Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Волгоградский государственный технический университет»

На правах рукописи 

Шакаев Вячеслав Дмитриевич

Моделирование воксельных ландшафтов для автоматизации проектирования систем виртуальной реальности

Специальность 05.13.12 – «Системы автоматизации проектирования»
(информационные технологии и промышленность)

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук

Научный руководитель:
д.т.н., доцент
Кравец Алла Григорьевна

Волгоград – 2019

Оглавление

Введение

Актуальность темы исследования. В различных системах САПР для геометрического моделирования, конструктивной твердотельной геометрии (CSG), для создания цифровых скульптур (скульптурное моделирование или 3D скульптинг) и синтеза виртуальной реальности (ВР) широко используется объёмное или воксельное представление трёхмерных объектов. Реалистичная визуализация ландшафтов требуется для проектирования геоинформационных систем, систем виртуальной реальности, военных и аэрокосмических тренажёров, симуляторов наземного транспорта, архитектурных и ландшафтных редакторов, видеоигр с большими открытыми пространствами.

В настоящее время воксельные данные рассматриваются как одно из перспективных представлений трёхмерных ландшафтов. Воксельные ландшафты в силу ряда преимуществ: возможность моделировать пещеры и «нависающие» части, внутреннюю геологическую структуру, расширенные возможности процедурной генерации, редактируемость всё чаще применяются вместо традиционных ландшафтов на картах высот. В последние годы в связи с ростом мощности компьютеров появилась возможность использовать воксельные данные для проектирования искусственных элементов ландшафта, таких как здания и крупные технические объекты. Результатом может быть комплексное решение, сочетающее в себе представление данных о рельефе ландшафта и информацию для реконструкции острых рёбер и конических вершин поверхности, необходимую для проектирования искусственных объектов.

В настоящий момент моделирование и визуализацию воксельных ландшафтов с острыми углами и рёбрами осуществляют несколько закрытых коммерческих пакетов, однако детальное описание используемых алгоритмов отсутствует в открытых источниках и недоступно для академического сообщества. Поэтому исследование и разработка методов моделирования и интерактивной визуализации воксельных ландшафтов с наличием острых рёбер и углов, необходимых для проектирования искусственных объектов в системах ВР, является актуальной научно-технической задачей.

Степень разработанности темы. Работы, связанные с интерактивной визуализацией больших изоповерхностей и ландшафтов растеризацией треугольных сеток проводились Толоком А.В., Авербухом В.Л., Бахтеревым М.О., Захаровой А.А., Жирковым А.О., Казаковым М.В., Манаковым Д.В., Пасько А.А., Скворцовым А.В., Bender J., Dachsbacher C., Forstmann S., Lengyel E., Löffler F., Müller A., Schumann H., Scholz M. и др. В вышерассмотренных работах представлены

методы сжатия, хранения и триангуляции скалярных данных, стратегии генерации, выбора и подгрузки уровней детализации, способы бесшовной «стыковки» блоков с различными уровнями детализации для «гладких», не содержащих острых рёбер и углов, изоповерхностей.

Задачей построения полигональной сетки с реконструкцией острых рёбер и конических вершин изоповерхности занимались такие учёные как Пасько А.А., Фрязинов О.В., Казаков М.В., Лебедев А.С., Чернышенко А.Ю., Данилов А.А., Сковпень А.В, а также зарубежные учёные Kobbelt L., Botsch M., Warren J., Schaefer S., Ju T., Gress A., Klein R., Lewiner T., Schmitz L., Kaufman A., Wang C., Denis F., Dupont. F., Wenger R. и др.

По упрощению полигональных сеток наиболее известны работы Акаева А.А., Hoppe H., Garland M., Heckbert P., Lindstrom P., Rossignac J., Borrel P., Cohen J., Luebke D., Cignoni P., Scopigno R., Schroeder W., Lorenson W. и др.

Цель диссертационного исследования состоит в совершенствовании процесса проектирования систем виртуальной реальности за счёт расширения функциональных возможностей САПР и повышения эффективности процессов моделирования и визуализации воксельных ландшафтов (увеличении производительности и снижении объёма потребляемой памяти). Для достижения поставленной цели выделены следующие **задачи**:

1) провести анализ существующих технологий проектирования систем ВР, включающих создание, моделирование и визуализацию воксельных ландшафтов;

2) разработать алгоритм триангуляции для визуализации воксельных ландшафтов с наличием острых углов и рёбер в интерактивном режиме;

3) разработать способы представления, методы редактирования и компактного хранения воксельных данных для работы с ландшафтами в системах ВР;

4) разработать методы визуализации больших воксельных ландшафтов, состоящих из блоков с различными уровнями детализации;

5) спроектировать, реализовать и провести апробацию прототипа системы, реализующего разработанные способы и алгоритмы;

6) экспериментально оценить показатели производительности и объёма потребляемой памяти предложенных алгоритмов и методов для моделирования и визуализации воксельных ландшафтов при проектировании систем ВР.

Объектом исследования являются воксельные ландшафты.

Предметом исследования являются процессы моделирования и интерактивной визуализации воксельных ландшафтов при автоматизации проектирования систем ВР.

Методы исследования. В диссертационном исследовании использованы методы автоматизации проектирования, машинной графики и синтеза ВР, методы вычислительной геометрии, численные методы линейной алгебры, методы параллельных вычислений и теории сложности алгоритмов, правила построения программного обеспечения.

Научная новизна. Научная новизна результатов, выносимых на защиту, заключается в разработке совокупности алгоритмов и методов обработки и визуализации воксельных ландшафтов в интерактивном режиме для проектирования систем ВР, включающей, в отличие от существующих САПР:

1) новый алгоритм адаптивной триангуляции изоповерхностей методом дуальных контуров, который отличается созданием когерентных треугольных сеток и подходит для проектирования сверхбольших воксельных ландшафтов с внешней памятью (п.п.8);

2) новый метод для компактного хранения уровней детализации воксельного ландшафта, отличающийся поддержкой проектируемых областей с острыми углами и различными материалами, а также возможностью генерации упрощённых уровней детализации (п.п.8);

3) новый алгоритм для адаптации иерархии уровней детализации к позиции камеры наблюдателя, который имеет более низкую вычислительную сложность по сравнению с традиционными рекурсивными алгоритмами, применяемыми при проектировании систем ВР (п.п.3);

4) метод для бесшовной триангуляции воксельного ландшафта по частям, отличающийся высокой производительностью и низким потреблением памяти по сравнению с существующими методами проектирования систем ВР (п.п.3);

5) модифицированный алгоритм дуальных контуров, отличающийся полным устранением зависимостей по данным, присутствующих в исходном алгоритме (п.п.3).

Теоретическая и практическая значимость. Теоретическая значимость проведенных исследований заключается в разработке научных основ построения средств САПР для современного и нетривиального объекта проектирования – систем виртуальной реальности. В ходе исследования разработаны и исследованы алгоритмы и методы для синтеза проектных решений в области систем ВР и анализа этих решений на основе визуализации больших воксельных ландшафтов, созданы и усовершенствованы научные подходы к построению методов геометрического моделирования и синтеза виртуальной реальности.

Практическая ценность работы заключается в разработке программных инструментов, которые могут быть использованы для проектирования систем ВР с поддержкой создания, редактирования и визуализации воксельных ландшафтов.

Предложенный алгоритм триангуляции может быть использован для визуализации объёмных данных в CAD/CAM/CAE-системах, где требуется триангуляция изоповерхности с восстановлением ее острых углов и рёбер.

Разработанное программно-инструментальное средство целесообразно использовать для проектирования рельефа ландшафта и зданий в таких сферах как системы ВР, системы архитектурной визуализации и геоинформационные системы (ГИС), аэрокосмические симуляторы и тренажёры, «серьёзные» игры и видеоигры с большими открытыми пространствами и изменяемым окружением.

В сравнении со стандартным рекурсивным вариантом новый алгоритм триангуляции потребляет в 3-4 раза меньше памяти, генерирует когерентные треугольные сетки (со значением ACMR < 0.75) и лучше подходит для триангуляции ландшафтов с внешней памятью. Новый метод компактного хранения уровней детализации воксельного ландшафта обеспечивает сжатие данных поверхности более чем в 3 раза, что позволяет уменьшить занимаемое на диске пространство и повысить скорость загрузки данных при триангуляции с внешней памятью.

Основные положения, выносимые на защиту:

1) алгоритм адаптивной триангуляции изоповерхностей, который обеспечивает триангуляцию воксельных ландшафтов с внешней памятью;

2) метод компактного хранения уровней детализации воксельного ландшафта с информацией для восстановления его острых углов и рёбер, поддержкой проектируемых областей с различными материалами, возможностью генерации упрощённых уровней детализации, высокой скоростью сжатия, декомпрессии и триангуляции;

3) алгоритм для адаптации иерархии уровней детализации, позволяющий учитывать позицию камеры наблюдателя при проектировании системы ВР;

4) метод для бесшовной триангуляции воксельного ландшафта по частям, который обеспечивает возможность «стыковки» блоков с различными разрешениями, высокое качество восстановления острых углов и рёбер поверхности, минимальное количество занимаемого на диске пространства;

5) модифицированный алгоритм дуальных контуров, который позволяет выполнять интерактивную триангуляцию смежных блоков параллельно.

Соответствие паспорту научной специальности. Основная область исследования соответствует паспорту специальности 05.13.12 — «Системы автоматизации проектирования», а именно пунктам 3 — «Разработка научных основ построения средств САПР, разработка и исследование моделей, алгоритмов и методов для синтеза и анализа проектных решений, включая конструкторские и технологические решения САПР и АСТПП», 8 – «Разработка научных основ построения средств компьютерной графики, методов геометрического моделирования проектируемых объектов и синтеза виртуальной реальности».

Достоверность изложенных в работе результатов и практических рекомендаций обусловлена корректным применением указанных методов исследования и экспериментальным тестированием алгоритмов и программ.

Реализация и внедрение. Разработанные алгоритмы и методы моделирования воксельных ландшафтов приняты для использования в ООО «ГОЛДИ С» при проектировании сцен для тестирования системы визуализации объёмных моделей ландшафтов и зданий. Разработанное программное обеспечение внедрено в ООО «Автоматизированные наукоёмкие технологии Груп» и ООО «ЦМИТ «ЛЮКС», что позволило повысить эффективность процесса проектирования систем виртуальной реальности и качество создаваемых продуктов. Разработанная «Система моделирования и визуализации воксельных ландшафтов» используется в учебном процессе в ВолгГТУ при проведении лабораторных работ на кафедре САПР и ПК по дисциплинам «Геометрическое моделирование САПР» и «Мультимедийные технологии» направлений 09.04.01, 09.03.01 «Информатика вычислительная техника».

Апробация работы. Основные результаты диссертации представлялись на конференциях “CIT&DS Creativity in intelligent technologies & data science (CITDS)” (Volgograd, Russia, 2017), Графикон 2016 (г. Нижний Новгород), INFO-2016 (г. Сочи), а также докладывались на и обсуждались совместном семинаре ИСП РАН и ИПМ РАН им. Келдыша и на семинарах кафедры «САПРиПК».

Публикации. Основные положения изложены в 10 печатных работах, в том числе 4 статьи в научных журналах, рекомендуемых ВАК РФ и 2 статьи в изданиях, индексируемых в базе научного цитирования Scopus. Получено свидетельство о государственной регистрации программы для ЭВМ.

Личный вклад автора. Все результаты диссертационной работы получены автором самостоятельно. Подготовка к публикации полученных результатов проводилась совместно с

соавторами, причем вклад диссертанта был определяющим. Программная реализация всех алгоритмов и методов выполнены лично автором.

Структура и объем диссертации. Диссертационная работа состоит из введения, пяти глав, заключения, списка литературы и приложений. Основной текст диссертационной работы (без приложений и списка литературы) составляет 177 страниц машинописного текста, в том числе 91 рисунок и 6 таблиц. Список литературы включает 160 наименований на 14 страницах.

Во введении обоснована актуальность темы диссертации, формулируются цель и задачи исследования, изложены научная новизна и практическая ценность полученных результатов, приведены основные положения, выносимые на защиту, описывается структура и излагается краткое содержание глав диссертации.

В первой главе анализируется современное состояние исследований в области интерактивной визуализации воксельных ландшафтов в системах ВР, рассматриваются существующие системы визуализации и моделирования ландшафтов, их возможности. Рассмотрены интерактивные ячеечные методы триангуляции, способные реконструировать острые углы и рёбра изоповерхности.

Во второй главе разработан новый адаптивный алгоритм триангуляции изоповерхностей методом дуальных контуров на основе линейных октодеревьев.

В третьей главе рассмотрены методы представления и форматы хранения воксельных данных, которые содержат информацию для реконструкции острых углов поверхности. Разработн метод для компактного хранения поверхности воксельного ландшафта.

В четвертой главе разработаны методы для визуализации больших воксельных ландшафтов с различными уровнями детализации. Разработан метод для «бесшовной стыковки» смежных блоков ландшафта, обладающих различными уровнями детализации. Разработана модификация алгоритма дуальных контуров, которая совершенно не требует создания бесшовного соединения.

В пятой главе приводится описание программной разработки: архитектуры, функциональной структуры, форматов хранения данных, технологий разработки и механизмов реализации.

В заключении диссертации приводятся основные научные и прикладные результаты, полученные в процессе выполнения диссертационной работы, и выделяются возможные направления дальнейших исследований.

В приложениях приведены: краткое разъяснение используемых в работе понятий и терминов; фрагменты программного кода, реализующие описанные алгоритмы; копия свидетельства о регистрации программы для ЭВМ.

Благодарности. Автор выражает благодарность Шабалиной Ольге Аркадьевне, кандидату технических наук, доценту кафедры «САПР и ПК» ВолгГТУ, за оказанную помощь и консультации в ходе выполнения диссертационной работы.

1 Анализ методов моделирования и визуализации ландшафтов в проектировании систем виртуальной реальности

1.1 Процесс проектирования систем виртуальной реальности

Концепция «виртуальной реальности» в широком смысле подразумевает создание интерактивных моделей реальной действительности, непосредственное влияющих на сознание человека через его органы чувств. В рамках данной работы виртуальная реальность (ВР) рассматривается, как искусственный трехмерный мир — киберпространство, созданное с помощью компьютера и современных информационных технологий, в особенности, реалистичной компьютерной графики, необходимой для создания зрительного эффекта присутствия в искусственно созданном мире. В настоящее время технологии виртуальной реальности успешно применяются для решения широкого круга практических задач: виртуальное проектирование ландшафтов и архитектуры, разработка всевозможных симуляторов и учебных тренажерных комплексов, виртуальная коммуникация для работы и деловых встреч, посещение виртуальных музеев, виртуальная археология и т.д. Погружаясь в виртуальный мир, человек обретает власть над цифровыми объектами. Применение технологий ВР в САПР позволяет вместо статичных и плоских изображений получать полноценные трёхмерные объекты, на которые можно влиять непосредственно [1, 2]. Также возможно успешное применение технологий ВР и компьютерной графики в образовательных целях [3, 4].

Одним из важных аспектов реализации концепции виртуальной реальности является виртуальное проектирование — проектирование виртуальных объектов, в частности, архитектурное параметрическое проектирование и виртуальное макетирование. Спроектированные виртуальные объекты зачастую отражают реальную действительность, и могут найти своё материальное воплощение (например, при создании проектов зданий). Использование виртуальных моделей позволяет существенно сократить время и средства, а также даёт архитектору или конструктору возможность детально рассмотреть различные варианты и выбрать наиболее оптимальный на стадии проектирования.

Ключевым этапом при создании трехмерной сцены в процессе виртуального проектирования строительных конструкций или больших открытых пространств для транспортных тренажерных комплексов является моделирование ландшафта. В данной работе под ландшафтом понимается совокупность естественного рельефа поверхности и крупных объектов искусственного происхождения, таких как здания и технические объекты.

Основными этапами ландшафтного проектирования являются: внешнее проектирование, эскизное проектирование и рабочее ландшафтное проектирование.

На этапе внешнего проектирования проводятся маркетинговые исследования и составляются требования технического задания.

На этапе эскизного (концептуального) проектирования проводятся определяются общая пространственная планировка и ландшафтный дизайн объекта — стиль участка, разбивка территории на функциональные зоны, размещение основных ландшафтных элементов, крупных форм и сооружений, древесно-кустарниковой растительности, цветников и газонов, водоёмов и ручьёв, прокладывается дорожно-тропиночная сеть участка, а также решается вопрос о трансформации или корректировке рельефа.

На этапе рабочего ландшафтного проектирования составляются детальные чертежи участка. Эта стадия включает в себя разработку генерального плана, создание разбивочного и посадочного чертежа, дендроплана, схемы размещения осветительных элементов, планов баланса площадей, малых форм и сооружений, а также другие рабочие чертежи, помогающие при дальнейших ландшафтных работах.

Технологии ВР позволяют значительно ускорить процесс создания ландшафта на этапе эскизного проектирования, предоставляя пользователю трёхмерное представление вариантов планировки участка и возможность чувствовать и контролировать имеющееся пространство. При этом использование объёмного, воксельного представления сцены позволяет естественным образом реализовать операции объемного геометрического моделирования, с помощью которых могут быть решены задачи отработки внешнего облика рельефа задолго до этапа его физической реализации в реальном мире и в соответствии с приёмами ландшафтного дизайна. В процессе виртуального проектирования ландшафта необходимость работы в интерактивном режиме накладывает жесткие ограничения на организацию программных средств [5].

Из-за высокой вычислительной сложности обработки воксельных моделей большинство существующих САПР, которые ориентированы на приложения в области архитектуры и строительства, используют граничные (B-Rep) представления объектов и поэтому не обладают функциональными возможностями для полноценного проектирования ландшафтов в 3D-пространстве [6,7]. В отличие от воксельного представления, многие операции геометрического моделирования над граничными представлениями объектов не определены или ненадёжны.

Можно сделать вывод, что применение технологий ВР и объемного геометрического моделирования позволяет расширить возможности САПР и повысить качество проектирования ландшафтов, что подтверждает целесообразность и актуальность настоящего исследования.

1.2 Современные проблемы виртуального проектирования ландшафтов

Для представления ландшафтов традиционно используются карты высот (height maps или elevation maps), или однозначные скалярные функции высот от планового положения точки. Подавляющее большинство программных пакетов, предназначенных для интерактивной визуализации ландшафтов в системах ВР, и по сей день работают только с картами высот, поскольку для них разработаны специализированные, весьма простые и эффективные алгоритмы редактирования, процедурной генерации, сжатия и распаковки, построения бесшовной, адаптивной и непрерывной триангуляции в различных разрешениях. Карты высот можно редактировать в любом редакторе изображений и создавать процедурно, при этом существует огромное количество готовых карт высот (например, высокоточные топографические карты, полученные с помощью лазерного сканирования).

В силу аппаратных ограничений современных настольных компьютеров «влобный» подход к отрисовке больших ландшафтов зачастую непрактичен или невозможен. На практике используются алгоритмы с внешней памятью или так называемые «внеядерные» (out-of-core) подходы, в которых части сцены могут подгружаться и выгружаться по мере необходимости, и техники визуализации с различными уровнями детализации (Levels of Detail, LoD) [8], обеспечивающие оптимальное распределение ограниченных вычислительных ресурсов.

В последнее время всё большее применение для представления ландшафтов находит использование объёмных или воксельных данных, в которых в явной форме содержатся сведения о принадлежности элементов объекта внутреннему или внешнему по отношению к объекту пространству. Вóксель или вóксел (voxel, сокр. от volume (или volumetric) pixel) — это элемент объёма, которому обычно сопоставляется индекс подобласти с определёнными свойствами (идентификатор материала) или физические свойства и характеристики среды (например, температура, цвет, плотность, расстояние до поверхности и т.п.). В отличие от традиционных карт высот, воксельные ландшафты позволяют представить сложные трёхмерные особенности ландшафтов и описать их внутреннее геологическое строение. Например, воксельные ландшафты обладают возможностью передавать пещеры и арки, отвесные и нависающие скалы, а при

использовании ландшафтов, основанных на картах высот, нельзя описать даже вертикальные колонны. Поскольку в настоящее время для моделирования и визуализации ландшафтов преимущественно используются карты высот, то 3D-объекты, которые не могут быть представлены картой высот, создаются во внешних САПР и импортируются в ландшафтный редактор. Использование унифицированного воксельного представления и для природного рельефа, и для зданий способно решить проблемы бесшовной интеграции архитектурных моделей с ландшафтом, устранения пересечений моделей с 3D-ландшафтом, задачи обеспечения параметрической связи вне одного пакета и т.д.

Однако использование воксельных ландшафтов сопряжено с рядом проблем. Воксельные данные занимают на порядок больше памяти, чем карты высот, поэтому возникает необходимость в эффективном сжатии, хранении, подгрузке и распаковке данных ландшафта. Для визуализации воксельных ландшафтов в интерактивном режиме (с частотой обновления не ниже 15-25 кадров в секунду) сперва строится треугольная сетка поверхности, которая затем отрисовывается (растеризуется) с помощью графического оборудования. При триангуляции воксельного ландшафта построенная сетка, как правило, имеет гораздо больше треугольников, чем при триангуляции ландшафта, основанного на картах высот, поэтому ещё более остро встают задачи упрощения сетки, а также проблемы генерации, выбора, подгрузки и смены уровней детализации.

Поскольку ландшафт обычно не умещается в оперативную память целиком, его разбивают на отдельные блоки, что позволяет подгружать, редактировать, изменять уровни детализации и отрисовывать только необходимые его части. При этом возникает сложная задача сшивки фрагментов ландшафта, обладающих различными уровнями детализации. Без решения этой проблемы на границах между смежными блоками ландшафта, обладающими различными уровнями детализации, возникают разрывы и самопересечения треугольной сетки.

В настоящее время для визуализации воксельных данных в большинстве интерактивных приложений (или приложений «мягкого» реального времени) наиболее практичными являются косвенные методы, основанные на растеризации треугольных сеток с помощью специального графического оборудования (GPU). Подавляющее большинство существующих интерактивных программных комплексов для представления, хранения и редактирования воксельного ландшафта используют воксели, а для отображения — треугольники. Таким образом, для визуализации воксельного ландшафта в интерактивном режиме необходимо сначала построить по исходным воксельным данным поверхностную сетку, аппроксимирующую поверхность ландшафта набором треугольников.

Результатом триангуляции воксельного ландшафта являются нерегулярные треугольные сетки (TIN), кусочно-линейно аппроксимирующие рельеф наборами треугольников. Существующие алгоритмы разбиения, бесшовной склейки и упрощения треугольных сеток обладают высокой вычислительной сложностью и малой надёжностью [9].

До настоящего времени проблема интерактивных моделирования и визуализации воксельных ландшафтов исследована сравнительно мало: по данной теме опубликовано две диссертации [10,11] и несколько работ [12–22]. Существует несколько программных продуктов [23–27], предоставляющих программный интерфейс (API) для написания производных приложений для работы с воксельными ландшафтами. Однако разработанные в указанных программных продуктах технические решения являются закрытыми.

В силу аппаратных ограничений компьютеров воксельные данные сначала использовались для представления грубых, «гладких» воксельных ландшафтов.

«Гладкие» воксельные ландшафты подходят для передачи органических форм, которые встречаются в природе (например, песчаные холмы, крупные валуны, пещеры с закруглёнными углами) (рисунок 1.1). Для триангуляции «гладких» воксельных ландшафтов наиболее часто используются алгоритмы Марширующих Кубиков (Marching Cubes, MC) [28], Марширующих Тетраэдров (Marching Tetrahedra, MT) [29] и Поверхностных Сеток (Surface Nets) [30].

Рисунок 1.1 – Примеры гладких воксельных ландшафтов. Для бесшовной триангуляции ландшафта, состоящего из частей с различными уровнями детализации, использовались методы (слева) Transvoxel [10,13] и (справа) Nested Clip Boxes [11,14,15].

Воксельные ландшафты с острыми рёбрами и углами в последнее время всё чаще используются для представления искусственных элементов ландшафта, таких как здания и

крупные технические объекты (рисунок 1.2). Для триангуляции воксельного ландшафта с наличием искусственных объектов требуются методы извлечения изоповерхностей, способные восстанавливать острые углы и рёбра. Для хранения информации для реконструкции острых углов ландшафта обычно применяются смешанные, гибридные «воксельно-геометрические» представления, в которых, помимо трёхмерных скалярных данных, также содержатся сведения и об особенностях поверхности на границе объекта.

Рисунок 1.2 – Примеры воксельных ландшафтов с искусственными объектами: а) Voxel Farm [23]; б) Upvoid Engine [24]. Для триангуляции с реконструкцией острых рёбер и углов поверхности использовались алгоритмы (слева) Dual Contouring [31] и (справа) Cubical Marching Squares [33].

Программные решения для воксельных ландшафтов с острыми углами. На момент написания работы Voxel Farm [23] является единственным коммерческим пакетом (с интеграцией в популярные игровые «движки» Unity и Unreal Engine 4), способным процедурно генерировать и в интерактивном режиме отображать большие и модифицируемые воксельные ландшафты с острыми рёбрами и углами. Voxel Farm использует для триангуляции воксельных данных модифицированный метод дуальных контуров (Dual Contouring, DC) [31]. В блоге проекта периодически появляются упоминания о новых результатах исследований, однако подробная информация о деталях реализации представлена крайне скудно. Задача визуализации воксельных ландшафтов с острыми углами частично решается в закрытых пакетах Ultimate Terrains (uTerrains) [25], TerrainEngine [26] и TerraVol [27], предназначенных для работы в среде Unity 3D.

Современные тенденции развития воксельных ландшафтов. Технологии воксельных ландшафтов и виртуальной реальности в основном применяются в индустрии видеоигр. Вышедшая в 2011 году видеоигра Minecraft, которая предоставляет игрокам возможности создавать свои

модели и исследовать процедурно генерируемые миры, представленные в виде вокселей кубической формы (boxel, bloxel или cuberille), привела к взрыву популярности воксельных «движков» и породила множество клонов. Воксельное представление начинает применяться в видеоиграх для реализации модифицируемого окружения (строительство и разрушение), процедурной генерации 3D-контента, симуляции физических процессов (моделирование мультифазных сред и жидкостей) и т.д. Например, видеоиграми, использующими воксели для представления ландшафтов, являются EverQuest Next Landmark (отменена в марте 2016 года) и Crowfall (используют Voxel Farm [23]) (на 2018 год ещё находится в разработке), The Tomorrow Children (использует Dual Contouring [31]; выпущена осенью 2016) [32], Planet Explorers! (2014), Subnautica (2017), UpvoidMiner [24] (использует Cubical Marching Squares [33]; разработка прекращена в 2014) и GeoMechanic (2015), No Man's Sky (2016), Miner Wars 2081 [17] (использует марширующие кубики [28]; выпущена в 2014) и Space Engineers (активно разрабатывается с 2013), Dual Universe (использует Dual Contouring [31] для полигонизации планетарных воксельных ландшафтов и искусственно созданных объектов, находится в разработке с 2014 года) и многие другие.

