатываются в параллельных потоках (раздел 3.3). Упрощение полигональной сетки (раздел 3.4) реализовано в классе *VertexMerger*. В классе *FanSimplifier* реализована дополнительная оптимизация: попытка обнаружить и упростить планарные вееры и полу-вееры треугольников. Также при упрощении сетки может использоваться удаление внутренних граней с помощью библиотеки *ExteriorFinder*. По завершении генерации кластеров запускается экспорт в формат 3D Tiles (метод *Generator::exportTiles*) при котором содержимое каждого кластера экспортируется в формат B3DM (класс *B3DMExporter*, использующий класс *glTFExporter* для записи glTF внутри B3DM). Для моделей с текстурами на данном этапе генерируются текстурные атласы (класс *TextureAtlas*). Весь процесс экспорта распараллелен. Финальным этапом экспорта является запись JSON файла тайлсета (метод *Generator::exportTileset* с использованием класса *TilesetExporter*).

На протяжение всего процесса для записи/чтения полигональных сеток и картинок на диск используется шаблонный класс *Serializer*. Сбор статистики осуществляется в классе *StatisticsCollector*. Валидация дерева кластеров реализована в классе *Validator*. Класс *Chronometer* используется для замеров времени. Вспомогательные шаблонные функции реализованы в заголовочном файле *Utils.h*.

5.3 Рендерер HDLOD

Рендеринг HDLOD реализован в библиотеке HDLODRenderer. Библиотека имеет C-интерфейс (приложение А.2) и C++ реализацию. Реализованы: консервативный рендеринг (класс *ConservativeRenderer*), интерактивный рендеринг (класс *InteractiveRenderer*) и рендеринг в основной памяти (класс *InMemoryRenderer*). Зачитывание HDLOD из формата 3D Tiles реализовано в классах: *TilesImporter* (чтение JSON файла тайлсета), *B3DMImporter* (чтение B3DM) и *glTFImporter* (чтение glTF внутри B3DM). Во время рендеринга содержимое кластеров зачитывается из B3DM файлов в структуру *LoadedMesh* в параллельном потоке, после чего полигональная сетка (и текстура) загружаются в видеопамять вызовом *GLRendererCreateMesh* к библиотеке *OpenGLRenderer* в главном потоке. Реализации пролётов по сцене для вычислительных экспериментов находятся в классах *Benchmark* и *BenchmarkPath*.

5.4 Рендеринг с использованием OpenGL

Рендеринг полигональных сеток с использованием API OpenGL версии 4.5 реализован в библиотеке OpenGLRenderer. Библиотека имеет C-интерфейс (приложение А.3) и C++ реализацию. Реализация представляет собой стандартную реализацию рендеринга на OpenGL, без каких-либо продвинутых оптимизаций. Для каждой полигональной сетки используется отдельный буфер, что позволяет легко управлять загрузкой/отгрузкой в видеопамять. Для отображения каждой полигональной сетки используется отдельный вызов отрисовки. Тем не менее, даже для такой простой реализации метод HDLOD демонстрирует хорошую производительность рендеринга. Производительность может быть ещё улучшена путём низкоуровневых оптимизаций под конкретное оборудование, однако это выходит за рамки данной работы.

5.5 Обнаружение внешних граней полигональной сетки

Реализация поиска внешних граней полигональной сетки находится в библиотеке *ExteriorFinder*. Библиотека имеет C-интерфейс (приложение А.4) и C++ реализацию. Для поиска внешних граней используется рендеринг полигональной сетки на OpenGL с разных ракурсов. Каждой грани присваивается уникальный цвет-идентификатор, после чего осуществляется

рендеринг полигональной сетки с 26 ракурсов. По завершении рендеринга с каждого ракурса происходит чтение данных из фреймбуфера и накопление попавших в кадр цветов-идентификаторов в множестве обнаруженных цветов-идентификаторов (используется *std::unordered_set*). В итоге те грани, цвет-идентификатор которых попал во множество, считаются внешними. Остальные — внутренними. Библиотека *ExteriorFinder* реализована таким образом, что её функции могут вызываться одновременно из нескольких параллельных потоков (потокобезопасна). Она используется при упрощении геометрии в библиотеке *HDLODGenerator*.

5.6 Импорт данных

Для импорта данных из различных файловых форматов реализованы три библиотеки: *HDLODFrom3D* (импорт трёхмерной геометрии в различных файловых форматах), *HDLODFromCityGML* (импорт моделей городов в формате CityGML [30]) и *HDLODFromIFC* (импорт моделей зданий и сооружений в формате IFC [14]). Библиотеки имеют C-интерфейс (приложение А.5) и C++ реализацию.

Для импорта 3D используется сторонняя библиотека ASSIMP. Данная библиотека зачитывает различные файловые форматы в своё универсальное C++ представление, после чего происходит обход этого представления и передача полигональных сеток (с текстурами, если они имеются) в библиотеку *HDLODGenerator*.

Для импорта CityGML используется сторонняя библиотека libcitygml, которая производит парсинг GML файлов (с использованием библиотеки xerces) и заполнение данными своего C++ представления модели в виде иерархии экземпляров класса *citygml::CityObject*. Осуществляется обход данной иерархии, полигоны одного *citygml::CityObject* объединяются в одну полигональную сетку (или в несколько, если у них разные текстуры, но так чтобы все полигоны с одинаковой текстурой объединились в одну сетку). Каждая полигональная сетка передаётся в *HDLODGenerator*, для неё также передаётся идентификатор соответствующего городского объекта, если он определён в файле.

Для импорта IFC используется сторонняя библиотека IfcOpenShell. Библиотека парсит IFC файл и осуществляет тесселяцию геометрии с использованием библиотеки OpenCASCADE. После чего последовательно обрабатываются все экземпляры сущности *IfcProduct* (продукты), и их геометрия передаётся в *HDLODGenerator*. На импорте из IFC не обрабатываются текстуры (они достаточно редко встречаются в таких моделях), но обрабатывается динамика. Модели IFC часто помимо геометрии содержат ещё и расписание (план строительства здания). В библиотеке *HDLODFromIFC* обрабатываются сущности *IfcTask* (задача в календарно-сетевом

планировании), ассоциированные с *IfcProduct*. Даты задач передаются в *HDLODGenerator* для определения динамического поведения объектов.

5.7 Консольное приложение

Консольное приложение *HDLODApp* предоставляет пользовательский интерфейс для генерации HDLOD, используя входные данные в виде файлов различного формата, а также для рендеринга HDLOD в формате 3D Tiles и проведения вычислительных экспериментов.

Глава 6. Вычислительные эксперименты

6.1 Масштабируемость в зависимости от размера на экране

Для того, чтобы оценить масштабируемость метода HDLOD была проведена серия вычислительных экспериментов, в которой модель визуализировалась с разных расстояний. В первом положении камера была размещена на максимально близком расстоянии, при котором сцена была видна полностью. В других положениях камера размещалась на таких расстояниях, что сцена занимала одну четвёртую и одну шестнадцатую области экрана. В данных экспериментах исключается фактор отсечения конусом видимости, модель всегда полностью попадает в область видимости, поэтому ускорение рендеринга достигается только за счёт упрощения геометрии. Измерялись значения времени рендеринга кадра при фиксированных моментах времени моделирования, а также среднее время кадра для анимации на протяжении всего периода моделирования. Были выбраны положения времени в начале и в конце моделируемого периода, а также в одной третьей и в двух третях модельного периода. В качестве тестовых были выбраны динамические сцены, визуализирующие процесс строительства небоскрёба, стадиона и бизнес-центра. Модель небоскрёба (рис. 6.1) состоит из 79 396 строительных элементов, представленных полигональными сетками, суммарно содержащими 3 632 126 треугольников. План строительства включает 67 099 активностей, каждая из которых отвечает за определённые работы на строительной площадке и порождает соответствующие события в динамической сцене. Модель стадиона (рис. 6.2) содержит 52 630 полигональных сеток и 9 799 257 треугольников, а план строительства представляется 48 902 активностями. Модели небоскрёба и стадиона содержат статические объекты (окружение) и псевдо-динамические объекты. Модель бизнес-центра (представленная на рис. 2.1 и 2.3 в главе 2) содержит статические, псевдо-динамические и динамические объекты; статистика числа объектов и треугольников представлена в таблице 5.

Вычислительные эксперименты проводились на компьютере типичной конфигурации: Intel Core i7-4790 CPU (3.6 GHz), 16 GB RAM, GeForce GTX 750 Ti (2 GB).

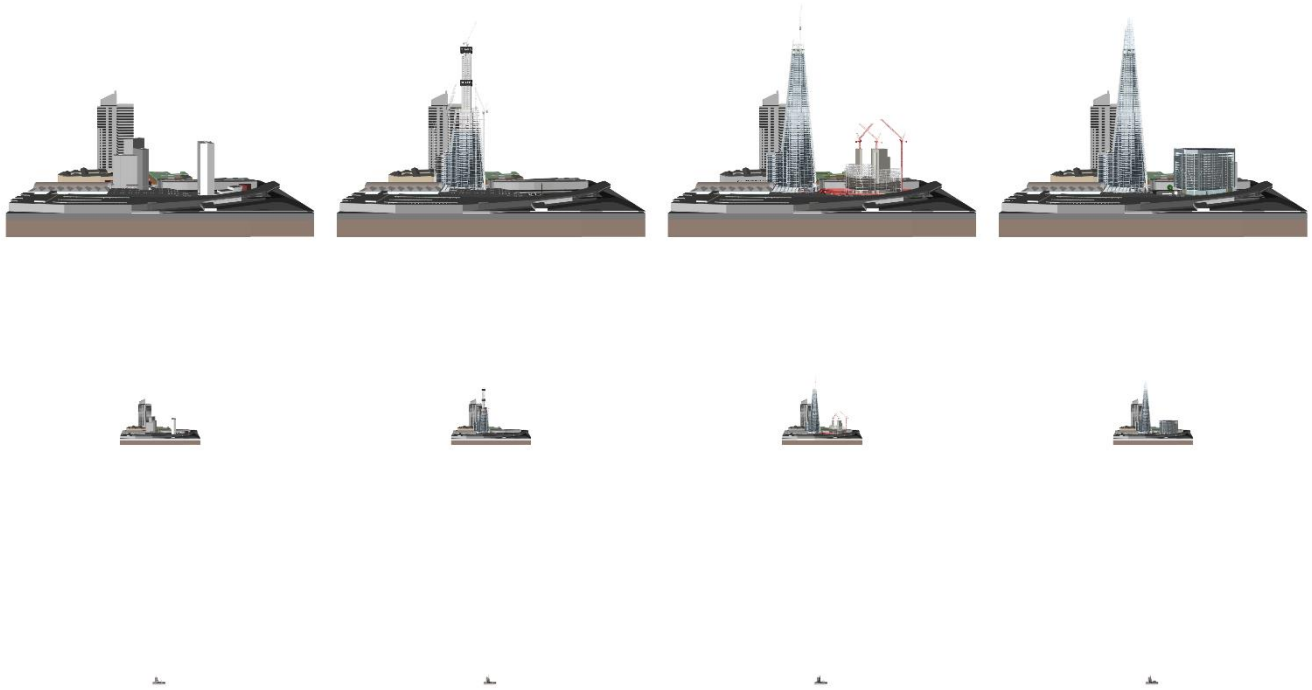


Рисунок 6.1 — Изображения, полученные в ходе экспериментов для сцены строительства небоскрёба. Слева-направо меняется модельное время (начало, 1/3, 2/3 и конец периода), сверху-вниз меняется расстояние до сцены (1/1, 1/4 и 1/16 экрана).

Таблица 3 — Время кадра (в миллисекундах) при визуализации сцены строительства небоскрёба (меньше — лучше).

	1/1 экрана	1/4 экрана	1/16 экрана	Без HDLOD
Начало	7,02	1,5	0,66	40,88
1/3 периода	8,7	2,75	0,71	82,9
2/3 периода	26,47	2,93	0,74	121,3
Конец	28,17	3,15	0,76	164,83
Анимация	18,63	2,87	0,72	105,4

Таблица 4 — Время кадра (в миллисекундах) при визуализации сцены строительства стадиона (меньше — лучше).

	1/1 экрана	1/4 экрана	1/16 экрана	Без HDLOD
Начало	0,37	0,32	0,31	26,81
1/3 периода	7,17	3,31	1,38	31,36
2/3 периода	21,9	7,35	4,4	124,8
Конец	1,73	0,79	0,6	142,4
Анимация	20,01	12,91	6,63	86,35

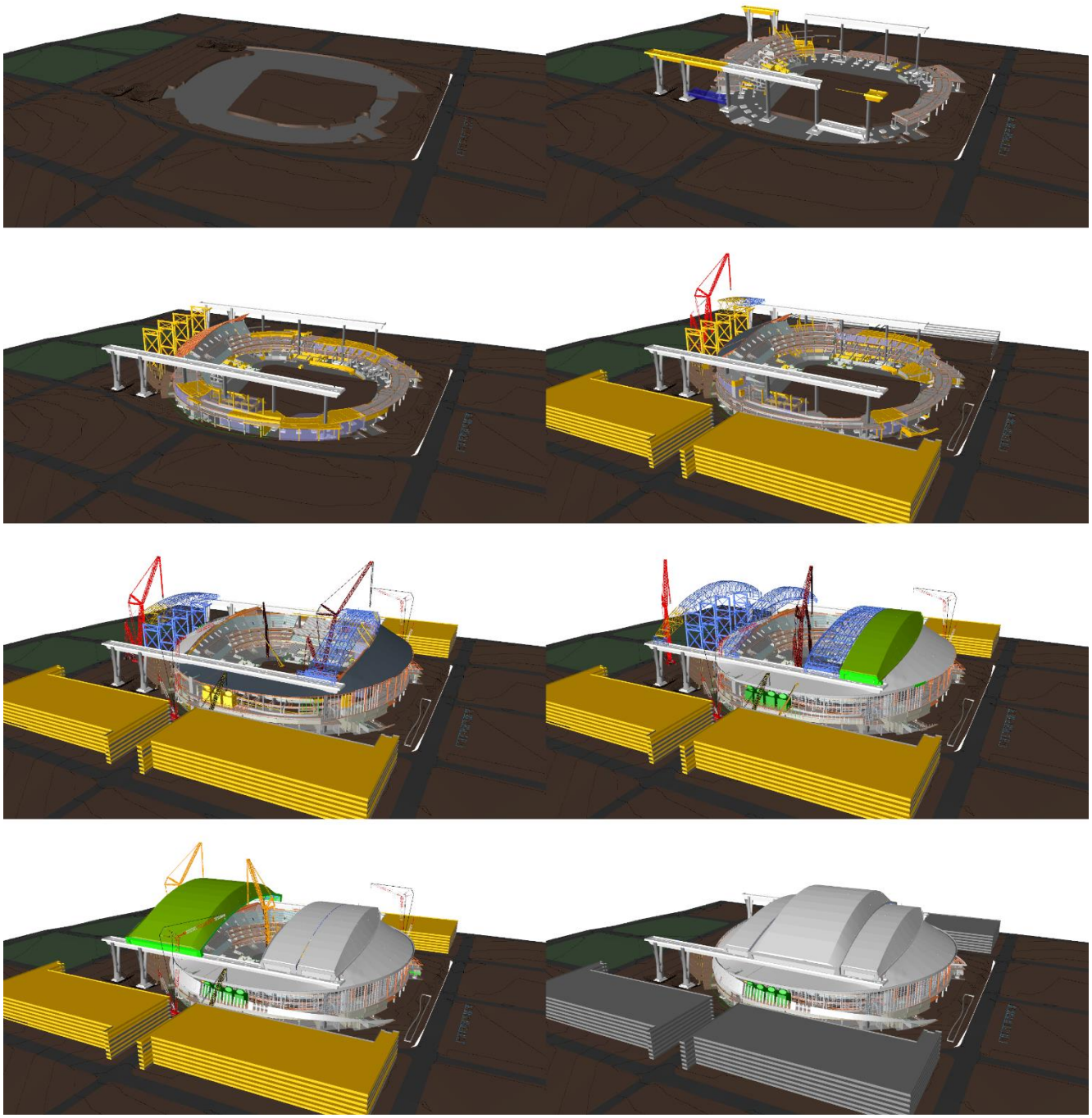


Рисунок 6.2 — Сцена строительства стадиона в разные моменты модельного времени.

Таблица 5 — Количество объектов и треугольников в модели бизнес-центра.

	Статические	Псевдо-динамические	Динамические	Всего
Объекты	498	40 010	15 649	56 157
Треугольники	66 784	2 879 506	1 313 767	4 260 057

Таблица 6 — Время кадра (в миллисекундах) при визуализации сцены строительства бизнес-центра (меньше — лучше).

	1/1 экрана	1/4 экрана	1/16 экрана	Без HDLOD
Начало	3,81	3,58	3,29	16,9
1/3 периода	17,47	17,31	15,59	35,2
2/3 периода	8,5	7,87	7,53	54,22
Конец	3,72	3,58	3,25	61,56
Анимация	9,59	8,51	7,94	47,43

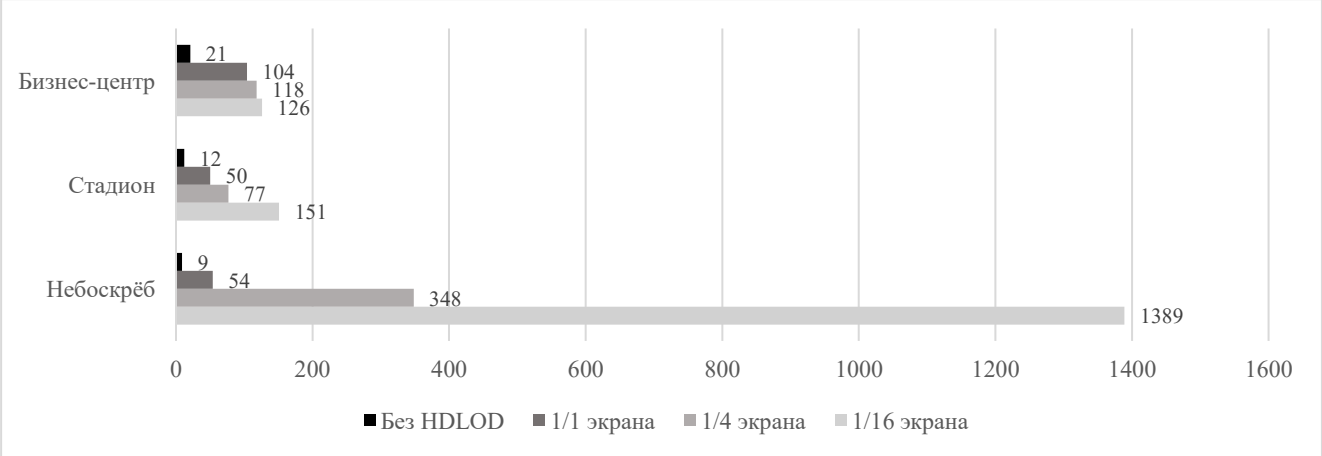


Рисунок 6.3 — Частота кадров (больше — лучше) при рендеринге моделей с разного расстояния, с использованием и без использования HDLOD метода.

В таблицах 3, 4 и 6 приведены результаты измерений производительности в ходе описанных экспериментов. В последней колонке содержатся результаты, полученные при рендеринге индивидуальных объектов без использования иерархии упрощённых представлений HDLOD. Очевидно, что применение HDLOD значительно повышает производительность, и по мере удаления камеры от сцены достигаемый эффект растёт. Подобное поведение наблюдается как при навигации по статической сцене, зафиксированной в выбранные моменты времени, так и при анимации сцены. Зависимость производительности рендеринга от размера сцены на экране является супер-линейной, что позволяет говорить о высокой масштабируемости метода HDLOD по отношению к сложности статических и динамических сцен (рис. 6.3).

Также можно заметить, что время кадра растёт от начала к концу временного периода. Это связано с тем, что в начале временного периода сцена содержит мало геометрии, а с течением модельного времени геометрия появляется в соответствии с планом строительства. На сцене стадиона наибольшее время рендеринга кадра наблюдается на 2/3 временного периода, а не в конце строительства. Такая же ситуация наблюдается и на сцене бизнес-центра на 1/3 временного

периода. Это связано с наличием строительного оборудования, которое исчезает к концу строительства, а также с функцией геометрической погрешности кластера $\delta(t)$. $\delta(t)$ принимает высокие значения в те моменты времени, когда часть объектов, вошедших в кластер, присутствует, а часть отсутствует, что вынуждает во время обхода дерева проваливаться в потомков кластера. Это приводит к тому, что отображаются более детальные кластеры. Тем не менее, производительность рендеринга с использованием HDLOD остаётся значительно более высокой, чем при рендеринге без уровней детализации.

Главным достоинством HDLOD является сохранение высокой производительности при анимации. Во время анимации значение модельного времени меняется с каждым кадром, в сцене происходят динамические изменения, но HDLOD, в отличие от HLOD, не требуют пересчётов, и позволяют сохранить высокую производительность рендеринга даже во время анимации. По сравнению с HLOD, которые потребовали бы больших вычислительных затрат на обновление, даже при небольших локальных изменениях в сцене, HDLOD подготавливаются заранее и не требуют никаких обновлений при анимации сцены на всём временном периоде.