1.3 Методы триангуляции изоповерхностей с реконструкцией острых углов

Наиболее полный обзор и анализ современных алгоритмов триангуляции изоповерхностей приведён в [34,35,36]. Далее будут рассмотрены наиболее важные и распространённые интерактивные ячеечные двойственные методы триангуляции, способные воспроизводить острые рёбра и углы изоповерхности [37] (ключевые понятия и определения предметной области приведены в [9,38,39] и в Приложении А, а обзор основных подходов к визуализации изоповерхностей и воксельных ландшафтов с различными уровнями детализации и использованием данных методов и алгоритмов триангуляции приведён в четвёртой главе).

1.3.1 Алгоритм «Extended Marching Cubes»

Алгоритм расширенных марширующих кубиков (Extended Marching Cubes, EMC**)** [40] является самым первым опубликованным в 2001 году алгоритмом извлечения изоповерхности, способным воспроизводить её острые рёбра и углы.

Вначале для каждой активной ячейки из кубической решётки разбиения находятся координаты $\mathbf{s}_i$ точек пересечения рёбер ячеек с изоповерхностью, а также единичные нормали $\mathbf{n}_i$ к поверхности в этих точках пересечения, которые служат для локализации её острых углов.

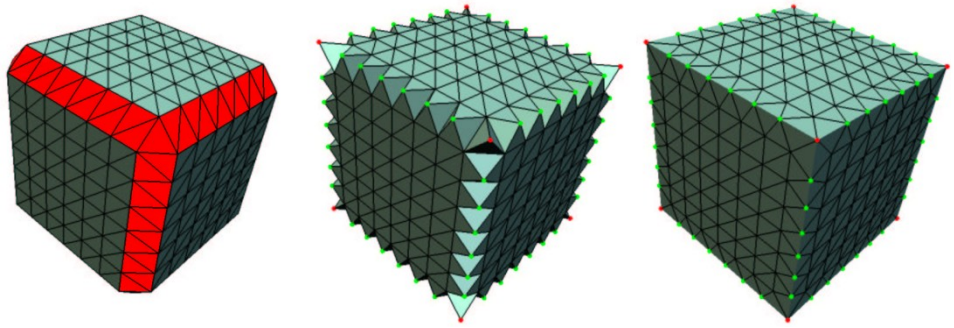
Работа алгоритма EMC может быть описана в три этапа (рисунок 1.3):

1) Определение *особенных* (содержащих острые углы поверхности) ячеек.

2) Триангуляция ячеек в зависимости от их типа.

3) Переворачивание рёбер («флип» от англ. edge flipping) для соединения вершин в смежных особенных ячейках и получения острых граней поверхности.

На первом этапе для определения типа ячейки (содержит ли ячейка острый угол или ребро поверхности?), в [40] предложено оценивать открытый угол конуса нормалей. Среди всех точек пересечения находится пара нормалей $\mathbf{n}_0$ и $\mathbf{n}_1$, образующих максимальный открытый угол: $\theta = \min_{i,j} (\mathbf{n}_i^{\mathrm{T}} \mathbf{n}_j)$. Если $\theta < \theta_{sharp} = 0.9$, то ячейка содержит острый угол или грань. Затем вычисляется максимальное отклонение φ нормалей $\mathbf{n}_i$ от векторного произведения $\mathbf{n}^* = \mathbf{n}_0 \times \mathbf{n}_1$: $\varphi = \max_i | \mathbf{n}_i^{\mathrm{T}} \mathbf{n}^* |$; если $\varphi > \varphi_{corner} = 0.7$, то ячейка содержит острый угол, а не ребро.



а) б) в)

Рисунок 1.3 – Этапы работы метода EMC [40]: а) определение ячеек, содержащих острый угол или острое ребро; б) нахождение вершин, лежащих на острых углах или рёбрах внутри ячеек; в) треугольная сетка после переворачивания рёбер для получения острых рёбер поверхности.

На втором этапе, если активная ячейка не является особенной (т.е. не содержит острого ребра или острого угла), то внутри неё поверхность строится стандартным алгоритмом марширующих кубиков (Marching Cubes, MC) [28]. Если активная ячейка содержит острый угол или ребро, то внутри неё создаётся вершина $\mathbf{v}$, расположенная на остром углу или на остром ребре. Затем вершина $\mathbf{v}$ веером треугольников (triangle fan) соединяется с другими вершинами, расположенными в точках пересечения $\mathbf{s}_i$ активных рёбер ячейки с изоповерхностью. Оптимальная позиция острой вершины $\mathbf{v}$ минимизирует сумму квадратов расстояний от $\mathbf{v}$ до касательных

плоскостей, образованных $\mathbf{s}_i$ и $\mathbf{n}_i$, которую можно представить в виде квадратичной функции погрешности (Quadratic Error Function, QEF) [41, 42, с. 44]:

$$E(\mathbf{v}) = \sum_i (\mathbf{n}_i \cdot (\mathbf{v} - \mathbf{s}_i))^2. \quad (1.1)$$

Минимуму QEF (1.1) удовлетворяет решение системы линейных уравнений:

$$[..., \mathbf{n}_i, ...]^T \mathbf{v} = [..., \mathbf{n}_i^T \mathbf{s}_i, ...], \quad (1.2)$$

где $\mathbf{s}_i$ — координаты i-ой точки пересечения ребёр ячейки с изоповерхностью, $\mathbf{n}_i$ — единичная нормаль к поверхности в этой точке. Поскольку данная система, как правило, является переопределённой, в [43,40] предложено искать нормальное псевдорешение системы (1.2), дополнительно минимизируя расстояние от $\mathbf{v}$ до центра кубической ячейки.

Поскольку острые вершины ячеек создаются независимо друг от друга, то для получения острых рёбер поверхности на последнем этапе необходимо перевернуть те рёбра получившейся треугольной сетки, которые бы потом соединяли острые вершины ячеек после переворачивания (см. рисунок 1.3, в).

EMC не всегда генерирует развёртываемые треугольные сетки, потому что в особенных ячейках создаётся только по одной острой вершине. Другими недостатками метода являются необходимость поддерживать структуру данных смежности треугольной сетки из-за необходимости переворачивания рёбер, проблема зависимости ячеек друг от друга (inter-cell dependency), возможность создания низкокачественных треугольных сеток (содержащих полувырожденные треугольники и самопересечения) и сложность реализации адаптивных вариантов. При ошибочном определении типа ячеек теряются острые углы и рёбра изоповерхности при её реконструкции. В [44] описывается адаптивный вариант алгоритма (Adaptive Extended Marching Cubes) на основе октодеревьев.

1.3.2 Метод «Dual Contouring»

Метод дуальных или двойственных контуров (Dual Contouring, DC) [31] объединяет алгоритмы EMC [40] и алгоритм поверхностных сеток (Surface Nets, SF) [30,45], самый первый опубликованный в 1999 г. двойственный алгоритм триангуляции. Двойственные методы триангуляции создают полигональные сетки, которые являются двойственными по отношению к сеткам, построенных с помощью прямых методов (см. Приложение А) (см. рисунки 1.4 и 1.5).

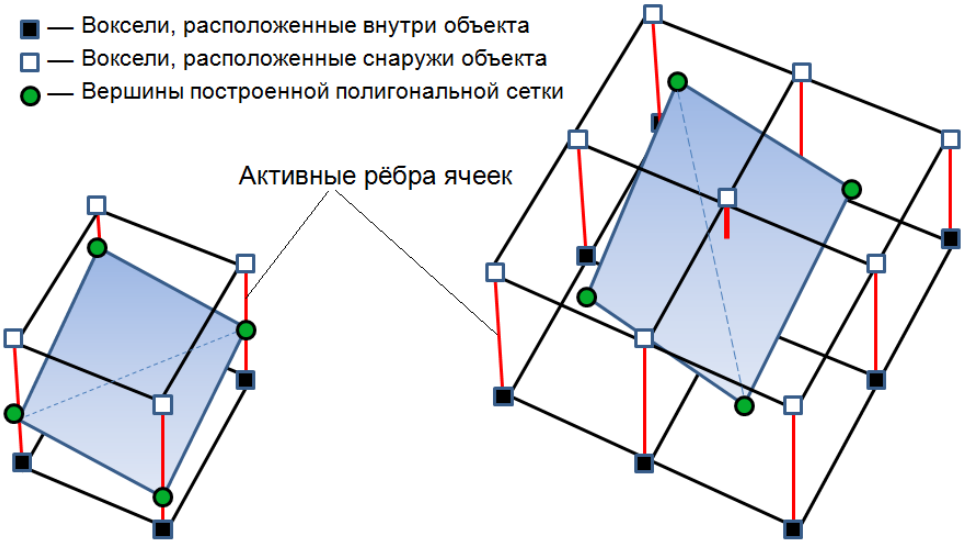


Рисунок 1.4 – а) Отдельно взятая ячейка и построенный в ней фрагмент треугольной сетки с помощью алгоритма марширующих кубиков (MC). б) Четыре смежные активные ячейки и четырёхугольник, созданный для их общего активного ребра алгоритмом Dual Contouring/Surface Nets. В алгоритме MC на каждом активном ребре создаётся по вершине сетки, в то время как алгоритм DC создаёт для каждого активного ребра ячейки по четырёхугольнику.

Для регулярных знакоопределённых кубических решёток с наличием Эрмитовых данных (точек пересечения рёбер ячеек с поверхностью и единичных нормалей к поверхности) на активных рёбрах каждой ячейки четырёхугольная сетка строится по следующим правилам (алгоритм Uniform Dual Contouring):

1) для каждой активной ячейки создаётся вершина, как в EMC [40];

2) для каждого активного ребра создаётся четырёхугольник, соединяющий вершины четырёх смежных ячеек, для которых данное ребро является общим. Ориентация четырёхугольника зависит от знаков на концах ребра, как в SF [30].

На первом этапе в каждой активной ячейке создаётся вершина, позиция $\mathbf{v}$ которой удовлетворяет минимуму QEF (1.1), т.е. является решением системы

$$\begin{pmatrix} \mathbf{n}_0^{\mathrm{T}} \\ \vdots \\ \mathbf{n}_N^{\mathrm{T}} \end{pmatrix} \mathbf{v} = \begin{pmatrix} \mathbf{n}_0^{\mathrm{T}} \mathbf{s}_0 \\ \vdots \\ \mathbf{n}_N^{\mathrm{T}} \mathbf{s}_N \end{pmatrix} \tag{1.3}$$

а также, в отличие от EMC, дополнительно минимизирует расстояние от $\mathbf{v}$ до центра масс $\mathbf{v}_{\mathbf{mass}}$ точек пересечения рёбер ячейки с изоповерхностью [46]:

$$\mathbf{v}_{\mathbf{mass}} = \frac{1}{N} \sum_{i=0}^{N} \mathbf{s}_i \,,$$

где $\mathbf{s}_i$ — координаты i-ой точки пересечения ребёр ячейки с изоповерхностью,
а N — количество точек пересечения (равное количеству активных рёбер ячейки). (В алгоритме SF $\mathbf{v}$ всегда помещается в $\mathbf{v}_{\mathbf{mass}}$.) При наличии погрешностей во входных данных или недостаточном разрешении решётки разбиения позиция острой вершины может выходить за пределы (AABB) ячейки [46]. В этих случаях координаты вершины $\mathbf{v}$ «обрезают» границами ячейки (clamping) или помещают в $\mathbf{v}_{\mathbf{mass}}$, что приводит к появлению зубцов и вмятин на острых рёбрах полученной поверхности [46,59,60]. Для качественного позиционирования острой вершины в систему (1.3) добавляют дополнительные ограничения [19,47,48,49].

На втором этапе создаются четырёугольники, соединяющие вершины сетки.

В [31,46] также предложен адаптивный алгоритм триангуляции методом DC (Adaptive Dual Contouring, ADC) на основе несбалансированных знакоопределённых октодеревьев, в которых размеры соседних ячеек могут отличаться более чем в два раза:

1) построение октодерева до максимальной глубины разбиения;

2) рекурсивное упрощение октодерева путем слияния его листовых узлов, если их суммарная невязка (значение QEF (1.1)) меньше заданной величины;

3) рекурсивный обход дерева и построение полигональной сетки.

Более подробное описание метода ADC можно найти в разделе 2.3.

В Extended Dual Contouring [50,51] и Enhanced Dual Contouring [52] на рёбрах ячейки может быть зарегистрировано до двух точек пересечения с изоповерхностью. В этих методах, а также в Manifold Dual Contouring (MDC) [53], в каждой активной ячейке может быть создано до четырёх вершин, что позволяет реконструировать более мелкие детали, чем DC, при одинаковом разрешении сетки разбиения, а также сохранить первоначальную топологию модели даже после агрессивного упрощения октодерева. В [54] описан адаптивный вариант DC на основе kD-деревьев (kD-trees), которые лучше адаптируются к особенностям поверхности, чем октодеревья. Помимо параллелепипедов, ячейки в методе дуальных контуров могут иметь форму любого выпуклого многогранника. В [55] описан алгоритм дуальных контуров на тетраэдральных ячейках, который всегда генерирует двусторонние развёртываемые (2-manifold) полигональные сетки.

Основные достоинства метода дуальных контуров: простота реализации, высокое быстродействие, широкая применимость. Поскольку в большинстве реализаций позиции вершин помещаются близко к центроиду точек пересечения рёбер ячейки с поверхностью [40,46], то создаваемые сетки обычно содержат меньше треугольников плохой формы [30,56], чем сетки, построенные прямыми методами. Для регулярных сеток метод DC допускает табличную

реализацию и хорошо переносится на GPU [57,18,19]. Недостатки метода DC и его адаптивного варианта подробно описаны в разделах 2.2 − 2.4.

1.3.3 Алгоритм «Dual Marching Cubes»

Алгоритм дуальных (двойственных) марширующих кубиков (Dual Marching Cubes, DMC) [56] создаёт четырёхугольные сетки, которые являются двойственными по отношению к сеткам, построенным с помощью марширующих кубиков (MC) [28] (рис. 1.5).

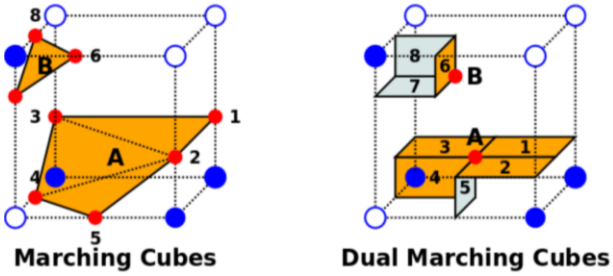


Рисунок 1.5 – DMC строит полигональную сетку, двойственную той, полученной с помощью MC: каждому созданному полигону (связной компоненте) в MC соответствует вершина в DMC, а каждой вершине на рёбре ячейки в MC — четырёхугольник в DMC. Изображение взято из [58].

В отличие от алгоритма DC, DMC может создавать в каждой активной ячейке до четырёх вершин. Каждому шаблону триангуляции из таблицы марширующих кубиков соответствует аналогичный вариант для DMC. Все $2^8 = 256$ вариантов с учётом вращательной симметрии сводятся к 23 базовым случаям, показанным на рисунке 1.6.

Каждый элемент таблицы в DMC содержит информацию о количестве вершин в ячейке и топологии их связи для построения четырёхугольной сетки $S^{\Diamond}$:

1) количество вершин, которое нужно создать внутри ячейки;

2) список активных рёбер для каждой вершины (или связной компоненты).

В каждой активной ячейке создаётся от одной до четырёх вершин сетки $S^{\Diamond}$, каждой вершине из $S^{\Diamond}$ соответствует от трёх до шести активных рёбер ячейки.

Процесс построения сетки $S^{\Diamond}$ алгоритмом DMC выполняется в два этапа:

1) в каждой ячейке создаются вершины согласно знаковой конфигурации;

2) для каждого активного ребра (из 12 возможных) каждой ячейки создаётся четырёхугольник, соединяющий вершины четырёх смежных ячеек, для которых это ребро является общим, аналогично DC (номер вершины известен из таблицы).

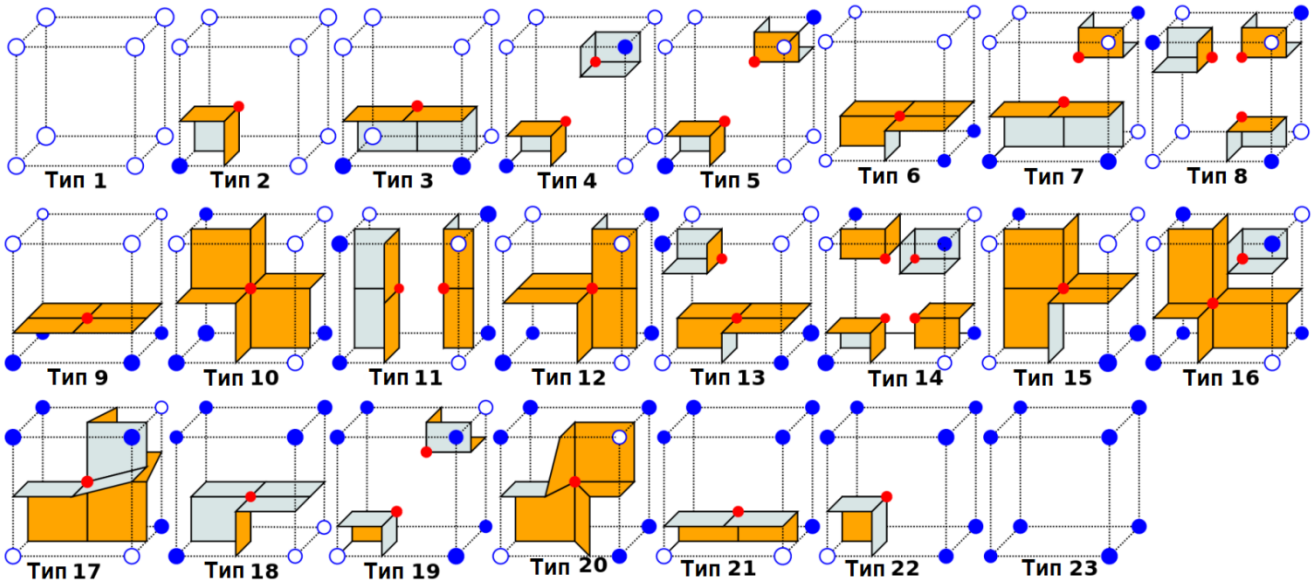


Рисунок 1.6 – Базовые знаковые конфигурации ячейки в DMC, из которых последовательными поворотами могут быть получены все элементы таблицы. Изображение взято из [58].

DMC способен реконструировать особенности поверхности при наличии Эрмитовых данных [53]. Из таблицы смежности для каждой вершины известен список соответствующих ей активных рёбер ячейки, поэтому позиции острых вершин могут быть оценены теми же способами, как и в алгоритмах DC и EMC.

DMC послужил основой для других двойственных алгоритмов [53,59,60]. В [48,49,35] показано, что в некоторых случаях алгоритм может создавать неразвёртываемые сетки, в которых ребро является общим для более, чем двух полигонов, и предлагается модификация алгоритма DMC для гарантированного создания развёртываемых сеток. Существует аналогичный алгоритм для тетраэдральных ячеек [61]. В целом, алгоритм DMC наследует достоинства и недостатки алгоритма дуальных контуров.

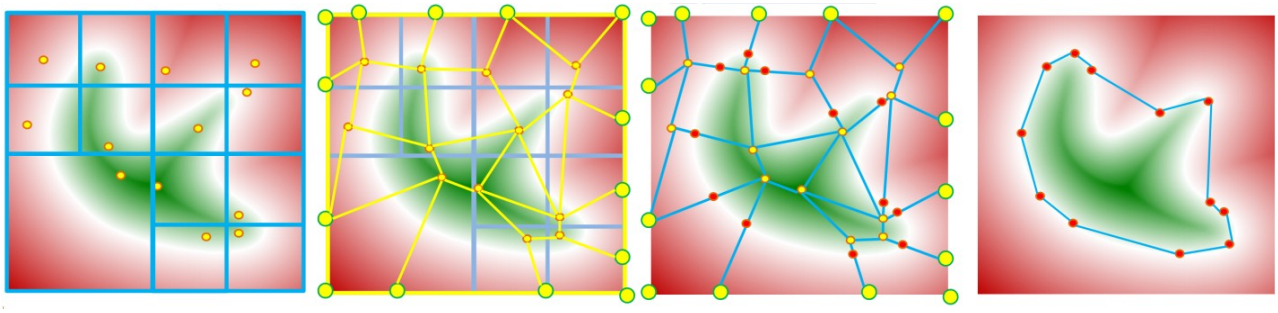
1.3.4 Метод «Dual Marching Cubes: Primal Contouring of Dual Grids»

Основная идея метода прямой триангуляции двойственных решёток (Primal Contouring of Dual Grids, PCDG) [62] заключается в построении из октодерева двойственной к нему шестигранной решётки, которая затем триангулируется с помощью алгоритма марширующих кубиков. Триангуляция скалярного поля ϕ с помощью метода PCDG выполняется в три этапа:

1) построение октодерева, с заданной точностью аппроксимирующего особенности ϕ: каждый листовой узел октодерева содержит вершину в особых точках ϕ (т.е. в стационарных или критических точках поля ϕ, где градиент $\nabla\phi$ не определен или обращается в нуль) и скалярное значение поля в этой точке;

2) создание решётки (dual grid), двойственной по отношению к октодереву;

3) триангуляция полученной решётки-«скелета» с помощью MC (рисунок 1.7).



а) б) в) г)

Рисунок 1.7 – Этапы работы PCDG: а) построение октодерева, в котором для каждого листового узла находится особая точка скалярного поля; б) построение двойственной к октодереву решётки (жёлтая); в) триангуляция двойственной решётки с помощью MC (красным показаны вершины контура, лежащие на изоповерхности); г) реконструированный контур изоповерхности (выбранное разбиение октодерева не позволяет передать все особенности поверхности).

На первом этапе особые точки скалярного поля ϕ могут находиться методом градиентного спуска или минимизацией QEF (1.1) (особая точка скалярного поля в ячейке минимизирует сумму квадратов расстояний до касательных плоскостей, образованных позициями углов ячейки и градиентами поля в этих точках). Особые точки скалярного поля ϕ становятся вершинами двойственной решётки.

На втором этапе построение двойственной к октодереву решётки может производиться стандартным рекурсивным нисходящим алгоритмом [62,63,31,46] или снизу вверх [64]. При этом ячейки двойственной решётки могут вырождаться в выпуклые многогранники с меньшим, чем у куба, количеством вершин. Однако они сохраняют топологию куба и поэтому могут быть триангулированы с помощью MC на третьем этапе, при этом позиции вершин рёбрах ячеек, полученные линейной интерполяцией значений поля в вершинах ячеек, будут лежать на нулевой изоповерхности скалярного поля. При помещении вершин двойственной решётки в стационарные точки поля её ячейки могут становиться невыпуклыми, и тогда их последующая триангуляция с помощью MC приводит к самопересечениям сетки [62,63]. На практике вершины двойственной

решётки часто помещают в центры ячеек октодерева, что устраняет самопересечения, но приводит к генерации низкокачественных треугольных сеток. В [62,63] для улучшения качества сетки предложено помещать вершины на изоповерхность, если они находятся от неё меньше некоторого порогового расстояния.

Преимущества метода PCDG по сравнению с адаптивным DC: отсутствие необходимости получения Эрмитовых данных (точек пересечения рёбер ячеек и единичных нормалей к поверхности в этих точках), гарантированное создание развёртываемых треугольных сеток, более высокая «разрешающая способность» на наклонных поверхностях при одинаковой степени измельчения октодерева (поскольку форма ячеек двойственной решётки адаптируется к особенностям скалярного поля), при изменении нулевого уровня исследуемого скалярного поля не нужно перестраивать октодерево и двойственную решётку (в DC пришлось бы обновлять знаки всех ячеек). Недостатки метода PCDG: высокая вычислительная сложность, усугубление проблемы inter-cell dependency, при редактировании поля необходимо перестраивать октодерево/решётку, отсутствие поддержки различных материалов, возможность создания треугольников плохой формы и неправильной передачи топологии исходного объекта при реконструкции, поскольку для триангуляции ячеек двойственной решётки используется табличный алгоритм.

Существует аналогичный алгоритм для тетраэдральных ячеек [63], который создаёт более детальные и свободные от самопересечений треугольные сетки за счёт увеличения количества треугольников и снижения их качества.

1.3.5 Алгоритм «Cubical Marching Squares»

Алгоритм кубических марширующих квадратов (Cubical Marching Squares, CMS) [33] является гибридом прямых и двойственных методов извлечения изоповерхности, разработанным для решения проблемы взаимозависимости соседних ячеек (inter-cell dependency) при построении триангуляции и топологической неточности (topological inconsistency) построенных поверхностей.

Триангуляция знакоопределённой кубической решётки с Эрмитовыми данными алгоритмом кубических марширующих квадратов выполняется в три этапа:

1) построение знакоопределённого октодерева с Эрмитовыми данными;

2) построение сегментов на гранях ячеек октодерева;

3) триангуляция каждой ячейки октодерева.

На первом этапе каждая ячейка октодерева измельчается, если не достигнут максимальный уровень разбиения, и верно хотя бы одно из следующих условий:

1) какое-либо её ребро имеет более двух точек пересечения с поверхностью;

2) ячейка содержит особенности поверхности (аналогично EMC).

На втором этапе на гранях каждой ячейки октодерева создаются ломаные (сегменты) с помощью алгоритма марширующих квадратов (Marching Squares, MS) (кубическая ячейка может быть развёрнута на шесть граней). Помимо нахождения острых углов, находящихся внутри ячейки, нормали на активных рёбрах ячейки также используются для разрешения 2D-неоднозначностей в алгоритме марширующих квадратов при выборе шаблона триангуляции на гранях ячейки (face ambiguities) и локализации острых вершин, лежащих на грани ячейки (рисунок 1.8).

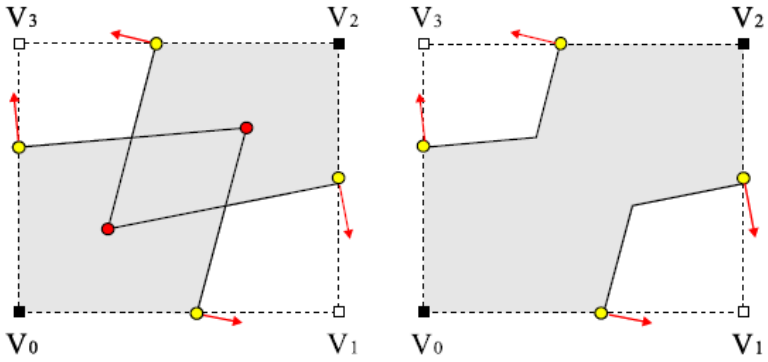


Рисунок 1.8 – Использование нормалей в точках пересечения рёбер ячейки с поверхностью для разрешения 2D-неоднозначностей в алгоритме марширующих квадратов при выборе способа соединения компонентов на гранях ячейки. Использование шаблона триангуляции слева приводит к появлению самопересечения в построенной поверхности, поэтому выбирается другой шаблон триангуляции, в котором сплошные области, включающие углы V_0 и V_2, являются связными.

На последнем этапе каждая ячейка октодерева триангулируется независимо от других:

1) сегменты на гранях объединяются в замкнутые циклические компоненты;

2) для каждой компоненты определяется коническая вершина поверхности;

3) полученные компоненты триангулируются веерами треугольников, 3D-неоднозначности (internal ambiguities) устраняются с помощью нормалей.

В [33] показано, что полученная полигональная сетка обладает большей точностью аппроксимации изоповерхности, чем EMC и DC.

Достоинства CMS: высокая точность аппроксимации изоповерхности, отсутствие inter-cell dependency при триангуляции ячеек, возможность поддержки различных материалов. Недостатки

CMS: высокая вычислительная сложность, сложность реализации. Подробное описание CMS можно найти в [65].

1.4 Основные выводы по первой главе

В первой главе проанализировано современное состояние исследований в области интерактивной визуализации воксельных ландшафтов в системах ВР, рассмотрены существующие системы визуализации и моделирования воксельных ландшафтов, а также используемые в них алгоритмы триангуляции.

С ростом требований к интерактивности и реалистичности в системах виртуальной реальности и видеоиграх технология воксельных ландшафтов становится популярной для моделирования модифицируемого окружения.

Большинство работ сфокусированы на «гладких» воксельных ландшафтах, и только в единственной работе [19] рассматривается проблема моделирования и визуализации ландшафтов с острыми рёбрами и углами, что необходимо для изображения искусственных объектов.

Существует небольшое число закрытых программных решений для моделирования и визуализации воксельных ландшафтов. На момент написания работы из платных решений возможностями моделирования ландшафтов с острыми углами обладают только Voxel Farm [23], Ultimate Terrains (uTerrains) [25], Terrain Engine [26] и TerraVol [27].

Для интерактивной визуализации воксельных ландшафтов используются методы извлечения изоповерхности, кусочно-линейно аппроксимирующие изоповерхность треугольными сетками, которые растеризуются с помощью GPU. Были подробно рассмотрены основные методы извлечения изоповерхностей с реконструкцией острых рёбер и углов. Показано, что прямая реализация рассмотренных методов для триангуляции воксельных ландшафтов недостаточно эффективна и мало исследована, а использование этих методов для триангуляции воксельных ландшафтов сопряжено с рядом трудностей и особенностей.

2 Разработка методов и алгоритмов триангуляции воксельных ландшафтов для проектирования систем ВР

В подразделе 2.4.2 описан разработанный автором алгоритм триангуляции октодеревьев, который может быть использован для адаптивной триангуляции изоповерхностей методами Dual Contouring [31] и Dual Marching Cubes [56]. Предложенный алгоритм разработан для триангуляции воксельного ландшафта с внешней памятью («по частям»), использует линейное представление октодерева и отличается простотой реализации и минимальным потреблением памяти. Проведено экспериментальное сравнение предложенного алгоритма с оригинальным вариантом [31,46] и ближайшим к нему алгоритмом [49].

2.1 Требования к методу триангуляции воксельного ландшафта

Как было упомянуто раннее, для отрисовки воксельного ландшафта в интерактивном режиме его сперва необходимо сконвертировать в треугольную сетку, которая аппроксимирует поверхность ландшафта. Построенная сетка затем может быть эффективно отрисована на специальном графическом оборудовании.

Для решения задач, сформулированных в первой главе, разрабатываемый метод триангуляции ландшафтов должен удовлетворять следующим требованиям:

1) высокая скорость триангуляции (для отображения изменений ландшафта в интерактивном режиме, после редактирования/смены уровней детализации);

2) поддержка различных материалов (для описания составных областей);

3) возможность генерации треугольных сеток с восстановлением острых рёбер и углов поверхности, что необходимо для отображения искусственных элементов ландшафта, таких как здания и крупные технические объекты;

4) высокое качество треугольной сетки (полученная сетка должна содержать как можно меньше треугольников плохой формы);

5) возможность триангуляции с внешней памятью (поскольку ландшафт, как правило, не умещается целиком в оперативную память компьютера).

Кроме того, желательно, чтобы используемый метод триангуляции ландшафта предоставлял следующие возможности:

1) возможность адаптивной триангуляции: поверхность ландшафта должна тесселироваться с минимальной избыточностью, как можно меньшим количеством треугольников, при сохранении визуального качества;

2) возможность генерации упрощённых представлений (грубых уровней детализации) (адаптивные методы триангуляции часто предоставляют эту возможность).

Адаптивная триангуляция позволяет уменьшить количество треугольников на плоских и однородных участках рельефа и, наоборот, измельчить сетку в районах высокой кривизны, сохранив при этом острые рёбра и углы поверхности. Это может повысить производительность дальнейших стадий, работающих с полученными треугольными сетками (например, подсистема оптимизации поверхностной сетки, подсистема рендеринга, подсистема обнаружения столкновений), поскольку вычислительная сложность, потребление памяти и время отрисовки кадра пропорциональны количеству созданных треугольников.

Возможность генерации упрощённых представлений требуется для визуализации больших и детализированных ландшафтов. Создание и перестроение дискретных уровней детализации должны происходить в интерактивном режиме и с сохранением особенностей поверхности, отражающих важные свойства объекта.

2.2 Проектирование метода триангуляции воксельного ландшафта

В данной работе на основе проведенного анализа в качестве базового метода для триангуляции воксельных ландшафтов был выбран метод дуальных или двойственных контуров (Dual Contouring, DC) [31,46] (см. раздел 1.2.2).

По сравнению с другими методами для интерактивной триангуляции изоповерхностей выбранный метод обладает рядом преимуществ:

1) универсальность: вершины создаваемой полигональной сетки могут свободно «плавать» внутри ячеек решётки разбиения пространства, что позволяет отображать как гладкие поверхности, так и объекты с острыми углами и рёбрами;

2) высокая скорость работы, алгоритм может быть перенесён на GPU;

3) приемлемое качество восстановления острых рёбер и углов поверхности;

4) метод легко расширяется до адаптивного варианта: DC содержит «встроенные» алгоритмы создания уровней детализации и генерации бесшовной и замкнутой полигональной сетки с различными уровнями детализации;

5) встроенная поддержка различных материалов;

6) относительная простота реализации и наличие большого количества реализаций с открытым кодом.

Недостатками и ограничениями метода дуальных контуров являются:

1) зависимости между ячейками при триангуляции (см. Приложение А);

2) необходимость в наличии нормалей к поверхности для её реконструкции;

3) высокая чуствительность к погрешностям во входных данных;

4) зависимость триангуляции от ориентации объекта относительно решётки разбиения, что приводит к «срезанию» (feature chamfering) острых углов и ребёр, наклоненных под большими углами к осям решётки разбиения, и избыточному измельчению октодерева в местах небольшой кривизны на плоских наклонных поверхностях (в стандартном адаптивном варианте метода);

5) возможность создания неразвёртываемых полигональных сеток, которые могут содержать пересекающиеся треугольники, сингулярные вершины и рёбра;

6) топологические неточности реконструированной поверхности из-за пропуска мелких деталей объекта при недостаточном разбиении.

Некоторые из вышеперечисленных недостатков не являются критичными для визуализации ландшафтов или могут быть частично устранены.

Зависимость ячеек друг от друга (inter-cell dependency) приводит к тому, что для получения непрерывной треугольной сетки смежные блоки воксельного ландшафта не могут быть триангулированы по отдельности друг от друга, даже если они имеют одинаковое разрешение или уровень детализации. Поэтому для бесшовного соединения отдельно взятого блока воксельного ландшафта со смежными к нему блоками необходимо всегда включать в процесс триангуляции ячейки соседних блоков. Решение данной проблемы обсуждается в разделе 4.3.

Для нахождения острых углов поверхности метод DC использует Эрмитовы данные (см. Приложение А), которые увеличивают и без того большой объём данных, требуемый для хранения воксельного ландшафта. Для качественной реконструкции острых углов объекта необходимы точные единичные нормали к поверхности, что на практике ограничивает возможную степень сжатия Эрмитовых данных. Проблема сохранения высокой точности при использовании сжатия решается применением специализированных алгоритмов сжатия или вычислением Эрмитовых данных из неявного представления (см. раздел 3.6). Кроме того, восстановление острых углов поверхности не требуется для передачи «гладких» воксельных ландшафтов.

При использовании зашумленных или неполных данных, или под влиянием вычислительных погрешностей найденные позиции острых углов могут выходить за пределы соответствующих ячеек, что может привести к созданию «плохих» треугольников, складок и пересечений в полученной сетке. Поэтому в DC позиции острых вершин ограничиваются охватывающими оболочками (AABB) ячеек. Однако на практике часто возникает ситуация, когда ячейкой «обрезается» действительная позиция острого угла. Например, в 3D возможна ситуация, когда коническая поверхность проходит сквозь грань ячейки, не пересекая никаких рёбер ячейки, что приводит к «срезанию» вершины конуса. Уменьшение размеров ячеек (увеличение разрешения решётки разбиения) делает такие дефекты менее заметными, но не позволяет полностью избавиться от «срезанных» углов. Для решения данной проблемы можно адаптировать форму ячеек к особенностям поверхности (как в методе PCDG [62,63]) или объединять острые вершины нескольких смежных ячеек (как методе SHREC [60]). На практике можно строить сетки приемлемого качества, допуская отклонение вершин в пределах допустимой величины, а возникшие дефекты устранять в процессе постробработки. Например, в Voxel Farm [23] допускается, чтобы позиции острых углов незначительно выходили за границы соответствующих ячеек (расширение «roaming vectors»[66]).

На наклонных и повёрнутых поверхностях адаптивный вариант метода DC теряет способность создавать адаптивную триангуляцию. Данное ограничение наследуется от структуры разбиения пространства, ячейки которой имеют форму кубов (в октодереве) или параллелепипедов (в *k*D-дереве), выровненных вдоль принципиальных координатных осей. Поэтому для реконструкции наклонных, тонких или протяжённых объектов лучше подходит метод PCDG [62,63]. В Voxel Farm [23] для уменьшения количества треугольников в построенной с помощью адаптивной триангуляции сетке дополнительно применяются методы упрощения (прореживания, редуцирования, десимации) треугольных сеток.

Для гарантированного создания развёртываемых полигональных сеток необходимо использовать другие двойственные методы триангуляции (например, Manifold DMC [35], CMS [33]), или модифицировать оригинальный алгоритм дуальных контуров (2-manifold correction) [54,51]. В подразделе 2.4.3 описано, как можно расширить DC до DMC без использования дополнительной информации, что позволяет создавать более детализированные и, в большинстве случаев, развёртываемые сетки. Отметим, что создание развёртываемых сеток не является обязательным требованием в большинстве приложений для визуализации ландшафтов, но делает

возможным или упрощает множество операций, которые могут выполняться над построенной поверхностной сеткой (см. Приложение А).

В рамках данной работы были разработаны несколько модификаций алгоритма адаптивной триангуляции методом дуальных контуров (Adaptive Dual Contouring, ADC) [31,46]. Данный алгоритм используется для интерактивной триангуляции воксельных ландшафтов с острыми углами в ряде коммерческих продуктов, в том числе и в Voxel Farm [23].

2.3 Описание традиционного подхода к адаптивной триангуляции

Прежде чем приступать к описанию предлагаемого способа триангуляции, рассмотрим стандартный алгоритм триангуляции в методе дуальных контуров [31,46]. Алгоритм ADC создаёт замкнутую и бесшовную треугольную сетку из знакоопределённого октодерева (signed octree), в котором размеры соседних листовых узлов могут различаться более чем в два раза (unrestricted octree).

2.3.1 Знакоопределённое октодерево

Октодерево [68,69] представляет собой древовидную структуру разбиения трёхмерного пространства, состоящую из внутренних (internal) и листовых (leaf) узлов. Каждый внутренний узел разделяет пространство на восемь октантов, и, соответственно, содержит восемь указателей на своих потомков. У внутреннего узла может быть от одного до восьми дочерних узлов. Листовые (т.е. терминальные, не имеющие потомков) узлы октодерева представляют собой ячейки (cells) разбиения пространства, участвующие в процессе триангуляции.

Построение октодерева для триангуляции алгоритмом ADC — это рекурсивный процесс, который может выполняться либо снизу вверх (bottom-up), либо сверху вниз (top-down). В первом случае октодерево строится до максимальной глубины разбиения и упрощается до достижения заданной ошибки аппроксимации путем объединения его листовых узлов [31,46]. Во втором случае, начиная с исходного, корневого узла, октодерево измельчается в необходимых местах, опираясь на различные эвристики [48,33]. Первый способ необходим для генерации уровней детализации при редактировании ландшафта, второй подход может быть использован для триангуляции процедурно сгенерированных ландшафтов. В обоих случаях листовые узлы-ячейки создаются только на границе области — все ячейки должны пересекать поверхность объекта (т.е.

не существует ячеек, которые находятся полностью внутри или полностью снаружи объекта). Каждая ячейка *cell* октодерева содержит вершину **v**, которая помещается в острый угол на поверхности внутри ячейки (рисунок 2.1). Для этого позиция вершины **v** должна минимизировать квадратичную функцию ошибки (Quadratic Error Function, QEF) (1.1) (подраздел 1.3.1) : сумму квадратов расстояний от **v** до касательных плоскостей *planes*(**v**), образованных координатами $\mathbf{s}_i$ точек пересечения активных рёбер ячейки *cell* с изоповерхностью и единичными нормалями $\mathbf{n}_i$ к поверхности в этих точках.

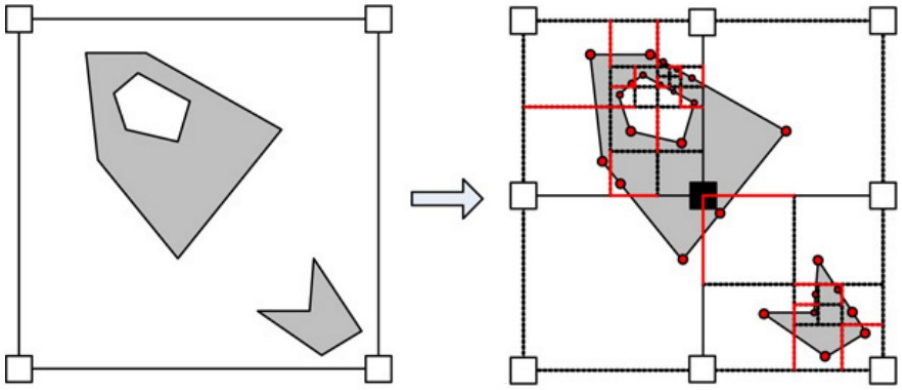


Рисунок 2.1 – Построение знакоопределённого октодерева для триангуляции алгоритмом ADC. Каждая граничная ячейка содержит позицию острой вершины на поверхности объекта.

В данной работе используется первый подход, основанный на упрощении октодерева снизу вверх [31,46]. Каждой вершине **v** (соответственно, ячейке *cell*) сопоставляется матрица **Q**, позволяющая вычислить сумму $E(\mathbf{v})$ квадратов расстояний от **v** до *planes*(**v**):

$$E(\mathbf{v}) = E\left(\begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}^{\mathrm{T}}\right) = \sum_{\mathbf{p}_i \in planes(\mathbf{v})} (\mathbf{p}_i{}^{\mathrm{T}}\mathbf{v})^2 = \mathbf{v}^{\mathrm{T}}\left(\sum_{\mathbf{p}_i \in planes(\mathbf{v})} \mathbf{p}_i\mathbf{p}_i^{\mathrm{T}}\right)\mathbf{v} = \mathbf{v}^{\mathrm{T}}\mathbf{Q}\mathbf{v}, \qquad (2.1)$$

где $\mathbf{p}_i = [a\ b\ c\ d]^{\mathrm{T}}$ представляет плоскость $ax + by + cz + d = 0$ (где a, b, c — компоненты единичной нормали $\mathbf{n}_i$ к плоскости $\mathbf{p}_i$ ($a^2 + b^2 + c^2 = 0$), d равно $-\mathbf{n}_i \cdot \mathbf{s}_i$),

а $\mathbf{Q} = \mathbf{p}_i\mathbf{p}_i^{\mathrm{T}} = \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{pmatrix}$ —квадратная симметричная 4x4 матрица.

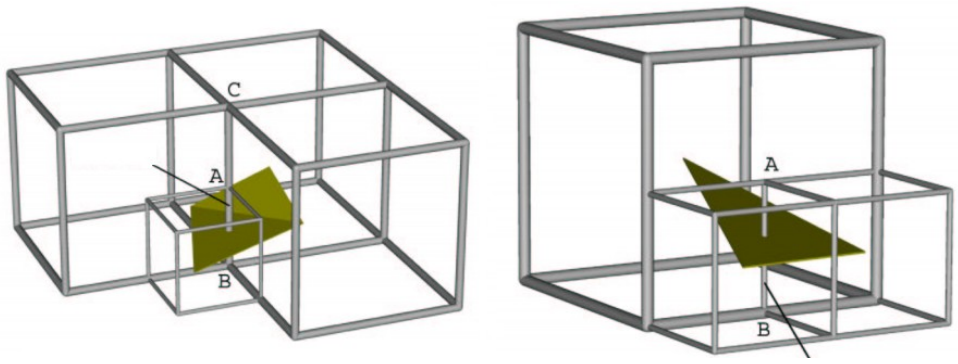
В процессе упрощения октодерева при слиянии ячеек $cell_i$ в $cell_{new}$ их матрицы $\mathbf{Q}_i$ складываются: $\mathbf{Q}_{new} = \sum_i \mathbf{Q}_i$. Полученная QEF-матрица $\mathbf{Q}_{new}$ позволяет оценить сумму квадратов расстояний от произвольной точки до плоскостей всех $cell_i$ [41–43]. Позиция вершины $\mathbf{v}_{new}$ новой ячейки $cell_{new}$ должна минимизировать квадратичную форму $E(\mathbf{v}_{new}) = \mathbf{v}_{new}^{\mathrm{T}}\mathbf{Q}_{new}\mathbf{v}_{new}$. Если

$E(\mathbf{v}_{new})$ превышает некоторое пороговое значение ε (QEF error threshold), то слияние ячеек запрещается. Таким образом, в алгоритме ADC для возможности упрощения октодерева для каждой его ячейки (или листового узла) хранится симметричная квадратная 4x4 матрица **Q** (для хранения **Q** достаточно 10 чисел). Нетрудно заметить, что **Q** содержит 3x3 матрицу ковариации нормалей $\mathbf{A} = \sum_i \mathbf{n}_i \mathbf{n}_i^{\mathrm{T}}$, которая используется для определения острых углов в подразделе 2.4.3.

В знакоопределённом октодереве в углах каждой кубической ячейки разбиения также известны знаки: значения «внутри»/«снаружи», которые могут компактно храниться в 8-битной маске. При описании составных областей для каждого угла ячейки обычно хранится не знак, а индекс подобласти (материал).

2.3.2 Стандартный алгоритм адаптивной триангуляции

Генерация полигональной сетки. В [31,46] показано, что для генерации замкнутой, «водонепроницаемой» сетки достаточно создать четырёхугольники только для *минимальных* активных рёбер октодерева (minimal edges), т.е. не содержащих полностью рёбра соседних ячеек. В октодереве минимальное ребро может входить в четыре или в три (adaptive case) смежные ячейки. Во втором случае минимальное ребро включается в три смежные ячейки различных размеров (т.е. располагающиеся на разных уровнях разбиения), и тогда возможно создание только одного треугольника (рисунок 2.2, б).



а) б)

Рисунок 2.2 – Триангуляция минимального ребра AB октодерева в алгоритме Adaptive Dual Contouring, приводящая к созданию (а) четырёхугольника и (б) треугольника.

Для нахождения минимальных рёбер ячеек октодерева в [31,46] предложена сложная рекурсивная процедура, включающая функции **CellProc(**n**)**, **FaceProc(**n1,n2**)** и

EdgeProc(n1,n2,n3,n4**)**, аргументами которых соответственно являются указатели на 1 узел, или 2 или 4 смежных узла октодерева (рисунок 2.3).

Для генерации полигональной сетки вызывается **CellProc** для корневого узла. Рекурсия завершается созданием четырёхугольника в функции **EdgeProc**, если все её аргументы — листовые узлы-ячейки.

Рекурсивная генерация сетки выполняется следующим образом:

– если аргументом **CellProc(** node **)** является внутренний узел, то 8 раз вызывается **CellProc** (для его 8 дочерних узлов), 12 раз **FaceProc** (для 4 пар потомков, имеющих смежные грани в трёх плоскостях XY, XZ и YZ) и 6 раз **EdgeProc** (для подузлов, включающих внутренние рёбра текущего узла);

– **FaceProc(** node1, node2 **)** вызывает 4 раза **FaceProc** (для 4 пар потомков, имеющих смежную грань в соответствующей плоскости) и 4 раза **EdgeProc** (для потомков, имеющих смежные грани в двух перпендикулярных плоскостях);

– если хотя бы один аргумент **EdgeProc(** node1, node2, node3, node4 **)** является внутренним узлом октодерева, то два раза вызывается **EdgeProc** (для двух пар потомков, имеющих смежную грань вдоль соответствующей плоскости);

– если все четыре аргумента **EdgeProc** являются листьями-ячейками, то находится минимальное ребро $\mathcal{E}$, которое принадлежит листовому узлу-ячейке наименьшего размера (или узлу с наибольшей глубиной залегания в октодереве). Если ребро $\mathcal{E}$ пересекает изоповерхность (т.е. его углы имеют разные значения «внутри»/«снаружи»), то для него создаётся четырёхугольник, соединяющий вершины четырёх смежных ячеек, включающих $\mathcal{E}$, и ориентированный наружу (в соответствии со знаками на концах ребра). Из-за адаптивной природы октодерева среди аргументов **EdgeProc** один и тот же узел может повторяться два раза, и тогда четырёхугольник вырождается в треугольник (рисунок 2.2, б). Это приводит к тому, что в переходных областях октодерева, где смежные ячейки имеют различные размеры, вершины, соответствующие ячейкам наименьшего размера, обладают высокой валентностью[1] (одна вершина может входить в большое количество треугольников).

На практике создание индексированной полигональной сетки путём ADC производится в два рекурсивных прохода октодерева. В первом проходе создаётся массив вершин (vertex buffer) полигональной сетки, и инициализируются индексы (порядковые номера) вершин, которые обычно хранятся в каждом листовом узле, помимо прочих данных. Во время второго прохода в процессе

[1] Валентностью или степенью вершины называется число инцидентных ей ребер.

выполнения вышеописанной процедуры перечисления минимальных рёбер генерируется массив индексов (index buffer) вершин — список треугольников, определяющий связность вершин в полигональной сетке.

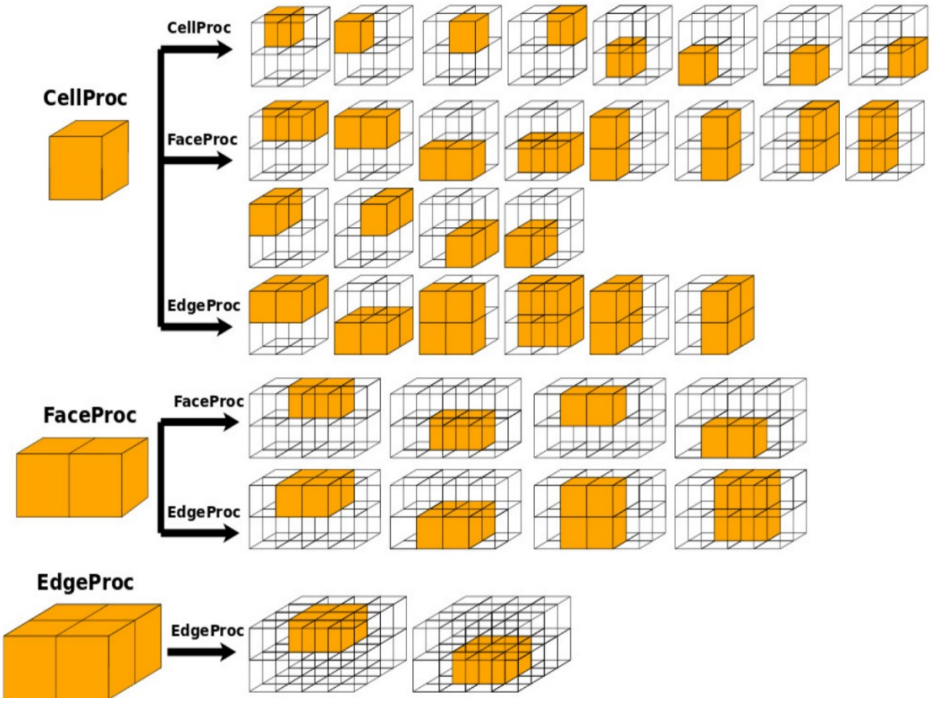


Рисунок 2.3 – Схема рекурсивного перечисления рёбер октодерева. Рисунок взят из [58].

Преимущества и недостатки алгоритма. Поскольку в процессе триангуляции единожды проходится каждый узел октодерева, то стандартный алгоритм триангуляции ADC обладает линейной сложностью, пропорциональной количеству узлов октодерева. Основным недостатком стандартного алгоритма триангуляции является громоздкость его практической реализации, включающей несколько рекурсивных функций с таблицами и ветвлениями для определения минимальных рёбер. Кроме того, исходный алгоритм триангуляции неприменим для внеядерной триангуляции данных, поскольку предполагает хранение всего октодерева в памяти. Задача триангуляции воксельных ландшафтов, является, по сути, задачей триангуляции с внешней памятью (поскольку ландшафт, как правило, не умещается в оперативную память целиком). Поэтому становится очевидной необходимость поиска более простых и эффективных алгоритмов для адаптивной триангуляции с внешней памятью.

2.4 Описание предлагаемого подхода к адаптивной триангуляции

В данном подразделе предлагается новый алгоритм триангуляции знакоопределённого октодерева, построенный на следующих идеях: 1) в процессе триангуляции участвуют только листовые узлы-ячейки октодерева; 2) если обходить ячейки октодерева по порядку возрастания их размеров, то можно избавиться от сложной рекурсивной процедуры нахождения минимального ребра.

Новый алгоритм использует линейное представление октодеревьев (linear octree representation) [67,69], что позволяет значительно уменьшить расход памяти и заменить сложную рекурсивную процедуру обхода октодерева более простой итерацией. В [64] предложены алгоритмы для построения двойственных решёток над линейными октодеревьями, которые могут быть использованы для триангуляции изоповерхностей методом PCDG [62]. Наиболее близкими к описанному алгоритму являются алгоритмы внеядерной триангуляции Adaptive Dual Marching Cubes (ADMC) [49,58].

2.4.1 Линейное представление октодеревьев

Линейное октодерево (Linear Octree) [67,69] не содержит указателей, а состоит только из листовых узлов октодерева и их уникальных идентификаторов (locational codes). Уникальный идентификатор или *адрес* узла — это битовая строка, хранящая путь, по которому можно дойти до данного листового узла, начиная с корня дерева. В качестве уникальных идентификаторов на практике чаще всего используются *коды Мортона* (*Morton codes*) [70], как самые дешёвые с вычислительной точки зрения. Каждому октанту (или квадранту в двумерном случае) присваивается 3-битный (или, соответственно, 2-битный в 2D) порядковый номер (рисунок 2.4).