6.2 Консервативный рендеринг

Была проведена серия экспериментов для алгоритма консервативного рендеринга HDLOD. Тестовая сцена была сгенерирована с использованием метода, предложенного в работе [50]. Она имитирует масштабный строительный проект и содержит 9 зданий, состоящих из кирпичей. Каждое здание насчитывает 30 кирпичей в длину, 30 в ширину и состоит из 30 этажей. В итоге сцена состоит из 89910 полигональных сеток и насчитывает 1078920 треугольников. Эксперименты были организованы следующим образом: камера пролетала через сцену по заданной траектории в течение 500 кадров, при этом записывалось время рендеринга кадра. В начале траектории камера находится далеко от сцены. В течение первых 100 кадров она приближается к сцене и оказывается на таком расстоянии, что вся сцена всё ещё полностью попадает в пирамиду видимости. В течение следующих 100 кадров камера поворачивается вокруг центра сцены вправо на 90 градусов. Следующие 100 кадров камера приближается к центральному зданию, пролетая через другое здание, часть сцены при этом отсекается пирамидой видимости. Далее, камера вращается вокруг центрального здания на 90 градусов вправо, при этом некоторые здания появляются в области видимости, а некоторые исчезают из неё. Наконец, в течение последних 100 кадров камера быстро удаляется от сцены, пролетая через одно из зданий.

Эксперименты были произведены для статического и динамического сценариев. Динамическое поведение было сгенерировано для объектов сцены, имитируя процесс

одновременного возведения зданий. Деревья HDLOD были сгенерированы для статической и динамической сцены и записаны на жёсткий диск. На рис. 6.4 представлены снимки экрана, полученные в течение эксперимента на динамической сцене, кластеры на них визуализированы разными цветами. Кластеры выбирались для рендеринга на основе экранной погрешности SSE, вычисленной из их геометрической погрешности $\delta_c(t)$. Для обеспечения высокой точности визуализации пороговым значением экранной погрешности было выбрано значение в один пиксель. Таким образом, разница между визуализацией оригинальной модели и визуализаций HDLOD была едва заметной.

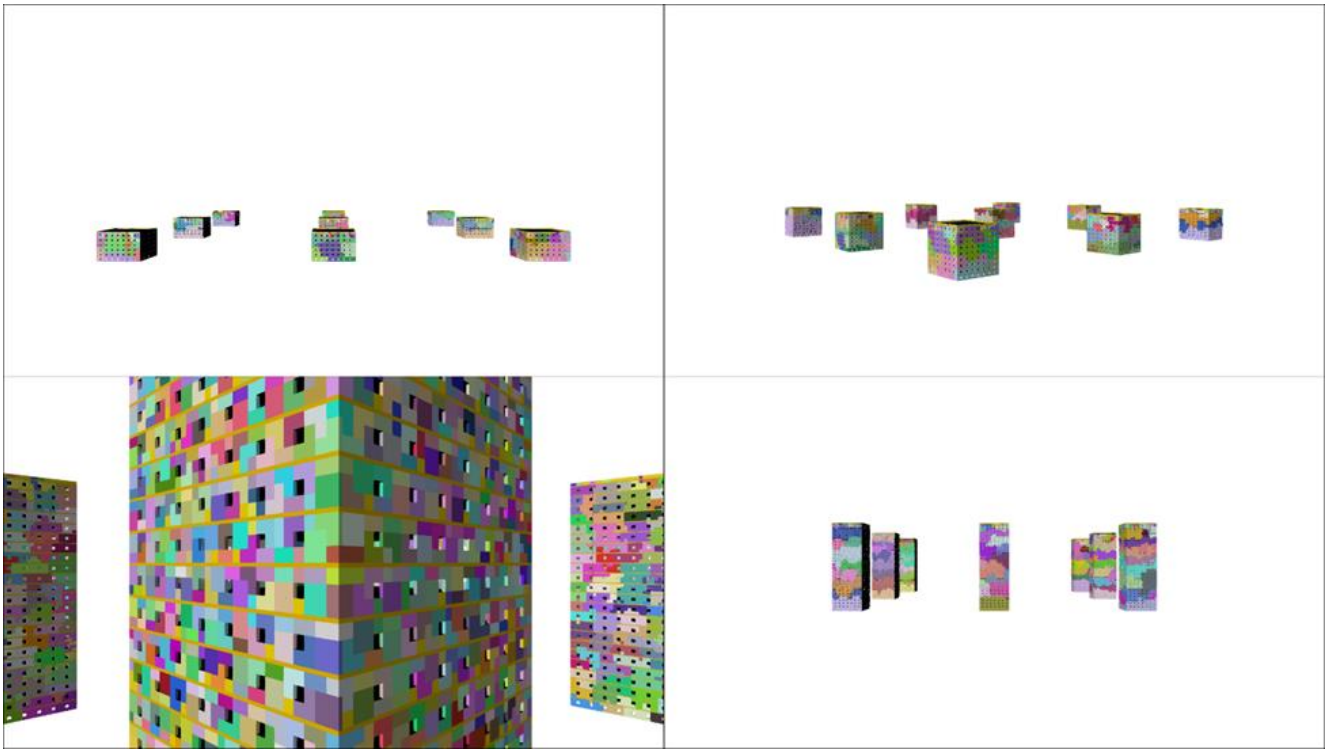


Рисунок 6.4 — Снимки экрана, полученные в ходе эксперимента с динамической сценой. Здания растут по мере того, как камера пролетает по сцене. Кластеры визуализированы разными цветами.

Измерения производились на типичной для массового рынка конфигурации оборудования Intel Core i7-4790 CPU (3.6 GHz), 16 GB RAM, GeForce GTX 750 Ti (2 GB), Seagate ST2000DM001 HDD (2 TB, 7200 RPM). Следует отметить, что производительность устройства хранения данных определяет скорость загрузки содержимого кластеров. Для апробации предложенного метода был выбран наихудший сценарий, когда содержимое кластеров находится на HDD. В последнее время всё большее распространение получают SSD, которые будут показывать значительно лучше результаты.

На рис. 6.5 и 6.6 представлены графики времени кадра, полученные в результате экспериментов. Рис. 6.5 позволяет сравнить производительность при рендеринге без HDLOD (когда отображались только листовые кластеры, соответствующие оригинальным объектам сцены), рендеринге HDLOD в основной памяти (когда все кластеры были предварительно загружены в основную память и оставались там в ходе всего эксперимента) и внешнем рендеринге HDLOD с бесконечным размером кэша (содержимое кластеров загружалось в основную память по ходу эксперимента и оставалось там до конца эксперимента). В случае «без HDLOD» времена кадров стабильные, за исключением части эксперимента, где работает отсечение по пирамиде видимости (кадры 200-400). Рендеринг с HDLOD в основной памяти показывает значительно лучшие результаты, поскольку для отображения выбираются упрощённые представления. Например, в начале и в конце эксперимента камера находится на большом удалении от сцены, поэтому в данных случаях выбираются наиболее крупные и (соответственно) наиболее упрощённые кластеры, что приводит к самым низким значениям времени кадра. Результаты внешнего рендеринга HDLOD сравнимы с результатами рендеринга HDLOD в основной памяти за исключением пиков на графиках. Эти пики вызваны загрузкой большого количества кластеров, что особенно заметно для кадров 250-280 и 400-410, так как в это время камера пролетает через здания, что приводит к необходимости загрузки большого числа листовых кластеров.

В динамических экспериментах происходит не только движение камеры, но и последовательное изменение модельного времени. Эксперимент начинается с пустой сценой. Далее, по мере изменения модельного времени происходит рост зданий и, соответственно, количества отображаемых полигональных сеток и треугольников, что обуславливает рост времени кадра в эксперименте без HDLOD. На графике для внешнего рендеринга HDLOD можно наблюдать большие пики по сравнению со статическим случаем. Они связаны с динамикой и изменением значений функций погрешности $\delta_c(t)$ с течением времени, а именно, с необходимостью загрузки большого числа дочерних кластеров в те моменты времени, когда с кластером происходят изменения и его погрешность оказывается велика.

В эксперименте с бесконечным кэшем кластеры никогда не удаляются из памяти. Серия экспериментов с различными размерами кэша позволяет получить представление о том, как размер кэша может влиять на производительность (рис. 6.6). Для простоты, размер кэша определён в терминах количества кластеров. В целом, результаты очень похожи на случай с бесконечным кэшем; существенная разница наблюдается только при очень маленьких размерах кэша, когда кэш не может вместить все кластеры, требуемые для рендеринга одного кадра. В таком случае кластеры удаляются из кэша практически сразу же после их рендеринга, для того чтобы обеспечить место для загрузки новых кластеров. На следующем кадре первые кластеры

снова загружаются, удаляя кластеры, загруженные в конце предыдущего кадра. Это приводит к перманентной загрузке. Например, для статического сценария с размером кэша 1000 это явление можно наблюдать для кадров 250-280 (пролёт камеры через здание), 290-400 (вращение вокруг центрального здания, расположенного близко к камере) и кадров около 410 кадра (пролёт через одно из зданий по мере быстрого удаления камеры от сцены). При размере кэша 100, количество кластеров, отображаемых на каждом кадре, практически всегда превышает размер кэша, что приводит с существенному снижению производительности, за исключением начала и конца эксперимента, когда камера находится далеко от сцены и отображается небольшое количество упрощённых кластеров.

В динамическом сценарии есть периоды модельного времени, когда в сцене не происходит изменений. Они соответствуют периодам с низкими значениями времени кадра для размеров кэша 100 и 1000. Это происходит потому, что в эти моменты модельного времени более упрощённые кластеры выбираются для отображения (их функция погрешности $\delta_c(t)$ принимает меньшие значения для этих моментов модельного времени), таким образом снижая общее количество отображаемых кластеров. Результаты измерений с размерами кэша более 2000 практически неотличимы от случая с бесконечным кэшем, поэтому не были включены в приведённые графики. Это означает, что политика замещения наиболее давно использовавшихся кластеров (LRU) работает успешно.

При очень маленьких размерах кэша производительность внешнего рендеринга HDLOD оказывается хуже производительности без HDLOD, однако необходимо помнить, что это экстремальный сценарий. На практике размер кэша следует выбирать настолько большим, насколько это возможно, чтобы максимально задействовать имеющийся объём основной памяти. Более того, в эксперименте «без HDLOD» в основной памяти находится содержимое 89910 листовых кластеров, что намного больше 100 или 1000. Наконец, эксперименты при экстремально малых размерах кэша показывают робастность метода HDLOD, способного осуществить визуализацию сцен, значительно превышающих объём доступной основной памяти, без ошибок, программных исключений или падения приложения.

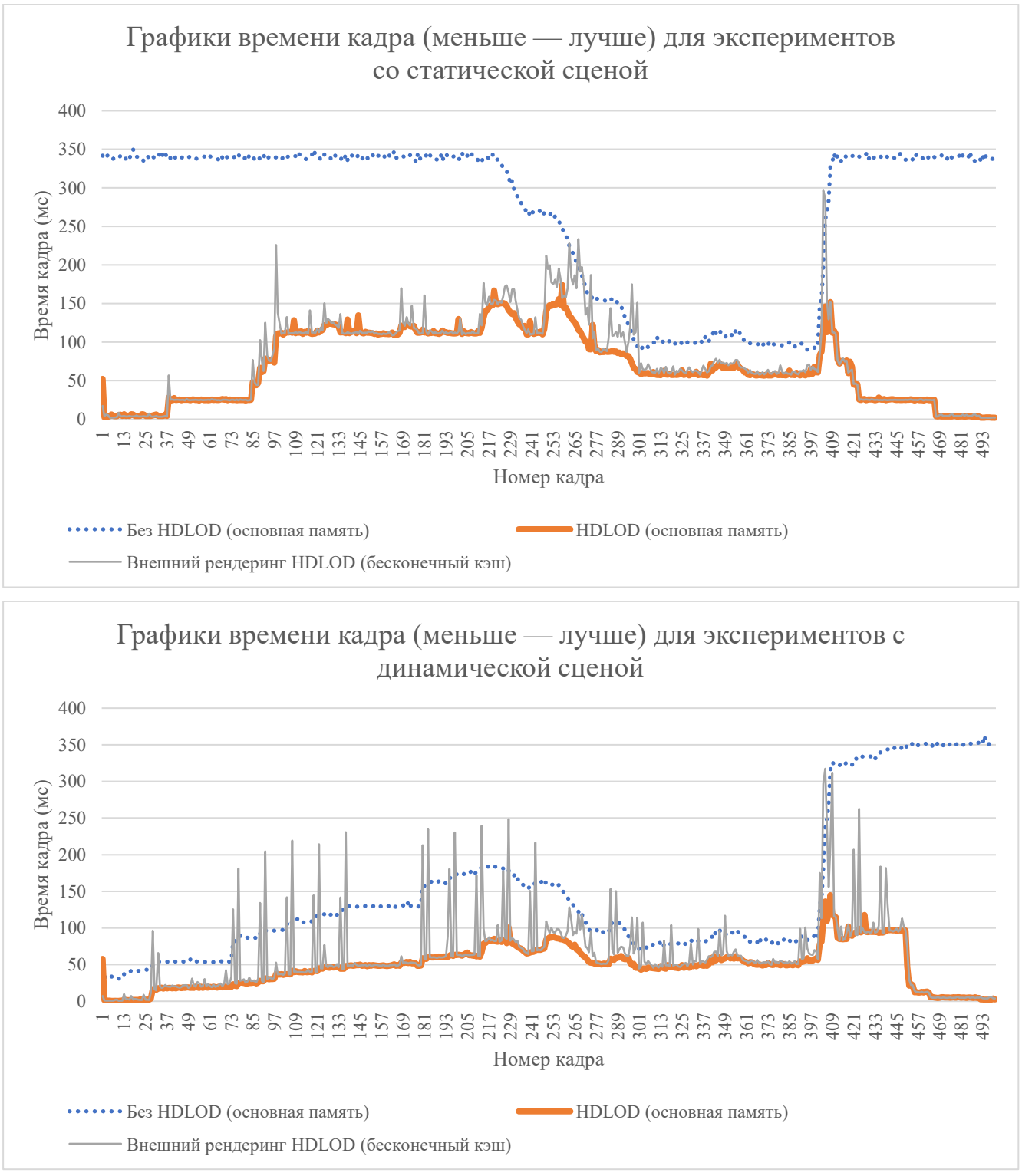
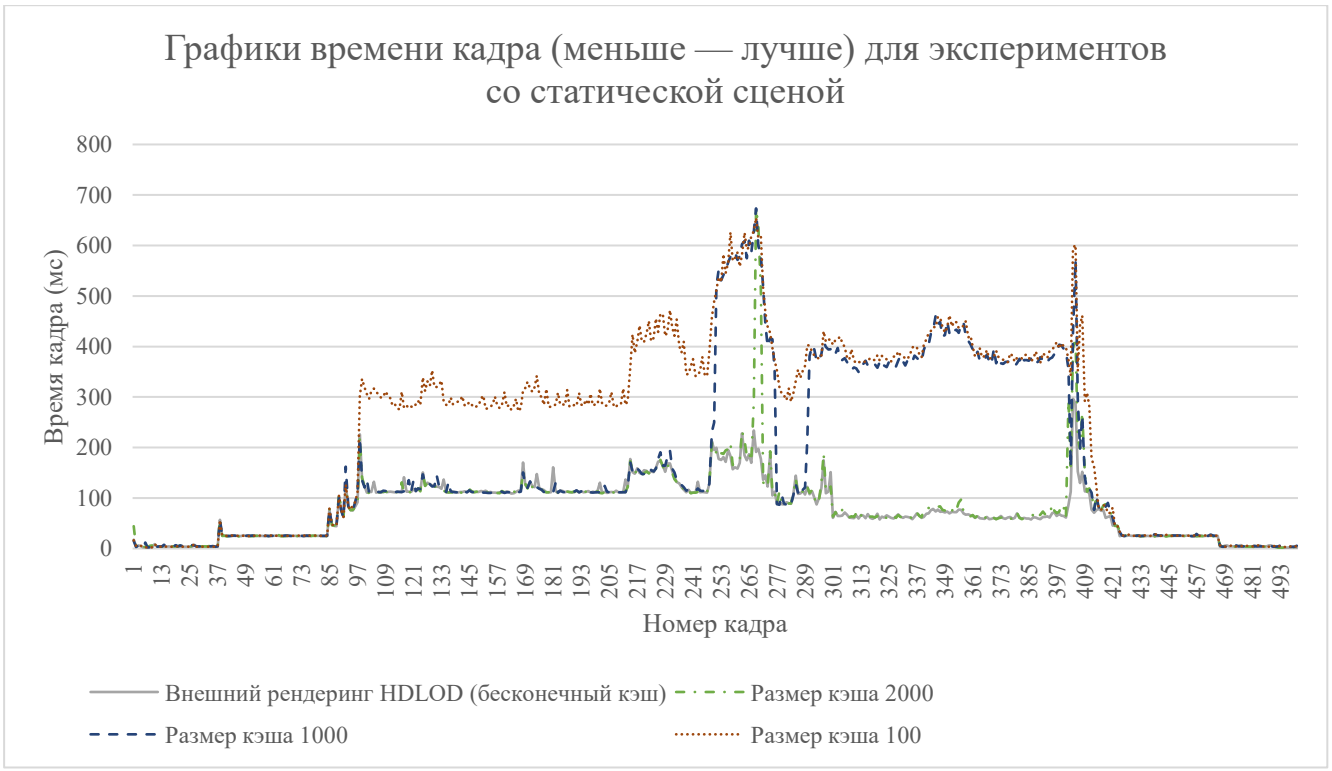


Рисунок 6.5 — Сравнение производительности рендеринга в основной памяти без HDLOD, с HDLOD и рендеринга HDLOD во внешней памяти с бесконечным размером кэша. Наихудший результат показывает рендеринг без HDLOD. Производительность внешнего рендеринга HDLOD с бесконечным размером кэша сравнима с производительностью HDLOD в основной памяти за исключением скачков времени кадра, обусловленных загрузкой новых кластеров.



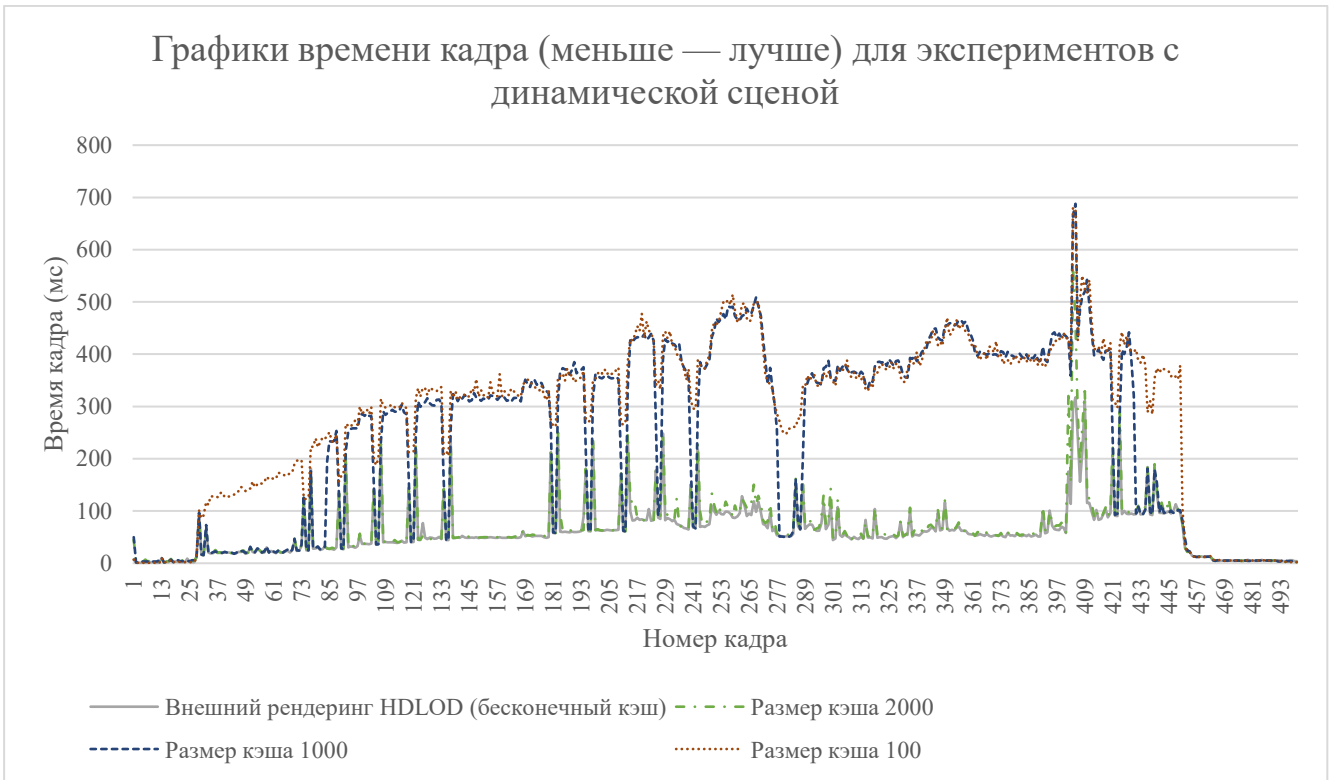


Рисунок 6.6 — Производительность внешнего рендеринга HDLOD при малых размерах кэша по сравнению с бесконечным размером кэша. Повышение времени кадра соответствует случаям, когда количество кластеров, выбранных для отображения, превышает размер кэша, что приводит к перманентной перезагрузке кластеров.

6.3 Многовариантный метод HDLOD

Для того, чтобы сравнить производительность многовариантного метода HDLOD с оригинальным, была проведена серия вычислительных экспериментов. Эксперимент проводился таким образом, что камера двигалась сквозь сцену по предопределённому пути на протяжении 500 кадров и замерялось время рендеринга каждого кадра. Сначала камера находится далеко от сцены, затем она подлетает к центру сцены, поворачивается и улетает вдаль с противоположной стороны сцены (рис. 6.7).

В качестве тестовых сцен были выбраны реальные индустриальные 4D модели строительства небоскрёба и стадиона. Сцена небоскрёба состояла из 79378 полигональных сеток с суммарным числом треугольников 3668091. Сцена стадиона насчитывала 53805 полигональных сеток и 7252950 треугольников. Тест производительности производился для двух сценариев: статика и анимация (рис. 6.8). Статический сценарий означает, что модельное время зафиксировано на дате окончания проектного плана, тем самым визуализируя полностью достроенное строение, в то время как камера облетает сцену. Сценарий с анимацией означает, что модельное время меняется от старта проекта к финишу, в то время как камера облетает сцену, визуализируя процесс строительства. HDLOD предварительно подготовлены и загружены в оперативную и видеопамять перед началом эксперимента, динамика не меняется во время эксперимента.