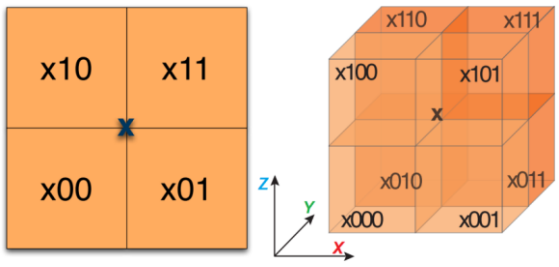


Рисунок 2.4 – Нумерация (слева) квадрантов и (справа) октантов для построения кодов Мортона в квадродереве и октодереве соответственно. Для получения адреса дочернего узла, расположенного в квадранте (октанте) Q, номер Q добавляется к адресу **x** родителя. Рисунок взят из [64].

Код корневого узла полагается равным **1**, а адрес произвольного дочернего узла может быть получен конкатенацией адреса родительского узла и октанта (квадранта в 2D), в котором расположен данный дочерний узел (рисунок 2.5).

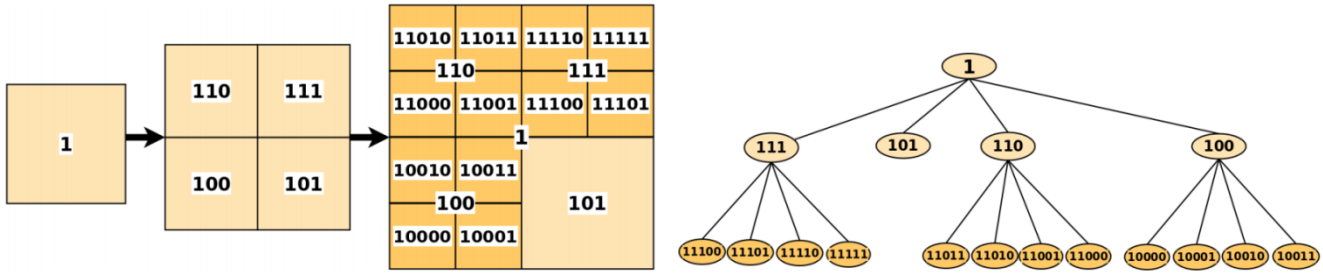


Рисунок 2.5 – Слева: использование 2-битных кодов Мортона для построения линейного квадродерева. Справа: эквивалентное квадродерево на указателях. Изображения взяты из [58].

Адрес P родительской ячейки получается удалением последних 3 бит (2 бит в двумерном случае) адреса M текущей ячейки: $P = M >> 3$. Позиция k старшего бита в адресе M и глубина d ячейки в октодереве связаны соотношением: $d = k / 3$.

Код Мортона M также может быть построен из координат (x,y,z) угла или центра ячейки в регулярной решётке разбиения с разрешением n+1 ячеек путём последовательного чередования бит дискретных координат: $M = z_n y_n x_n \ ... \ z_1 y_1 x_1 \ z_0 y_0 x_0$. *Линеаризованные* таким способом координаты ячеек располагаются вдоль линии, называемой кривой Лебега, кривой Мортона или *Z-кривой* (*Z-curve*) (рисунок 2.6).

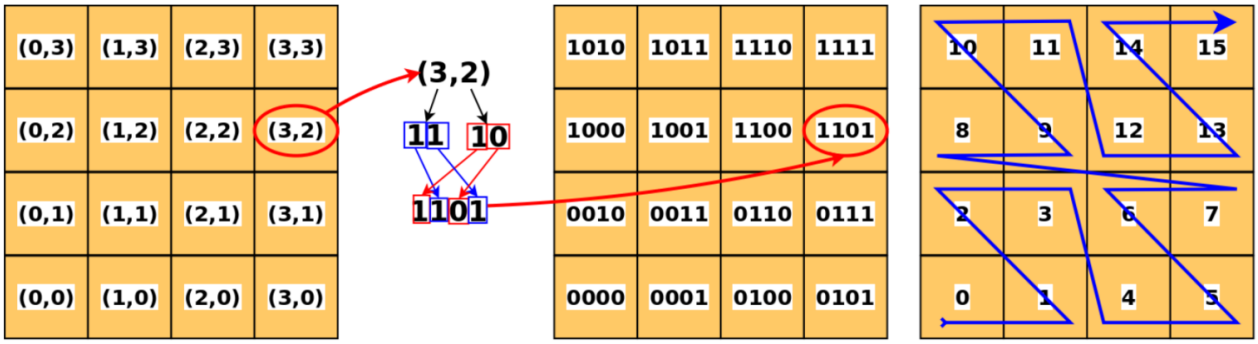


Рисунок 2.6 – Для построения кода Мортона ячейки на регулярной решётке достаточно перемешать координаты ячейки в битовом представлении. Полученные коды образуют Z-кривую. Рисунок взят из [58].

Коды Мортона обладают замечательным и полезным свойством: ячейки с близкими адресами, как правило, находятся близко друг к другу на Z-кривой (и в пространстве). Свойство пространственной когерентности (spatial coherency) часто применяется для повышения

производительности систем с кэшированием данных за счёт увеличения локальности доступа к данным, независимо от размера кэша (такие алгоритмы называют «cache oblivious»). Линейное представление октодеревьев хорошо подходит для эффективного поиска ближайших соседей (nearest neighbor searching) [69,71,64], который лежит в основе предложенного алгоритма триангуляции. Линейное октодерево, для каждой ячейки которого известны знаки её углов (значения «внутри»/«снаружи»), будем называть знакоопределённым линейным октодеревом (Signed Linear Octree).

2.4.2 Алгоритм триангуляции знакоопределённого линейного октодерева

Входными данными для предложенного алгоритма триангуляции является знакоопределённое линейное октодерево, представляющее собой одномерный массив из N ячеек. Помимо стандартного набора данных, которым должен обладать листовой узел (позиция острой вершины поверхности внутри ячейки, знаки в углах ячейки и т.д.), каждая ячейка содержит соответствующий ей код Мортона, который используется как ключ для сортировки и адрес для поиска [72].

Генерация полигональной сетки. Алгоритм построения индексированной полигональной сетки из знакоопределённого линейного октодерева выполняется в три шага:

1) сортировка ячеек по убыванию соответствующих кодов Мортона;

2) создание вершин полигональной сетки поверхности;

3) создание четырёхугольников для активных рёбер ячеек.

Шаг 1: сортировка ячеек по мортон-кодам в убывающем порядке.

На первом шаге все ячейки октодерева сортируются по кодам Мортона в порядке убывания. В силу конструкции мортон-кодов (позиция k старшего бита адреса ячейки зависит от глубины d залегания ячейки в октодереве: $k = 3d$) ячейки наименьшего размера (или ячейки, расположенные на максимальной глубине измельчения октодерева) окажутся в самом начале отсортированного массива.

Данный шаг требуется выполнять только после модификации октодерева. Статичные октодеревья могут храниться в отсортированном виде (см. раздел 3.6).

Шаг 2: создание вершин полигональной сетки.

На втором шаге генерируется массив вершин полигональной сетки.

Для каждой ячейки октодерева создаётся соответствующая ей вершина полигональной сетки (при использовании DMC [56] будет создано от одной до четырёх вершин, в зависимости от знаковой конфигурации ячейки). Поскольку для экономии памяти координаты вершин обычно хранятся в квантованном виде (относительно ограничивающей оболочки (AABB) ячейки) (см. раздел 3.6), на этом шаге происходит расжатие позиций вершин. AABB ячейки может быть вычислен из мортон-кода ячейки и AABB октодерева (см. Приложение Б).

На данном шаге рекурсивная процедура обхода октодерева, используемая в оригинальном алгоритме для создания вершинного буфера, заменена более эффективной итерацией.

Шаг 3: создание четырёхугольников для активных рёбер ячеек.

На третьем шаге генерируется массив индексов полигональной сетки.

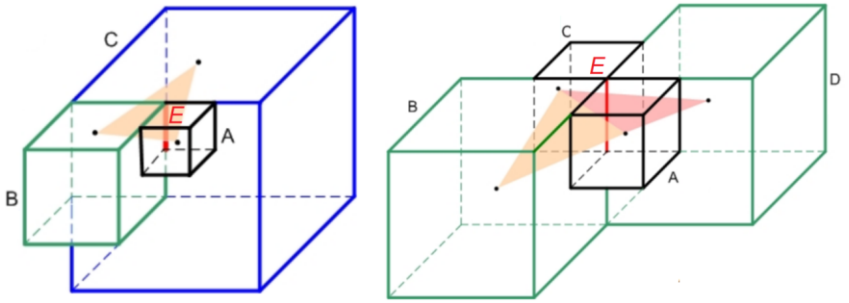
Рассматривается каждая ячейка A с мортон-кодом M_A и порядковым номером i (i варьируется в диапазоне [1..N]). Для каждого активного ребра $\mathcal{E}$ текущей ячейки A ищутся остальные три ячейки B, C и D наименьшего размера, включающие данное ребро и смежные к A. Для поиска смежной к A ячейки X (B, C или D) на основе M_A строится её мортон-код M_X. Если M_X является невалидным адресом (указывает на несуществующую ячейку, находящуюся за пределами октодерева), то создание четырёхугольника уже невозможно, и рассматривается следующее активное ребро текущей ячейки A.

Чтобы пропустить текущую ячейку A и все предыдущие, уже пройденные ячейки (для предотвращения повторного создания одинаковых полигонов), поиск ячейки X с адресом M_X должен стартовать со следующей ячейки (т.е. индекс ячейки X должен находиться в интервале [i+1..N]).

Если хотя бы одна ячейка из B, C и D, включающая текущее ребро $\mathcal{E}$, не найдена, то создание четырёхугольника для ребра $\mathcal{E}$ становится невозможным, и тогда рассматривается следующее активное ребро текущей ячейки A.

Если найдены остальные три смежные к A и включающие ребро $\mathcal{E}$ ячейки: B, C и D, то для ребра $\mathcal{E}$ создаётся четырёхугольник, соединяющий вершины ячеек A, B, C и D. Как и в оригинальном алгоритме ADC, поиск соседней ячейки может дважды вернуть одну и ту же смежную к A ячейку X (когда размер X превышает размер A), и тогда четырёхугольник вырождается в треугольник (рисунок 2.7, а).

Если рассмотрены все активные рёбра (из 12 возможных) текущей ячейки A, то алгоритм переходит к следующей по порядку ячейке A_{i+1} (с размером равным или превышающим размер ячейки A, потому что все ячейки отсортированы по размеру). Если знаки смежных ячеек октодерева согласованы друг с другом, то рассмотрев все ячейки, алгоритм сгенерирует закрытую полигональную сетку.



а) б)

Рисунок 2.7 – Триангуляция минимального ребра E, приводящая к созданию а) одного и б) двух треугольников. В обоих случаях текущая ячейка A не превышает по размеру смежные ячейки.

Пояснение на примере. Рассмотрим работу алгоритма в двумерном случае.

Пусть требуется восстановить контур некоторого объекта. В двумерном случае контур аппроксимируется ломаными линиями, а четырёхугольники становятся отрезками ломаных. Построим решётку, вершинам которой приписаны значения «внутри»/«снаружи» объекта, а на рёбрах отмечены точки пересечения с границей объекта и единичные нормали к границе в этих точках (рисунок 2.8, а). Из решётки построим максимально измельчённое знакоопределённое октодерево и упростим его до требуемой точности путём рекурсивного слияния листовых узлов (см. подраздел 2.3.1). Оставив только листовые узлы-ячейки и присвоив им уникальные адреса в виде кодов Мортона, получим знакоопределённое линейное октодерево (рисунок 2.8, б), над которым и будет выполняться алгоритм построения контура.

Первый шаг: сортировка ячеек.

Отсортируем ячейки в порядке убывания соответствующих адресов (кодов Мортона), тогда ячейки наименьшего размера (расположенные на максимальной глубине) будут помещены в начало отсортированного массива (рисунок 2.9, а).

Второй шаг: создание вершин контура.

Создадим массив вершин контура из соответствующих им острых вершин ячеек F, E, D, C, B, A (т.е. порядок вершин совпадает с порядком ячеек в отсортированном массиве).

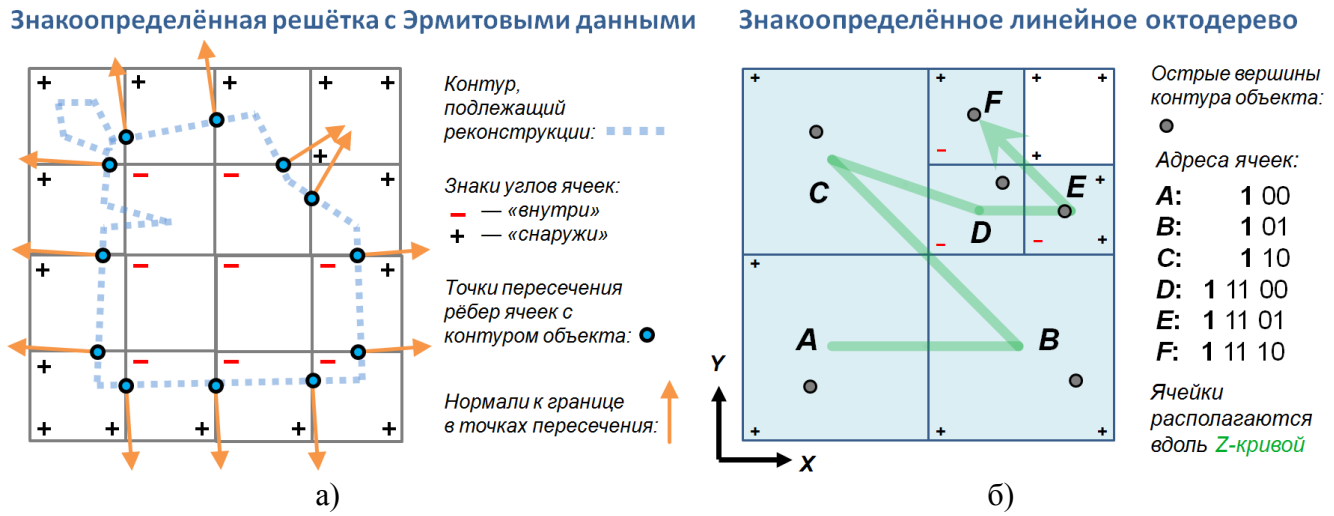


Рисунок 2.8 – Линейное октодерево (б), построенное из знакоопределённой решётки (а). (Из-за недостаточного разрешения решётки разбиения были пропущены мелкие детали объекта в ячейке *C*.)

Третий шаг: построение линий контура.

Рассмотрим каждую ячейку отсортированного массива.

Первая ячейка F имеет адрес $M_F = \mathbf{1}\ 11\ 10_2 = M(d = 2; y = 11_2; x = 10_2)$,

где d − глубина (уровень) ячейки в дереве, x и y − делинеаризованные координаты ячейки на текущем уровне, а $M(x, y)$ − функция построения мортон-кода ячейки, которая перемешивает биты координат и приписывает слева бит глубины (в данном случае его позиция $k = d \cdot 2 = 4$).

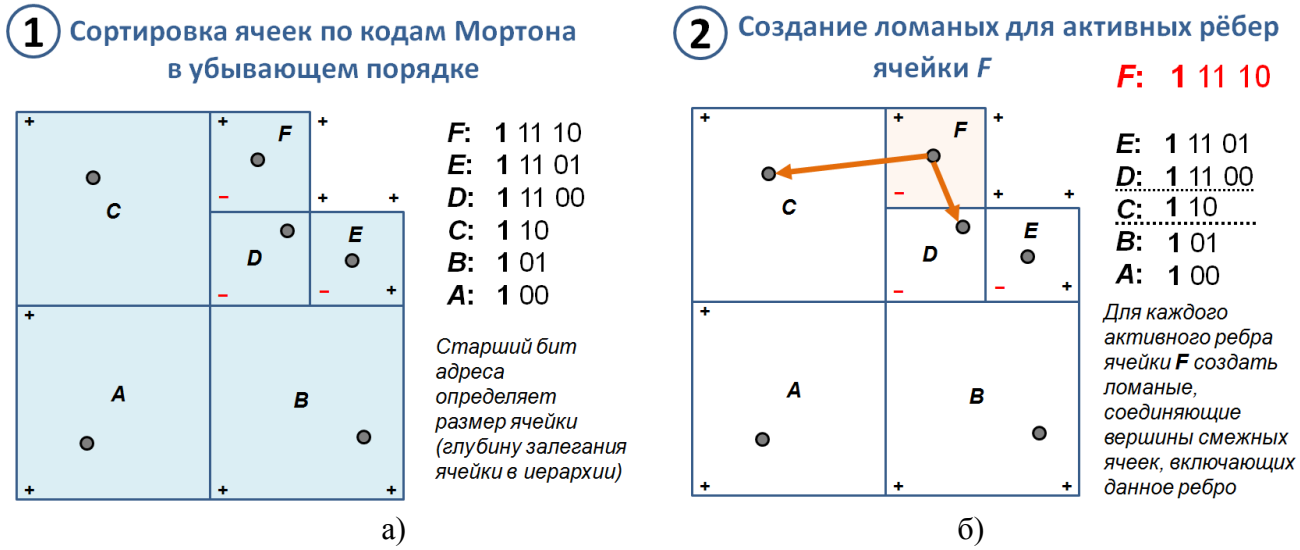
Ячейка F имеет два активных ребра: слева и снизу. Найдём смежную к F ячейку X наименьшего размера, включающую нижнее активное ребро. Построим адрес M_X смежной ячейки X, отняв от адреса $M_F = \mathbf{1}\ 11\ 10_2$ ячейки F единичное смещение $I_Y = M(y\colon 1;\ x\colon 0) = \underline{1}0_2$ вдоль вертикальной оси Y (нечётные биты):

$$M_X = M_F - I_Y = \mathbf{1}\ 11\ \underline{1}0_2 - \underline{1}0_2 = M(d\colon 2;\ y\colon \underline{11}_2;\ x\colon 10_2) - M(y\colon \underline{1};\ x\colon 0) =$$
$$= M(d\colon 2;\ y\colon \underline{10}_2;\ x\colon 01_2) = \mathbf{1}\ 11\ \underline{00}_2.$$

Среди оставшихся ячеек такой адрес имеет ячейка D, поэтому соединим вершины контура, соответствующие смежным ячейкам F и D, отрезком ломаной.

Построим адрес M_X смежной ячейки X, включающей активное ребро слева, отняв от адреса M_F ячейки F горизонтальное смещение $I_X = M(y\colon 0;\ x\colon 1) = 0\underline{1}_2$:

$$M_X = M_F - I_X = \mathbf{1}\ \underline{11}\ \underline{1}0_2 - 0\underline{1}_2 = M(d\colon 2;\ y\colon 11_2;\ x\colon \underline{10}_2) - M(y\colon 0;\ x\colon \underline{1}) =$$
$$= M(d\colon 2;\ y\colon 11_2;\ x\colon \underline{01}_2) = \mathbf{1}\ \underline{10}\ \underline{11}_2.$$



а) б)

Рисунок 2.9 – Иллюстрация работы алгоритма триангуляции линейного октодерева: а) сортировка ячеек по адресам; б) создание элементов контура для активных рёбер первой ячейки *F*.

На текущем уровне ($d = 2$) дерева ячейки с таким адресом не существует, поэтому продолжим поиск на следующем уровне ($d = 1$), т.е. будет искать ячейку под адресом **1** 10_2. По этому адресу находится *C*, поэтому соединим вершины ячеек *F* и *C* (рис. 2.9, б).

а) б)

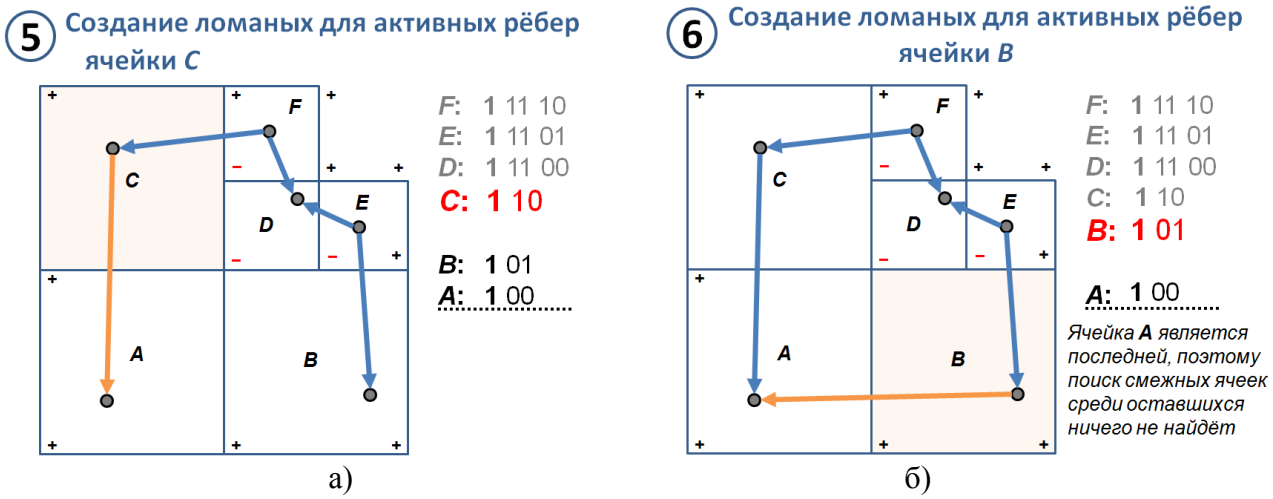
Рисунок 2.10 – Иллюстрация работы алгоритма триангуляции (продолжение):
а) создание элементов контура для активных рёбер ячейки *E*; б) рассмотрение ячейки *D*.

Следующая ячейка *E* также имеет активные рёбра слева и снизу, которым соответствуют смежные ячейки *D* и *B*. Соединим вершину *E* с вершинами *D* и *B* (рисунок 2.10, а). Рассмотрим следующую по порядку ячейку *D*. Она имеет два активных ребра (справа и сверху), но поскольку

соответствующие им смежные ячейки (*E* и *F*) уже были пройдены, то поиск смежных ячеек (среди оставшихся ячеек) ничего не найдёт, и новых отрезков создано не будет (рисунок 2.10, б).

Аналогичные процедуры выполняются для ячеек *C* и *B*, имеющих по два активных ребра (рисунок 2.11, а). Ячейка *A* имеет два активных ребра (сверху и справа), но поскольку она является последней, то поиск ячеек, соответствующих её активным рёбрам, ничего не найдёт среди оставшихся ячеек (рисунок 2.11, б).

В результате работы алгоритма получается кусочно-линейное приближение исходного контура (рисунок 2.9, а) с восстановлением его острых углов.



а) б)

Рисунок 2.11 – Иллюстрация работы алгоритма триангуляции (продолжение): создание элементов контура для активных рёбер а) ячейки *C* и б) ячейки *B*.

Надо сказать, что в данном упрощённом примере контур объекта целиком расположен «внутри» октодерева, что довольно редко встречается на практике. Обычно при триангуляции октодерева, построенного из отдельного блока воксельного ландшафта, получается поверхность с открытой границей[2]. (Примеры треугольных сеток с открытой границей показаны на рисунках 4.11 — 4.15.) Если ячейка лежит на границе октодерева, то возможна ситуация, когда адрес, составленный для поиска смежной к ней ячейки, указывает на несуществующую ячейку (находящуюся за пределами октодерева). В новом алгоритме проверка валидности адреса ячейки (проверки мортон-кода на переполнение (overflow) и выход за нижнюю границу (underflow)) реализована проще и эффективней (одной битовой операцией и ветвлением, см. Приложение Б), чем в аналогичных алгоритмах [64,49]: длина адреса (в битовом представлении) искомой ячейки не

[2] Под границей триангулированной поверхности понимается множество ребер, на каждое из которых опирается только по одному треугольнику. Вершины, инцидентные указанным ребрам, называются граничными.

должна превышать длину адреса текущей ячейки, потому что все ячейки отсортированы по убывающим адресам (соответствующим кодам Мортона).

На рисунке 2.12 приведёна принципиальная блок-схема алгоритма триангуляции.

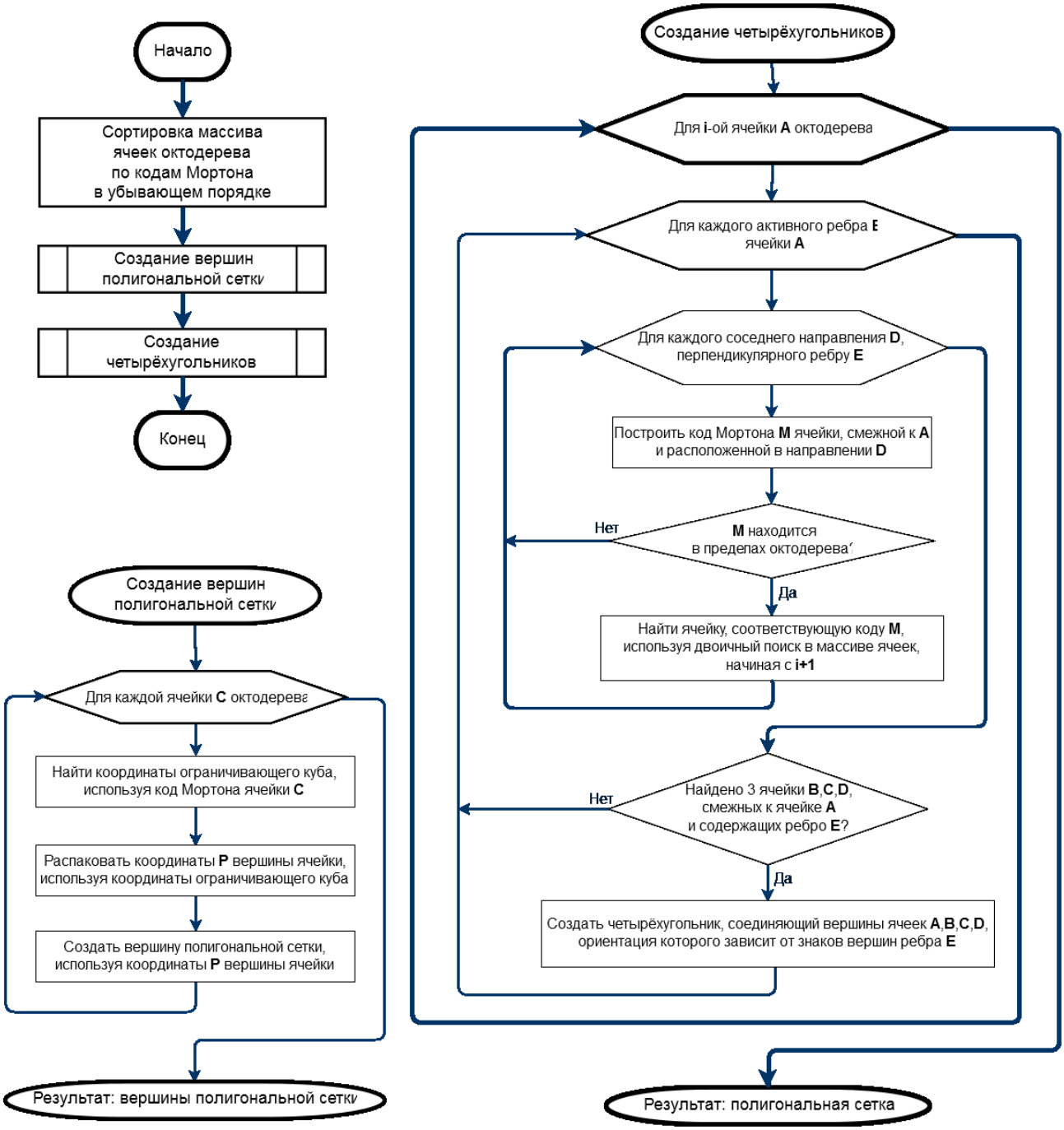


Рисунок 2.12 – Блок-схема предложенного алгоритма триангуляции

Преимущества и недостатки предложенного алгоритма. Во-первых, можно отметить простоту предложенного алгоритма. Оригинальный алгоритм триангуляции в ADC является нисходящим рекурсивным: начиная с корневого узла октодерева, алгоритм рассматривает все внутренние рёбра октодерева. В предложенном восходящем нерекурсивном алгоритме рассматриваются только активные рёбра каждой ячейки. При этом, в отличие от оригинального алгоритма триангуляции, в новом алгоритме при создании четырёхугольника не требуется определять ячейку наименьшего размера для нахождения минимального ребра. Все ячейки октодерева отсортированы по размеру в порядке возрастания, поэтому текущая ячейка *A* никогда не превышает по размеру смежные к ней ячейки *B*, *C, D* и, таким образом, всегда полностью содержит минимальное ребро (см. рисунок 2.7).

В предложенном алгоритме всегда выполняется только последовательный доступ к памяти, что оптимальным образом использует механизмы кэширования памяти. Кроме того, новый алгоритм является более эффективным и простым в реализации для триангуляции по частям сверхбольших воксельных ландшафтов, которые не умещаются целиком в оперативной памяти (см. раздел 4.3). Основная идея алгоритма внеядерной триангуляции ландшафта заключается во включении в процесс триангуляции каждого блока приграничных ячеек его соседей. Для этого из ячеек блока и прилегающих к нему ячеек из соседних блоков строится октодерево, над которым и выполняется алгоритм триангуляции. В отличие от традиционных октодеревьев на указателях, построение линейного октодерева не требует операций выделения динамической памяти для создания новых узлов октодерева и манипуляций с указателями для вставки ячеек в нужные его ветви. Создание октодерева сводится к построению новых адресов (Мортон-кодов) ячеек с помощью битовых операций и сортировке полученного массива ячеек.

Из-за использования кодов Мортона и сортировки предложенный алгоритм создаёт полигональные сетки с более высокой пространственной когерентностью по сравнению с сетками, полученными стандартным подходом. Это повышает степень локальности доступа к вершинным данным при растеризации сетки, что, в свою очередь, увеличивает эффективность кэша вершин и скорость отрисовки. Кэш трансформированных вершин (post-transform vertex cache) — это устройство графического процессора (GPU), запоминающее результаты выполнения вершинного шейдера (GPU-программы, обрабатывающей данные вершин). Если при рендеринге индексированной сетки обработанная вершина уже присутствует в кэше (индекс вершины — ключ для поиска в кэше), то для неё не требуется выполнять вершинный шейдер. Кэш имеет ограниченный размер, поэтому для его оптимального использования нужно, чтобы в

индексированной полигональной сетке на каждую вершину ссылалось как можно большее количество смежных полигонов. Оптимизация треугольных сеток под вершинный кэш путём их организации в виде длинных полос треугольников (triangle strips) или вееров (triangle fans) часто используется на практике для ускорения рендеринга, а также для снижения размера сеток в памяти [73]. В [74] описан простой алгоритм оптимизации треугольных сеток для рендеринга перестановкой треугольников и вершин по порядку мортон-кодов вершин. Применение предложенного алгоритма для триангуляции может дать ощутимый выигрыш при рендеринге результатов научного моделирования или воксельного ландшафта с большим количеством треугольников и «тяжёлым» вершинным шейдером[3]. В данной работе свойство пространственной когерентности данных ячеек, упорядоченных вдоль Z-кривой, используется для повышения коэффициента сжатия линейных октодеревьев (см. раздел 3.6).

Из недостатков нового алгоритма в первую очередь следует отметить его более высокую вычислительную сложность (квадратичную) по сравнению с $\mathcal{O}(N)$ сложностью оригинального алгоритма, где N — число ячеек октодерева. В предложенном алгоритме определяющей является сложность операции поиска смежных ячеек, которая составляет $\mathcal{O}(N \cdot h)$ для линейного поиска, но может быть приближена до $\mathcal{O}(h \cdot \log_2 N)$ использованием бинарного поиска и до $\mathcal{O}(h)$ с помощью хэш-таблицы, где h — средняя глубина (количество уровней) линейного октодерева.

Также, теоретически возможна ситуация, когда знаки ячеек не согласованы друг с другом, тогда алгоритм создаст сетку с изолированными вершинами. Кроме того, хотя построенные сетки и обладают повышенной пространственной когерентностью, они не оптимизированы для повышения эффективности буфера глубины (уменьшения overdraw).

Дополнительные оптимизации. Несложно видеть, что на третьем шаге алгоритма можно не рассматривать последнюю ячейку, потому что смежные (к текущей ячейке) ячейки всегда ищутся только среди оставшихся ячеек.

Поскольку все ячейки октодерева отсортированы по их адресам, то для нахождения ячеек разумно использовать целочисленный бинарный поиск. Для ускорения бинарного поиска можно использовать вспомогательную таблицу, которая для каждого уровня октодерева указывает начальное смещение и количество ячеек, или хранить ячейки для каждого уровня в отдельном массиве. За счёт дополнительных затрат памяти можно сократить асимптотическое время поиска до $\mathcal{O}(h)$ с помощью хэш-таблицы [48,49,64]. Построение адресов для поиска ячеек целесообразно

[3] Для корректного освещения сетки, содержащей и острые углы, и гладкие участки, необходимо «разбивать» и дублировать её острые вершины, что снижает эффективность вершинного кэша. Новым вершинам присваиваются нормали инцидентных к ним граней, поскольку нормали к поверхности не определены на её острых углах.

выполнять непосредственно над мортон-кодами в «разреженной» (dilated) форме [75,76,64], без перевода мортон-кодов в соответствующие координаты и обратно.

В нашем алгоритме всегда выполняется процедура поиска смежных ячеек, включающих активные рёбра текущей ячейки, даже если эти смежные ячейки уже были пройдены (см. рис. 2.10, б). Для предотвращения «холостого» поиска можно создать для каждой ячейки 12-битную маску *Edges*, ненулевые биты которой указывают на номера активных рёбер ячейки, для которых ещё не были созданы четырёхугольники. При обработке очередной ячейки *A* из её маски *Edges* и из масок *Edges* всех найденных (смежных к *A*) ячеек удаляются рёбра, для которых были созданы четырёхугольники [77,49]. Использование маски активных рёбер не только является мощной оптимизацией, но и также позволяет обрабатывать все ячейки в произвольном порядке без риска повторного создания одинаковых полигонов.

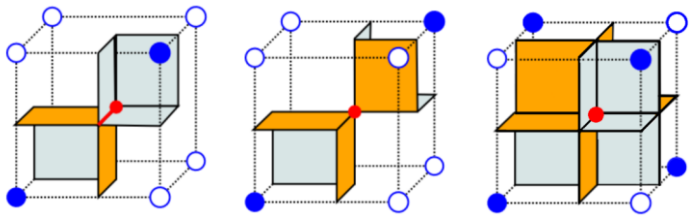
В Приложении Б приведены псевдокод и программная реализация базовой версии предложенного алгоритма, включающая следующие оптимизации: бинарный поиск, составление мортон-кодов в «разреженной» форме и использование масок активных рёбер.

2.4.3 Улучшение качества полученной полигональной сетки

Улучшенная реконструкция тонких стенок и создание развёртываемых полигональных сеток. Поскольку в каждой активной ячейке DC создаёт только по одной вершине, то построенная полигональная поверхность не всегда является развёртываемой (2-manifold). Ячейки с неоднозначной топологией (с наличием более одной связной компоненты) будем называть *неоднозначными* (*ambiguous*) (при триангуляции таких ячеек алгоритмом марширующих кубиков возникают неоднозначности выделения замкнутых контуров на гранях (face ambiguities) или внутри ячейки (internal/voxel ambiguities)). Неоднозначные ячейки появляются в результате недостаточного разрешения решётки разбиения. В неоднозначных ячейках DC создаёт сингулярные вершины или сингулярные рёбра (рисунок 2.13).

В алгоритме двойственных марширующих кубиков (DMC) [56] для каждой неоднозначной ячейки создаётся от двух до чётырёх вершин, по одной вершине для каждой связной компоненты. Например, в кубической ячейке с четырьмя связными компонентами (с 12 активными рёбрами, рисунок 2.13, в) будет создано четыре вершины (подраздел 1.3.3). Поэтому DMC в большинстве случаев создаёт развёртываемые полигональные сетки с двусторонней топологией [53,48,35] и при

одинаковом разрешении решётки разбиения способен реконструировать более мелкие геометрические детали изоповерхности, чем DC.



а) б) в)

Рисунок 2.13 – Примеры знаковых конфигураций ячеек, в которых DC создаёт (а) сингулярные рёбра и (б,в) сингулярные вершины. Изображения взяты из [58].

В ходе исследования был разработан простейший способ, позволяющий в каждой неоднозначной ячейке оценить позиции вершин, соответствующих каждой связной компоненте, на основе координат единственной вершины, созданной для триангуляции методом DC. Это, в свою очередь, позволяет использовать DMC для триангуляции линейного октодерева, «заточенного» под DC. В неоднозначных ячейках каждую вершину будем помещать в центроид точек пересечения соответствующих ей активных рёбер ячейки с поверхностью. Поскольку в линейном октодереве не хранятся в явном виде координаты точек пересечения рёбер ячеек с поверхностью, будем использовать точки пересечения активных рёбер ячеек с построенной в результате триангуляции полигональной сеткой, как при выполнении булевых операций над точечными представлениями с неявной связностью [79] (см. раздел 3.6).

Алгоритм триангуляции линейного октодерева методом DMC аналогичен алгоритму триангуляции методом DC, описанному в подразделе 2.4.2. На втором шаге алгоритма в неоднозначных ячейках для каждой вершины создаётся счётчик количества смежных к ней вершин, а её позиция обнуляется. На третьем шаге после создания четырёхугольника рассматриваются четыре смежные ячейки, включающие текущее активное ребро. Если хотя бы одна их них является неоднозначной, то находится точка пересечения текущего активного ребра с четырёхугольником. Координаты точки пересечения добавляются к позициям вершин неоднозначных ячеек, и соответствующие счётчики увеличиваются. После генерации полигональной сетки координаты вершин в неоднозначных ячейках усредняются, используя значения соответствующих счётчиков.

На рисунке 2.14 приводится сравнение результатов реконструкции тонкой стенки с помощью методов DC, DMC и предложенного гибридного подхода.

Предложенный подход генерирует полигональные сетки с улучшенным визуальным качеством по сравнению с сетками, полученными методом DC, при незначительном снижении производительности. (Зазубренные края являются результатом «обрезания» (clamping) острых вершин, выходящих за границы ячеек в двойственных методах). В частности, новый подход способен обеспечить разделение более мелких деталей и создаёт сетки с более гладкой поверхностью.

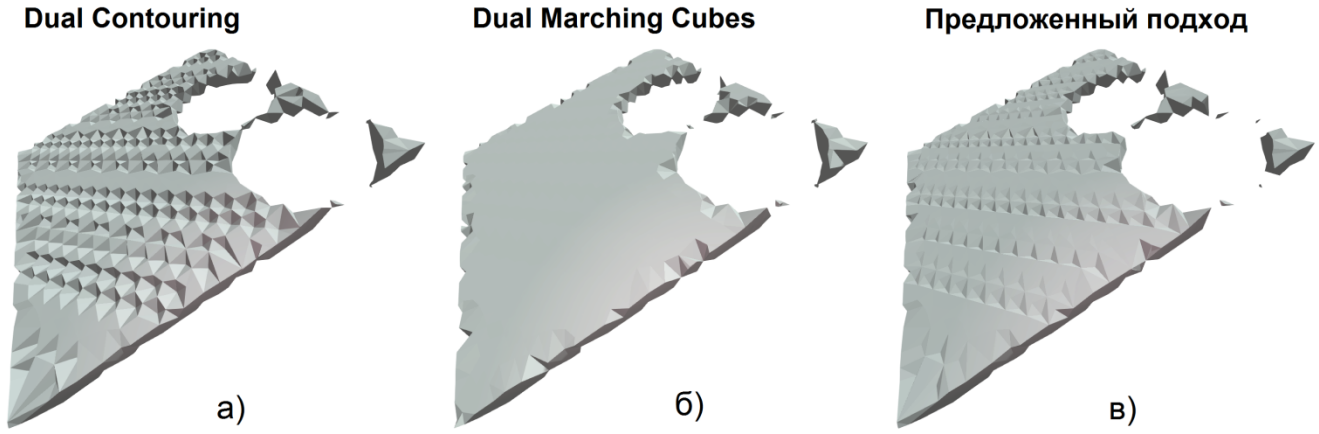


Рисунок 2.14 – Результаты триангуляции тонкой наклонной стенки с помощью а) DC, б) DMC и в) гибридного подхода. DC создаёт только по одной вершине в неоднозначных ячейках, поэтому реконструированная поверхность имеет «ячеистый» вид. В DMC для локализации вершин используются данные на соответствующих активных рёбрах. В гибридном подходе позиции вершин в неоднозначных ячейках оцениваются на основе координат единственной вершины.

Сохранение острых углов и рёбер поверхности. На кубической решётке разбиения двойственные методы создают четырёхугольники, вершины обычно которых не лежат в одной плоскости. Для получения треугольной сетки каждый полученный четырёхугольник необходимо разбить на два треугольника. Чтобы восстановить острые рёбра поверхности, четырёхугольник нужно разбить по диагонали, соединяющей острые углы поверхности [80]. Без соблюдения этого правила острые рёбра могут получиться «иззубренными» и «сколотыми».

Данной проблеме наиболее подвержены алгоритмы триангуляции на регулярных решётках (Uniform DC/DMC), поскольку они, в отличие от адаптивных вариантов, всегда генерируют четырёхугольники. В ADC октодерево сгущается к поверхности, и в переходных областях создаются треугольники, причём острые углы, как правило, являются вершинами этих треугольников. Поэтому появление «сколотых» рёбер менее вероятно. Стоит отметить, что данная

проблема отсутствует в алгоритме CMS [33] и в том случае, когда при дискретизации пространства используется симплициальный комплекс (тетраэдральная решётка в 3D).

В настоящей работе разбиение четырёхугольников для сохранения острых рёбер поверхности было реализовано следующим образом. Ячейки, содержащие острые углы, помечаются флагом. Для определения ячеек, содержащих острые рёбра или углы поверхности, могут использоваться два способа:

1) оценка открытого угла конуса нормалей (см. подраздел 1.3.1) [40];

2) определение ранга матрицы, составленной из нормалей [80].

В [40] утверждается, что первый способ является более надёжным, чем второй.

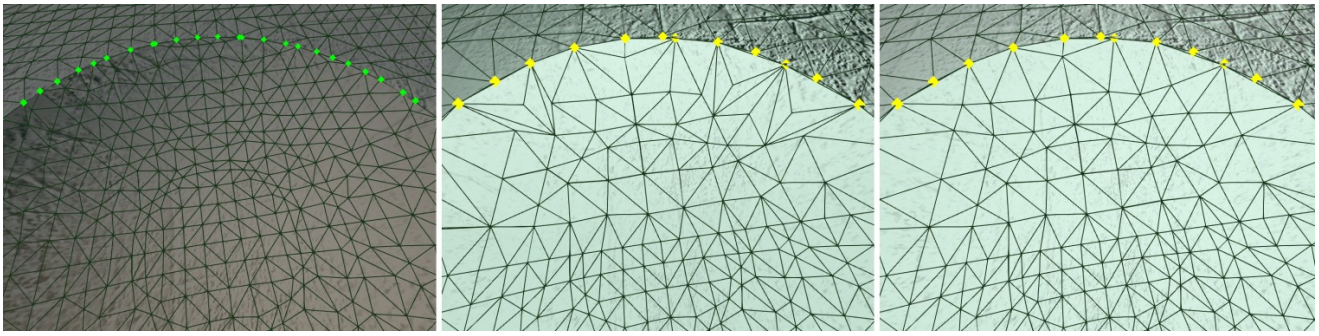
Во втором способе находятся собственные значения ($\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0$) симметричной 3x3 матрицы **A** нормалей, составленной из нормалей $\mathbf{n}_i$ в точках пересечения активных ребёр ячейки с изоповерхностью: $\mathbf{A} = \sum_i \mathbf{n}_i \mathbf{n}_i^{\mathrm{T}}$. Те значения λ, которые меньше некоторого порогового значения ε (в работах [31,46] предложено использовать $\varepsilon = 0.1$), полагаются равными нулю. Если все три собственных числа больше нуля, то вершина ячейки лежит на остром углу поверхности, если **A** имеет два ненулевых собственных числа, то вершина расположена на остром ребре поверхности, иначе вершина расположена на плоском участке поверхности. Если для нахождения позиций острых углов используется сингулярное разложение (SVD) матрицы **A** нормалей, то её собственные числа доступны как побочные продукты его вычисления, поэтому в работе используется этот способ.

При разбиении (невырожденного) четырёхугольника флаги его вершин комбинируются в четырёхбитовую маску *F*. Если все вершины четырёхугольника острые (*F = 0xFF*) или «тупые» (*F = 0*), или только одна вершина является острой, то четырёхугольник разбивается по кратчайшей диагонали для улучшения качества треугольников (рисунок 2.15).

В случае смешанных вершин *F* служит индексом в таблицу, определяющую диагональ разбиения четырёхугольника для каждой комбинации *F*. Поскольку вершины четырёхугольника обычно не лежат в одной плоскости, то полученная таким образом диагональ может пересекаться с треугольниками, образованными делением других четырёхугольников. Стоит отметить, что в некоторых реализациях воксельных ландшафтов, использующих для триангуляции алгоритмы Surface Nets / Dual Contouring, для уменьшения артефактов[4] освещения, связанных с интерполяцией вершинных нормалей, диагональ для разбиения четырёхугольника выбирается

[4] Под артефактами в компьютерной графике понимаются визуально-различимые нежелательные особенности сгенерированного компьютером изображения.

таким образом, чтобы максимизировать скалярное произведение нормалей получившихся треугольников.

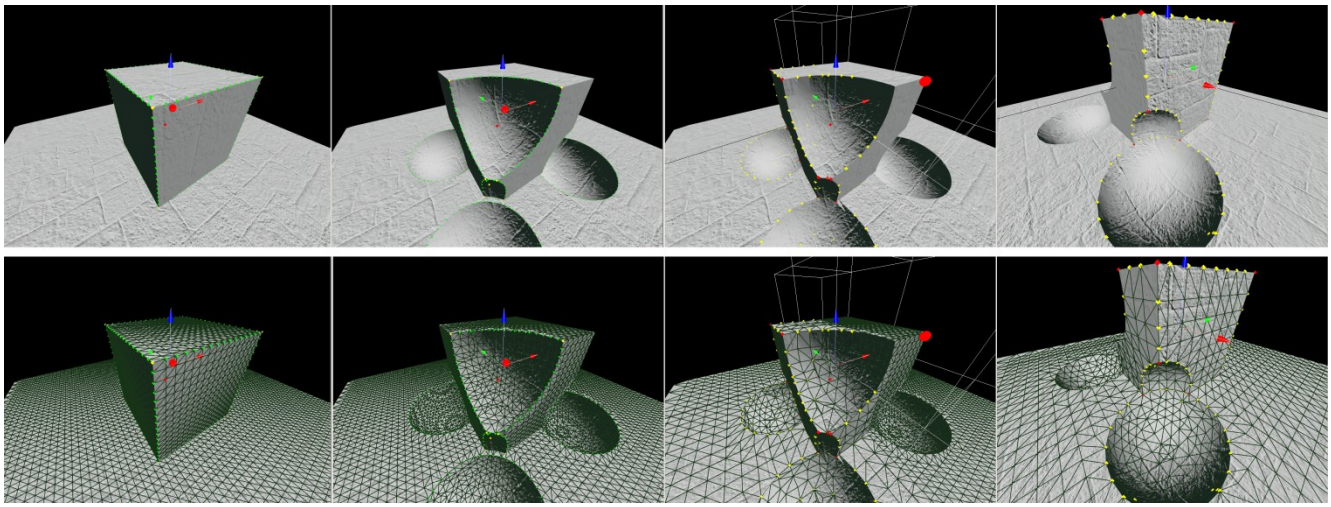


а) б) в)

Рисунок 2.15 – Результаты триангуляции воксельного ландшафта с реконструкцией острых рёбер. а) Исходный ландшафт на самом детальном уровне детализации (острые вершины помечены зелёным цветом). б) Триангуляция упрощённого уровня детализации с разбиением четырехугольников только вдоль «острой» диагонали, что часто приводит к появлению треугольников плохой формы, и в) триангуляция, в которой при неоднозначном выборе диагонали для разбиения каждого четырехугольника выбирается его более короткая диагональ.

На рисунке 2.16 показаны результаты работы вышеуказанных алгоритмов определения острых углов и генерации треугольной сетки с разбиением каждого четырёхугольника вдоль острой или кратчайшей диагонали.

В [81] предлагается находить пересечения и разбивать четырёхугольники на четыре, а не два треугольника. В [80,82] предложены более эффективные эвристики для генерации непересекающихся треугольных сеток, что, однако, не всегда приводит к сохранению острых граней и углов поверхности. В алгоритме CMS [33] при обнаружении самопересечений контура две пересекающиеся (конусообразные) части контура внутри ячейки объединяются в цилиндрическую поверхность для устранения пересечения.



а) б) в) г)

Рисунок 2.16 – Результаты триангуляции воксельного ландшафта с реконструкцией его острых углов и рёбер. Зелёным и жёлтым цветом помечены вершины треугольной сетки, расположенные на острых рёбрах и острых углах ландшафта, соответственно. Красным отмечены острые углы, реконструированные в процессе генерации упрощённых уровней детализации. а) Исходный ландшафт, построенный из неявного описания (объединение плоскости и куба). б) Ландшафт на самом детальном уровней детализации после применения булевых операций (CSG-вычитания сфер). в) Генерация упрощённых уровней детализации с сохранением острых углов и рёбер при движении наблюдателя слева направо. г) Полученный воксельный ландшафт на более грубом уровне детализации.

2.5 Результаты экспериментов

Детали реализации. Выбор структур данных и детали низкоуровневой реализации оказывают существенное влияние на производительность алгоритмов. Для уменьшения объёма рабочей памяти на 64-битных системах в авторской реализации классического октодерева вместо указателей используются 32-битные индексы. Самый старший бит каждого индекса представляет тип узла. Внутренние и листовые узлы хранятся в двух отдельных массивах. Стандартный алгоритм триангуляции [31,46] реализован в виде 13 громоздких рекурсивных функций и занимает более 600 строк кода.

В линейном октодереве хранятся только листья-ячейки и их мортон-коды, что приводит к значительной экономии памяти, поскольку в традиционном октодереве количество внутренних

узлов обычно примерно втрое превышает число листовых узлов[5]. Для каждой ячейки линейного октодерева хранится соответствующий ей код Мортона, позиция острой вершины поверхности внутри ячейки и 8-битная маска со значениями «внутри»/«снаружи» в углах ячейки (см. Приложение Б). В целях экономии памяти позиция острой вершины внутри ячейки хранится в виде квантованных в пространстве ячейки координат, что уменьшает размер позиций вершин в 4 раза (с 12 байт до 3 байт) без заметной потери визуального качества. Кроме того, квантование делает незаметным стыки между смежными блоками ландшафта (см. раздел 4.3), которые бы возникли из-за численных погрешностей (обусловленных использованием чисел с плавающей точкой), если бы позиции вершин хранились с полной точностью.

В настоящей реализации линейного октодерева используются 30-битные коды Мортона, что ограничивает максимальное разрешение октодерева 1024 ячейками (на каждое измерение отводится по десять бит). Это ограничение не является существенным для триангуляции воксельного ландшафта, где разрешение каждого блока, как правило, находится в пределах $16^3..64^3$. Программная реализация предложенного алгоритма триангуляции также гораздо проще классического подхода и занимает около 170 строк кода. Простейшая работающая реализация алгоритма приведена в Приложении Б.

Используемые сцены. Для исследования эффективности предложенного алгоритма триангуляции использовались стандартные тестовые полигональные модели из открытых хранилищ OpenCamLib, Computer Graphics Group RWTH-Aachen[6] и Stanford 3D scanning repository[7] (рисунок 2.17).

Измерения проводились на настольном компьютере, оснащённом процессором Intel® Core™ i7-2600K @ 3.40 ГГц, 16 Гб оперативной памяти. Все методы триангуляции были реализованы на языке C++ для исполнения на CPU и скомпилированы в 64-битном режиме с помощью Microsoft Visual C++ 2013.

---

[5] Как было показано ранее, в процессе триангуляции участвуют только листовые узлы-ячейки, которые пересекают изоповерхность, а внутренние узлы октодерева являются, по сути, соединяющим «клеем».