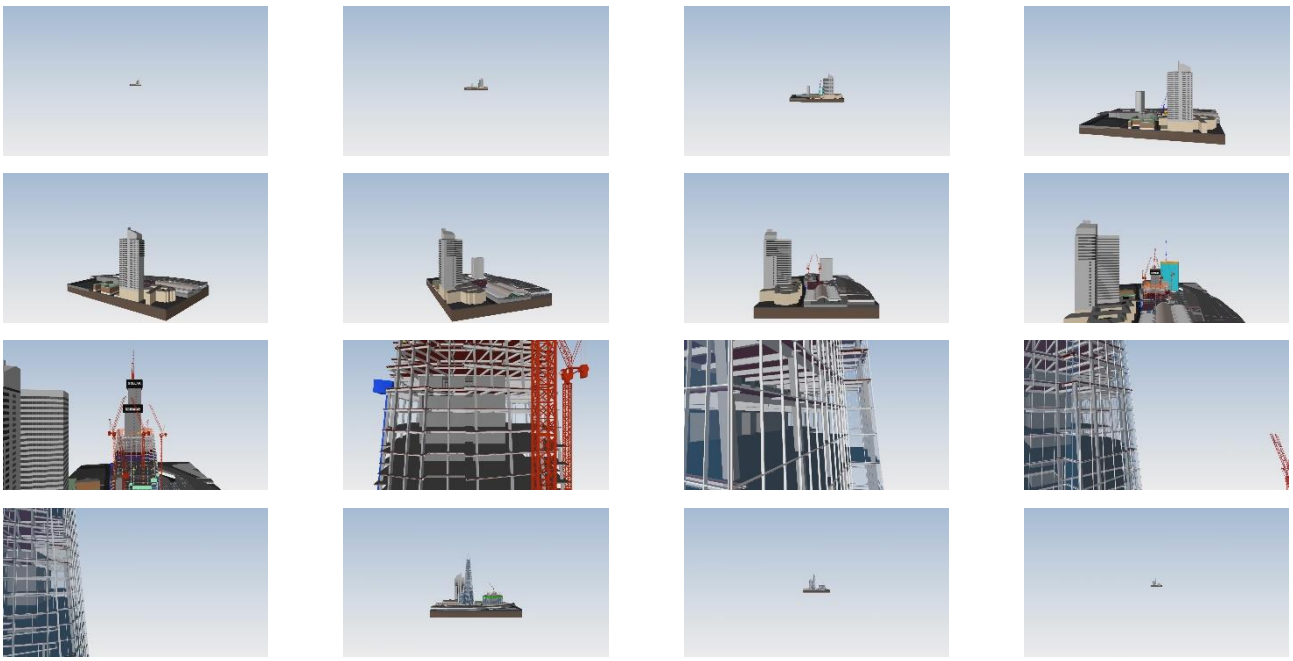


Рисунок 6.7 — Движение камеры по сцене во время эксперимента.

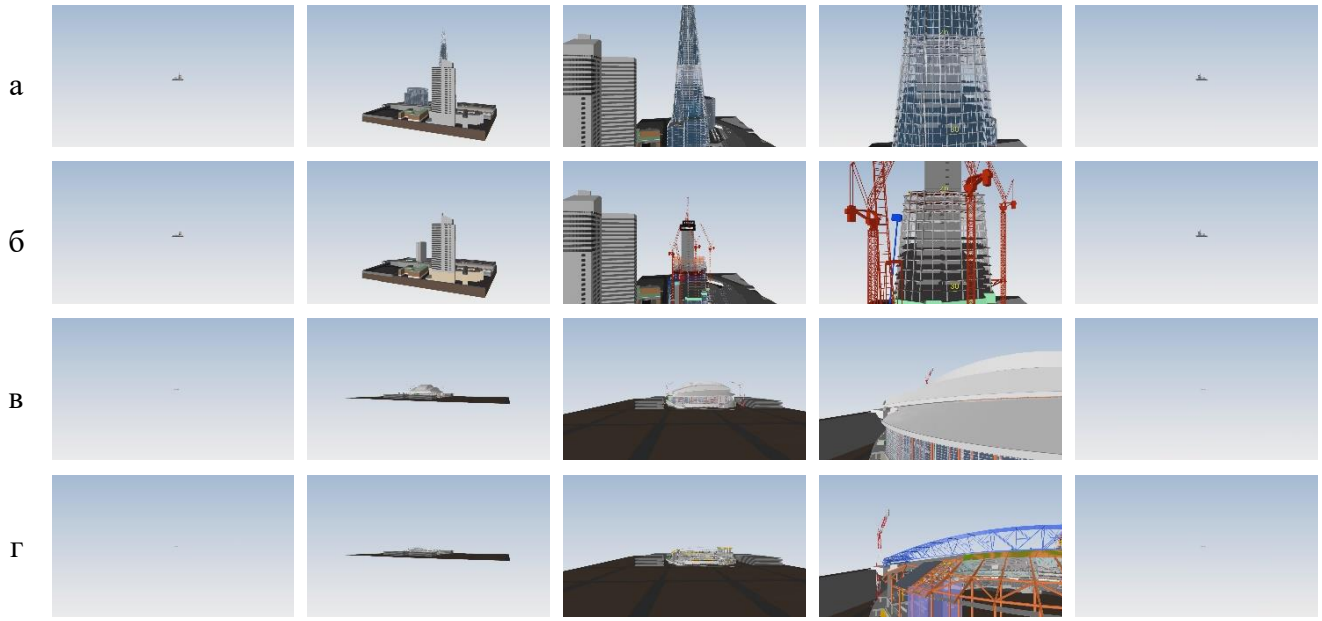


Рисунок 6.8 — Скриншоты, сделанные в ходе эксперимента: а) небоскрёб статика, б) небоскрёб анимация, в) стадион статика, г) стадион анимация.

Таблица 7 — Сравнение производительности рендеринга оригинального и многовариантного HDLOD методов.

Эксперимент	Мин. кадр/с	Макс. кадр/с	Сред. кадр/с	Сумм. время
Небоскрёб статика оригинальный	1.08	1076	36.4	13 с
Небоскрёб статика многовариантный	6.57	257	20.4	24 с
Небоскрёб статика без LOD	6.31	67.8	13.2	37 с
Небоскрёб анимация оригинальный	15.2	1065	77.7	6 с
Небоскрёб анимация многовариантный	7.17	251	20.1	24 с
Небоскрёб анимация без LOD	1.25	7.23	2.65	3 м 8 с
Стадион статика оригинальный	7.86	677	33.3	14 с
Стадион статика многовариантный	6.69	549	22.0	22 с
Стадион статика без LOD	7.36	86.0	21.1	23 с
Стадион анимация оригинальный	11.3	2666	65.1	7 с
Стадион анимация многовариантный	6.12	544	21.5	23 с
Стадион анимация без LOD	1.65	7.23	3.39	2 м 27 с

В таблице 7 приведены итоговые результаты. В ней показаны результаты замеров производительности, полученные с использованием оригинального и многовариантного HDLOD методов с одинаковой предельно допустимой погрешностью, а также производительность без

использования уровней детализации. В качестве метрик производительности приведены минимальные, максимальные и средние значения скорости рендеринга в кадрах в секунду, а также суммарное время, которое потребовалось на рендеринг 500 кадров.

Как можно видеть, оригинальный HDLOD метод является наиболее быстрым во всех экспериментах. Как и ожидалось, многовариантный метод является чуть более медленным за счёт более высоких накладных расходов на обработку кластеров во время рендеринга (таких как вычисление присутствия частей кластеров и передача данных о цвете и прозрачности на графический процессор). Оригинальный метод показывает наивысшую производительность для сценария анимации. Это происходит потому, что в оригинальном методе у каждого кластера есть своя функция присутствия. В начале анимации многие кластеры отсутствуют, и поэтому не отображаются. Во время анимации сооружение строится, появляются элементы конструкций, и вместе с этим растёт число отображаемых кластеров. Но в целом, итоговое число отображённых полигональных сеток и треугольников у оригинального метода оказывается меньше, чем у многовариантного. В многовариантном же методе, кластеры выбираются для отображения только с учётом их геометрической погрешности. Отсутствие объектов реализовано как сокрытие граней полигональных сеток в шейдере. Поэтому, для многовариантного метода, в сценариях «статика» и «анимация» отображаются одинаковое число сеток и треугольников, что выливается в одинаковую производительность. Метод рендеринга без уровней детализации, который использовался в данных экспериментах — это не просто наивный обход графа сцены. Он использует алгоритм, который предварительно подготавливает структуры данных для эффективного отсечения по области видимости и минимизации обхода графа сцены. Поэтому он показывает достаточно хорошую производительность при статическом сценарии, хотя и хуже HDLOD. Однако на сценарии «анимация» он демонстрирует совершенно неудовлетворительную производительность, поскольку требует обновления структур данных при изменении статуса присутствия объектов. С другой стороны HDLOD — это уже пространственно-временная структура данных, оптимизированная для динамического сценария.

Таблица 8 — Сравнение времени генерации HDLOD для многовариантного и оригинального методов.

Метод	Небоскрёб	Стадион
Многовариантный HDLOD	3 м 53 с	2 м 57 с
Оригинальный HDLOD	6 м 3 с	3 м 17 с

Также, были произведены замеры времени генерации HDLOD. Результаты представлены в таблице 8. При использовании многовариантного метода генерация происходит

быстрее, поскольку не учитывается временной фактор, в то время как в оригинальном методе необходимо производить вычисления временного расхождения d_T и функций $p(t)$ и $\delta(t)$. Но, что более важно, многовариантный метод не требует перегенерации в случае изменения динамического поведения, в то время как оригинальный потребует ждать несколько минут каждый раз, когда происходят изменения в динамике (например, пользователь редактирует проектный план в приложении 4D моделирования). Другим важным достоинством многовариантного метода является более низкий расход памяти в случае, когда сцена отображается в нескольких окнах с разными динамическими поведениями, поскольку при использовании многовариантного метода достаточно хранить кластеры только один раз (таблица 9).

Таблица 9 — Расход памяти на окна с разными динамическими поведениями одной сцены.

Метод	Одно окно	Два окна	Три окна	Четыре окна
Многовариантный HDLOD	186 МБ	238 МБ	439 МБ	587 МБ
Оригинальный HDLOD	203 МБ	579 МБ	884 МБ	1219 МБ

6.4 Влияние параметров генерации HDLOD

Для исследования влияния параметров генерации HDLOD была проведена серия экспериментов, в которых производилась генерация HDLOD с разными значениями параметров и сравнивалось время генерации, объём полученных данных на диске, а также производительность рендеринга.

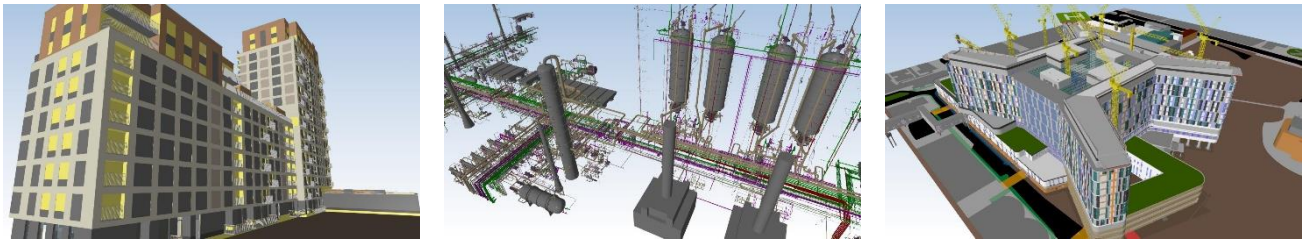
Эксперименты производились на трёх реальных индустриальных моделях, содержащих архитектурную и инженерную составляющую:

- модель жилого дома — 51 712 полигональных сеток, 1 556 416 треугольников (рис. 6.9а);
- модель нефтеперерабатывающего завода — 225 305 полигональных сеток, 19 729 304 треугольников (рис. 6.9б);
- модель клиники — 67 084 полигональных сеток, 1 934 955 треугольников (рис. 6.9в).

Конфигурация оборудования, на котором производилась генерация HDLOD:

- Для модели жилого дома: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz (4 ядра, 8 потоков), 32 ГБ ОЗУ, 1 ТБ SATA HDD 7200 RPM.

– Для модели завода: 11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz (6 ядер, 12 потоков), 64 ГБ ОЗУ, 10 ТБ SATA HDD 7200 RPM.
– Для модели клиники: Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz (6 ядер, 12 потоков), 32 ГБ ОЗУ, 500 ГБ NVMe SSD.



а) б) в)

Рисунок 6.9 — Модели: а) жилого дома, б) нефтеперерабатывающего завода, в) клиники.

Эксперименты по производительности рендеринга были проведены для двух сценариев:

1. десктопный интерактивный рендеринг, когда содержимое кластеров находится на диске и асинхронно загружается в видеопамять в ходе навигации по сцене;
2. веб-рендеринг был реализован с использованием Docker-контейнеризации и эмулирует реальные условия взаимодействия клиент-сервер с необходимостью пересылки содержимого кластеров по сети во время навигации по сцене.

Эксперимент проводился таким образом, что камера двигалась сквозь сцену по предопределённому пути на протяжении 500 кадров и замерялось время рендеринга каждого кадра. Сначала камера находится далеко от сцены, затем она подлетает к центру сцены, поворачивается и улетает вдаль с противоположной стороны сцены (рис. 6.10). Времена кадров, измеренные за весь пролёт, усреднялись и переводились в частоту кадров.

Спецификации оборудования, на котором производились эксперименты рендеринга:

– десктопный рендеринг моделей жилого дома и клиники, а также веб-рендеринг: ноутбук с Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz (8 ядер, 16 потоков), NVIDIA Quadro RTX 3000 6 ГБ, 32 ГБ ОЗУ, 1 ТБ NVMe SSD;
– десктопный рендеринг модели завода: 11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz (6 ядер, 12 потоков), NVIDIA GeForce GT 710 2 ГБ, 64 ГБ ОЗУ, 10 ТБ SATA HDD 7200 RPM;

Далее будут приведены таблицы с результатами экспериментов, отдельно для каждого параметра генерации HDLOD.

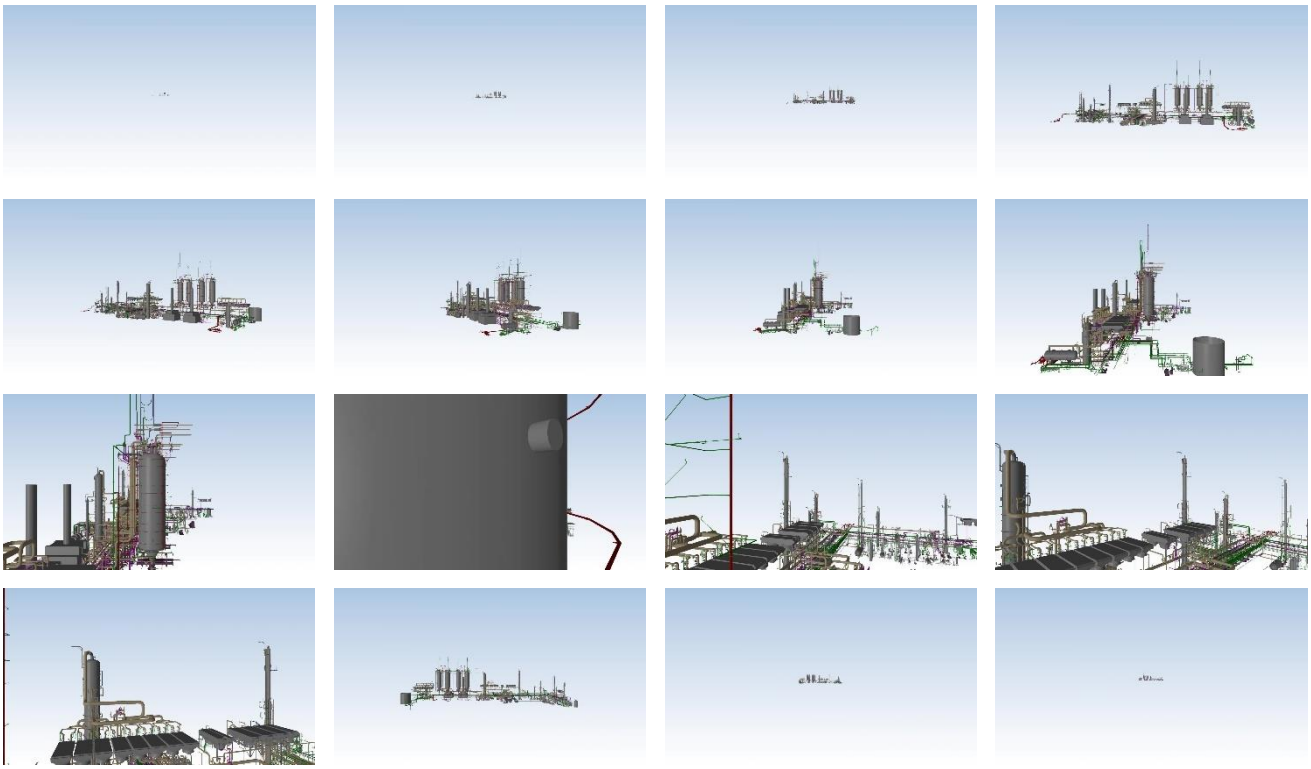


Рисунок 6.10 — Движение камеры в ходе эксперимента рендеринга модели завода.

6.4.1 **Размер кластера и общее число кластеров**

При генерации HDLOD можно управлять размерами кластеров. Сцену можно представить в виде большого числа мелких кластеров, или же малого числа крупных. Для оценки влияния этого фактора была проведена серия вычислительных экспериментов, в которой варьировался размер листового кластера. Напомним определения, приведённые в главе 3. Пусть W — размер сцены (длина диагонали ограничивающего параллелепипеда). Целевой размер каждого кластера на уровне l обозначается как $w(l)$. При этом выполняется соотношение $w(l+1) > w(l)$. В данном эксперименте варьировался размер листового кластера, то есть размер кластера на первом уровне дерева $w(1)$. Использовалось четыре значения $w(1) = 0{,}003W$, $w(1) = 0{,}03W$ и $w(1) = 0{,}3W$, а также предельное значение $w(1) = w_{min}$, где w_{min} — размер наименьшего из объектов сцены, что приводит к тому, что каждый объект сцены формирует свой листовой кластер. Для всех других уровней, кроме первого, размер кластера растёт в геометрической прогрессии согласно формуле: $w(l+1) = 2w(l)$. Результаты приведены в таблице 10.

Можно видеть, что увеличение числа кластеров негативно сказывается на производительности рендеринга. Это связано с ростом размера дерева и увеличением продолжительности его обхода, а также с ростом ассоциативных массивов, списков и прочих структур данных, используемых при кэшировании для отслеживания загруженных и недавно

использовавшихся кластеров, что увеличивает время поиска в этих структурах. Также, с ростом числа кластеров увеличивается общий размер файлов кластеров, что требует больше места для хранения, а также увеличивает объём читаемых данных (десктоп рендеринг), пересылаемых по сети данных (веб-рендеринг) и записываемых данных (время генерации). Большинство файловых систем и интернет-протоколов показывают лучшую производительность при работе с небольшим числом крупных файлов, нежели чем со множеством мелких файлов.

Таблица 10 — Результаты экспериментов: влияние числа кластеров на время генерации, скорость рендеринга и объём данных.

		$w(1) = w_{min}$	$w(1) = 0{,}003W$	$w(1) = 0{,}03W$	$w(1) = 0{,}3W$
Жилой дом	Число кластеров	64 117	63 499	17 530	74
	Суммарный размер кластеров	278 МБ	274 МБ	169 МБ	96 МБ
	Средний размер кластера	4 КБ	4 КБ	10 КБ	1 МБ 309 КБ
	Время генерации	1 м 57 с	1 м 45 с	1 м 57 с	2 м 27 с
	Рендеринг десктоп	111 кадр/с	105 кадр/с	173 кадр/с	3312 кадр/с
	Рендеринг веб	98 кадр/с	69 кадр/с	107 кадр/с	259 кадр/с
Завод	Число кластеров	340 986	66 265	5 289	125
	Суммарный размер кластеров	5 ГБ 452 МБ	1 ГБ 1019 МБ	820 МБ	784 МБ
	Средний размер кластера	17 КБ	32 КБ	159 КБ	6 МБ 276 КБ
	Время генерации	23 м 18 с	5 м 29 с	2 м 7 с	1 ч 18 м 40 с
	Рендеринг десктоп	273 кадр/с	300 кадр/с	389 кадр/с	2020 кадр/с
	Рендеринг веб	200 кадр/с	162 кадр/с	287 кадр/с	379 кадр/с
Клиника	Число кластеров	89 262	58 021	3 509	54
	Суммарный размер кластеров	429 МБ	355 МБ	142 МБ	113 МБ
	Средний размер кластера	5 КБ	6 КБ	41 КБ	2 МБ 93 КБ
	Время генерации	2 м 16 с	2 м 13 с	39 с	9 м 15 с
	Рендеринг десктоп	196 кадр/с	196 кадр/с	411 кадр/с	2807 кадр/с
	Рендеринг веб	140 кадр/с	97 кадр/с	119 кадр/с	229 кадр/с

Таким образом, для увеличения производительности рендеринга можно рекомендовать использовать более крупные кластеры. Однако предельный случай (поместить всю сцену в один кластер) тоже не будет работать. Для того чтобы визуализировать масштабные сцены, превышающие объём видеопамяти, разбиение на кластеры является необходимым, чтобы позволить загружать в видеопамять только релевантную порцию данных. Также, кластеры не должны быть слишком большими, чтобы отсечения невидимых поверхностей работали эффективно: слишком большой размер кластера может приводить к тому, что какая-то его часть всегда будет видна, что будет препятствовать его отсечению проверками видимости. В итоге, кластеры должны иметь разумный размер, чтобы представлять собой адекватное разбиение сцены на части, которые будут загружаться и выгружаться при навигации по сцене, а также позволят отсекать большие участки сцены проверками видимости. Другим недостатком крупных кластеров может стать необходимость единомоментно загружать большой объём данных в видеопамять, что может приводить к задержкам при навигации. Даже если в алгоритме интерактивного рендеринга (раздел 4.3) задать ограничение на загрузку не более одного кластера на кадр, в случае очень крупных кластеров, загрузка кластера в видеопамять может осуществляться долго и приводить к простою во время рендеринга. Поэтому более мелкий размер кластера может помочь производить загрузку сцены более плавно, без резких скачков.

Наконец, говоря про время генерации HDLOD, из таблицы можно заметить, что при самых крупных кластерах $w(1) = 0{,}3W$ генерация длится дольше всего. Это связано с необходимостью упрощать большие полигональные сетки, что увеличивает время генерации кластеров, а также расход оперативной памяти. Но при очень мелких кластерах $w(1) = w_{min}$ время генерации может быть больше, чем, например, при $w(1) = 0{,}03W$. Это связано с накладными расходами на обработку каждого кластера, в том числе открытие/закрытие файлов и операции с большим числом мелких файлов в ходе работы алгоритма генерации во внешней памяти.

6.4.2 **Прогрессия роста размеров кластеров**

В данном эксперименте исследовалось влияние шага геометрической прогрессии роста размеров кластеров. Целевой размер листового кластера был зафиксирован со значением $w(1) = 0{,}03W$, изменялся шаг p геометрической прогрессии роста размеров кластеров уровней дерева начиная со второго $w(l+1) = pw(l)$. Были выбраны значения $p = 1{,}5; p = 2; p = 3$. Результаты приведены в таблице 11.

Таблица 11 — Результаты экспериментов: влияние прогрессии роста размеров кластеров на скорость рендеринга.

		$w(l+1) = 1{,}5w(l)$	$w(l+1) = 2w(l)$	$w(l+1) = 3w(l)$
Жилой дом	Число кластеров	20 599	17 530	16 033
	Суммарный размер кластеров	217 МБ	169 МБ	143 МБ
	Средний размер кластера	11 КБ	10 КБ	9 КБ
	Время генерации	1 м 45 с	1 м 57 с	1 м 30 с
	Рендеринг десктоп	195 кадр/с	173 кадр/с	108 кадр/с
	Рендеринг веб	120 кадр/с	107 кадр/с	112 кадр/с
Завод	Число кластеров	6 451	5 289	4 799
	Суммарный размер кластеров	859 МБ	820 МБ	801 МБ
	Средний размер кластера	136 КБ	159 КБ	171 КБ
	Время генерации	2 м 5 с	2 м 7 с	2 м 24 с
	Рендеринг десктоп	409 кадр/с	389 кадр/с	295 кадр/с
	Рендеринг веб	380 кадр/с	287 кадр/с	207 кадр/с
Клиника	Число кластеров	4 212	3 509	3 145
	Суммарный размер кластеров	165 МБ	142 МБ	129 МБ
	Средний размер кластера	40 КБ	41 КБ	42 КБ
	Время генерации	39 с	39 с	36 с
	Рендеринг десктоп	390 кадр/с	411 кадр/с	349 кадр/с
	Рендеринг веб	160 кадр/с	119 кадр/с	162 кадр/с

Влияние на скорость рендеринга не очень значительное, однако в целом более низкое значение шага прогрессии показывает более высокую производительность. Это связано с тем, что при высоких значениях шага прогрессии размеры кластеров растут очень быстро с ростом уровня в дереве и для отображения объектов сцены может не найтись оптимального уровня детализации: будет выбран слишком качественный кластер с избыточным числом треугольников. Для того, чтобы выбирались наиболее оптимальные представления в терминах геометрическая погрешность/число треугольников, желательно иметь больше уровней детализации, то есть больше уровней дерева. Однако, как показал предыдущий эксперимент — слишком большое число кластеров — это тоже плохо, так как растут объёмы данных, которые приходится загружать/выгружать, и накладные расходы на обработку большого числа кластеров.

6.4.3 **Относительная погрешность**

В ходе процесса генерации кластеров происходит упрощение геометрии до целевой погрешности ε (см. раздел 3.3). Целевая погрешность кластера c выбирается относительно его фактического размера по формуле: $\varepsilon_c = rw_c$, где w_c — фактический размер кластера (длина диагонали его ограничивающего параллелепипеда), а r — параметр, называемый относительной погрешностью.

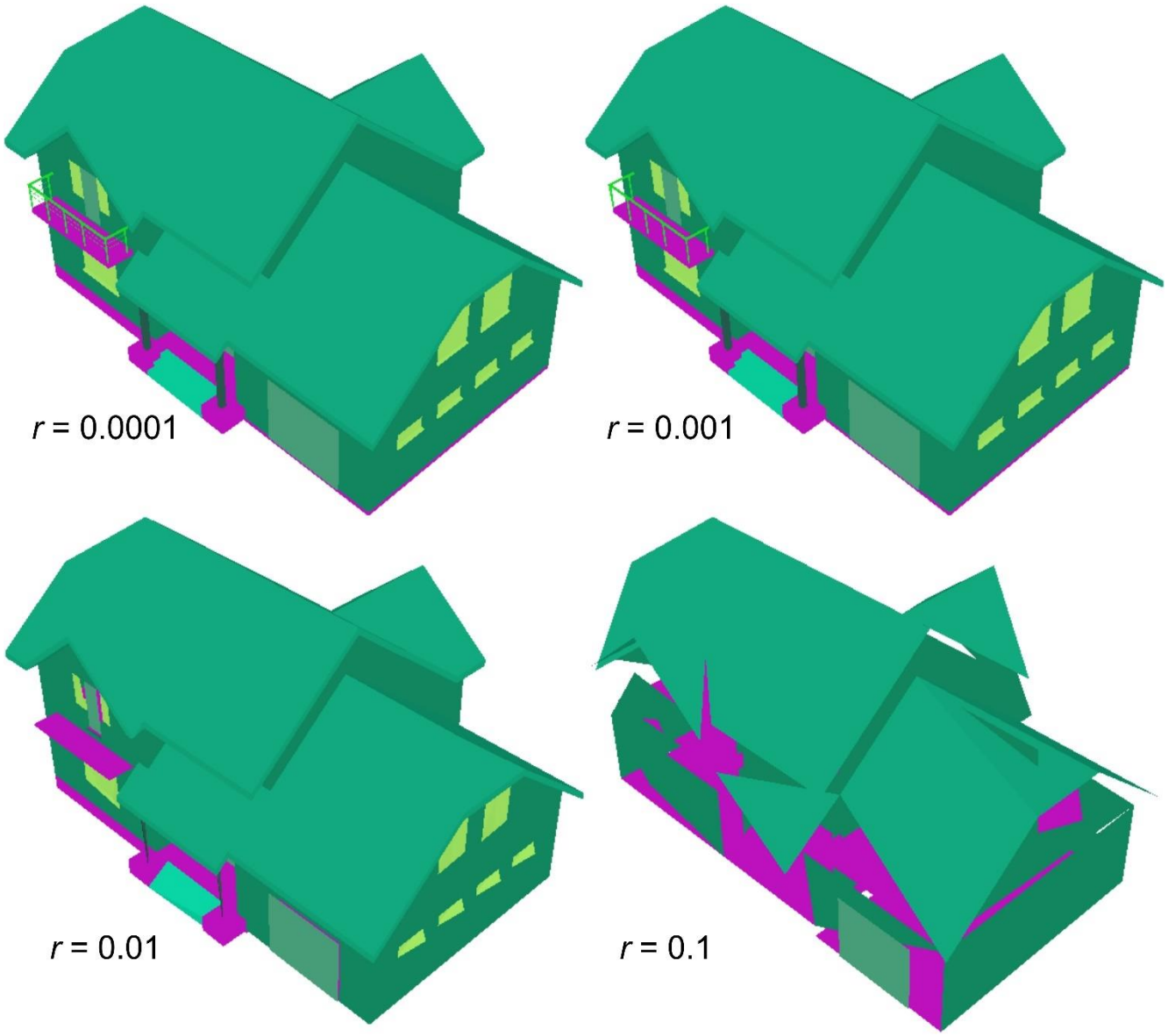


Рисунок 6.11 — Упрощение модели с разными значениями относительной погрешности.

Была проведена серия вычислительных экспериментов по исследованию влияния параметра относительной погрешности r на производительность рендеринга. Использовались значения $r = 0{,}1; r = 0{,}01; r = 0{,}001$. Пример влияния относительной погрешности на степень

упрощения модели представлен на рисунке 6.11. Результаты экспериментов приведены в таблице 12. Общий вывод: уменьшение относительной погрешности приводит к ускорению рендеринга.

Таблица 12 — Результаты экспериментов: влияние относительной погрешности упрощения кластеров на скорость рендеринга.

		$r = 0{,}1$	$r = 0{,}01$	$r = 0{,}001$
Жилой дом	Число кластеров	17 547	17 530	17 540
	Суммарный размер кластеров	120 МБ	169 МБ	341 МБ
	Средний размер кластера	7 КБ	10 КБ	20 КБ
	Время генерации	2 м 7 с	1 м 57 с	4 м 10 с
	Рендеринг десктоп	58 кадр/с	173 кадр/с	1385 кадр/с
	Рендеринг веб	56 кадр/с	107 кадр/с	259 кадр/с
Завод	Число кластеров	5 290	5 289	5 306
	Суммарный размер кластеров	791 МБ	820 МБ	1 ГБ 938 МБ
	Средний размер кластера	153 КБ	159 КБ	379 КБ
	Время генерации	2 м 55 с	2 м 7 с	5 м 41 с
	Рендеринг десктоп	199 кадр/с	389 кадр/с	1077 кадр/с
	Рендеринг веб	38 кадр/с	287 кадр/с	410 кадр/с
Клиника	Число кластеров	3 504	3 509	3 493
	Суммарный размер кластеров	117 МБ	142 МБ	370 МБ
	Средний размер кластера	34 КБ	41 КБ	109 КБ
	Время генерации	37 с	39 с	2 м 40 с
	Рендеринг десктоп	247 кадр/с	411 кадр/с	979 кадр/с
	Рендеринг веб	143 кадр/с	119 кадр/с	450 кадр/с

При высоких значениях относительной погрешности кластеры сильно упрощаются, что делает их подходящими для отображения только с больших расстояний. Поэтому, при близком рассмотрении объектов сцены, приходится использовать большое число низкоуровневых кластеров, что замедляет рендеринг за счёт увеличения времени обхода дерева и увеличения числа вызовов отрисовки. С другой стороны, если относительная погрешность низкая, кластеры могут использоваться даже при близком подлёте камеры к кластеру. Это уменьшает число кластеров на кадр и положительно влияет на производительность, так как сцена представляется небольшим числом крупных кластеров. Однако слишком низкие значения относительной погрешности могут приводить к тому, что кластеры будут слабо упрощаться и будут слишком

сложными. Поэтому на практике можно рекомендовать использовать значения порядка $r = 0{,}001$.

6.5 Сравнение интерактивного и консервативного рендеринга

Была проведена серия вычислительных экспериментов по интерактивному и консервативному рендерингу моделей из приложений информационного моделирования зданий (англ. building information modelling, BIM), а также информационного моделирования городов (англ. city information modelling, CIM). Были выбраны следующие динамические модели зданий: клиника, развлекательный комплекс, атомная электростанция. Модели городов были получены из публичных репозиториев в формате CityGML: модель города Монреаль (с текстурами) и модель города Таллин (без текстур).

Все модели представлены на рисунке 6.12. Их характеристики, а также характеристики сгенерированных HDLOD представлены в таблице 13.

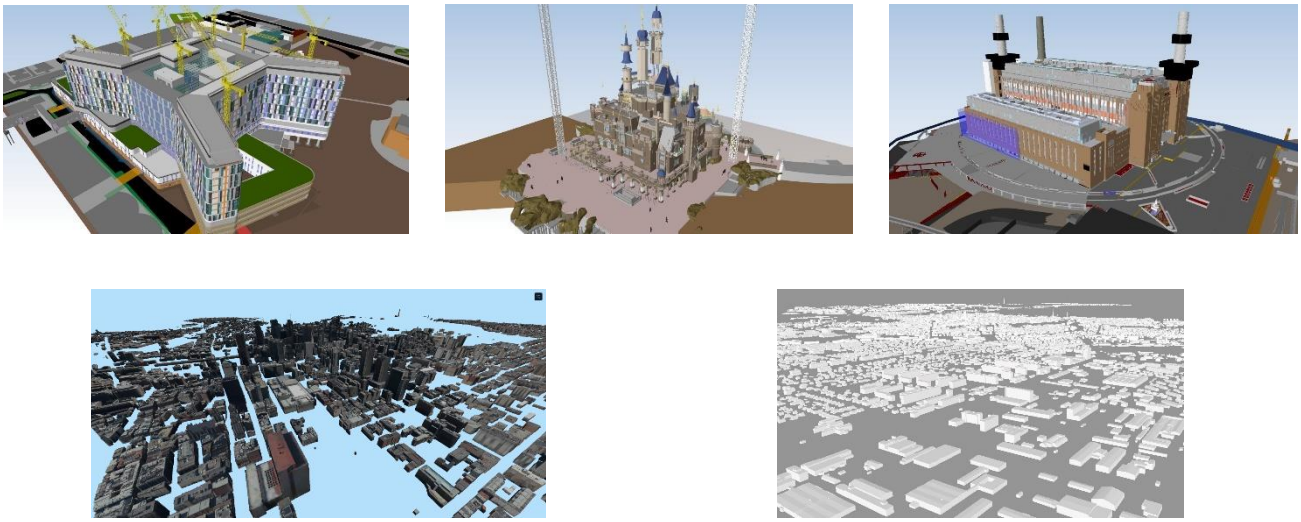


Рисунок 6.12 — Модели зданий, сооружений и городской инфраструктуры, используемых в экспериментах (слева-направо, сверху-вниз): клиника, развлекательный комплекс, атомная электростанция, урбанистические модели Монреаля и Таллина.

Каждый эксперимент был устроен следующим образом: начальное положение камеры находится в углу ограничивающего объёма сцены. В огранивающем объёме сцены случайным образом выбирается точка и камера плавно движется к этой точке в течение 10 секунд. После этого выбирается следующая точка, и процесс повторяется. Общая длительность эксперимента: 5 минут. Во время движения камера всегда смотрит в центр сцены. Точки выбираются с

использованием генератора случайных чисел, однако семя генератора фиксировано, то есть последовательность точек одинакова для каждого запуска эксперимента. В моделях с динамикой в течение эксперимента модельное время плавно изменялось от начала к концу проекта, то есть проигрывалась анимация постройки здания.

Таблица 13 — Характеристики моделей и полученных HDLOD представлений.

Модель	Число полиг. сеток в исх. модели	Сумм. число треуг. в исх. модели	Число клас-теров	Сумм. число треуг. в кластерах	Размер HDLOD на диске	Динамика	Текстуры
Клиника	57723	1913255	1184	3728959	411 МБ	Да	Нет
Развлекательный комплекс	17938	19788295	966	24794493	2,47 ГБ	Да	Нет
Электростанция	67345	42606836	2218	50865364	5,03 ГБ	Да	Нет
Модель Монреаля	113657	4937427	468	9067142	1,89 ГБ	Нет	Да
Модель Таллина	49172	3117368	234	4628654	574 МБ	Нет	Нет

Таблица 14 — Производительность рендеринга моделей в консервативном и интерактивном режимах.

Модель	Режим рендеринга	Мин. кадр/с	Сред. кадр/с	Макс. кадр/с	Итого кадров
Клиника	Консервативный	56	59	61	17 990
	Интерактивный	57	59	60	17 997
Развлекательный комплекс	Консервативный	8	52	60	15 867
	Интерактивный	26	53	60	16 040
Электростанция	Консервативный	0	38	61	11660
	Интерактивный	3	43	61	12 992
Модель Монреаля	Консервативный	0	32	60	10 283
	Интерактивный	38	58	62	17 657
Модель Таллина	Консервативный	40	59	60	17 953
	Интерактивный	59	59	60	17 999

В ходе эксперимента замерялась минимальная, максимальная и средняя частота кадров, а также общее число показанных кадров. Максимальная частота кадров была ограничена частотой вертикальной развёртки дисплея 60 герц. Рендеринг производился в разрешении 1920x1080. Конфигурация оборудования, на котором производились эксперименты: ноутбук с Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz (8 ядер, 16 потоков), NVIDIA Quadro RTX 3000 6 ГБ, 32 ГБ ОЗУ, 1 ТБ NVMe SSD. Размер кэша был ограничен значением 4 ГБ, чтобы избежать превышения объёма доступной видеопамяти с некоторым запасом. Результаты экспериментов представлены в таблице 14.

Интерактивный режим обеспечивает хорошую среднюю частоту кадров в секунду, близкую к 60. Консервативный режим по метрике средней частоты кадров несколько отстаёт от интерактивного. Однако, если смотреть на минимальную частоту кадров в секунду, то разница между режимами оказывается более значительной. Это связано с тем, что при консервативном рендеринге могут возникать моменты, когда требуется одномоментно загрузить большое количество кластеров для обеспечения заданного уровня качества, что приводит к задержкам в ходе рендеринга. При интерактивном рендеринге производится загрузка не более одного кластера на кадр, что позволяет сгладить неравномерности частоты кадров при навигации по сцене, однако в такие моменты происходит падение качества (так как кластеры требуемого качества отображаются не сразу).

На рис. 6.13 представлен график частоты кадров, который позволяет подробней сравнить производительность консервативного и интерактивного режимов в ходе экспериментов с моделью развлекательного комплекса. Можно видеть, что для консервативного режима характерны периодические провалы производительности, связанные с подгрузкой данных. Интерактивный режим также демонстрирует некоторые резкие провалы, связанные с загрузкой крупных кластеров в видеопамять, однако они происходят реже, и не такие серьёзные как при консервативном режиме. В ходе первой половины эксперимента оба режима показывают достаточно стабильные 60 кадров в секунду. Во второй половине производительность обоих режимов ниже — это связано с увеличением общей сложности модели. Так как модель динамическая, в ходе эксперимента растёт число отображаемой геометрии (по мере строительства появляются новые конструктивные элементы), что приводит к увеличению числа отображаемых треугольников и увеличению загруженности графического процессора.

На рис. 6.13 также представлен график качества в интерактивном режиме. Он получен как отношение числа треугольников, отображённых в интерактивном режиме, к числу треугольников, отображённых в консервативном режиме. В начале эксперимента качество интерактивного режима ниже, но нарастает, пока не достигнет целевого качества. Это связано с тем, что в интерактивном режиме дерево кластеров загружается последовательно сверху-вниз.

То есть, чтобы отобразить нужные кластеры, сначала должны быть загружены все их родители. В то время как в консервативном режиме загружаются только те кластеры, которые нужны в данный момент. Далее, на графиках можно видеть, что падения качества в интерактивном режиме происходят в те моменты времени, когда происходят падения производительности. Это моменты во время навигации, когда камера приближается к объектам и требуется подгрузить большое число новых детальных кластеров (которые до этого не использовались и их нет в кэше), или же необходимость загрузки новых кластеров возникает в связи с динамическими изменениями в сцене (добавилось большое число конструктивных элементов). За счёт таких падений качества интерактивный режим позволяет сгладить падения производительности.

Рисунок 6.13 — Графики частоты кадров (консервативного и интерактивного режимов) и качества (интерактивного режима) в эксперименте с моделью развлекательного комплекса.

По результатам экспериментов можно сделать следующий вывод: интерактивный режим следует рекомендовать для визуализации в ходе навигации пользователя по сцене, для обеспечения более плавного пользовательского опыта и предотвращения задержек в реакции программы на пользовательский ввод. Консервативный режим следует рекомендовать для процессов оффлайн рендеринга, результатами которых являются видео или серии изображений (видео с демонстрацией процесса строительства, презентационное видео облёта модели, визуализация результатов научной симуляции).

6.6 Сравнение программной реализации HDLOD с Cesium Ion

Иерархические уровни детализации применяются во многих областях, а их реализации имеются во многих программных продуктах. Однако, как правило — это проприетарные реализации, встроенные в исходный код графического движка. Исключение составляет графический веб-движок Cesium и его открытая спецификация 3D Tiles [12]. Cesium позволяет визуализировать иерархические уровни детализации в формате 3D Tiles. Также, разработчики Cesium предлагают веб-сервис Cesium Ion, который может генерировать иерархические уровни детализации в формате 3D Tiles. Поддерживается ряд 3D форматов, а также форматы для цифровых моделей зданий и сооружений, такие как IFC [14]. В Cesium Ion можно загрузить трёхмерную модель и получить 3D Tiles тайлсет. К сожалению, графический интерфейс Cesium Ion реализован таким образом, что он не позволяет оценить время генерации. Другим ограничением Cesium Ion является отсутствие поддержки какой-либо динамики в моделях. Наконец, эксперименты с Cesium Ion позволили понять, что генерируемые им тайлсеты не являются иерархическими уровнями детализации в традиционном смысле. Дело в том, что спецификация 3D Tiles допускает два способа уточнения геометрии (REFINE и ADD) при переходе от родительских тайлов к дочерним (тайл в терминологии 3D Tiles — аналог кластера в терминологии HDLOD). Способ REFINE соответствует тому, как работает метод HDLOD: дочерние тайлы заменяют собой родителя. Способ ADD говорит о том, что геометрия дочерних тайлов должна быть дорисована в дополнение к геометрии родителя. Для всех моделей, выбранных для данного эксперимента, при генерации тайлсета Cesium Ion применил только способ уточнения ADD. Это означает, что тайлсет содержит упрощённый корневой тайлы, а все остальные тайлы в нём — это детали, которые должны быть дорисованы в дополнение к корневому тайлу. Невозможно сказать, в какой степени производилось полигональное упрощение. Возможно, всё множество треугольников было просто поделено между тайлами.