[6] https://www.graphics.rwth-aachen.de/research/

[7] http://graphics.stanford.edu

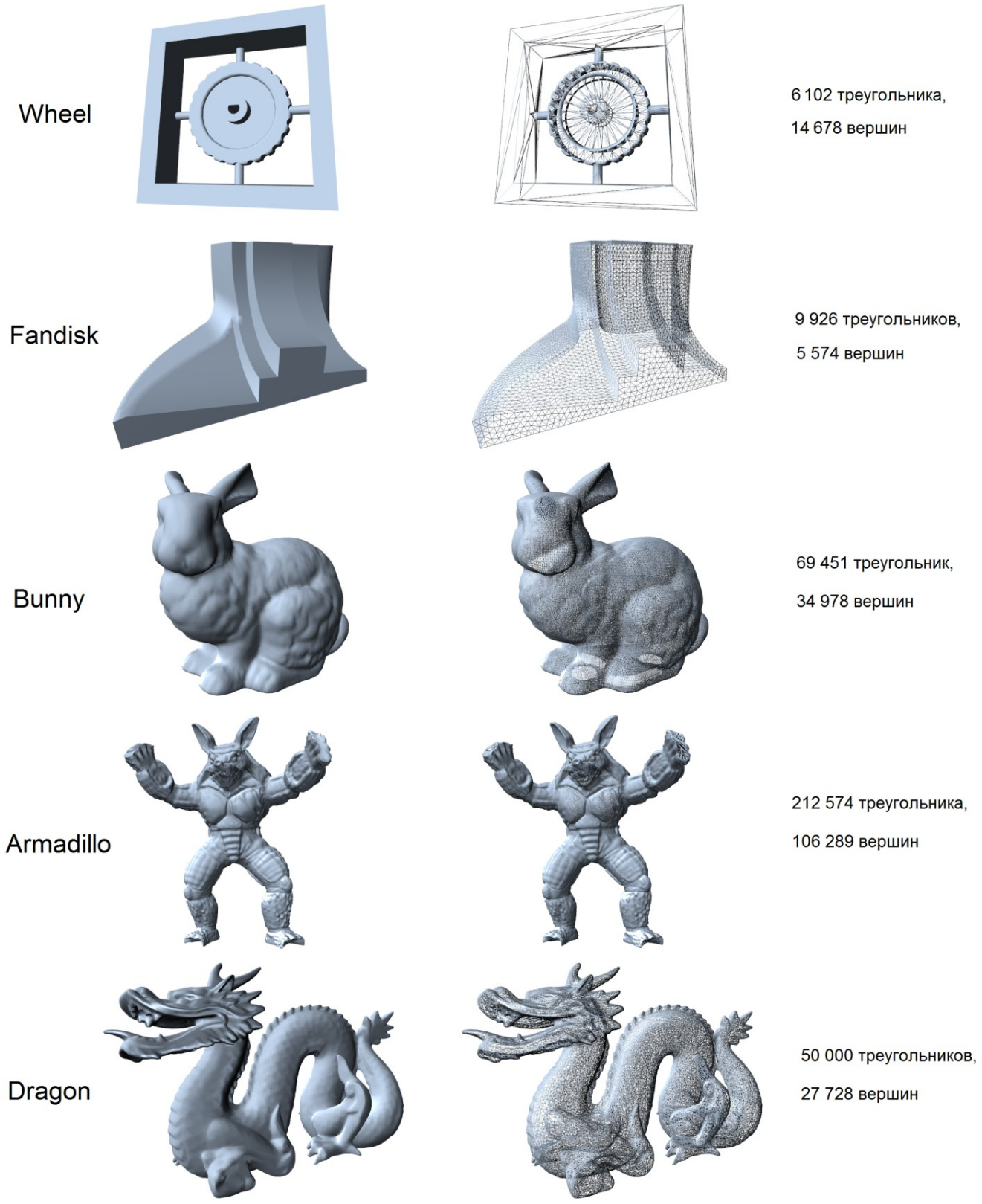
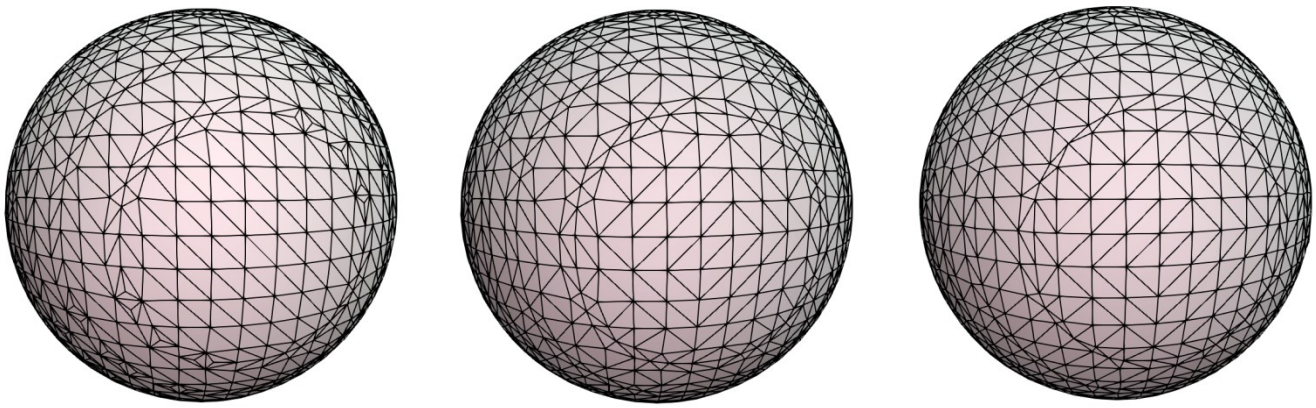


Рисунок 2.17 – Используемые модели (сверху-вниз): Wheel, Fandisk, Bunny, Armadillo, Dragon. Первые две модели содержат острые рёбра и углы.

Во всех экспериментах из исходной полигональной модели строилось лучевое представление (см. раздел 3.4), к которому затем применялись различные методы триангуляции. Время построения лучевого представления из исходной модели пренебрежительно мало по сравнению со временем, затрачиваемым на создание и упрощение октодерева, и на построение адаптивной триангуляции. При тестировании предложенных методов триангуляции для упрощения линейных октодеревьев использовался специальный итеративный восходящий алгоритм, который потребляет меньше памяти, чем стандартный рекурсивный алгоритм. Критерием остановки для упрощения октодерева в экспериментах было выбрано достижение абсолютной нормы невязки (QEF error threshold) $\varepsilon = 10^{-3}$.

Сравнение построенных сеток и потребления рабочей памяти. В первом эксперименте проводилось сравнение полученных полигональных сеток и проверялось потребление рабочей памяти. Стандартный [31,46] и предложенный (см. подраздел 2.4.2) алгоритмы триангуляции генерируют сетки с одинаковым количеством полигонов и вершин. Позиции вершин также совпадают, но при триангуляции изоповерхностей, не содержащих острых углов, различается порядок соединения вершин сеток. На рис. 2.18 показан результат триангуляции поверхности сферы стандартными и предложенным алгоритмами триангуляции.



а) б) в)

Рисунок 2.18 – Сравнение сеток, построенных методом дуальных контуров а) на регулярной решётке, б) с помощью стандартного адаптивного алгоритма и в) с помощью предложенного алгоритма триангуляции. В последних двух случаях не производилось упрощение октодерева, что эквивалентно выполнению триангуляции на регулярной решётке. Все треугольные сетки содержат одинаковое количество вершин и треугольников, но вершины соединены по-разному.

Для оценки когерентности треугольных сеток использовалось отношение количества трансформированных (не найденных в вершинном кэше) вершин к количеству растеризованных

треугольников (average cache miss ratio, ACMR) [73]. В наихудшем случае все три вершины каждого треугольника должны быть трансформированы, и тогда значение ACMR равно 3. В идеальном случае при рендеринге индексированной треугольной сетки трансформируется одна новая вершина каждого треугольника и используются результаты двух предыдущих вершин, и тогда значение ACMR стремится к 0.5 (обычно число треугольников примерно в два раза больше количества вершин). Для большинства моделей, созданных в САПР, ACMR обычно составляет около 1.5, а после их оптимизации под вершинный кэш (путём создания длинных полос треугольников) — 0.6−0.7. В среднем, стандартный алгоритм триангуляции для кэша вершин (post-transform vertex cache) размером 24 вершины генерирует треугольные сетки с ACMR **станд.** = 0.97 (что соответствует примерно 67% «попаданий» в вершинный кэш), предложенный — с ACMR **лин.** = 0.73 (76% кэш-попаданий). В знакоопределённом октодереве число листовых узлов-ячеек в два-четыре раза превышает количество внутренних узлов. В текущей реализации для хранения линейного октодерева требуется более чем в десять раз меньше памяти, чем для хранения традиционного октодерева. Результаты эксперимента представлены в таблице 2.1.

Таблица 2.1 – Распределение типов узлов октодерева и объём занимаемой памяти.

Название модели и разрешение воксельной решётки	**Число внутренних узлов октодерева**	**Число листовых узлов октодерева (ячеек)**	**Оценка эффективности вершинного кэша**		**Число треугольников и вершин в построенной сетке**
			ACMR станд.	**ACMR лин.**	
Fandisk (65^3)	4096	6 330	0.973	0.723	12 652 Δ 6 330 V
Wheel (129^3)	32768	17 915	0.924	0.736	35 846 Δ 17 915 V
Bunny (129^3)	32768	48 401	0.973	0.730	96 507 Δ 48 401 V
Armadillo (257^3)	65536	48 401	0.978	0.731	258 476 Δ 48 401 V
Dragon (513^3)	262144	398 613	0.971	0.734	797 758 Δ 398 613 V

Построенные треугольные сетки представлены на рисунке 2.19.

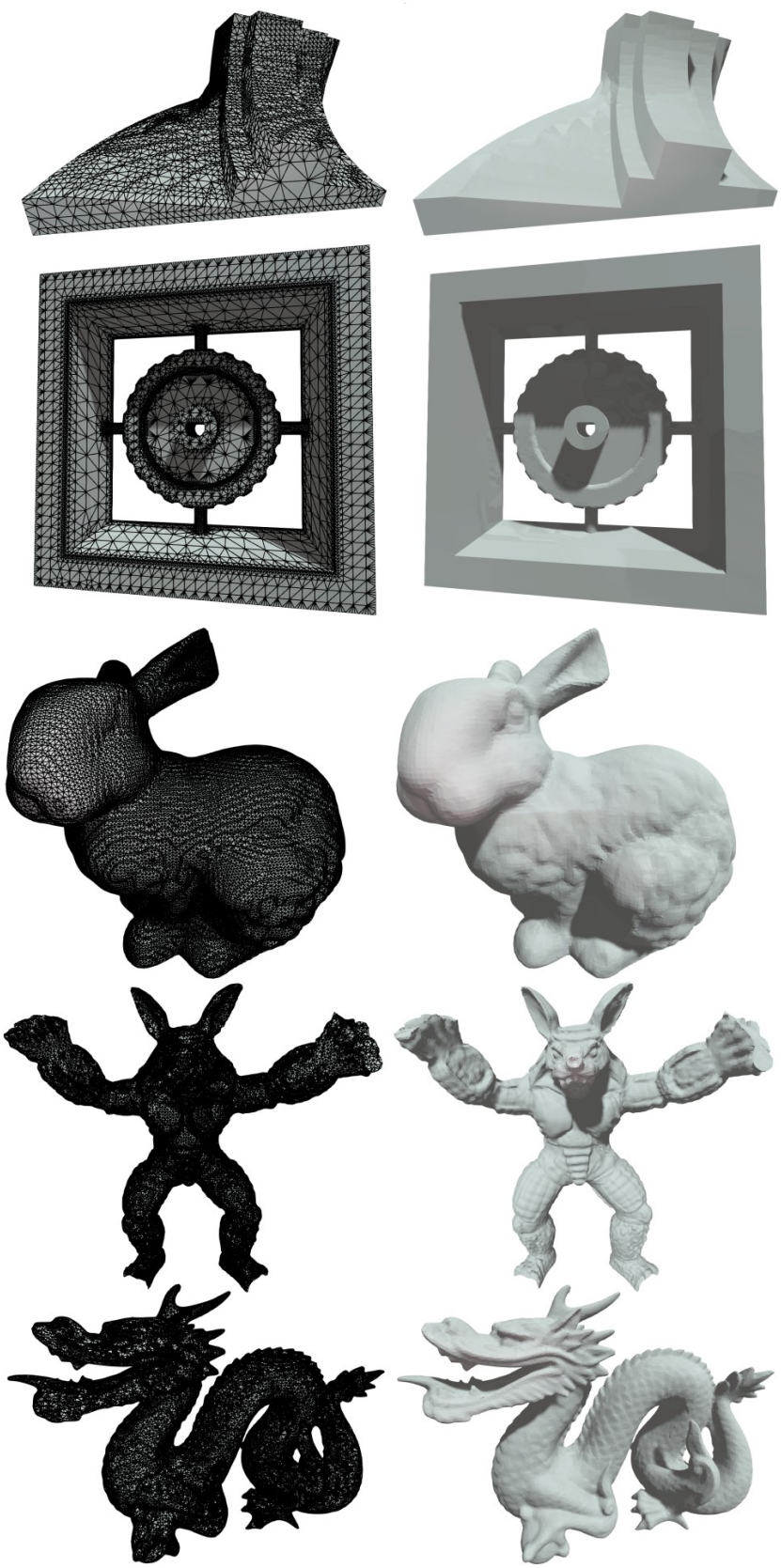


Рисунок 2.19 – Реконструкция тестовых моделей методом дуальных контуров с помощью предложенного алгоритма адаптивной триангуляции.

Сравнение скорости триангуляции. Во втором эксперименте проводилось сравнение скорости работы различных алгоритмов триангуляции. В таблице 2.2 приведены усреднённые со 100 запусков результаты измерения скорости работы алгоритмов на модели Стэнфордского кролика для сеток разрешениями воксельной решётки от 33^3 до 513^3. Для адаптивных методов триангуляции приведено только время триангуляции октодерева; с учётом времени построения и упрощения октодерева адаптивные методы выполняются дольше, чем методы триангуляции на регулярной решётке.

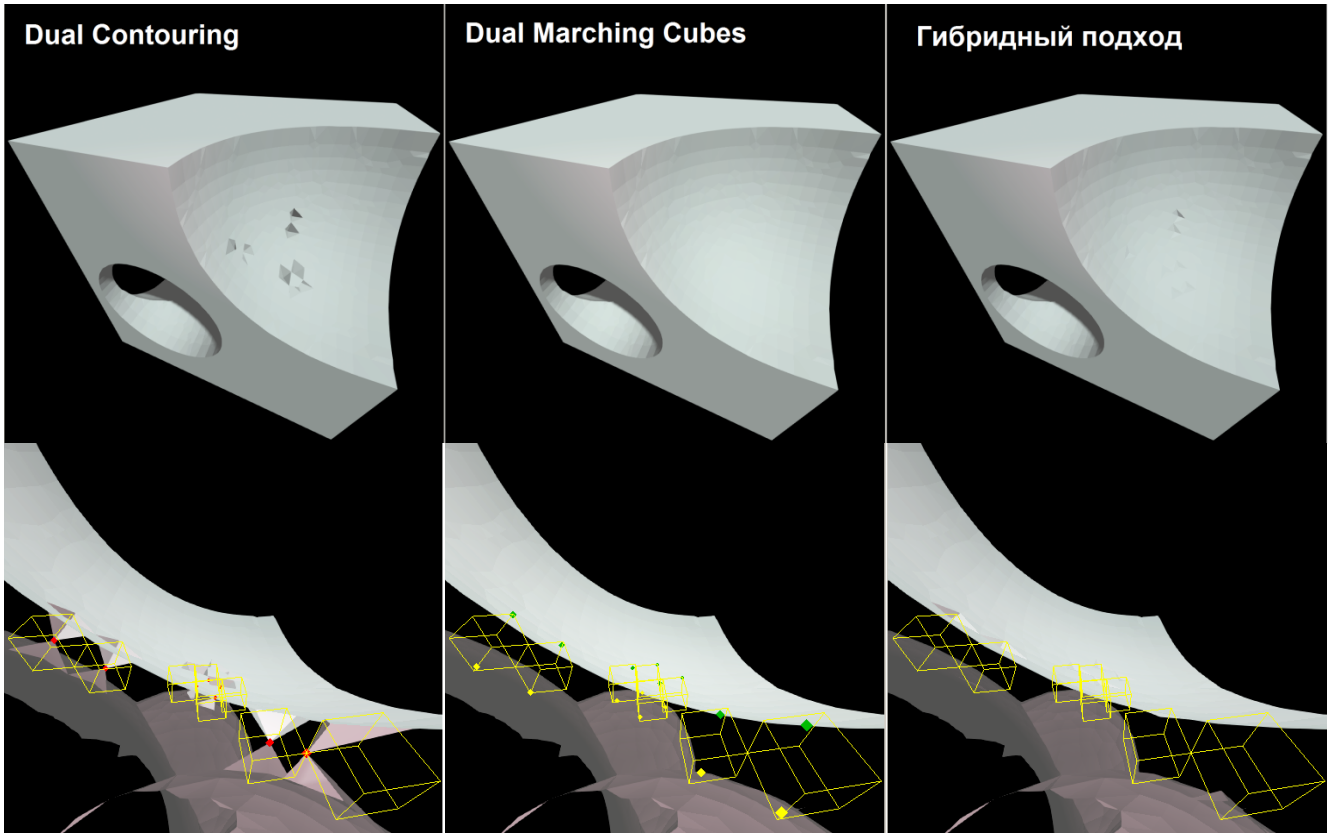
Таблица 2.2 – Сравнительное время работы алгоритмов для триангуляции модели "Bunny" в различных разрешениях: ADC (Adaptive Dual Contouring) — стандартный алгоритм адаптивной триангуляции на основе классических октодеревьев [31,46]; ADC лин., бин. поиск — реализация предлагаемого подхода с триангуляцией линейных октодеревьев, использованием бинарного поиска и масок активных рёбер (см. подраздел 2.4.2); ADC лин., хэш-табл. — реализация близкого к предложенному алгоритма, использующего хэш-таблицу для поиска [49]; ADMC лин., бин. поиск — модификация предложенного алгоритма для создания более качественных сеток [58] (см. подраздел 2.4.3); DC [31] и DMC [56] — двойственные методы триангуляции на регулярной решётке.

Разрешение воксельной решётки	**Время выполнения алгоритма триангуляции, в миллисекундах**					
	ADC	**ADC лин., бин. поиск**	**ADC лин., хэш-табл.**	**ADMC лин., бин. поиск**	**DC**	**DMC**
33	2	2	1	2	4	4
65	3	5	4	5	22	23
129	13	20	15	21	127	132
257	54	82	68	87	833	864
513	153	278	251	296	5819	6160

Из таблицы можно увидеть, что время работы адаптивных алгоритмов триангуляции растет пропорционально площади поверхности (ячейки октодерева пересекают поверхность области, а для гладкой и не содержащей мелких деталей модели кролика площадь поверхности зависит от квадрата линейных размеров области). На больших объёмах предложенный алгоритм триангуляции линейных октодеревьев уступает по скорости работы, однако, при типичном для

блока воксельного ландшафта разрешении (32-64 ячейки) разница во времени межу стандартным и предложенным алгоритмами триангуляции несущественна.

Сравнение качества сеток. В третьем эксперименте сравнивалось качество полигональных сеток, построенных методами DC [31,46], DMC [56] и с использованием предложенного гибридного подхода (см. подраздел 2.4.3). На рис. 2.20 показана реконструкция модели куба, в котором сериями булевых операций были образованы полости с тонкими стенками. В неоднозначных ячейках DC создаёт по сингулярной вершине (поверхность имеет форму песочных часов), в то время как DMC обеспечивает разделение связных компонент и развёртываемую топологию сетки. Предложенный гибридный подход создаёт триангуляцию методом DMC, оценивая позиции дополнительных вершин в неоднозначных ячейках на основе координат единственной вершины в DC.

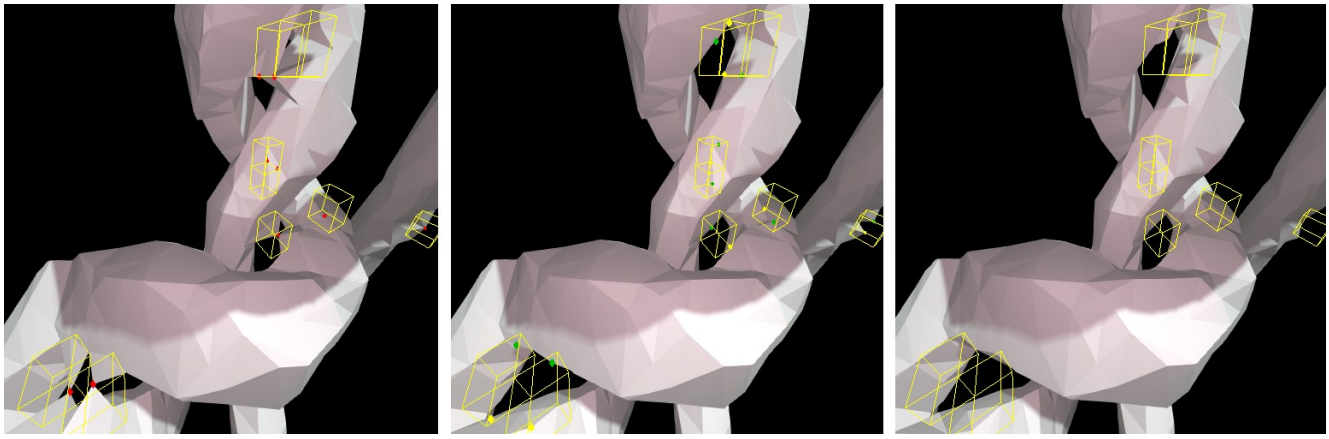


а) б) в)

Рисунок 2.20 – Результаты реконструкции тонкостенной поверхности с помощью а) DC, б) DMC и в) гибридного подхода. В нижнем ряду показан вид на тонкую стенку сверху.

На рис. 2.21 показаны результаты реконструкции модели перекрученного узла на грубой решётке разбиения, в результате чего полученная сетка содержит много узких (с почти

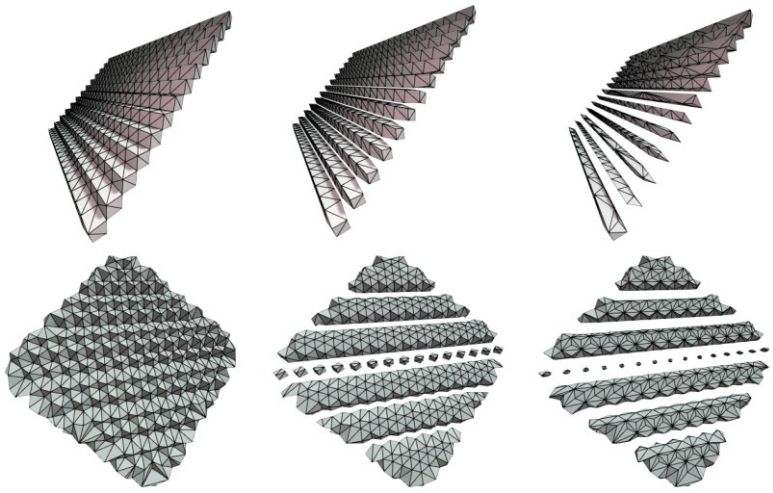
соприкасающимися поверхностями) участков. Нетрудно заметить, что предложенный подход создаёт более детализированные сетки. Однако, будучи табличным методом, в неоднозначных ячейках DMC иногда выбирает шаблон триангуляции с топологией, которая не всегда соответствует действительности.



а) б) в)

Рисунок 2.21 – Результаты реконструкции поверхности перекрученных нитей с помощью а) DC, б) DMC и в) гибридного подхода. DC не обеспечивает разделения мелких деталей.

На рис. 2.22 показаны случаи, в которых DC создаёт топологически связную поверхность с наличием сингулярных рёбер и вершин, в то время как DMC и гибридный подход генерируют развёртываемую, но несвязную сетку. Связная поверхность может быть построена при другом расположении наклонной стенки, увеличении разрешения решётки разбиения или использовании для триангуляции алгоритма CMS [33], в котором для разрешения неоднозначных ситуаций и выбора способа соединения контура используются нормали к границе области. По сравнению с DC, алгоритм DMC и предложенный подход создают сетки лучшего качества, незначительно увеличивая общее время триангуляции и количество вершин (количество треугольников в обоих случаях совпадает).



а) б) в)

Рисунок 2.22 – Реконструкция тонкой стенки, наклоненной по углом 45° к одной (сверху) и двум (снизу) координатным плоскостям с помощью а) DC, б) DMC и в) гибридного метода.

2.6 Основные выводы по второй главе

Разработан адаптивный алгоритм триангуляции линейных октодеревьев методом дуальных контуров, отличающийся от оригинального [31] простотой реализации, меньшим объёмом потребляемой памяти и меньшим размером исполняемого кода. В отличие от известных автору алгоритмов триангуляции, разработанный алгоритм создаёт треугольные сетки с большей пространственной когерентностью и может быть использован для триангуляции сверхбольших изоповерхностей и воксельных ландшафтов по частям (см. раздел 4.3). Подход на основе линейных октодеревьев может быть расширен и до других двойственных алгоритмов триангуляции (например, MDC [53] и ADMC [48,49]). Поскольку алгоритм адаптивной триангуляции не использует сложных иерархических структур данных, то он может быть с успехом реализован на GPU. Проведено сравнение разработанного алгоритма с аналогичными алгоритмами триангуляции и экспериментально показана его высокая эффективность.

Разработаны методы улучшения построенных треугольных сеток, а именно, улучшения качества реконструкции острых рёбер и углов и тонкостенных поверхностей.

3 Разработка методов представления и форматов хранения воксельных ландшафтов в САПР ВР

В данной главе рассмотрены способы представления трёхмерных объектов и форматы хранения объёмных (воксельных) данных, которые в настоящее время применяются для геометрического моделирования в САПР и могут быть использованы для работы с воксельными ландшафтами. Все форматы хранения содержат информацию для реконструкции острых углов поверхности, и у каждого есть свои достоинства и недостатки.

В разделе 3.6 предлагается метод для компактного хранения и внеядерной триангуляции изоповерхности воксельного ландшафта. Метод заключается в использовании линейных октодеревьев, ячейки которых содержат позицию острой вершины на поверхности и значения «внутри»/«снаружи» в углах ячейки.

3.1 Проектирование форматов хранения воксельных ландшафтов

Для решения задач, сформулированных в первой главе, формат хранения воксельных ландшафтов должен удовлетворять следующим требованиям:

1) возможность разбиения ландшафта на блоки (для редактирования, сохранения, загрузки и визуализации только необходимых частей ландшафта);

2) высокая скорость построения адаптивной триангуляции;

3) возможность хранения информации для восстановления острых рёбер и углов поверхности, что необходимо для отображения искусственных элементов ландшафта, таких как здания и крупные технические объекты;

4) поддержка различных материалов (для описания составных сред);

5) компактность представления в памяти;

6) высокая скорость сжатия (компрессии) и распаковки (для обеспечения интерактивности при редактировании ландшафта);

7) высокая степень сжатия (для компактности хранения в базе данных);

8) возможность генерации уровней детализации: визуализация больших и детализированных ландшафтов немыслима без использования LoD-схем;

9) возможность создания из полигональных моделей (для интеграции с существующими системами).

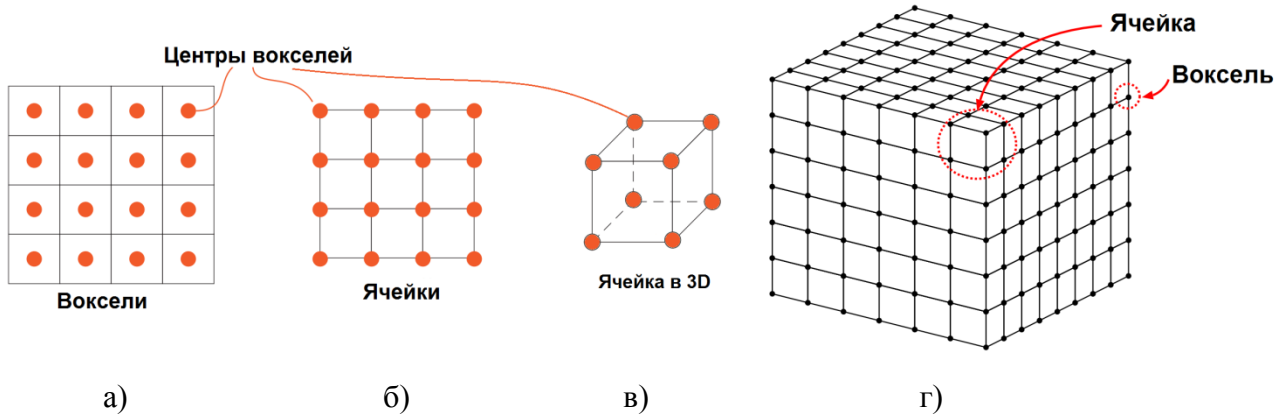
Кроме того, для обеспечения возможности интерактивной модификации ландшафта формат хранения воксельных данных должен предоставлять:

1) возможность выполнения операций редактирования (например, закраска материалом, CSG-операции) ландшафта в интерактивном режиме;

2) возможность интерактивной генерации уровней детализации.