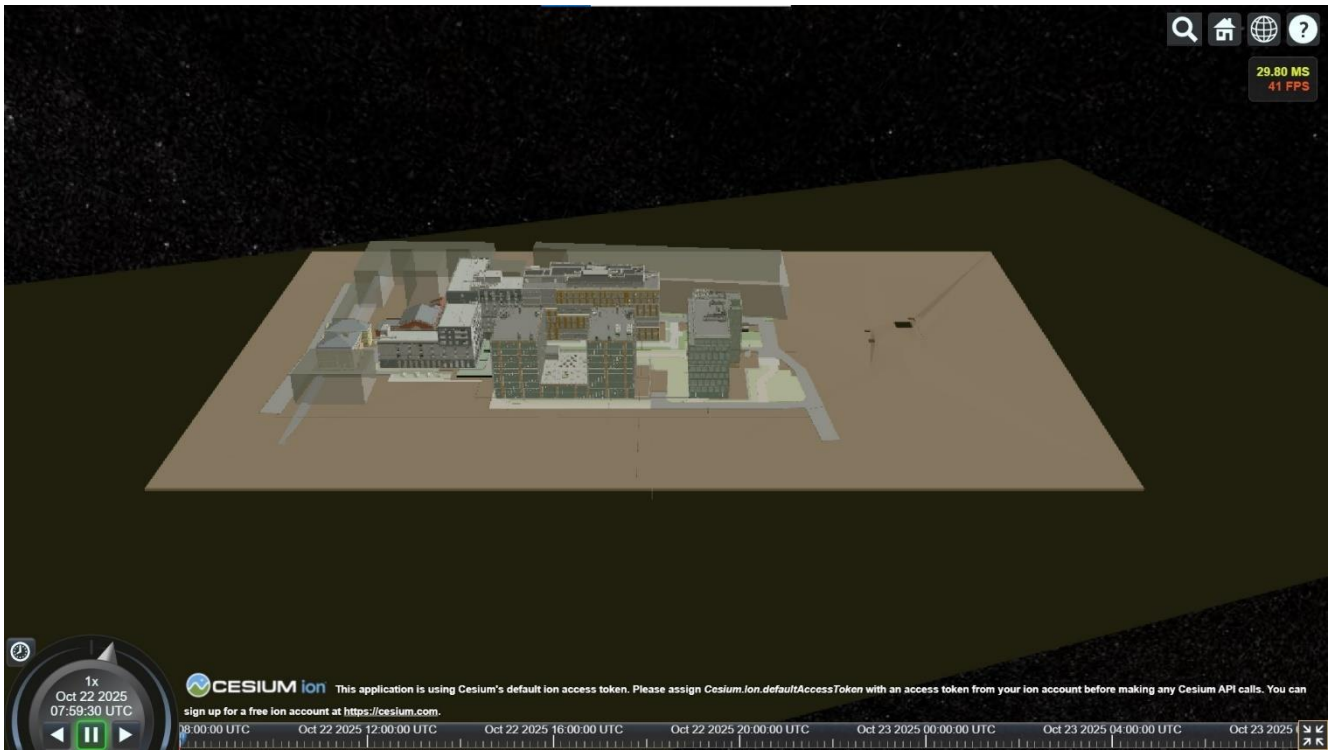


Рисунок 6.14 — Снимок экрана в ходе эксперимента (модель отеля).

С учётом описанных особенностей, тем не менее была произведена серия вычислительных экспериментов. В Cesium загружались тайлсеты для моделей, полученные с помощью Cesium Ion и по методу HDLOD. Размеры тайлсетов представлены в таблице 15. Все сцены были статическими. Использовались модели из предыдущих экспериментов: клиника, стадион, развлекательный комплекс, нефтеперерабатывающий завод, а также модель отеля (рис. 6.14). При визуализации камера устанавливалась таким образом, чтобы вся сцена попадала в область видимости. Предельно допустимая экранная погрешность была зафиксирована со значением в 1 пиксель. Замерялась производительность рендеринга в кадрах в секунду. Конфигурация оборудования Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz, NVIDIA Quadro RTX 3000 6 ГБ, 32 ГБ ОЗУ, 1 ТБ NVMe SSD. Использовался браузер Google Chrome. Рендеринг производился в двух разрешениях: HD 1280x720 (результаты в таблице 16) и 4K 3840x2160 (результаты в таблице 17). При фиксированной предельной экранной погрешности и фиксированном положении камеры это означает, что при более высоком разрешении сцена отображается с большей степенью детализации.

Таблица 15 — Размер файлов в формате 3D Tiles.

	Клиника	Стадион	Развлекательный комплекс	Завод	Отель
HDLOD	376 МБ	999 МБ	2,28 ГБ	2,53 ГБ	3,03 ГБ
Cesium Ion	125 МБ	655 МБ	1,64 ГБ	1,82 ГБ	1,72 ГБ

Таблица 16 — Производительность рендеринга (в кадрах/с) в разрешении 1280x720.

	Клиника	Стадион	Развлекательный комплекс	Завод	Отель
HDLOD	174	275	389	451	384
Cesium Ion	218	54	33	101	41

Таблица 17 — Производительность рендеринга (в кадрах/с) в разрешении 3840x2160.

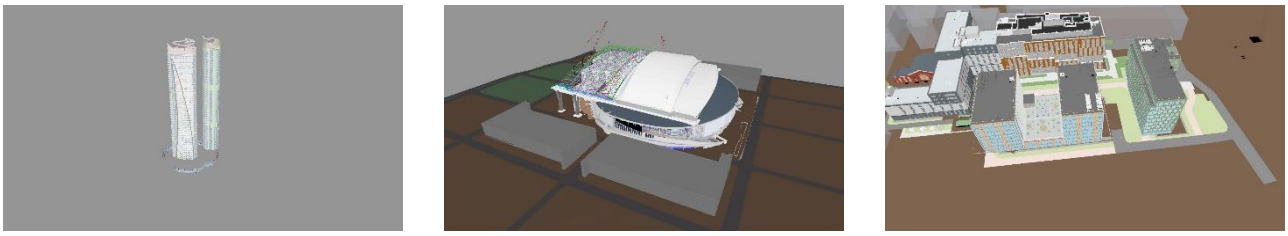
	Клиника	Стадион	Развлекательный комплекс	Завод	Отель
HDLOD	36	47	45	68	69
Cesium Ion	44	37	21	37	27

Таблица 15 позволяет сравнить размеры файлов при генерации уровней детализации по методу HDLOD и с использованием коммерческого решения Cesium Ion. Cesium Ion достигает меньшего размера файлов, что может быть связано с использованием способа уточнения ADD. Такой способ позволяет избежать дублирования представлений, в то время как способ REFINE, используемый в методе HDLOD обеспечивает наличие нескольких альтернативных представлений для одних и тех же объектов. Поскольку реализация Cesium Ion является закрытой, нет возможности установить, как происходит формирование тайлов: это лишь разбиение множества треугольников на кластеры, или же упрощение геометрии с выделением простых форм и дополнительных деталей.

Таблицы 16 и 17 позволяют сравнить производительность рендеринга уровней детализации, вычисленных по методу HDLOD, с коммерческим решением Cesium Ion. На всех моделях, кроме клиники (самая маленькая модель), метод HDLOD демонстрирует уверенное преимущество. Это может быть связано с тем, что способ уточнения ADD, используемый Cesium Ion, требует большего числа вызовов отрисовки во время рендеринга, так как одновременно отображаются родительские и дочерние узлы дерева. Более того, способ REFINE, используемый в методе HDLOD, предоставляет упрощённые представления сразу для больших групп объектов, в то время как ADD может потребовать дорисовывать множество деталей в дополнение к упрощённому представлению.

6.7 Оптимизации генерации HDLOD

Были исследованы различные способы ускорения генерации HDLOD и проведены серии вычислительных экспериментов: эксперименты с распараллеливанием, эксперименты с удалением внутренних граней и эксперименты с инкрементальными обновлениями. Все они проводились на компьютере с типовой конфигурацией Intel(R) Core(TM) i9-9880 CPU @ 3.60GHz (8 ядер, 16 потоков), 32 ГБ ОЗУ, 2 ТБ NVME SSD. Использовался единый набор тестовых сцен, реконструированных по реальным цифровым моделям строительных проектов с соответствующими архитектурными, конструктивными и инженерными элементами (рис. 6.15). Сцена высотного здания содержала 146 461 объект с тесселированной геометрией в виде полигональных сеток и общим числом треугольников 3 426 267. Сцены стадиона и отеля содержали 41 451 и 47 160 объектов, 7 960 512 и 29 131 139 треугольников соответственно.



(а) (б) (в)

Рисунок 6.15 — Цифровые модели строительных объектов: а) высотное здание б) стадион в) отель.

6.7.1 **Распараллеливание**

Для оценки эффективности параллельных оптимизаций была проведена серия экспериментов. Производилась генерация HDLOD с распараллеливание и без. Напомним, что обработка уровней выполняется последовательно. При этом кластеры одного уровня обрабатываются параллельно в несколько потоков, поскольку их представления зависят только от кластеров нижних уровней. После завершения всех потоков на очередном уровне начинается параллельная обработка кластеров следующего уровня. На первом уровне дерева кластеры формируются из оригинальных представлений объектов сцены. В данных экспериментах распараллеливание на первом уровне не применялось из-за особенностей реализации (сложность с организацией параллельного доступа к представлениям объектов в базе данных). Параллельная обработка начиналась со второго уровня, где, собственно, и имеет место вычислительно

затратное упрощение полигональных сеток. В таблице 18 приведено сравнение времени генерации уровней детализации без распараллеливания и с распараллеливанием для трёх тестовых сцен. В ней же указано количество кластеров на каждом уровне дерева, начиная со второго.

Таблица 18 — Эксперименты с распараллеливанием.

Модель	Кол-во кластеров на уровнях иерархии	Последовательное выполнение	Параллельное выполнение	Ускорение
Здание	17, 9, 4, 2, 1, 1	7м 57с	3м 2с	2.6x
Стадион	27, 15, 8, 5, 2, 1, 1	36м 28с	19м 56с	1.8x
Отель	127, 64, 32, 16, 8, 4, 2, 1	12ч 51м 38с	3ч 37м 18с	3.6x

Отметим, что в данных экспериментах использовалась самая простая реализация: обработка дерева кластеров осуществлялась в цикле по уровням, а обработка кластеров одного уровня — во вложенном цикле, который был распараллелен буквально с применением одной инструкции на языке C++. Тем не менее, даже такая простая реализация позволяет добиться существенного ускорения (например, с 12 до 3 часов в случае модели отеля). Также следует отметить, что в данном случае параметры генерации HDLOD были установлены таким образом, что кластеры были крупными, а их число было относительно небольшим. Увеличение числа кластеров усилило бы эффект от распараллеливания и поспособствовало ускорению генерации, но негативно сказалось бы на производительности рендеринга (см. раздел 6.4.1). Дальнейшего улучшения эффективности распараллеливания можно добиться путём усложнения реализации: использовать пул потоков и очередь задач, чтобы динамически балансировать нагрузку между потоками. Главное помещать кластер в очередь задач только после того, как все его дети обработаны, чтобы избежать состояний гонки.

6.7.2 **Удаление внутренних граней**

Вторая серия экспериментов по ускорению генерации предназначалась для оценки техники полигонального упрощения с удалением внутренних граней. Обнаружение внутренних граней было реализовано с использованием техники бросания лучей. Лучи выпускались из точек, равномерно распределенных по ограничивающей сфере кластера, в направлении ее центра. Для поиска пересечений лучей с гранями полигональной сетки применялась библиотека Embree, в которой данные операции эффективно реализованы с использованием пространственного

индексирования на основе иерархий ограничивающих объёмов. Грани, в которые попал хотя бы один луч, помечались как внешние, остальные — как внутренние. После завершения процесса бросания лучей и поиска пересечений все грани, помеченные как внутренние, удалялись. Для каждого кластера поиск и удаление внутренних граней осуществлялись сразу после формирования его полигональной сетки (путём объединения сеток дочерних кластеров) и перед запуском процедуры упрощения полигональной сетки.

Достоверность идентификации внутренних граней описанным способом существенно зависит от числа выпущенных лучей. С увеличением числа лучей она будет монотонно повышаться, однако процесс потребует больше времени. При этом будет обнаружено больше внешних граней и меньше внутренних. Число лучей в проводимых экспериментах варьировалось, начиная от десяти тысяч и заканчивая миллиардом лучей (на каждый кластер). Для каждой тестовой сцены замерялось время генерации всех уровней (начиная со второго) с использованием техники распараллеливания. Также определялись общее количество удалённых граней во всех кластерах, процент от общего числа граней в кластерах без удаления внутренних граней и размер полученных иерархических уровней детализации, сохранённых на диске в формате 3D Tiles. Полученные результаты приведены в таблице 19. В качестве референсной информации также укажем общее число граней в кластерах HDLOD иерархии без применения процедуры удаления внутренних граней: 6 826 213 (высотное здание), 13 853 526 (стадион) и 57 705 028 (отель).

Удаление внутренних граней имеет значительный эффект, который выражается как в ускорении времени генерации HDLOD, так и в меньшем объеме оперативной памяти и видеопамяти, необходимой для хранения и отображения уровней детализации. Ускорение достигается за счёт сокращения числа граней в полигональной сетке и более быстрой последующей работы алгоритма упрощения (с меньшим числом итераций, необходимым для достижения целевого числа треугольников, и меньшим размером вспомогательных индексных структур, поиск в которых, как правило, осуществляется за логарифмическое время от числа элементов). Удаление внутренних граней имеет накопительный эффект в рамках применяемой стратегии кластеризации. Поскольку при объединении представлений кластеров часть треугольников перекрывается и удаляется, на каждом следующем уровне иерархии всё меньше треугольников участвует в полигональных упрощениях. Упрощение уровней детализации HDLOD при их прежнем визуальном качестве улучшает также и производительность рендеринга за счёт сокращения общего числа отображаемых треугольников и сокращения объёмов загружаемых данных.

Таблица 19 — Эксперименты с удалением внутренних граней.

		Без удаления внутренних граней	10 000 лучей	100 000 лучей	1 000 000 лучей	10 000 000 лучей	100 000 000 лучей	1 000 000 000 лучей
Здание	Время генерации	3м 2с	0.58с	2.41с	16.68с	1м 11с	4м 5с	24м 57с
	Кол-во удалённых граней	0	3416063 (50%)	3378397 (49%)	3267249 (48%)	3072979 (45%)	2815698 (41%)	2589898 (38%)
	Размер файлов	969 МБ	482 МБ	508 МБ	554 МБ	613 МБ	654 МБ	714 МБ
Стадион	Время генерации	19м 56с	0.86с	1.56с	13.77с	57с	5м 59с	1ч 3м 22с
	Кол-во удалённых граней	0	7942012 (57%)	7905560 (57%)	7778853 (56%)	7567357 (55%)	7267930 (52%)	6889677 (50%)
	Размер файлов	2.24 ГБ	1.26 ГБ	1.27 ГБ	1.3 ГБ	1.35 ГБ	1.41 ГБ	1.47 ГБ
Отель	Время генерации	3ч 37м 18с	2.72с	3.41с	10.81с	1м 4с	7м 44с	1ч 7м 59с
	Кол-во удалённых граней	0	29103008 (50%)	29044720 (50%)	28876971 (50%)	28480811 (49%)	27715526 (48%)	26732145 (46%)
	Размер файлов	11.4 ГБ	5.78 ГБ	5.81 ГБ	5.86 ГБ	5.98 ГБ	6.18 ГБ	6.42 ГБ

Конкретные значения ускорения существенно зависят от достоверности идентификации внутренних граней и варьируются в диапазоне от нескольких единиц до трех порядков. Максимальное ускорение в 4 793 раз достигается на тестовой сцене отеля при числе лучей 10 000, однако достоверность определения внутренних граней невелика и, по крайней мере, 4% граней идентифицируется неверно (оценка получается путем сравнения доли внутренних или внешних граней, обнаруженных при числе лучей 10 000 и 1 000 000 000). Это может привести к нежелательным артефактам, например, к дырам во внешних поверхностях или удаленным

видимым элементам внутри сцены. При числе лучей 10 000 000 вычислительные затраты на определение пересечений возрастают и частично нивелируют эффект от сокращения числа полигонов. Ускорение на тестовой сцене отеля составляет уже 203. Дальнейшее увеличение числа лучей до 1 000 000 000 приводит к снижению ускорения, а для сцен с высотным зданием и стадионом эффект от применяемой оптимизации и вовсе становится отрицательным.

Результаты экспериментов показывают, что для разных тестовых сцен и разных кластеров одни и те же оценки достоверности достигаются при существенно разном числе лучей. Поэтому на практике их довольно сложно контролировать. Возможная итеративная схема для получения оценки заключается в кратном увеличении числа лучей для каждого обрабатываемого кластера до тех пор, пока изменения в доле внутренних или внешних граней не зафиксируются с заданной погрешностью. В любом случае, компромисс между производительностью генерации уровней детализации HDLOD и достоверностью их отображения должен искаться с учетом требований, предъявляемых к графическому приложению. Однако основным выводом является то, что при правильном подборе параметров удаления внутренних граней, можно обеспечить и ускорение генерации, и уменьшение объёма данных, и ускорение рендеринга. Таким образом, удаление внутренних граней настоятельно рекомендуется к применению в приложениях, где оно применимо (где не требуется заглядывать внутрь объектов, проводить секущие плоскости или придавать прозрачность объектам).

6.7.3 **Инкрементальные обновления**

Финальная серия экспериментов предназначалась для оценки техники инкрементального обновления HDLOD при локальных изменениях в сцене. Эксперименты были организованы следующим образом: некоторые объекты сцены помечались как изменившиеся, после чего запускалось обновление HDLOD. По набору изменившихся объектов определялись листовые кластеры первого уровня, требующие обновления. Затем итеративно определялись кластеры, требующие обновления на следующих уровнях, согласно такому принципу: кластер требует обновления, если хотя бы один из его детей изменился. После чего запускался процесс генерации представлений для помеченных кластеров. Заметим, что положение объектов, их габариты и ограничивающие объемы не менялись, поэтому все вычисления выполнялись с прежней иерархией кластеров. Было реализовано два способа моделирования изменений в сцене: путем случайного отбора объектов и путем выбора объектов, компактно расположенных в центре сцены при минимизации их общего огранивающего объёма. Эксперименты проводились для следующих случаев: когда изменился один случайный объект, когда изменилось 5% объектов

сцены (смежных и случайных) и когда поменялось 10% объектов сцены (смежных и случайных). Замерялось время генерации кластеров всех уровней, начиная со второго, а также число кластеров второго уровня, которые потребовали обновления. В таблице 20 приведены полученные результаты. В экспериментах полигональные упрощения кластеров проводились параллельно. Техника удаления внутренних граней не применялась, чтобы не исказить эффект от самих инкрементальных обновлений.

Таблица 20 — Эксперименты с инкрементальными обновлениями.

		Полная генерация	Изменение 1 объекта	5% смежных изменений	5% случайных изменений	10% смежных изменений	10% случайных изменений
Здание	Время	3м 2с	1м 6с	1м 20с	2м 16с	1м 18с	2м 26с
	Кол-во объектов	146464	1	7323	7323	14646	14646
	Кол-во кластеров уровня 2	17	1	3	12	3	12
Стадион	Время	19м 56с	3м 1с	12м 53с	20м 1с	12м 59с	20м 32с
	Кол-во объектов	41451	1	2072	2072	4145	4145
	Кол-во кластеров уровня 2	27	1	7	24	8	26
Отель	Время	3ч 37м 18с	10м 27с	44м 17с	1ч 36м 1с	1ч 48с	2ч 9м 20с
	Кол-во объектов	47160	1	2358	2358	4716	4716
	Кол-во кластеров уровня 2	127	1	9	39	13	53

Данная серия экспериментов подтверждает важность инкрементальных обновлений для приложений, нуждающихся в оперативном пересчете уровней детализации. Ускорение на

тестовых сценах высотного здания, стадиона и отеля с одним изменившимся объектом составило 2.75, 6.6 и 20.8 соответственно.

Ключевым фактором, влияющим на эффективность инкрементальных обновлений, оказывается пространственная локализация изменений, а не просто их число. В самом деле, если изменения затронули объекты только одного листового кластера, то обновление будет осуществлено относительно быстро в результате распространения изменений по родительской ветке вверх по иерархии. И наоборот, если изменившиеся объекты находятся в разных листовых кластерах, то потребуется инкрементальная обработка большой части кластеров, что может оказаться более затратным, чем полный пересчет всей иерархии HDLOD заново.

Так, в экспериментах со сценой высотного здания время обновления при 5% изменившихся объектов и при 10% не отличается, поскольку изменения затрагивают одинаковое число кластеров (3 в случае смежных изменений и 12 в случае изменения случайных объектов). Более того, на всех сценах 10% изменившихся смежных объектов затрагивают меньше кластеров, чем 5% изменившихся случайных объектов, и, как следствие, обновление для 10% смежных происходит быстрее, чем для 5% случайных.

Необходимо также учитывать фактор параллельности в проведенных экспериментах, который несколько завуалирует эффект от инкрементальной обработки кластеров. Если изменился, к примеру, один листовой кластер, это потребует обработать его и всех его родителей (по пути от листа до корня). При этом на каждом уровне будет обрабатываться только один кластер, поэтому параллельность не будет задействована. Если изменилось несколько кластеров на одном уровне дерева, включится фактор параллельности.