Удовлетворить всем этим требованиям одновременно трудно, поэтому было разработано несколько способов представления и форматов хранения воксельных данных, которые строятся на основе некоторого компромисса и имеют свои преимущества и недостатки.

Триангуляция. Все описанные в данной главе форматы данных для хранения воксельного ландшафта были разработаны с учётом особенностей выбранного метода триангуляции — метода дуальных контуров (Dual Contouring, DC) [31]. Минимальным элементом разбиения пространства в DC является двойственная ячейка (dual cell). Геометрия решётки разбиения (dual grid) является двойственной по отношению к воксельной решётке (voxel grid), т.е. центры вокселей расположены в углах ячеек (или узлах решётки разбиения), и наоборот. В кубической решётке разбиения двойственные ячейки образованы блоками из восьми (2×2×2) смежных вокселей. Таким образом, если массив воксельных данных состоит из n^3 ячеек, то в нём содержатся $(n+1)^3$ вокселей (рисунок 3.1).



а) б) в) г)

Рисунок 3.1 – Принцип двойственности на регулярной кубической решётке разбиения:
а) воксельная решётка в 2D и б) двойственная к ней решётка из ячеек
(оранжевыми точками отмечены центры вокселей); в) отдельная ячейка в 3D;
г) схематичное изображение решётки для дискретизации пространства (объёма) в 3D.

В данной работе для дискретизации пространства используются кубические решётки или равномерные декартовы сетки (uniform cartesian grids), как наиболее удобные и эффективные для

работы с объёмными данными. Например, в отличие от симплициального (тетраэдрального) разбиения, в кубической решётке такого же объёма содержится меньшее количество ячеек. Это приводит к тому, что для кубической решётки требуется хранить гораздо меньше данных, а в результате триангуляции будет создано меньшее количество треугольников, среди них также будет меньше треугольников плохой формы [83]. Количество вершин куба равно степени двойки, что позволяет разрабатывать компактные схемы адресации ячеек и эффективные программные реализации, в том числе с использованием SIMD-вычислений [84]. Недостатками кубической решётки являются неоднозначности триангуляции (ambiguous cases) и громоздкость таблиц с 2^8 шаблонами триангуляции (для алгоритмов MC [28] и DMC [56]) и топологией связи элементов куба (для CMS [44]), таблиц для обхода и упрощения октодерева с сохранением топологии (алгоритмы ADC, ADMC и MDC). В двойственных методах триангуляции возникают двузначности при разбиении на треугольники созданных для рёбер ячеек четырёхугольников, в то время как аналогичный тетраэдральный метод [61] свободен от этих недостатков.

Вокселям обычно сопоставляются индексы подобластей с различными свойствами — идентификаторы материалов (где нулевое значение, как правило, обозначает пустое пространство), а также физические свойства и характеристики среды (например, температура, цвет, плотность, расстояние до поверхности и т.п).

Ячейки, все углы которых расположены внутри одной подобласти (имеют одинаковый материал), называются *однородными* (*homogeneous*) ячейками. Ячейки с наличием различных подобластей (материалов) называются *неоднородными* (*inhomogeneous*) или смешанными (mixed). Границы между подобластями с различными материалами называются *интерфейсами*.

Для возможности реконструкции острых углов изоповерхности воксельного ландшафта в данной работе используется не воксельное представление в чистом виде, а различные смешанные представления, в которых, помимо воксельной решётки, дополнительно могут храниться или точки пересечения рёбер ячеек с изоповерхностью и единичные нормали к поверхности в этих точках, или позиции острых вершин на изоповерхности внутри ячеек, или градиенты скалярного поля.

При триангуляции двойственными методами внутри неоднородных ячеек создаётся по одной или несколько вершин (representative vertices, isopoints), расположенных на интерфейсах подобластей (поверхностях, представляющих границу раздела подобластей/сред с различными материалами). Помещая вершины в острые углы изоповерхности, двойственные методы способны создавать треугольные сетки с восстановлением особенностей изоповерхности внутри ячеек (см. Приложение А).

Сжатие объёмных данных. Хранение трёхмерных массивов вокселей в несжатом виде приводит к недопустимо высокому потреблению памяти, поэтому на практике применяются различные алгоритмы сжатия без потерь. Для обеспечения интерактивности при редактировании ландшафта помимо быстрой распаковки воксельных данных необходимо также быстрое сжатие. Данным требованиям удовлетворяют алгоритмы кодирования длин серий и словарные алгоритмы сжатия. Более изощрённые методы сжатия обладают высокой вычислительной сложностью и не всегда способны обеспечить лучшую степень сжатия для блоков небольшого размера. Как правило, «интересные» ландшафты обладают высокой степенью избыточности, что обусловливает высокую степень сжатия — до нескольких десятков раз [10,11,18]. В целом, для сжатия воксельных данных хорошо подходят алгоритмы сжатия растровых изображений.

Градиенты или Эрмитовы данные, необходимые для восстановления острых рёбер и углов изоповерхности, необходимо хранить только вблизи поверхностной оболочки или на границах раздела между различными материалами. Тогда количество потребляемой памяти будет расти пропорционально площади поверхности, а не величине объёма, занимаемого объектом. Компрессия и распаковка Эрмитовых данных должны происходить с минимальными потерями точности, поскольку двойственные методы триангуляции очень чувствительны к погрешностям в данных нормалей. На практике нормали обычно коррелированы друг с другом (это нарушается вблизи особенностей поверхности), поэтому для сжатия данных нормалей подходит схема квантование $\rightarrow$ дельта-кодирование $\rightarrow$ энтропийное кодирование. Как и в остальных случаях, для повышения коэффициента сжатия следует сжимать части блока с разными статистическими свойствами (например, позиции вершин, нормали, материалы вокселей) отдельно.

Редактирование ландшафта. Во всех разработанных форматах хранения воксельных данных операции редактирования выполняются над распакованными данными и сводятся к простым операциям над точками (вокселями) и отрезками (Эрмитовыми данными). В отличие от граничного представления (B-Rep), все описанные виды представления трёхмерных объектов замкнуты относительно теоретико-множественных операций, что обеспечивает корректность получаемых в результате редактирования объёмных тел. Например, булевы операции над поверхностными сетками и BSP-деревьями в теории позволяют получать точные результаты, но на практике являются ненадёжными из-за ошибок округления [85], в то время как булевы операции над воксельными решётками абсолютно надёжны (робастны), но работают только с дискретными аппроксимациями исходных тел.

Далее следует описание разработанных форматов воксельных данных, предназначенных для хранения отдельных блоков или *чанков* (*chunk*) ландшафта. Проблемы многомасшабной визуализации больших ландшафтов, а также задача создания бесшовной треугольной сетки для соединения блоков с отличающимися размерами и различными уровнями детализации рассмотрены в четвёртой главе.

3.2 Знакоопределённое поле расстояний с градиентами

Основой для самого простого из предложенных формата хранения данных воксельного ландшафта послужило знакоопределённое поле расстояний (Signed Distance Field, SDF) [86]. В этом формате каждый блок ландшафта представлен трёхмерным массивом вокселей, где для каждого вокселя хранится (скалярное) значение функции расстояния со знаком (Signed Distance Function), а также её градиент для возможности отслеживания особенностей изоповерхности.

Функция $d_s(\mathbf{p}) : \mathbb{R}^3 \mapsto \mathbb{R}$ расстояния от границы со знаком для каждой точки $\mathbf{p} \in \mathbb{R}^3$ возвращает минимальное по модулю эвклидово расстояние d со знаком (signed distance) от этой точки до границы ∂S объекта S :

$$d_s(\mathbf{p}) = \begin{cases} -\inf\limits_{\forall \mathbf{q} \in \partial S} \|\mathbf{p} - \mathbf{q}\|_2, & \mathbf{p} \in S \\ \inf\limits_{\forall \mathbf{q} \in \partial S} \|\mathbf{p} - \mathbf{q}\|_2, & \mathbf{p} \notin S \end{cases} \qquad (3.1)$$

Таким образом, $d_s(\mathbf{p}) < 0$, если точка $\mathbf{p}$ находится внутри объекта S, $d_s(\mathbf{p}) > 0$, если $\mathbf{p}$ лежит снаружи S, и $d_s(\mathbf{p}) = 0$, если точка $\mathbf{p}$ находится на его поверхности.

Формально, каждый блок воксельного ландшафта имеет форму куба, на котором задана равномерная декартова сетка, содержащая n^3 ячеек, и представлен воксельной решёткой V, которая задана набором значений v вокселей в узлах этой сетки: $V = \{ v_{i,j,k \in \mathbb{N}} \mid 0 \le i \le n,\ 0 \le j \le n,\ 0 \le k \le n \}$. Для возможности нахождения острых углов поверхности каждый воксель содержит, помимо значения $d_s(\mathbf{p}) = d$ наименьшего по модулю эвклидова расстояния со знаком от центра $\mathbf{p}$ вокселя до поверхности ландшафта S, нормализованный градиент $\hat{\mathbf{n}}$ функции d_s расстояния: $\hat{\mathbf{n}} = \frac{\nabla d_s}{|\nabla d_s|}$ (который внутри тела направлен к поверхности, а снаружи — направлен от поверхности). Таким образом, в данном формате представления воксельного ландшафта для каждого вокселя в решётке V хранится пара значений: $v = \{\hat{\mathbf{n}}; d\}$.

Триангуляция. Из всех разработанных форматов хранения объёмных данных, формат SDF предоставляет наибольшую гибкость в выборе методов триангуляции. При триангуляции методом

DC для нахождения острой вершины поверхности внутри ячейки нормали берутся из вокселей, расположенных в углах ячейки. Поэтому реконструированные особенности в полученной сетке обладают худшим качеством по сравнению с триангуляцией, использующей полноценные Эрмитовы данные (точные позиции точек пересечения рёбер активных ячеек с изоповерхностью и единичные нормали к поверхности в этих точках).

Способы создания. SDF может быть легко построено по заданной неявной функции F расстояния от границы со знаком. Для гладких (дифференцируемых, C^1-непрерывных) функций расстояния со знаком градиент может быть вычислен аналитически или оценен конечно-разностными методами:

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right) \approx$$

$$\approx \left(\frac{F_{(x_{i+1}, y_j, z_z)} - F_{(x_{i-1}, y_j, z_z)}}{2 \cdot dx}, \frac{F_{(x_i, y_{j+1}, z_z)} - F_{(x_i, y_{j-1}, z_z)}}{2 \cdot dy}, \frac{F_{(x_i, y_j, z_{z+1})} - F_{(x_i, y_j, z_{z-1})}}{2 \cdot dz}\right). \qquad (3.2)$$

Для оценивания градиентов в окрестности точки **p** в областях с острыми рёбрами и углами (с C^1-разрывами, где градиент не определён) может быть использован алгоритм ReliableGrad (ReliGrad) [59,60]. При использовании неявного описания объекты очень сложной формы могут быть скомбинированы из описаний более простых областей путём выполнения теоретико-множественных операций (CSG), поворотов и повторов, масштабирования, скручивания (bending), офсеттинга, гладкого сопряжения (smooth blend), метаморфозиса, дилатации, эрозии и т.д. Получение SDF из двусторонних полигональных моделей является гораздо более трудоёмким процессом [40,86], поскольку для каждого вокселя требуется нахождение наименьшего по модулю расстояния со знаком от центра вокселя до поверхностной сетки, представляющей границу объекта, и корректного градиента.

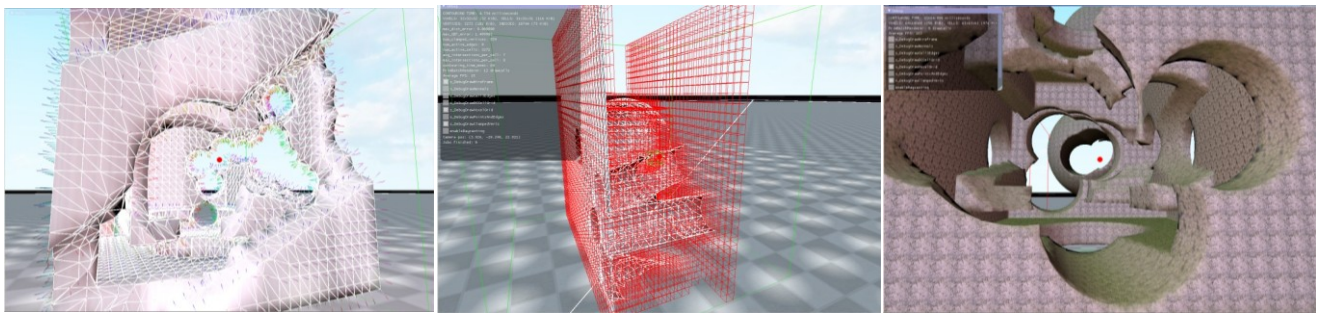
Редактирование. SDF поддерживает все вышеперечисленные операции над функциями расстояния со знаком. Например, булевы операции объединения, пересечения и разности выполняются для каждого вокселя (соответствующего значения ($\hat{\mathbf{n}}$; d)) по правилам, указанным в таблице 3.1.

Таблица 3.1 – Правила выполнения булевых операций в SDF (взято из [40,86]).

Тип булевой операции	Новое значение расстояния d	Новое значение градиента $\mathbf{n}$
A ∪ B (объединение)	$\min(d_a, d_b)$	if $d_a < d_b$ then $\mathbf{n_a}$ else $\mathbf{n_b}$
A ∩ B (пересечение)	$\max(d_a, d_b)$	if $d_a < d_b$ then $\mathbf{n_b}$ else $\mathbf{n_a}$
A \ B (разность)	$\max(d_a, -d_b)$	if $d_a < -d_b$ then $-\mathbf{n_b}$ else $\mathbf{n_a}$
A Δ B (симметрическая разность)	$\min(-d_a, -d_b)$	if $d_a < d_b$ then $-\mathbf{n_b}$ else $-\mathbf{n_a}$

Сжатие. Поскольку в SDF для хранения трёхмерного массива вокселей необходимо большое количество памяти, то было принято решение хранить пары значений ($\hat{\mathbf{n}}$; d) только вблизи поверхности объекта (т.е. модуль d никогда не превышает половины r длины диагонали вокселя: $d = \min(\max(-r, d_s(\mathbf{p})), r)$, что фактически превращает воксели со значениями ($\hat{\mathbf{n}}$; d) в поверхностные элементы — *сёрфели* или *сурфелы* (англ. *surfels*, сокр. от s*urface elements*) (см. рис. 3.2, б).

В настоящей реализации SDF отсутствует поддержка слоистых, составных областей (с наличием различных материалов), т.е. блок ландшафта в формате SDF представлен бинарной воксельной решёткой, в которой значения вокселей которых могут принимать только значения 1 (если воксель находится «внутри» объекта) или 0 («снаружи»), и разреженным массивом сёрфелей. В случае составных областей необходимо ввести дополнительный трёхмерный массив, содержащий материал каждого вокселя. При этом значения ($\hat{\mathbf{n}}$; d) должны храниться только у границы раздела между различными материалами. Воксель считается лежащим на границе подобласти, если хотя бы один из его 26 соседей имеет отличный от него индекс материала. Данная схема компактного хранения делает непригодным использование методов ReliGrad [59] и MergeSharp/SHREC [60], которые для отслеживания особенностей поверхности требуют наличие градиентов в 5x5x5-окрестности (и более) каждой ячейки. Кроме этого, теряется информация о градиенте поля внутри объекта, которая могла бы быть полезной, например, в системе обработки столкновений (collision detection) для «расталкивания» взаимопроникающих невыпуклых тел. Компромиссным решением будет хранение ($\hat{\mathbf{n}}$; d) для вокселей у границы объекта, и скалярных значений d расстояния для вокселей полностью внутри объекта, тогда градиенты поля расстояний внутри объекта могут быть оценены на основе значений d.



а) б) с)

Рисунок 3.2 – Использование SDF в качестве объёмных данных. а) Визуализация SDF: цветными линиями показаны нормали к поверхности. б) Воксели хранятся только на границах раздела двух сред. в) Фрагмент воксельного ландшафта после выполнения серии булевых операций.

Эксперименты показали, что «наивное» квантование нормализованных градиентов (с вычислением третьей компоненты через квадратный корень) приводит к созданию ребристой поверхности, а использование октодерева, в котором $\hat{\mathbf{n}}$ и d можно найти трилинейной интерполяцией значений в ключевых точках [86], создаёт «ступенчатые» артефакты вблизи особенностей поверхности.

Упрощение. Очевидным способом редуцирования данных в формате SDF является простое усреднение, где значения ($\hat{\mathbf{n}}$; d) каждого вокселя в упрощённой модели берутся, как средние соответствующих значений вокселей в исходной модели (с нормализацией усреднённой нормали). Однако этот способ не позволяет передать острые углы в упрощённой модели (рисунок 3.3).

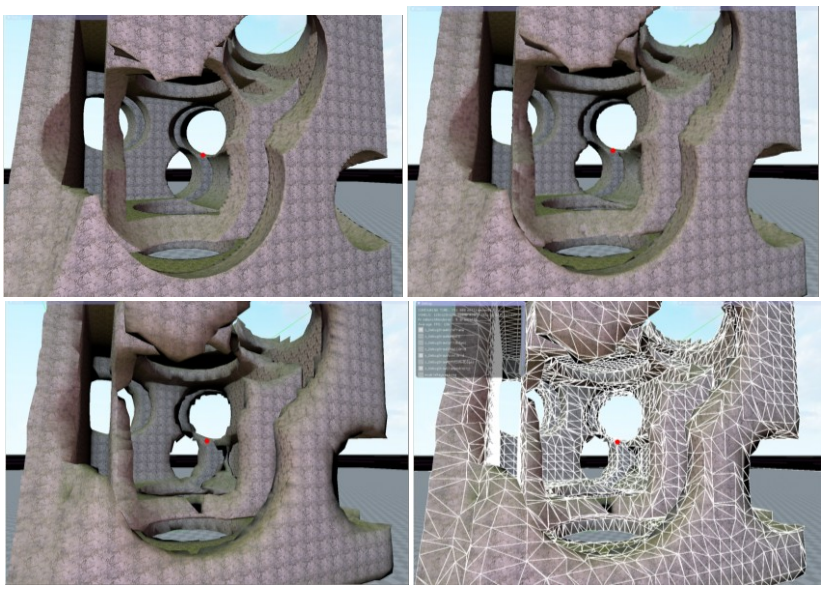


Рисунок 3.3 – Результаты упрощения SDF путём простого усреднения.

Преимущества формата SDF:

1) значительная гибкость в выборе методов триангуляции;

2) очень простая и регулярная структура;

3) простота и высокая скорость создания из аналитического описания;

4) простота реализации и высокая скорость операций редактирования.

Недостатки формата SDF:

1) худшее качество восстановления острых рёбер и углов по сравнению с форматами объёмных данных, использующими полноценные Эрмитовые данные;

2) низкая скорость создания из полигональных моделей;

3) сложность реализации составных, мультиматериальных областей;

4) большой объём занимаемой памяти в несжатом виде;

5) сложные схемы сжатия (требуются градиенты высокой точности).

Комментарии. Будучи наиболее «неявной» формой представления трёхмерных объектов из всех разработанных форматов, SDF позволяет очень просто реализовать все стандартные операции над функциями расстояния со знаком, включая операции конструктивной сплошной геометрии. Поскольку при триангуляции используются градиенты вместо полноценных Эрмитовых данных, острые углы и рёбра реконструированной поверхности становятся сглаженными и ребристыми, однако их качества должно быть достаточно для изображения в видеоиграх ударных кратеров и воронок от взрывов).

В демосцене популярны воксельные «движки», в которых сцена неявно описывается комбинацией SDF-примитивов. Вместо дискретизованного объёма хранится декларативное описание в виде дерева или списка из описаний простых областей и операций над ними (например, булевы операции, повторение области (domain repetition), гладкое сопряжение [87]), что обычно занимает гораздо меньше места и позволяет получать качественные полигональные сетки с практически неограниченным уровнем детализации. Для качественной реконструкции острых углов и рёбер при триангуляции позиции вершин целесообразно передвигать в направлении градиента SDF и помещать на изоповерхность [88]. Для устранения плохих треугольников и самопересечений можно дополнительно решать задачи «расталкивания» смежных вершин и распутывания вывернутых элементов сетки. Главным недостатком данного подхода является высокая вычислительная сложность определения расстояний и нахождения пересечений — избыточная детализация снижает эффективность работы с точным неявным представлением, а «интересные» сцены, как правило, состоят из большого числа SDF-примитивов и операций

редактирования, порядка нескольких тысяч. Эта проблема решается организацией списка в иерархическую структуру разбиения пространства (например, иерархию ограничивающих объёмов, октодерево), которая позволяет быстро определить участки пространства, содержащие границу области [89]. В силу различных причин (простота и размер кода, качество изображения), в демосцене для рендеринга сцены обычно используются не методы извлечения изоповерхности, а прямые методы визуализации: трассировка сфер (sphere tracing), бросание «снежков» (splatting), трассировка лучей (ray tracing), бросание лучей (ray casting) и т.д. При наличии SDF эти методы позволяют очень просто реализовать мягкие тени и эффекты глобального освещения (global illumination).

3.3 Воксельная решётка с Эрмитовыми данными

Для поддержки мульти-материальных, составных областей с качественной реконструкцией острых рёбер и углов поверхности был исследован формат воксельных данных, в котором помимо вокселей (индексов подобластей или материалов) присутствуют также и Эрмитовы данные (Hermite data) [31,46]: позиции точек пересечения рёбер ячеек с изоповерхностью и единичные нормали к поверхности в этих точках (см. Приложение А). В данном формате каждый блок ландшафта представлен «плотным» трёхмерным массивом вокселей (идентификаторов материалов) и разреженным массивом Эрмитовых данных. Последние хранятся только для активных рёбер ячеек, т.е. рёбер, концы которых имеют различные знаки (значения «внутри» и «снаружи» в случае бинарной воксельной решётки) или отличающиеся индексы подобластей (идентификаторы материалов).

В формальном виде каждый блок воксельного ландшафта имеет форму куба, на котором задана равномерная декартова сетка C, состоящая из n^3 ячеек. Вершинам (узлам) сетки C с координатами $\mathbf{v}_{i,j,k}$, где i, j, k — индексы вершины в координатах сетки, приписаны значения вокселей $v \in \mathbb{Z}$: индексы подобластей или материалов (см. рис. 3.1). На каждом ребре $\mathcal{E}$ сетки C, концы которого имеют отличающиеся значения v, отмечена точка $\mathbf{s}$ пересечения ребра $\mathcal{E}$ с интерфейсом (границей между подобластями) или внешней границей области, и единичная нормаль $\hat{\mathbf{n}}$ к границе в этой точке. Пример таких данных показан на рисунке 2.8а. Для экономии памяти координаты $\mathbf{s}$ хранятся как *направленные расстояния* (*directed distances*) [40] — значения $d \in [0..1)$ вдоль соответствующих рёбер ячейки (рисунок 3.4). Например, позиция точки $\mathbf{s}$ на ребре между вершинами $\mathbf{v}_{i,j,k}$ и $\mathbf{v}_{i+1,j,k}$ может быть восстановлена следующим образом:

$$\mathbf{s} = \left(1 - d_{i,j,k}\right) \mathbf{v}_{i,j,k} + d_{i,j,k}\, \mathbf{v}_{i+1,j,k}\,. \qquad (3.3)$$

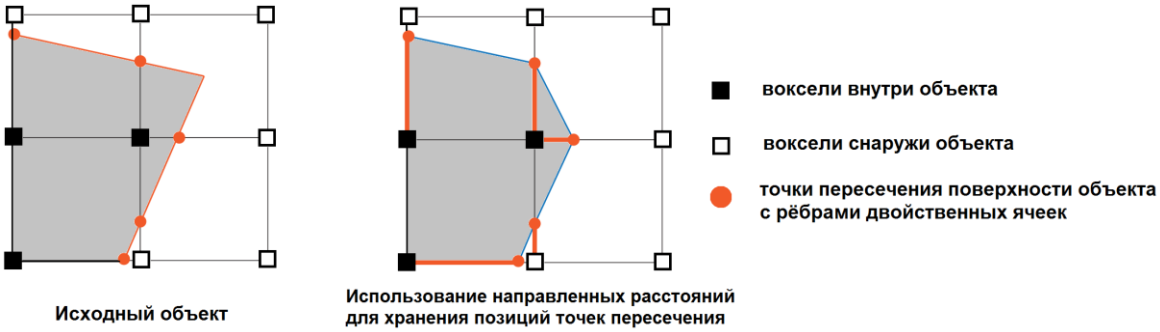


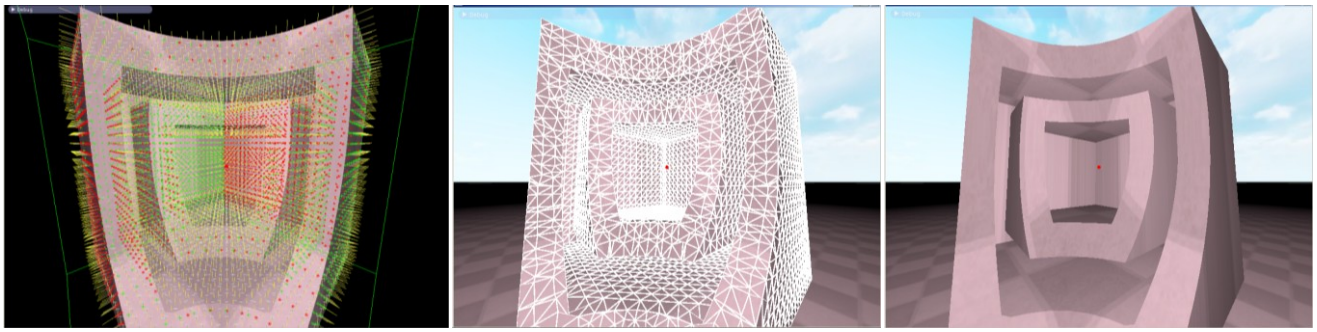
Рисунок 3.4 – Использование направленных расстояний для компактного хранения координат точек пересечения активных рёбер ячеек с поверхностью.

Таким образом, в данном формате каждый блок воксельного ландшафта представлен воксельной решёткой V, которая задана набором значений $v \in \mathbb{Z}$ в узлах кубической сетки C: $V = \{v_{i,j,k \in \mathbb{N}} \mid 0 \le i \le n,\ 0 \le j \le n,\ 0 \le k \le n\}$, и набором Эрмитовых данных $\{d; \hat{\mathbf{n}}\}$ на активных рёбрах ячеек сетки C.

В регулярной кубической решётке разбиения большинство рёбер являются общими для соседних ячеек. Поэтому Эрмитовы данные $\{d; \hat{\mathbf{n}}\}$ (четыре числа типа float) достаточно хранить только для каких-либо трёх смежных рёбер каждой ячейки (при этом для удобства вводится дополнительный слой ячеек) [28].