В целом, инкрементальные обновления необходимы в приложениях, в которых большие сцены подвергаются перманентным изменениям и изменения локализованы в пространстве.

Глава 7. Приложения метода HDLOD

В настоящей главе рассматриваются графические приложения, в которых метод HDLOD был успешно применен для рендеринга сложных динамических трехмерных сцен, а именно: сервис для управления замечаниями в архитектурно-строительных проектах, система визуального пространственно-временного моделирования индустриальных проектов и приложение для градостроительного планирования.

7.1 Сервис для управления замечаниями в архитектурно-строительных проектах

Сервис управления замечаниями является одним из ключевых компонентов «Национальной технологической платформы для информационного моделирования», разрабатываемой в Институте системного программирования им. В.П. Иванникова РАН для поддержки технологически сложных проектов в архитектурно-строительной отрасли [17, 9]. Открытые и общедоступные веб-сервисы [15] основаны на применении открытых информационных стандартов и предназначены для комплексного решения задач цифровизации требований (рис. 7.1), включая спецификацию требований с использованием стандарта IDS (Information Delivery Specification) [63], верификацию и валидацию цифровых информационных моделей зданий и сооружений (ЦИМ), определяемых стандартом IFC (Industry Foundation Classes) [14], а также управление замечаниями, представленными в соответствии со стандартом BCF (BIM Collaboration Format) [24]. В настоящее время поддерживаются актуальные версии стандартов IFC 2x3, IFC 4, IFC 4.3, IDS 1.0 и BCF 3.0. Для сервиса управления замечаниями получено свидетельство о государственной регистрации программы для ЭВМ [12].

Сервис редактирования машинно-интерпретируемых требований предназначен для просмотра, модификации и документирования формализованных требований, представленных в соответствии с открытыми стандартами IFC и IDS [18]. Сервис верификации предназначен для проверки ЦИМ, представленных стандартом IFC, на соответствие синтаксису файлового формата SPF (STEP Physical File – ISO 10303-21:2016) и семантике формальной схемы IFC, специфицированной на языке объектно-ориентированного моделирования EXPRESS (ISO 10303-11:2004) [16].

ПРОВЕРКА ЦИМ НА НАЦИОНАЛЬНОЙ ТИМ ПЛАТФОРМЕ ИСП РАН

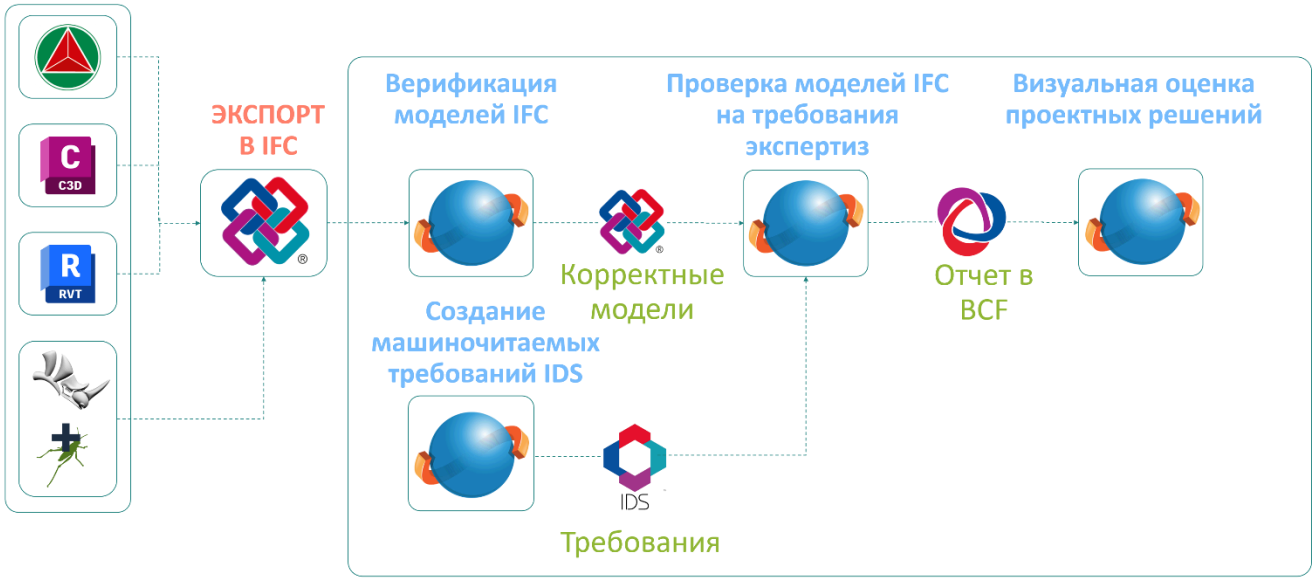


Рисунок 7.1 — Функциональность открытых веб-сервисов.

Сервис валидации (рис. 7.2) предназначен для проверки ЦИМ, представленных стандартом IFC, на соответствие требованиям участников проектной деятельности, спецификации которых подготовлены на основе стандарта IDS [16]. Сервис позволяет проконтролировать полноту объектного и атрибутного состава ЦИМ, согласованность элементов ЦИМ, правил именования объектов, наличие отношений композиции между объектами, назначенных свойств, классификаторов и материалов, ожидаемые области значений для атрибутов и свойств. Выявленные сервисом ошибки заносятся в журнал и отправляются пользователю в виде HTML документа или BCF файла для дальнейшего анализа и просмотра в графических интерактивных инструментах.

Сервис управления замечаниями предназначен для просмотра и редактирования журналов замечаний, представленных в соответствии с актуальной версией формата BCF 3.0, сгенерированных сервисом валидации ЦИМ или подготовленных в ПО ТИМ третьих сторон. При открытии вместе с файлом BCF соответствующей IFC модели сервис обеспечивает интерактивную 3D визуализацию замечаний непосредственно в окне веб-браузера. Это осуществляется путем отображения модели с ракурсов, предустановленных в выбранном замечании, и применения предписанных техник отображения к элементам модели, указанным в замечании (рис. 7.3). При редактировании замечания в сервисе можно скорректировать список элементов, установить новые точки обзора с графическими или текстовыми пометками в 3D окне, задать иные техники отображения, а также прикрепить к замечанию сгенерированные

изображения. Поскольку интерактивный рендеринг больших 3D сцен, порождаемых IFC моделями, вызывает известные технологические проблемы для веб-приложений, в реализации сервиса применён метод иерархических динамических уровней детализации HDLOD.

Сервис управления замечаниями предназначен для просмотра и редактирования журналов замечаний, представленных в соответствии с актуальной версией формата BCF 3.0, сгенерированных сервисом валидации ЦИМ или подготовленных в ПО ТИМ третьих сторон [19]. При открытии вместе с файлом BCF соответствующей IFC модели сервис обеспечивает интерактивную 3D визуализацию замечаний непосредственно в окне веб-браузера. Это осуществляется путем отображения модели с ракурсов, предустановленных в выбранном замечании, и применения предписанных техник отображения к элементам модели, указанным в замечании (рис. 7.3). При редактировании замечания в сервисе можно скорректировать список элементов, установить новые точки обзора с графическими или текстовыми пометками в 3D окне, задать иные техники отображения, а также прикрепить к замечанию сгенерированные изображения. Поскольку интерактивный рендеринг больших 3D сцен, порождаемых IFC моделями, вызывает известные технологические проблемы для веб-приложений, в реализации сервиса применён метод иерархических динамических уровней детализации HDLOD.

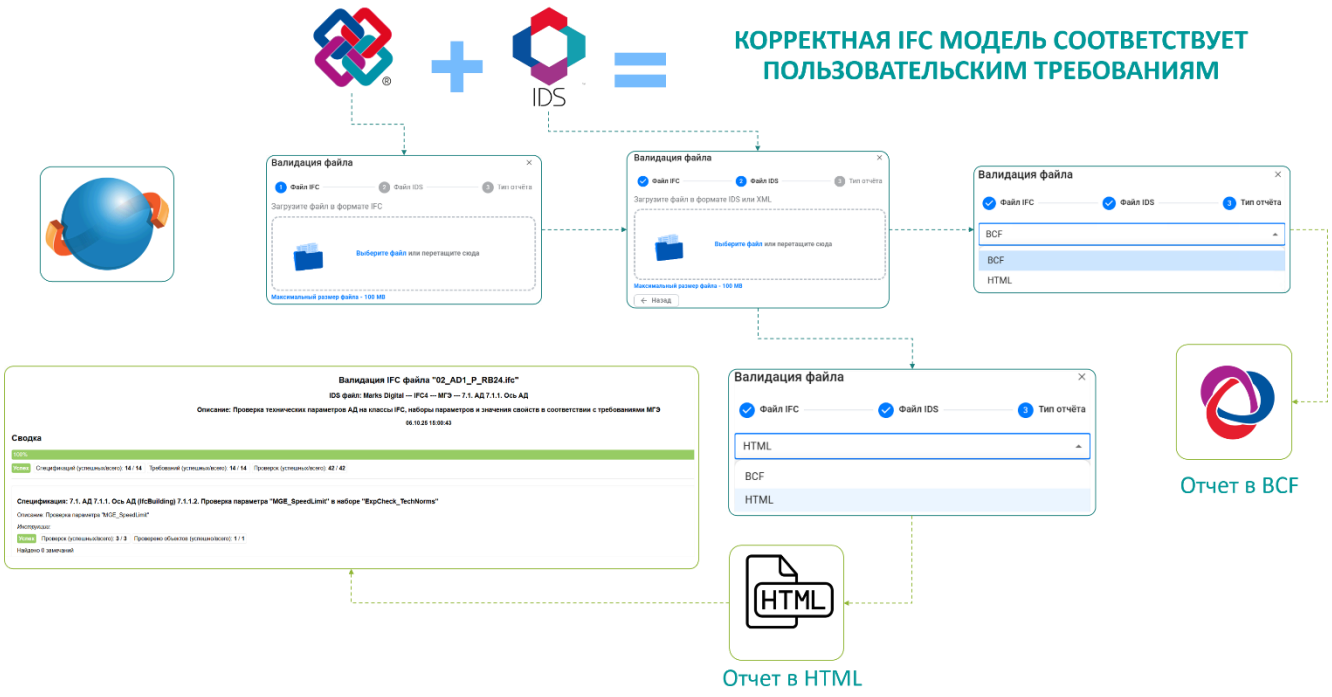


Рисунок 7.2 — Функциональность сервиса валидации.

Применение метода HDLOD в сервисе управления замечаниями обеспечило эффективную реализацию его ключевых функций:

– визуальный просмотр замечаний с фиксированных точек просмотра;
– учёт динамических параметров в замечаниях, в частности, модельного времени и состояния объектов;
– оперативное обновление сцены при переключении между замечаниями и смены модельного времени;
– визуализация больших промышленных и инфраструктурных проектов, состоящих из сотен IFC моделей.

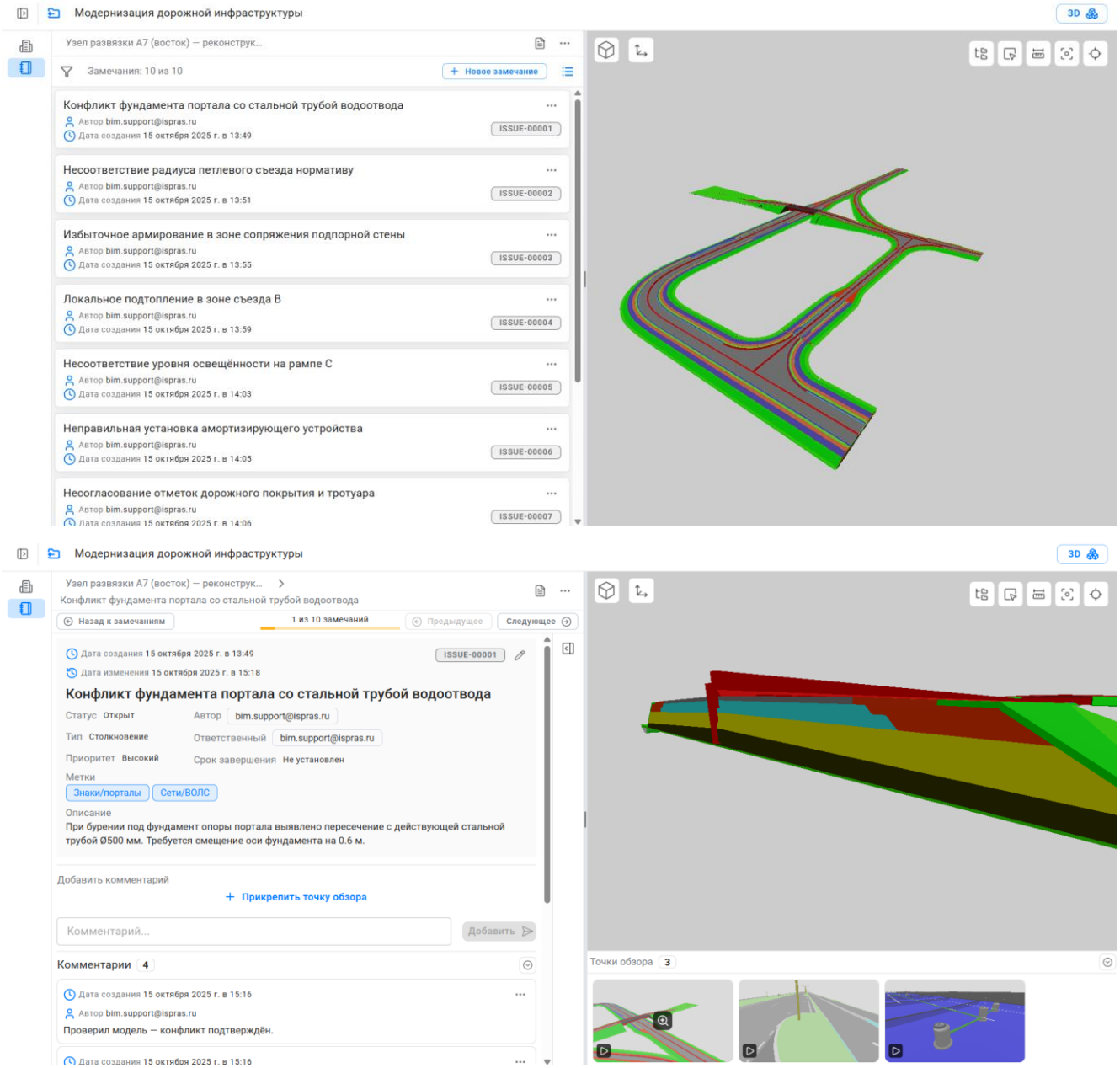


Рисунок 7.3 — Снимки окна сервиса управления замечаниями.

7.2 Система визуального пространственно-временного моделирования индустриальных проектов

HDLOD метод нашел применение в системе визуального пространственно-временного моделирования индустриальных проектов «Synchro 4D», разработанной в рамках совместного научно-исследовательского проекта ИСП РАН. Назначение и базовые принципы системы описаны в работах [107-111]. Текущая версия системы и основные функции представлены на сайте продукта [115].

Система позволяет визуально промоделировать ход реализации проекта. На диаграмме Ганта отображаются задачи со сроками, длительностями и взаимосвязями. Есть возможность создать несколько окон с 3D видами сцены, что позволяет просматривать модель с разных ракурсов. Более того, в 3D окнах могут использоваться разные фильтры и стили элементов, а также разные версии расписания. Например, в одном окне визуализация может осуществляться согласно запланированным датам, а в другом — согласно актуальным. В процессе строительства пользователь производит актуализацию плана, что позволяет сравнить процесс строительства, как он был изначально запланирован, и как он реализуется на самом деле. Представления в 3D окнах синхронизированы с временным репером на диаграмме Ганта. Перемещая временной репер, пользователь может оценивать состояние модели на разные моменты времени (в том числе согласно разным версиям расписания). Снимок окна приложения представлен на рис. 7.4.

Основным назначением системы является комплексное визуальное моделирование и планирование проектов с учетом технологической и пространственно-временной согласованности проектных работ, условий их логистического и финансового обеспечения.

Благодаря развитым графическим средствам визуализации и анимации проектных работ система обладает важными преимуществами над традиционными инструментами управления проектами (в том числе, популярными программными продуктами MS Project, Oracle Primavera, Spider), а именно:

- повышает достоверность планирования проектов и составления календарно-сетевых графиков с учетом пространственно-временных, ресурсных и стоимостных факторов;
- улучшает координацию и взаимодействие участников проектной деятельности;
- обеспечивает эффективный мониторинг хода проекта и анализа альтернативных сценариев реализации проекта;
- позволяет принимать обоснованные управленческие решения со снижением рисков, затрат и сроков выполнения проектных работ.

Применение метода HDLOD при реализации графических средств рендеринга системы обеспечило возможность визуализации и анимации сложных динамических сцен с большим числом объектов, имеющих индивидуальные полигональные представления и поведенческие характеристики. Другой важной возможностью стало отображение в нескольких окнах альтернативных сценариев, основанных на разных календарно-сетевых графиках и использующих разные визуальные стили представления объектов (например, с учетом опережения или запаздывания соответствующих проектных работ). С этой целью была успешно применена многовариантная версия HDLOD метода.

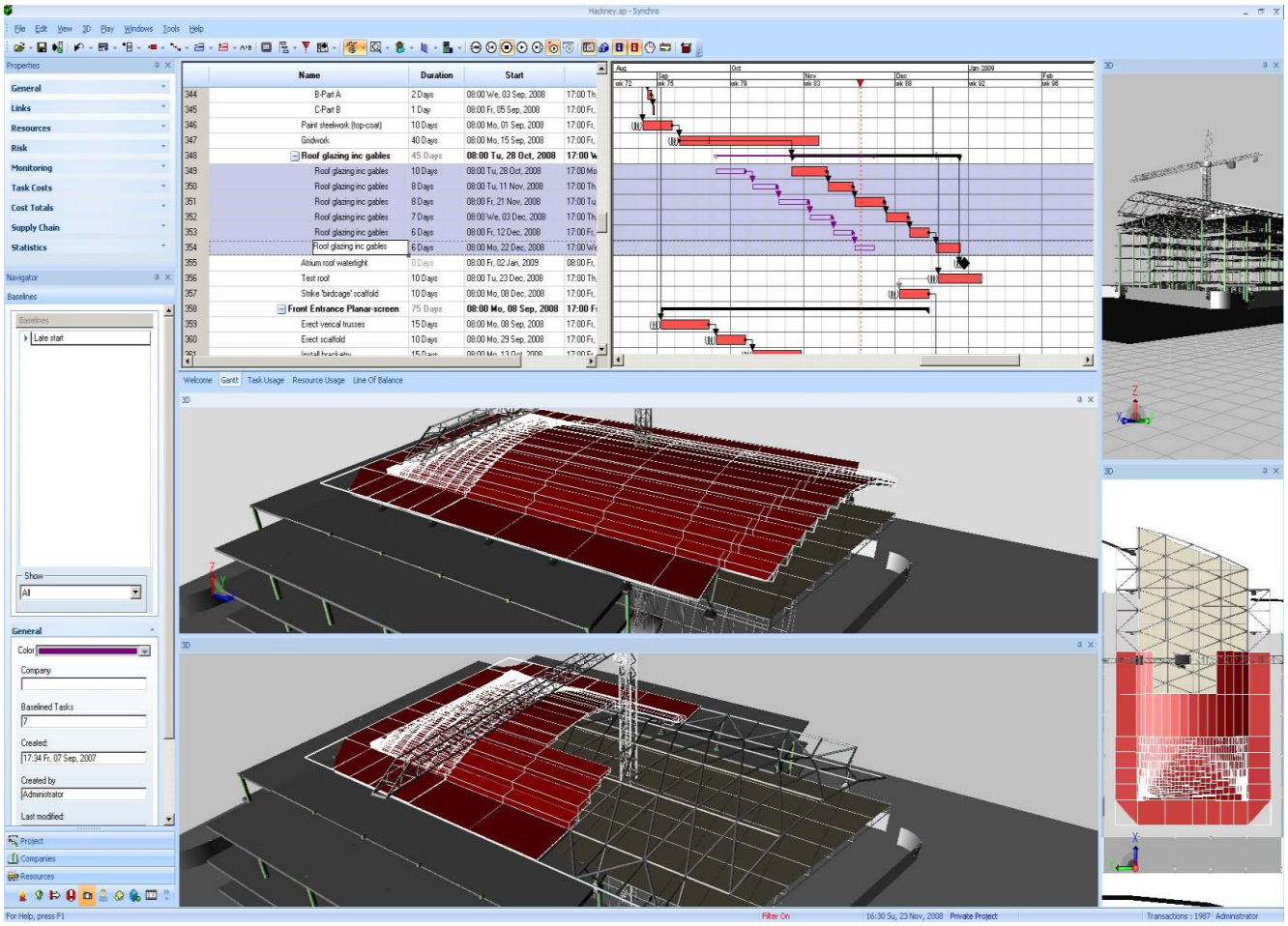


Рисунок 7.4 — Снимок окна приложения для визуального пространственно-временного моделирования индустриальных проектов.

7.3 Приложение для градостроительного планирования

Метод HDLOD был успешно применен для рендеринга больших динамических сцен в составе разработанного приложения для градостроительного планирования. Основные

принципы функционирования приложения в результате консолидации ЦИМ для городской инфраструктуры и объектов капитального строительства (ОКС), представленных в форматах CityGML и IFC соответственно, опубликованы в работе [6].

Приложение имеет клиент-серверную архитектуру и обеспечивает визуальное моделирование городской инфраструктуры и процессов комплексной застройки непосредственно в веб-браузере (рис. 7.5). Как правило, в геоинформационных системах применяются условные обозначения или сильно упрощённые модели для представления ОКС в виде простых контуров или оболочек без внутренних элементов, что существенно ограничивает возможности для детального планирования строительных, ремонтных и эксплуатационных работ. Разработанное приложение позволяет визуализировать модели городов с высокой степенью детализации на уровне элементов ЦИМ для отдельных зданий и сооружений с общим объемом данных, исчисляемым сотнями гигабайт. Подобные ЦИМ содержат не только подробную информацию о конструкции здания (стены, перекрытия, лестницы и т.д.), но и детальные модели элементов коммуникаций (сантехника, электрика, лифтовое оборудование и т.д.), что позволяет использовать ЦИМ на протяжении всего жизненного цикла ОКС, включая проектирование, строительство, эксплуатацию и демонтаж. Таким образом, приложение реализует единую концепцию цифрового двойника города и цифровых двойников ОКС, сочетая в себе функции градостроительного планирования и управления строительными проектами ОКС.

Для хранения моделей ОКС и обеспечения доступа к ним используется база данных на сервере, а веб-приложение загружает с сервера и визуализирует HDLOD представления, сгенерированные для индивидуальных моделей ОКС. Поддерживаются средства навигации по сцене, поиска здания по заданным атрибутам (например, по местоположению, адресу, назначению, этажности, профилю располагающихся в нем организаций и т.п.), вывода полной информации о здании и его элементах. Приложение также обеспечивает анимацию истории застройки города и процессов сооружения ОКС, что является важным как для управления отдельными строительными проектами, так и для градостроительного планирования в целом.

Реализация HDLOD метода в составе описанного выше приложения обеспечила возможность рендеринга сверхбольших и сверхсложных сцен. Для этого использовалась многоуровневая вложенная организация HDLOD представлений, при которой листовой кластер одного представления (например, наиболее детального представления здания на уровне городской модели) может ссылаться на корневой кластер другого представления (например, на наиболее упрощенное представление индивидуальной модели здания). При этом генерировалась одно HDLOD представление для модели всего города и по одному HDLOD представлению на каждую индивидуальную модель здания.

Отметим, что рендеринг сцены, порождаемой цифровой моделью одного объекта гражданского строительства или промышленного сооружения с сотнями тысяч, а иногда и миллионами элементов, представляет собой серьезную технологическую проблему. Приблизительно ту же сложность имеют цифровые модели городов, в которых десятки и сотни тысяч зданий представлены относительно простыми текстурированными полигональными сетками. Тем не менее, HDLOD метод с описанной многоуровневой организацией успешно справляется с задачами рендеринга статических и динамических сцен, порождаемыми детальными моделями городов и зданий.

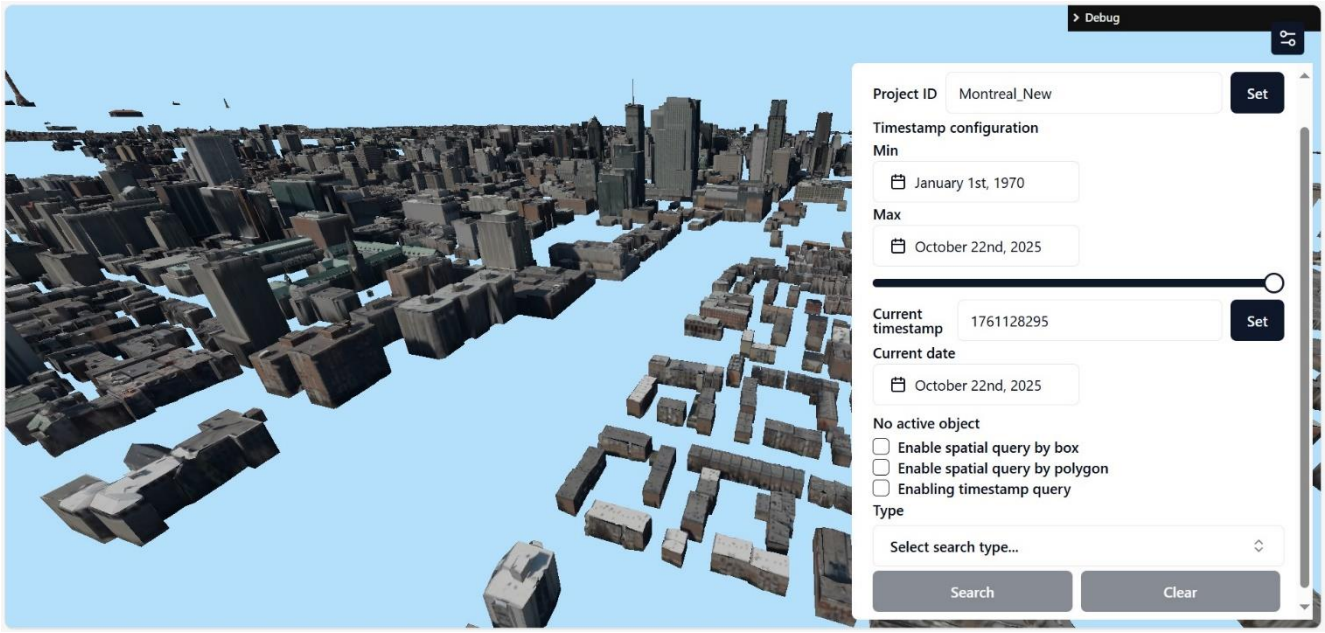


Рисунок 7.5 — Снимок окна веб-приложения для градостроительного планирования.

Таким образом, разработанный метод HDLOD имеет важные, индустриально значимые приложения.

Заключение

Основные результаты работы заключаются в следующем:

1. Проведен анализ существующих подходов к рендерингу сложных сцен, на основе которого определены возможности обобщения методов уровней детализации для класса полигональных сцен с детерминированным дискретно-непрерывным характером динамики, имеющего важные индустриальные приложения.
2. Предложен и разработан метод иерархических динамических уровней детализации HDLOD для представления и эффективного рендеринга динамических полигональных сцен выделенного класса.
3. Разработаны и исследованы алгоритмы вычисления уровней детализации HDLOD во внешней памяти на основе классификации объектов сцены, многоуровневой кластеризации и серий полигональных упрощений. Вычисленные уровни детализации позволяют оперативно формировать альтернативные упрощенные представления сцены с учетом положения пространственной камеры и временного репера, разрешения устройства отображения и заданной визуальной точности.
4. Разработаны и исследованы алгоритмы консервативного (гарантирующего заданную пространственную и временную точность) и интерактивного (стремящегося обеспечить максимально возможную частоту генерации изображений при приемлемом уровне реализма) рендеринга сцен во внешней памяти. Алгоритмы предусматривают раздельное хранение иерархии уровней и геометрических моделей, а также кэширование моделей в основной и видеопамяти с асинхронной загрузкой и вытеснением. Благодаря возможности переключения между консервативным и интерактивным режимами непосредственно во время исполнения целевого приложения обеспечивается сбалансированное по производительности и качеству отображение сложных сцен.
5. Разработана и исследована многовариантная модификация HDLOD метода для приложений, в которых происходят частые изменения динамических характеристик объектов или требуется одновременное отображение динамической сцены с альтернативными характеристиками объектов в нескольких окнах.
6. Разработанный HDLOD метод и связанные с ним алгоритмы вычисления и рендеринга уровней детализации реализованы в виде библиотеки программ на языке C++ с использованием графического интерфейса OpenGL и предложенного расширения формата 3D Tiles для хранения уровней детализации.

7. Проведена серия вычислительных экспериментов на синтетических и реальных индустриальных сценах, результаты которых подтверждают высокую эффективность разработанных метода и библиотеки и определяют область применения для широкого класса графических приложений.
8. Благодаря достигнутым характеристикам HDLOD метод нашел применение в ряде прикладных проектов, имеющих целью создание приложений визуального пространственно-временного моделирования индустриальных проектов, градостроительного планирования, управления требованиями и замечаниями в архитектурно-строительной отрасли.

Список литературы

1. Semenov V., Shutkin V., Zolotov V. Visualization of complex industrial products and processes using hierarchical dynamic LODs //Advances in Transdisciplinary Engineering, 2019, 10, pp. 655–664.
2. Semenov V., Shutkin V., Zolotov V. Faster rendering of large pseudo-dynamic scenes using hierarchical dynamic LODs //Multi Conference on Computer Science and Information Systems, MCCSIS 2019 - Proceedings of the International Conferences on Interfaces and Human Computer Interaction 2019, Game and Entertainment Technologies 2019 and Computer Graphics, Visualization, Computer Vision and Image Processing 2019, 2019, pp. 275–282.
3. Семёнов В.А., Шуткин В.Н., Золотов В.А., Морозов С.В. Расширение метода иерархических уровней детализации для динамических сцен с детерминированным характером событий //Труды конференции ГрафиКон-2019, стр. 37-41, DOI: 10.30987/graphicon-2019-1-37-41.
4. 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, 46(3), pp. 223–232.
5. Семенов В.А., Шуткин В.Н., Морозкин Н.К. Эффективный подход к 4D-визуализации масштабных строительных проектов и программ на основе иерархических динамических уровней детализации //Материалы IV Международной научно-практической конференции BIM-моделирование в задачах строительства и архитектуры (BIMAC 2021), 2021, СПб, стр. 385-393. DOI: 10.23968/BIMAC.2021.048.
6. Shutkin V., Morozkin N., Zolotov V., Semenov V. City and building information modelling using IFC standard //ECPPM 2021 – eWork and eBusiness in Architecture, Engineering and Construction: Proceedings of the 13th European Conference on Product & Process Modelling (ECPPM 2021), 2021, Taylor & Francis Group, London, ISBN 978-1-032-0438-9, pp. 406-413, DOI:10.1201/9781003191476-56.
7. V. Semenov, V. Shutkin, V. Zolotov. Conservative Out-of-Core Rendering of Large Dynamic Scenes Using HDLODs //Proceedings of the 31st International Conference on Computer Graphics and Vision (GraphiCon 2021), 2021, Nizhny Novgorod, Russia, pp. 105-115, DOI:10.20948/graphicon-2021-3027-105-115.
8. Shutkin V., Semenov V., Zolotov V., Morozkin N. Alternative HDLOD method for large scenes with multiple dynamic behaviors //Proceedings of 17th International Conference on Computer

Graphics, Visualization, Computer Vision and Image Processing, Porto, Portugal 16 – 18 July 2023, pp. 141-148.

9. Морозов С.В., Шуткин В.Н., Гринченко А.И. Открытые сервисы верификации и валидации цифровых информационных моделей в архитектуре и строительстве //Материалы VIII Международной научно-практической конференции Информационное моделирование в задачах строительства и архитектуры (BIMAC 2025), 2025, СПб, стр. 135-144. DOI: 10.23968/BIMAC.2025.020.
10. Шуткин В.Н., Морозкин Н.К., Семенов В.А., Тарлапан О.А. Оптимизации генерации иерархических уровней детализации для масштабных полигональных сцен //Труды Института системного программирования РАН. – 2025. – Т. 37. – №. 3. – С. 311-324.
11. 11. Шуткин В.Н., Морозкин Н.К., Семенов В.А., Тарлапан О.А. Эффективная генерация иерархических уровней детализации: параллельная обработка, инкрементальные обновления, полигональные упрощения на основе видимости //Труды конференции ГрафиКон-2025, 2025, Йошкар-Ола, Россия, стр. 261-275, DOI: 10.25686/978-5-8158-2474-4-2025-261-275.
12. Семенов В.А., Аришин С.В., Гринченко А.И., Морозкин Н.К., Тарлапан О.А., Шуткин В.Н., «Сервис управления замечаниями в строительстве», Российская Федерация, Свидетельство о государственной регистрации программы для ЭВМ № 2025668968, дата поступления заявки 18 июня 2025 г., дата государственной регистрации в Реестре программ для ЭВМ 21 июля 2025 г.
13. 3D Tiles Standard | Open Geospatial Consortium Publications. URL: https://www.ogc.org/standards/3dtiles/ (дата обращения: 20.06.2025).
14. ГОСТ Р 10.0.02-2019/ИСО 16739-1:2018 Система стандартов информационного моделирования зданий и сооружений. Отраслевые базовые классы (IFC) для обмена и управления данными об объектах строительства.
15. Национальная ТИМ Платформа. URL: https://bim.ispras.ru/ (дата обращения 27.11.2025).
16. Семенов В.А., Аришин С.В., Тарлапан О.А. Верификация и валидация информационных моделей на основе стандарта IFC в сложных проектах //BIM-моделирование в задачах строительства и архитектуры: материалы VI Международной научно-практической конференции / под общ. ред. А. А. Семенова. — Санкт-Петербургский государственный архитектурно-строительный университет, Санкт-Петербург: 2023. — С. 187–195. — DOI: 10.23968/BIMAC.2023.026.
17. Семенов В.А., Золотов В.А., Рогачев И.В. Национальная технологическая платформа информационного моделирования. Концепция управления данными и документами //BIM-моделирование в задачах строительства и архитектуры: материалы VI

Международной научно-практической конференции / под общ. ред. А. А. Семенова. — Санкт-Петербургский государственный архитектурно-строительный университет, Санкт-Петербург: 2023. — С. 120–126. — DOI: 10.23968/BIMAC.2023.017.

18. Семенов В.А., Морозов С.В., Шерстенников И.А. Методологические основы перехода к машиночитаемым стандартам в строительстве //Информационное моделирование в задачах строительства и архитектуры: материалы VII Международной научно-практической конференции / под общ. ред. А. А. Семенова. — Санкт-Петербургский государственный архитектурно-строительный университет, Санкт-Петербург: 2024. — С. 153–160. — DOI: 10.23968/BIMAC.2024.021.
19. Семенов Г.В., Гринченко А.И., Морозкин Н.К. Перспективные сценарии управления требованиями и замечаниями в строительных проектах на основе стандартов IFC и BCF //BIM-моделирование в задачах архитектуры и строительства. Материалы VI Международной научно-практической конференции по технологиям информационного моделирования в архитектуре и строительстве BIMAC 2023, Санкт-Петербург, 2023, с. 127-135.
20. Aliaga D. et al. MMR: An interactive massive model rendering system using geometric and image-based acceleration //Proceedings of the 1999 symposium on Interactive 3D graphics. – 1999. – С. 199-206.
21. Aspert N., Santa-Cruz D., Ebrahimi T. Mesh: Measuring errors between surfaces using the hausdorff distance //Proceedings. IEEE international conference on multimedia and expo. – IEEE, 2002. – Т. 1. – С. 705-708.
22. Assarsson U., Moller T. Optimized view frustum culling algorithms for bounding boxes //Journal of graphics tools. – 2000. – Т. 5. – №. 1. – С. 9-22.
23. Balázs Á., Guthe M., Klein R. Fat borders: gap filling for efficient view-dependent LOD NURBS rendering //Computers & Graphics. – 2004. – Т. 28. – №. 1. – С. 79-85.
24. BCF Standard | BIM Collaboration Format – buildingSMART International. URL: https://www.buildingsmart.org/standards/bsi-standards/bim-collaboration-format/ (дата обращения: 27.11.2025).
25. Caplan P. C. Tessellation and interactive visualization of four-dimensional spacetime geometries //Computer-Aided Design. – 2025. – Т. 178. – С. 103792.
26. Chen J. et al. Optimally redundant, seek-time minimizing data layout for interactive rendering //The Visual Computer. – 2017. – Т. 33. – С. 139-149
27. Chhugani J. et al. vLOD: High-fidelity walkthrough of large virtual environments //IEEE Transactions on Visualization and Computer Graphics. – 2005. – Т. 11. – №. 1. – С. 35-47.

28. Ciampalini A., Cignoni P., Montani C. & Scopigno R. Multiresolution decimation based on global error //The Visual Computer 13(5), 1997, pp. 228–246, DOI:10.1007/s003710050101.
29. Cignoni P., Rocchini C., Scopigno R. Metro: measuring error on simplified surfaces //Computer graphics forum. – Oxford, UK and Boston, USA : Blackwell Publishers, 1998. – Т. 17. – №. 2. – С. 167-174.
30. CityGML стандарт | Публикации Open Geospatial Consortium. URL: https://www.ogc.org/ru/standards/citygml/ (дата обращения: 25.06.2025).
31. Clark J. H. Hierarchical geometric models for visible surface algorithms //Communications of the ACM. – 1976. – Т. 19. – №. 10. – С. 547-554.
32. Cohen J., Varshney A., Manocha D., Turk G., Weber H., Agarwal P., Brooks F. & Wright W. Simplification envelopes //Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH '96), 1996, Association for Computing Machinery, New York, NY, USA, pp. 119–128, DOI:10.1145/237170.237220.
33. Cohen J., Olano M. & Manocha D. Appearance-preserving simplification //Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98), 1998, Association for Computing Machinery, New York, NY, USA, pp. 115–122, DOI:10.1145/280814.280832.
34. Continuous level of detail mesh library [Электронный ресурс] URL: [https://github.com/nvpro-samples/nv_cluster_lod_builder] (дата обращения: 28.10.2025).
35. Correa W., Klosowski J. & Silva C. iWalk: Interactive Out-of-Core Rendering of Large Models //Technical Report TR-653-02, 2002, Princeton University, USA.
36. Corrêa W. T., Klosowski J. T., Silva C. T. Visibility-based prefetching for interactive out-of-core rendering //IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003. – IEEE, 2003. – С. 1-8.
37. Cozzi P.J. Visibility Driven Out-of-Core HLOD Rendering //Master’s thesis, 2008, University of Pennsylvania, Philadelphia, USA.
38. Dey T.K., Edelsbrunner H., Guha S. & Nekhayev D. Topology preserving edge contraction //Publications de l'Institut Mathematique 66(80), 1999, pp. 23–45.
39. El-Sana J. & Varshney A. Generalized View-Dependent Simplification //Comput. Graph. Forum, 18(3), 1999, pp. 83–94, DOI:10.1111/1467-8659.00330.
40. Erikson C., Manocha D. & Baxter W.V. HLODs for faster display of large static and dynamic environments //Proceedings of the 2001 symposium on Interactive 3D graphics (I3D '01), 2001, Association for Computing Machinery, New York, NY, USA, pp. 111–120, DOI:10.1145/364338.364376.

41. Everitt C. Interactive order-independent transparency //White paper, nVIDIA. – 2001. – Т. 2. – №. 6. – С. 7.
42. Funkhouser T. A., Séquin C. H. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments //Proceedings of the 20th annual conference on Computer graphics and interactive techniques. – 1993. – С. 247-254.
43. Garland M. & Heckbert P.S. Surface simplification using quadric error metrics //Proceedings of the ACM SIGGRAPH Conference on Computer Graphics, 1997, pp. 209–216, DOI:10.1145/258734.258849.
44. Garland M, & Heckbert P.S. Simplifying surfaces with color and texture using quadric error metrics //Proceedings of the conference on Visualization '98 (VIS '98), 1998, IEEE Computer Society Press, Washington, DC, USA, pp. 263–269.
45. Garland M., Willmott A. & Heckbert P.S. Hierarchical face clustering on polygonal surfaces //Proceedings of the 2001 symposium on Interactive 3D graphics (I3D '01), 2001, Association for Computing Machinery, New York, NY, USA, pp. 49–58, DOI:10.1145/364338.364345.
46. Garland M. & Zhou Y. Quadric-based simplification in any dimension //ACM Trans. Graph. 24(2), April 2005, pp. 209–239, DOI:10.1145/1061347.1061350.
47. Ge Y. et al. A Novel LOD Rendering Method with Multi-level Structure Keeping Mesh Simplification and Fast Texture Alignment for Realistic 3D Models //IEEE Transactions on Geoscience and Remote Sensing. – 2024.
48. Ghazanfarpour A. et al. Proximity-aware multiple meshes decimation using quadric error metric //Graphical Models. – 2020. – Т. 109. – С. 101062.
49. Gieng T., Hamann B., Joy K., Schussman G. & Trotts I. Constructing Hierarchies for Triangle Meshes. //IEEE Trans. Vis. Comput. Graph. 4(2), 1998, pp. 145–161, DOI:10.1109/2945.694956.
50. Gonakhchyan V., Tarlapan O. & Semenov V. Generating Dynamic 3D Scenes for Rendering Benchmarks //Proceedings of the International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2019 (CGVCVIP 2019), 2019, pp. 485–488. DOI:10.33965/cgv2019_201906P076.
51. Guéziec A., Taubin G., Horn B. & Lazarus F. A framework for streaming geometry in VRML //IEEE Computer Graphics and Applications 19(2), 1999, pp. 68–78, DOI:10.1109/38.749125.
52. Guthe M., Borodin P., Klein R. Real-time out-of-core rendering //International Journal of Image and Graphics. – 2005.
53. Guthe M. et al. Real-time appearance preserving out-of-core rendering with shadows //Rendering Techniques. – 2004. – Т. 2004. – С. 69-79.

54. Hamann B. A data reduction scheme for triangulated surfaces //Computer Aided Geometric Design 11(2), 1994, pp. 197–214, DOI:10.1016/0167-8396(94)90032-9.
55. Hasselgren J. et al. Appearance-Driven Automatic 3D Model Simplification //EGSR (DL). – 2021. – С. 85-97.
56. Haydel J., Yuksel C., Seiler L. Locally-adaptive level-of-detail for hardware-accelerated ray tracing //ACM Transactions on Graphics (TOG). – 2023. – Т. 42. – №. 6. – С. 1-15.
57. Heckbert P., Garland M. Multiresolution modeling for fast rendering //Graphics Interface. – Canadian Information Processing Society, 1994. – С. 43-43.
58. Hinker P. & Hansen C. Geometric optimization //Proceedings of the 4th conference on Visualization '93 (VIS '93), 1993, IEEE Computer Society, USA, pp. 189–195.
59. Hoppe H., DeRose T., Duchamp T., McDonald J. & Stuetzle W. Mesh optimization //Proceedings of the 20th annual conference on Computer graphics and interactive techniques (SIGGRAPH '93), 1993, Association for Computing Machinery, New York, NY, USA, pp. 19–26, DOI:10.1145/166117.166119.
60. Hoppe H. Progressive meshes //Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH '96), 1996, Association for Computing Machinery, New York, NY, USA, pp. 99–108, DOI:10.1145/237170.237216.
61. Hoppe H. View-dependent refinement of progressive meshes //Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97), 1997, ACM Press/Addison-Wesley Publishing Co., USA, pp. 189–198, DOI:10.1145/258734.258843.
62. Hoppe H. Smooth view-dependent level-of-detail control and its application to terrain rendering //Proceedings of the conference on Visualization '98 (VIS '98), 1998, IEEE Computer Society Press, Washington, DC, USA, pp. 35–42.
63. IDS Standard | Information Delivery Specification – buildingSMART International. URL: https://www.buildingsmart.org/standards/bsi-standards/information-delivery-specification-ids/ (дата обращения: 27.11.2025).
64. Kalvin A. & Taylor R.H. Superfaces: Polygonal mesh simplification with bounded error //IEEE Computer Graphics and Applications 16(3), 1996, pp. 64–77, DOI:10.1109/38.491187.
65. Kerbl B. et al. A hierarchical 3d gaussian representation for real-time rendering of very large datasets //ACM Transactions on Graphics (TOG). – 2024. – Т. 43. – №. 4. – С. 1-15.
66. Kircher S., Garland M. Progressive multiresolution meshes for deforming surfaces //Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation. – 2005. – С. 191-200.
67. Laine S., Karras T. Two Methods for Fast Ray-Cast Ambient Occlusion //Computer Graphics Forum. – Oxford, UK : Blackwell Publishing Ltd, 2010. – Т. 29. – №. 4. – С. 1325-1333.

68. Lakhia A. Efficient interactive rendering of detailed models with hierarchical levels of detail //Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. – IEEE, 2004. – С. 275-282.
69. Lee C. H., Varshney A., Jacobs D. W. Mesh saliency //ACM SIGGRAPH 2005 Papers. – 2005. – С. 659-666.
70. Levenberg J. Fast view-dependent level-of-detail rendering using cached geometry //IEEE Visualization, 2002. VIS 2002. – IEEE, 2002. – С. 259-265.
71. Li M., Nan L. Feature-preserving 3D mesh simplification for urban buildings //ISPRS Journal of Photogrammetry and Remote Sensing. – 2021. – Т. 173. – С. 135-150.
72. Liang J. et al. InfNeRF: Towards Infinite Scale NeRF Rendering with O (log n) Space Complexity //SIGGRAPH Asia 2024 Conference Papers. – 2024. – С. 1-11.
73. Liang Y., He F., Zeng X. 3D mesh simplification with feature preservation based on whale optimization algorithm and differential evolution //Integrated Computer-Aided Engineering. – 2020. – Т. 27. – №. 4. – С. 417-435.
74. Lin W. et al. Visual saliency and quality evaluation for 3D point clouds and meshes: An overview //APSIPA Transactions on Signal and Information Processing. – 2022. – Т. 11. – №. 1.
75. Lindstrom P. Out-of-core simplification of large polygonal models //Proceedings of the 27th annual conference on Computer graphics and interactive techniques. – 2000. – С. 259-262.
76. Lindstrom P. & Turk G. Image-driven simplification //ACM Trans. Graph. 19(3), July 2000, pp. 204–241, DOI:10.1145/353981.353995.
77. Lodi A., Martello S., Monaci M. Two-dimensional packing problems: A survey //European journal of operational research. – 2002. – Т. 141. – №. 2. – С. 241-252.
78. Loubet G., Neyret F. Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets //Computer Graphics Forum. – 2017. – Т. 36. – №. 2. – С. 431-442.
79. Low K.-L. & Tan T.-S. Model simplification using vertex-clustering //Proceedings of the 1997 symposium on Interactive 3D graphics (I3D '97), 1997, Association for Computing Machinery, New York, NY, USA, pp. 75–ff, DOI:10.1145/253284.253310.
80. Luebke D. & Erikson C. View-dependent simplification of arbitrary polygonal environments //Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97), 1997, ACM Press/Addison-Wesley Publishing Co., USA, pp. 199–208, DOI:10.1145/258734.258847.
81. Luebke D. et al. Level of detail for 3D graphics. – Elsevier, 2002.
82. Mammen A. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique //IEEE Computer graphics and Applications. – 1989. – Т. 9. – №. 4. – С. 43-55.

83. Martínez Bayona J. Space-optimized texture atlases : дис. – Universitat Politècnica de Catalunya, 2009.
84. McGuire M. and Bavoil L., 2013. Weighted Blended Order-Independent Transparency //Journal of Computer Graphics Techniques (JCGT), Vol. 2, No. 2, pp. 122-141.
85. Melero F. J., Cano P., Torres J. C. Bounding-planes Octree: A new volume-based LOD scheme //Computers & Graphics. – 2008. – Т. 32. – №. 4. – С. 385-392.
86. Michaud C., Mellado N., Paulin M. Mesh simplification with curvature error metric //Eurographics 2017. – 2017.
87. Morozov S. et al. Indexing of Hierarchically Organized Spatial-Temporal Data Using Dynamic Regular Octrees //Perspectives of System Informatics. Lecture Notes in Computer Science vol. 10742, 2018, pp. 276–290.
88. Nanite: A Deep Dive — презентация с конференции Siggraph 2021 [Электронный ресурс] URL: [https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf] (дата обращения: 28.10.2025).
89. Pantazopoulos I., Tzafestas S. Occlusion culling algorithms: A comprehensive survey //Journal of Intelligent and Robotic Systems. – 2002. – Т. 35. – С. 123-156.
90. Pečnik S., Žalik B. Real-time visualization using GPU-accelerated filtering of lidar data //World Acad. Sci., Eng. Technol., Int. J. Comput., Elect., Automat., Control Inf. Eng. – 2014. – Т. 8. – №. 12. – С. 2108-2112.
91. Peng C., Cao Y. A GPU-based approach for massive model rendering with frame-to-frame coherence //Computer Graphics Forum. – Oxford, UK : Blackwell Publishing Ltd, 2012. – Т. 31. – №. 2pt2. – С. 393-402.
92. Peng C., Cao Y. Parallel LOD for CAD model rendering with effective GPU memory usage //Computer-Aided Design and Applications. – 2016. – Т. 13. – №. 2. – С. 173-183.
93. Peng C. Real-time Visualization of Massive 3D Models on GPU Parallel Architectures : дис. – Virginia Polytechnic Institute and State University, 2013.
94. Popović J. & Hoppe H. Progressive simplicial complexes // Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97), 1997, ACM Press/Addison-Wesley Publishing Co., USA, pp. 217–224, DOI:10.1145/258734.258852.
95. Ponchio F., Hormann K. Interactive rendering of dynamic geometry //IEEE Transactions on Visualization and Computer Graphics. – 2008. – Т. 14. – №. 4. – С. 914-925.
96. Ramos F., Ripolles O., Chover M. Continuous level of detail for large scale rendering of 3d animated polygonal models //International Conference on Articulated Motion and Deformable Objects. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – С. 194-203.

97. Rossignac J. & Borrel P. Multi-resolution 3D approximation for rendering complex scenes //Geometric Modeling in Computer Graphics, 1993, Springer Verlag, Genova, Italy, pp. 455-465, DOI:10.1007/978-3-642-78114-8_29.

98. Ryu J., Kamata S. An efficient computational algorithm for Hausdorff distance based on points-ruling-out and systematic random sampling //Pattern Recognition. – 2021. – Т. 114. – С. 107857.

99. Sahm J. A client-server-scenegraph for the visualization of large and dynamic 3d scenes //Journal of WSCG, Vol.12, No.1-3, ISSN 1213-6972, WSCG2004, February 2-6, 2004, Plzen, Czech Republic.

100. Sahm J., Soetebier I., Birthelmer H. Efficient representation and streaming of 3D scenes //Computers & Graphics. – 2004. – Т. 28. – №. 1. – С. 15-24.

101. Sajadi B. et al. A novel page-based data structure for interactive walkthroughs //Proceedings of the 2009 symposium on Interactive 3D graphics and games. – 2009. – С. 23-29.

102. Salinas D., Lafarge F., Alliez P. Structure-aware mesh decimation //Computer Graphics Forum. – 2015. – Т. 34. – №. 6. – С. 211-227

103. Samet H. Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., San Francisco, 2006, 1024 p.

104. Sander P. V., Mitchell J. L. Out-of-core rendering of large meshes with progressive buffers //ACM SIGGRAPH 2006: Proceedings of the conference on SIGGRAPH 2006 course notes. – 2006. – С. 1-18.

105. Schroeder W.J., Zarge J.A. & Lorensen W.E. Decimation of triangle meshes //Proceedings of the 19th annual conference on Computer graphics and interactive techniques (SIGGRAPH '92), 1992, Association for Computing Machinery, New York, NY, USA, pp. 65–70. DOI:10.1145/133994.134010.

106. Schroeder W.J. A topology modifying progressive decimation algorithm //Proceedings of the 8th conference on Visualization '97 (VIS '97), 1997, IEEE Computer Society Press, Washington, DC, USA, pp. 205–ff.

107. Semenov V., Anichkin A., Morozov S., Tarlapan O. & Zolotov V., Effective Project Scheduling Under Workspace Congestion and Workflow Disturbance Factors. //Australasian Journal of Construction Economics and Building Conference Series (ISSN: 2200-7679), Vol. 2, No. 1, 2014, pp. 35-50.

108. Semenov V., Anichkin A., Morozov S., Tarlapan O. & Zolotov V. Effective project scheduling under workspace congestion and workflow disturbance factors. //Proceedings of the 13th International Conference on Construction Applications of Virtual Reality, 30-31 October 2013, London, UK (ISBN: 978-0-9927161-0-3); pp. 239-252.

109. Semenov V.A., Kazakov K.A., Morozov S.V., Tarlapan O.A., Zolotov V.A., Dengenis T. 4D modeling of large industrial projects using spatio-temporal decomposition. //eWork and eBusiness in Architecture, Engineering and Construction, eds. K. Menzel and R. Scherer, CRC Press, Taylor & Francis Group,London, UK, 2010, pp. 89-95.

110. Semenov V.A., Kazakov K.A., Zolotov V.A., Dengenis T. Virtual Construction: 4D Planning and Validation. // Proceedings of the XI International Conference on Construction Applications of Virtual Reality, 3-4 November, 2011, Weimar, Germany, p.135-142.

111. Semenov V.A., Kazakov K.A., Zolotov V.A., Jones H., Jones S. Combined strategy for efficient collision detection in 4D planning applications. //Computing in Civil and Building Engineering, Proceedings of the International Conference, W. Tizani (Editor), 30 June-2 July, 2010, Nottingham, UK, Nottingham University Press, p. 31-39, ISBN 978-1-907284-60-1.

112. Shamir A., Pascucci V., Bajaj C. Multi-resolution dynamic meshes with arbitrary deformations //Proceedings Visualization 2000. VIS 2000 (Cat. No. 00CH37145). – IEEE, 2000. – С. 423-430.

113. Shamir A., Pascucci V. Temporal and spatial level of details for dynamic meshes //Proceedings of the ACM symposium on Virtual reality software and technology. – 2001. – С. 77-84.

114. Song R. et al. Mesh saliency via spectral processing //ACM Transactions On Graphics (TOG). – 2014. – Т. 33. – №. 1. – С. 1-17.

115. SYNCHRO: Digital Construction Delivery Software. URL: https://www.bentley.com/software/synchro/ (дата обращения 27.11.2025).

116. Tang M., Lee M., Kim Y. J. Interactive Hausdorff distance computation for general polygonal models //ACM Transactions on Graphics (TOG). – 2009. – Т. 28. – №. 3. – С. 1-9.

117. Toledo L., De Gyves O., Rudomín I. Hierarchical level of detail for varied animated crowds //The Visual Computer. – 2014. – Т. 30. – С. 949-961.

118. Varadhan G. & Manocha D. Out-of-core rendering of massive geometric environments // Proceedings of the conference on Visualization '02 (VIS '02), 2002, IEEE Computer Society, USA, pp. 69–76.

119. Verdie Y., Lafarge F., Alliez P. LOD generation for urban scenes //ACM Transactions on Graphics. – 2015. – Т. 34. – №. 3. – С. 30.

120. Wald I., Dietrich A., Slusallek P. An interactive out-of-core rendering framework for visualizing massively complex models //ACM SIGGRAPH 2005 Courses. – 2005. – С. 17-es.

121. Wand M., Straßer W. Multi-resolution rendering of complex animated scenes //Computer Graphics Forum. – Oxford, UK : Blackwell Publishing, Inc, 2002. – Т. 21. – №. 3. – С. 483-491.

122. Wang R. et al. GPU-based out-of-core many-lights rendering //ACM Transactions on Graphics (TOG). – 2013. – Т. 32. – №. 6. – С. 1-10.

123. Westover L. Interactive volume rendering //Proceedings of the 1989 Chapel Hill workshop on Volume visualization. – 1989. – С. 9-16.

124. Xia J.C. & Varshney A. Dynamic view-dependent simplification for polygonal models //Proceedings of the 7th conference on Visualization '96 (VIS '96), 1996, IEEE Computer Society Press, Washington, DC, USA, pp. 327–ff.

125. Xie J. et al. Automatic simplification and visualization of 3D urban building models //International Journal of Applied Earth Observation and Geoinformation. – 2012. – Т. 18. – С. 222-231.

126. Xu D. & Tian Y. A Comprehensive Survey of Clustering Algorithms //Annals of Data Science vol. 2, 2015. pp. 165–193.

127. Yoon S. E. et al. Quick-vdr: Interactive view-dependent rendering of massive models //ACM SIGGRAPH 2004 Sketches. – 2004. – С. 22.

128. Zhang H., Cao L., Peng C. UltraMeshRenderer: Efficient Structure and Management of GPU Out-of-core Memory for Real-time Rendering of Gigantic 3D Meshes //ACM Transactions on Graphics (TOG). – 2025. – Т. 44. – №. 4. – С. 1-19.

129. Zhang L. et al. Web-based visualization of large 3D urban building models //International Journal of Digital Earth. – 2014. – Т. 7. – №. 1. – С. 53-67.

130. Zhang X. et al. Size-Adaptive Texture Atlas Generation and Remapping for 3D Urban Building Models //ISPRS International Journal of Geo-Information. – 2021. – Т. 10. – №. 12. – С. 798.

131. Zhao T., Jiang J., Guo X. A Novel Quadratic Error Metric Mesh Simplification Algorithm For 3D Building Models Based On ‘Local-Vertex’ Texture Features //The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. – 2022. – Т. 48. – С. 109-115

132. Zheng Z., Prakash E., Chan T. Interactive view-dependent rendering over networks //IEEE transactions on visualization and computer graphics. – 2008. – Т. 14. – №. 3. – С. 576-589.

133. Zhou S. et al. A hybrid level-of-detail representation for large-scale urban scenes rendering //Computer Animation and Virtual Worlds. – 2014. – Т. 25. – №. 3-4. – С. 243-253

134. Zhou Y. et al. Efficient Scene Appearance Aggregation for Level-of-Detail Rendering //arXiv preprint arXiv:2409.03761. – 2024.

135. Zhou Z., Chen K., Zhang J. Efficient 3-D scene prefetching from learning user access patterns //IEEE Transactions on Multimedia. – 2015. – Т. 17. – №. 7. – С. 1081-1095.

Приложение А. Интерфейсы программных библиотек

А.1 Интерфейс библиотеки генерации HDLOD

```
typedef enum HDLODTextureMagFilter {
        MAG_NEAREST = 0,
        MAG_LINEAR
} HDLODTextureMagFilter;

typedef enum HDLODTextureMinFilter {
        MIN_NEAREST = 0,
        MIN_LINEAR,
        MIN_NEAREST_MIPMAP_NEAREST,
        MIN_LINEAR_MIPMAP_NEAREST,
        MIN_NEAREST_MIPMAP_LINEAR,
        MIN_LINEAR_MIPMAP_LINEAR
} HDLODTextureMinFilter;

typedef enum HDLODTextureWrap {
        CLAMP_TO_EDGE = 0,
        MIRRORED_REPEAT,
        REPEAT
} HDLODTextureWrap;

typedef struct HDLODImage {
        int width;
        int height;
        int comp;
        const unsigned char* bytes;
} HDLODImage;

typedef struct HDLODTexture {
        long imageID;
        const char* imagePath;
```

```
        HDLODTextureMagFilter magFilter;
        HDLODTextureMinFilter minFilter;
        HDLODTextureWrap wrapS;
        HDLODTextureWrap wrapT;
    } HDLODTexture;

    typedef struct HDLODMesh {
        int numVertices;
        int numIndices;
        const double* points;
        const float* normals;
        const float* texCoords;
        const unsigned char* colors;
        const unsigned int* indices;
        long textureID;
        long objectID;
    } HDLODMesh;

    int HDLODGeneratorInit(const char* directory);
    long HDLODGeneratorCreateImage(const HDLODImage* image);
    long HDLODGeneratorCreateTexture(const HDLODTexture* texture);
    long HDLODGeneratorCreateObject(const char* globalID);
    int HDLODGeneratorCreateMesh(const HDLODMesh* mesh);
    int HDLODGeneratorSetPresenceFunction(long objectID, int size, const time_t* timePoints,
const int* values);
    int HDLODGeneratorSetRelativeError(double error);
    int HDLODGeneratorSetNumLevels(int numLevels);
    int HDLODGeneratorSetTextureAtlasSize(int pixels);
    int HDLODGeneratorSetRemoveInteriorFaces(int enabled);
    int HDLODGeneratorSetColorClusters(int enabled);
    int HDLODGeneratorSetExportGLTF(int enabled);
    int HDLODGeneratorGenerate();
    void HDLODGeneratorCleanup();
```

A.2 Интерфейс библиотеки рендеринга HDLOD

```
typedef enum HDLODRenderMode {
        HDLODRENDERMODE_CONSERVATIVE = 0,
        HDLODRENDERMODE_INTERACTIVE,
        HDLODRENDERMODE_INMEMORY,
        HDLODRENDERMODE_NOLOD
} HDLODRenderMode;

int HDLODRendererInit(const char* filePath, int width, int height, HDLODRenderMode mode,
unsigned long long memoryLimit);
int HDLODRendererRender();
int HDLODRendererBenchmark(const char* directory);
void HDLODRendererCleanup();
```

A.3 Интерфейс библиотеки OpenGL рендеринга

```
typedef enum GLTextureMagFilter {
        MAG_NEAREST = 0,
        MAG_LINEAR
} GLTextureMagFilter;



typedef enum GLTextureMinFilter {
        MIN_NEAREST = 0,
        MIN_LINEAR,
        MIN_NEAREST_MIPMAP_NEAREST,
        MIN_LINEAR_MIPMAP_NEAREST,
        MIN_NEAREST_MIPMAP_LINEAR,
        MIN_LINEAR_MIPMAP_LINEAR
} GLTextureMinFilter;
```

```
typedef enum GLTextureWrap {
        CLAMP_TO_EDGE = 0,
        MIRRORED_REPEAT,
        REPEAT
} GLTextureWrap;

typedef struct GLTexture {
        int width;
        int height;
        int comp;
        const unsigned char* bytes;
        GLTextureMagFilter magFilter;
        GLTextureMinFilter minFilter;
        GLTextureWrap wrapS;
        GLTextureWrap wrapT;
} GLTexture;

typedef struct GLMesh {
        int numVertices;
        int numIndices;
        const float* positions;
        const float* normals;
        const float* texCoords;
        const unsigned char* colors;
        const unsigned int* indices;
        long textureID;
} GLMesh;

int GLRendererInit(int width, int height);
long GLRendererCreateTexture(const GLTexture* texture);
long GLRendererCreateMesh(const GLMesh* mesh);
int GLRendererDestroyTexture(long textureID);
int GLRendererDestroyMesh(long meshID);
int GLRendererFocusCameraOnBounds(float minx, float miny, float minz, float maxx, float
maxy, float maxz, float dirx, float diry, float dirz);
```

int GLRendererStartRendering();

int GLRendererRenderMesh(long meshID);

int GLRendererFinishRendering();

double GLRendererGetFrameTime();

long long GLRendererGetNumRenderedTriangles();

long long GLRendererGetNumRenderedMeshes();

void GLRendererGetCameraPosition(float x, float* y, float* z);*

void GLRendererGetCameraForward(float x, float* y, float* z);*

void GLRendererGetCameraUp(float x, float* y, float* z);*

void GLRendererGetCameraRight(float x, float* y, float* z);*

int GLRendererGetCameraWidth();

int GLRendererGetCameraHeight();

float GLRendererGetCameraFOV();

float GLRendererGetCameraNear();

float GLRendererGetCameraFar();

void GLRendererSetCameraPosition(float x, float y, float z);

int GLRendererSetCameraOrientation(float fwdx, float fwdy, float fwdz, float upx, float upy, float upz);

int GLRendererIntersectBoundsFrustum(float minx, float miny, float minz, float maxx, float maxy, float maxz);

unsigned long long GLRendererEstimateMeshVmemUsage(int numVertices, int numIndices);

unsigned long long GLRendererEstimateTextureVmemUsage(int width, int height, int comp);

void GLRendererDestroy();

A.4 Интерфейс библиотеки поиска внешних граней полигональной сетки

int ExteriorFinderInit(int quality);

int ExteriorFinderProcessMesh(const float positions, int numIndices, const unsigned int* indices, int* numResultInidices, unsigned int* resultIndices);*

void ExteriorFinderCleanup();

A.5 Интерфейсы библиотек импорта

int HDLODFrom3D(const char filePath);*

int HDLODFromCityGMLImportFile(const char filePath, unsigned int lod);*

int HDLODFromIFCImportFile(const char filePath);*