Триангуляция. Данный формат позволяет без каких-либо модификаций использовать любой ячеечный метод триангуляции, принимающий в качестве входных данных знакоопределённую решётку с Эрмитовыми данными. В частности, для триангуляции могут быть использованы описанные в первой главе алгоритмы и методы EMC [40], DC [31], DMC [56], CMS [33] и их разновидности.

В целом, данный формат обеспечивает высокое качество восстановления острых углов поверхности (рисунок 3.5), но для хранения набора Эрмитовых данных на регулярной решётке разбиения требуется в среднем в три раза больше памяти, чем для хранения приграничных вокселей (сёрфелей) в формате SDF.



а) б) в)

Рисунок 3.5 – Использование воксельной решётки с Эрмитовыми данными для качественной реконструкции острых рёбер и углов: а) визуализация точек пересечения и нормалей к поверхности); б) треугольная сетка поверхности; в) финальное изображение.

Способы создания. В [40] и [31] показано, что знакоопределённые решётки с Эрмитовыми данными могут быть построены из большинства наиболее распространённых типов входных данных: полигональные модели, неявно заданные поверхности, функции расстояния со знаком (SDF) и облака точек. На получение Эрмитовых данных затрачивается меньше вычислений, чем на создание воксельных данных в формате SDF, поскольку требуется находить точки пересечения с поверхностью луча в одном направлении (вдоль оси X, Y или Z) вместо поиска кратчайшего расстояния до поверхности во всех направлениях. Результаты вокселизации полигональных моделей зданий с помощью лучевой трассировки построенных из них BSP-деревьев показаны на рисунке 3.6.

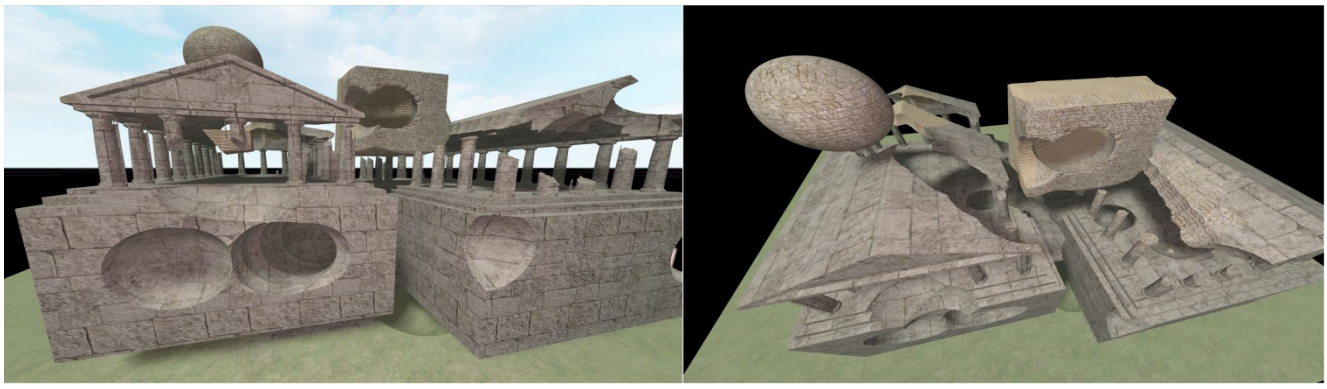


Рисунок 3.6 – Результаты вокселизации полигональных моделей в формат «воксельная решётка с Эрмитовыми данными с помощью лучевой трассировки BSP-деревьев. На сцену были помещены различные модели, к которым затем были применены операции закраски различными материалами и булевы операции вычитания.

Редактирование. Булевы операции сводятся к операциям над воксельной решёткой (массив индексов материалов) и 1D-сегментами (Эрмитовыми данными на активных рёбрах ячеек) [40]. Для простоты выполнения CSG-операций направленные расстояния могут быть дополнены знаками: значение расстояния получает отрицательный знак, если стартовая позиция отрезка расположена внутри объекта [40]. При этом правила выполнения CSG-операций остаются такими же, как и в случае с SDF (см. таблицу 3.1).

Одним из преимуществ данного формата является то, что булевы операции над блоками ландшафта могут выполняться полностью параллельно и независимо друг от друга, потому что для обеспечения целостности данных на границах блоков при редактировании не требуется подгружать данные с соседних блоков, как, например, в точечных представлениях с неявной связностью (раздел 3.5).

Одним из недостатков данного формата является сложность реализации операций сглаживания, которые создают зависимости между смежными блоками.

Сжатие. В данной работе для быстрых сжатия и распаковки воксельного массива используются алгоритмы группового (RLE) и словарного[8] кодирования, при этом коэффициент сжатия на практике достигает 25-30. Эрмитовы данные хранятся без сжатия: квантованные до трёх–шести байт единичные нормали, как показали эксперименты, являются непригодными для качественной реконструкции острых углов (на реконструированной поверхности возникают вмятины и «складки»).

Упрощение. Для редуцирования воксельных данных было реализовано простое усреднение, где значения $\{d; \hat{\mathbf{n}}\}$ каждого активного ребра в упрощённой модели берутся, как средние соответствующих значений на активных рёбрах в исходной модели (с нормализацией усреднённой нормали).

В отличие от SDF, данный способ не приводит к катастрофической потере острых углов в упрощённой модели (см. рис. 3.7), но возникает проблема выбора приоритетов материалов (иначе могут наблюдаться такие нежелательные артефакты, как, например, просвечивание земли сквозь траву в упрощённом блоке ландшафта) [10, p.53]. Подобный способ упрощения использовался для генерации уровней детализации в Upvoid Engine [24].

[8] http://www.lz4.org

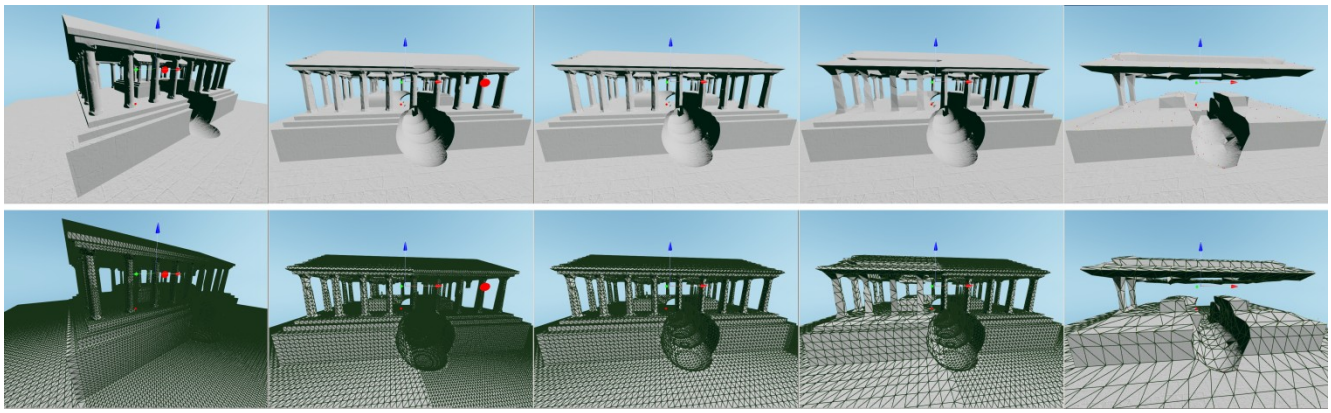


Рисунок 3.7 – Результаты генерации упрощённых уровней детализации воксельного ландшафта, представленного в виде воксельной решётки с Эрмитовыми данными, после применения булевых операций.

Преимущества воксельных решёток с Эрмитовыми данными:

1) гибкость: можно без каких-либо модификаций использовать любой ячеечный (cube-based) метод триангуляции (MC, EMC, DC, DMC, CMS и т.д.);

2) простая и регулярная структура;

3) высокое качество восстановления острых рёбер и углов;

4) встроенная поддержка составных, мультиматериальных областей;

5) простота выполнения CSG-операций;

6) высокая скорость создания из большинства типов входных данных;

7) возможность быстрой генерации грубых уровней детализации путём усреднения, при этом возможно восстановление острых углов поверхности;

8) возможность хранения дополнительных данных в точках пересечения рёбер ячеек с поверхностью (например, текстурных координат, материалов).

Недостатками данного формата являются:

1) высокое потребление памяти в распакованном виде (необходимо сжатие);

2) высокие требования к точности хранящихся единичных нормалей;

3) при редактировании и триангуляции нужно рассмотреть каждую ячейку;

4) отсутствие поддержки операций сглаживания;

5) дублирование данных между смежными блоками на их границах.

Комментарий. Воксельные решётки с Эрмитовыми данными являются наиболее популярным способом хранения воксельных данных для триангуляции двойственными методами (DC, DMC, CMS) с восстановлением острых углов и рёбер поверхности. В частности, данный

формат используется с 2012 года для представления ландшафта в Upvoid Engine [24] (для триангуляции используется алгоритм CMS [33]) и видеоигре Dual Universe (на 2018 год находящейся в разработке) для хранения планетарного ландшафта, пользовательских построек и космических станций (триангуляция выполняется алгоритмом DC [31]), где данный формат назван Hermite Data Field (HDF).

3.4 Лучевое представление с нормалями

Возможно, наиболее эффективным на сегодняшний момент представлением трёхмерных сплошных тел для быстрого выполнения булевых операций является лучевое представление (Ray Representation, Ray-Rep) [90]. Для возможности реконструкции острых углов поверхности объекта лучевое представление может быть дополнено нормалями к поверхности. Полученную форму представления можно рассматривать как дальшейшее развитие Marching Intersections [91] и расширение тройного лучевого представления (Triple Ray-Rep, Tri-Dexel) [92], в которых объект кодируется интервалами вдоль трёх координатных осей.

Лучевое представление с нормалями можно описать, как набор «стержней», пронизывающих объект вдоль каждого координатного направления, на каждом из которых хранятся координаты точек пересечения с поверхностью объекта и единичные нормали к поверхности объекта в этих точках [93−101] (рисунок. 3.8).

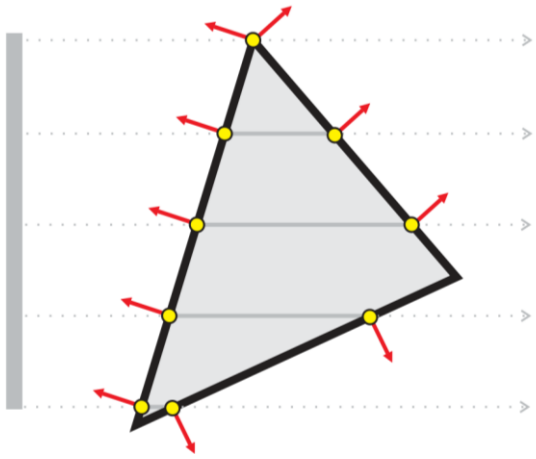


Рисунок 3.8 – Лучевое представление двумерного объекта. Показаны лучи вдоль одного координатного направления (из двух возможных). Чёрной сплошной линией обозначен контур объекта, жёлтыми точками отмечены точки пересечения лучей с поверхностью объекта, красными стрелками — нормали к поверхности в этих точках. Изображение взято из [101].

При построении лучевого представления из закрытого объекта на каждом луче будет создано чётное количество точек пересечения с поверхностью объекта — *сёрфелей* (*surfels*), или, соответственно, нечетное количество «сплошных», расположенных внутри объекта интервалов — *декселей* (*dexel*, сокр. от *depth pixel*). Для описания неоднородных, составных объектов сёрфели или дексели могут хранить индекс соответствующего материала [98].

В данной работе было реализовано тройное лучевое представление (далее Ray-Rep), дополненное нормалями для возможности реконструкции особенностей поверхности и материалами для возможности описания составных сред (рис. 3.9).

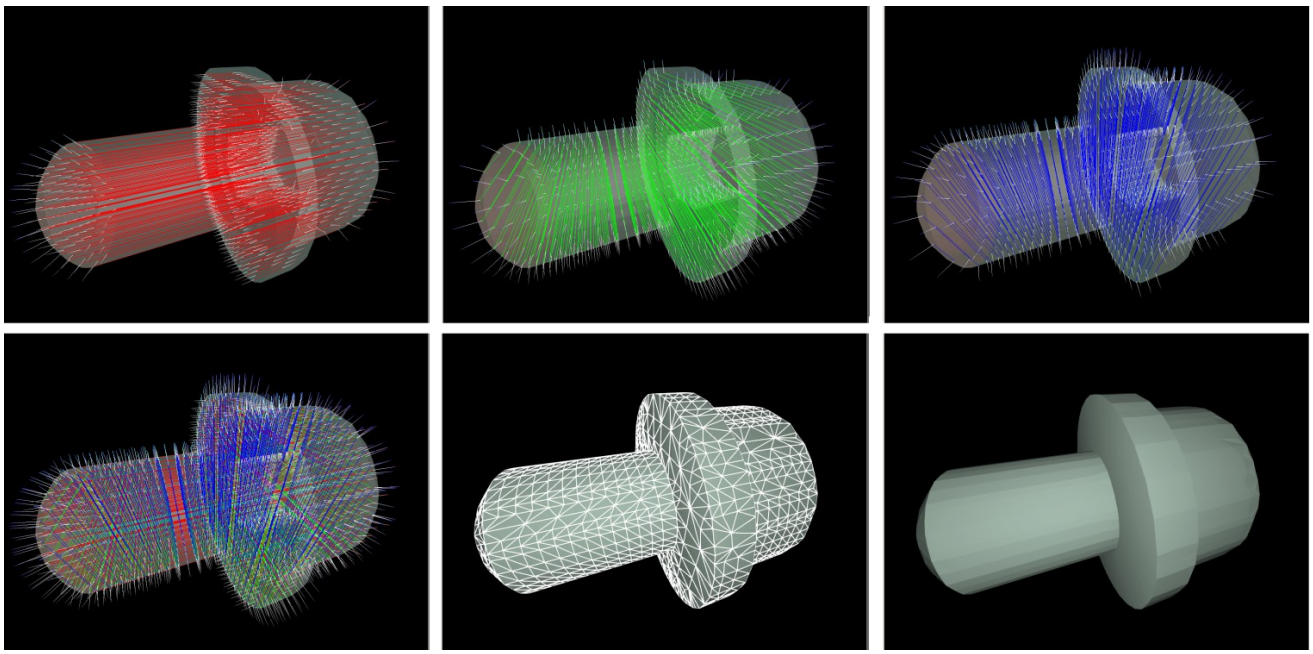


Рисунок 3.9 – Тройное лучевое представление с нормалями и реконструированная поверхность. Трёхмерный объект представлен набором интервалов вдоль трёх координатных направлений, на концах интервалов хранятся точки пересечения с поверхностью тела и единичные нормали.

Триангуляция. Как и в предыдущем формате данных, для триангуляции Ray-Rep могут быть использованы любые ячеечные методы: MC, EMC, DC, DMC, CMS и т.д. При этом рёбра кубических ячеек располагаются вдоль лучей Ray-Rep, а центры вокселей (или углы ячеек) находятся на пересечениях лучей. Неоднозначности типа «внутри»/«снаружи» разрешаются «голосом большинства» (majority voting): центр вокселя считается расположенным внутри объекта, если он находится на пересечении как минимум двух декселей (интервалов, расположенных полностью внутри объекта).

В отличие от воксельной решётки, для триангуляции которой необходимо рассмотреть все ячейки, Ray-Rep позволяет быстро отбросить пустые участки пространства и идентифицировать только ячейки, пересекающие поверхность. Пропуск пустого пространства может применяться для ускорения адаптивных методов триангуляция, в которых иерархическая структура строится сверху вниз на основе эвристических правил. При этом в качестве структуры разбиения пространства более предпочтительными, чем октодеревья, являются *k*D-деревья, поскольку они лучше адаптируются к особенностям поверхности.

Способы создания. Ray-Rep может быть создано теми же способами, что воксельные решётки с Эрмитовыми данными: в основе получения Ray-Rep лежит операция пересечения луча с поверхностью. Среди всех разработанных форматов трёхмерных данных, Ray-Rep обладает самой высокой скоростью построения из полигональных моделей. Ray-Rep могут быть получены с любой заданной точностью с помощью трассировки лучей или растеризации полигональной сетки.

В первом случае для создания Ray-Rep с разрешением n вокселей вдоль каждого координатного направления бросается $n \times n$ лучей, проходящих через центры вокселей (и вдоль рёбер ячеек решётки разбиения). На каждом луче записываются все точки пересечения луча с поверхностью объекта и единичные нормали к поверхности. Нечётное число точек пересечения указывает на ошибки в исходной модели или наличие в ней слишком мелких деталей. Расхождения в классификации вокселей разрешаются «голосом большинства» [102].

Второй подход предпочтителен для использования на GPU и позволяет почти в интерактивном режиме мультипроходным рендерингом (Depth Peeling) [93−100] или более современным методом Order Independent Translucency (OIT) [103] получать детальные Ray-Rep из массивных полигональных моделей.

После создания Ray-Rep для исправления неоднозначностей, возникающих из-за избыточности Ray-Rep, рекомендуется выполнять регуляризацию [99, 100].

Редактирование. Лучевое представление позволяет просто и эффективно реализовать операции конструктивной сплошной геометрии (CSG), офсеттинга, построения суммы и разницы Минковского. На GPU процессы создания Ray-Rep из полигональной модели, выполнения вышеперечисленных операций и триангуляции полученного Ray-Rep могут быть выполнены в реальном времени даже для Ray-Rep с разрешениями 257-513 лучей (или вокселей). Авторская реализация на CPU позволяет в интерактивном режиме выполнять серии булевых операций над Ray-Rep с разрешением в 65 лучей (раздел 5.4).

Булевы операции над Ray-Rep сводятся к операциям над лучевыми сегментами: слияние, сортировка и удаление нулевых областей [90−93,96,97,100,101].

В результате редактирования может нарушиться целостность Ray-Rep, поэтому для поддержания согласованности необходимо выполнять *регуляризацию* [99, 100]: удаление декселей с длиной меньше ребра ячейки, удаление или слияние изолированных декселей для исправления неоднозначностей и поддержания не более одной точки пересечения на рёбрах ячеек, поддержание чётного количества точек пересечения на каждом луче в замкнутой модели и т.д. В данной работе было реализовано хранение блоков воксельного ландшафта в виде Ray-Rep, что обернулось большой ошибкой. После выполнения длинных серий булевых операций интервалы на лучах Ray-Rep «рассинхронизируются», что приводит к появлению различных артефактов на границах между смежными блоками ландшафта (рисунок 3.10).

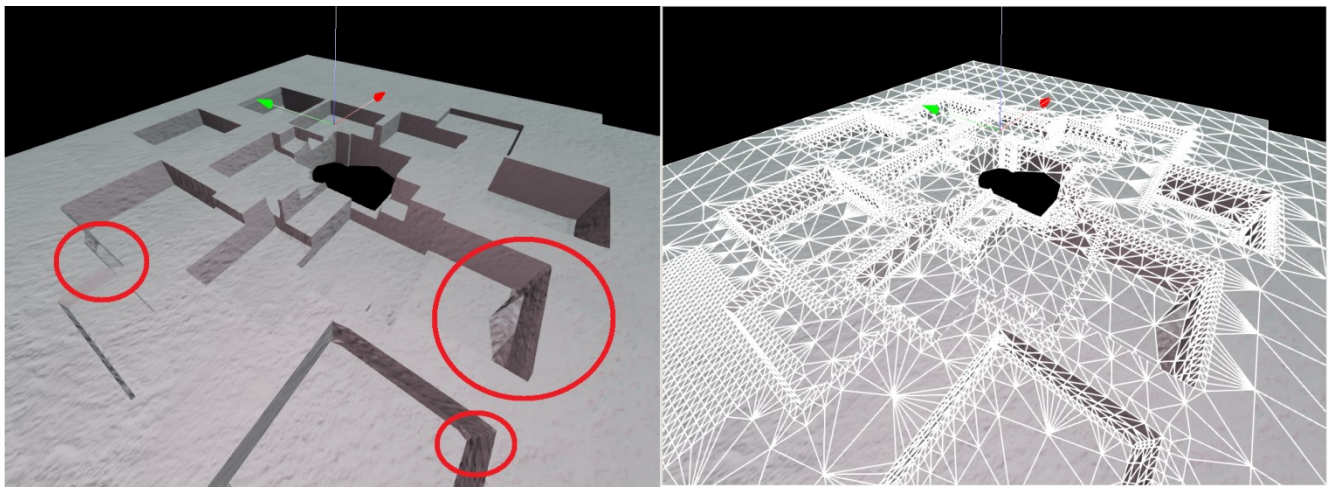


Рисунок 3.10 – *Слева*: артефакты на границах блоков ландшафта из-за рассинхронизации их данных, хранящихся в виде лучевого представления. *Справа*: треугольная сетка поверхности.

Сжатие. Ray-Rep является компактной, разреженной структурой данных, обладающей высокой избыточностью и не предназначенной для долговременного хранения и сжатие. Для снижения объёма рабочей памяти в авторской реализации Ray-Rep расстояния вдоль лучей огрубляются до 16-битных целых чисел, а нормали к поверхности в точках пересечения лучей с границей объекта хранятся с полной точностью (по три числа типа float на каждую нормаль).

Упрощение. Ray-Rep не предназначены для упрощения. Обычно Ray-Rep строится из полигональной модели для быстрого выполнения булевых операций и затем «выбрасывается». Генерация уровней детализации, сжатие и хранение Ray-Rep нецелесообразны.

Преимуществами лучевого представления с нормалями являются:

1) гибкость: можно без каких-либо модификаций использовать любой ячеечный (cube-based) метод триангуляции (MC, EMC, DC, DMC, CMS и т.д.);

2) очень простая и регулярная структура, допускающая параллельную реализацию как на CPU, так и на GPU;

3) простота и высокая скорость операций редактирования (самая высокая скорость выполнения булевых операций среди всех разработанных форматов);

4) высокая скорость создания из большинства типов входных данных (самая высокая скорость построения из сложных полигональных моделей);

5) высокое качество реконструкции острых рёбер и углов;

6) возможность хранения дополнительных данных в точках пересечения рёбер ячеек с поверхностью (например, текстурных координат, материалов);

7) низкое потребление рабочей памяти из-за разреженной структуры данных;

8) высокая скорость построения адаптивной триангуляции за счёт пропуска пустых областей.

Недостатки Ray-Rep вытекают в первую очередь из-за его избыточности:

1) избыточность приводит к неоднозначностям в классификации вокселей;

2) необходимость в регуляризации для поддержания целостности и исправления неоднозначностей после создания или редактирования;

3) высокая чуствительность к ошибкам округления;

4) отсутствие поддержки операций сглаживания;

5) при хранении объём занимаемого пространства выше, чем у других форматов объёмных данных, которые могут сжиматься в десятки раз.

Комментарии. Лучевое представление обладает избыточностью и поэтому нуждается в регуляризации. Поэтому отдельные блоки воксельного ландшафта целесообразно хранить в виде треугольных сеток и конвертировать в Ray-Rep и обратно при необходимости (современное графическое оборудование позволяет выполнять все эти операции «на лету»). Для представления объёмного ландшафта в лучевой форме с минимальной избыточностью может быть использовано только одно координатное направление (например, вертикальное, по аналогии с воксельными колонками [18]). Полученный таким образом гибрид между картами высот и воксельным представлением может быть триангулирован с помощью алгоритма [104]. Реализация Ray-Rep на GPU получила название многослойных карт глубины с нормалями (Layered Depth-Normal Images, LDNI) [93−101], поскольку до появления технологии GPGPU для хранения глубины пикселя

(расстояние от опорной плоскости до точки пересечения с поверхностью) и нормали использовались массивы текстур, как в многослойных картах глубины (Layered Depth Images, LDI). С появлением технологий CUDA, OpenCL и вычислительных шейдеров стало возможным на GPU строить списки точек для каждого фрагмента изображения [103]. В видеоигре The Tomorrow Children [32] данная технология используется для моделирования разрушаемого окружения, а многослойные карты глубины с нормалями названы Layered Depth Cube (LDC). В [101] лучевое представление с нормалями названо Z-list-buffer. Исходя из анализа находящейся в открытом доступе документации [66], можно предположить, что Voxel Farm [23] использует лучевое представление с нормалями для вокселизации архитектурных моделей.

3.5 Точечные представления с неявной связностью

Среди предложенных способов представлений для воксельных ландшафтов самой широкой функциональностью обладают точечные представления с неявной связностью (Point with Implicit Connectivity, PIC) [79]. В данном представлении 3D-объект хранится, как воксельная решётка и множество приграничных ячеек с соответствующими им точками, лежащими на поверхности объекта, для которых не заданы в явном виде понятия топологии связей или непрерывной поверхности. Объёмные данные организованы в готовом виде для создания поверхностной триангуляции каким-либо двойственным методом (например, DC или DMC). Точки становятся вершинами полигональной сетки, а топология их связи (информация о смежности, необходимая для построения сетки) неявно следует из воксельных данных: из знаков (значений «внутри»/«снаружи» в случае бинарной воксельной решётки) или индексов материалов, известных в углах каждой ячейки [78].

Эрмитовы данные также не хранятся в PIC в явном виде, а вычисляются в случае необходимости (например, перед выполнением операций редактирования). Для получения Эрмитовых данных выбранным методом триангуляции создаётся фрагмент полигональной сетки — частичный контур (partial contour), который затем пересекается с (активными) рёбрами, для которых требуется найти точки пересечения с поверхностью и единичные нормали к поверхности в этих точках.

Регулярное разбиение. PIC на регулярной решётке формально можно описать в виде воксельной решётки V и облака точек S. Каждый блок воксельного ландшафта имеет форму куба, на котором задана равномерная декартова сетка C, состоящая из n^3 ячеек. Воксельная решётка V

задана набором значений вокселей (индексов подобластей или материалов) в узлах сетки C (в углах ячеек $c_{i,j,k}$):

$$V = \{v_{i,j,k} \in \mathbb{Z} \mid \ i,j,k \in \mathbb{N},\ 0 \le i,j,k \le n\}.$$

Облако точек S задано множеством вершин, которые существуют только внутри неоднородных (с наличием различных материалов или знаков) ячеек сетки C (т.е. внутри ячеек, углы которых не имеют совпадающие материалы или знаки):

$$S = \{\mathbf{p}_{i,j,k} \in \mathbb{R}^3 \mid i,j,k \in \mathbb{Z},\ 0 \le i,j,k < n \ \wedge c_{i,j,k} \text{ — неоднородная ячейка}\}.$$

Позиции вершин $\mathbf{p}_{i,j,k}$ помещаются вблизи острых углов поверхности, представляющей границу раздела между различными материалами. Если для триангуляции используется метод DC, то внутри каждой граничной ячейки будет создано по одной острой вершине, если DMC, то может быть создано от одной до четырёх острых вершин в зависимости от знаковой конфигурации ячейки.

Таким образом, каждый блок воксельного ландшафта в формате PIC хранится, как 3D-массив вокселей и разреженный 3D-массив точек, которые становятся вершинами поверхностной сетки. В целях экономии памяти позиции острых вершин квантуются относительно границ (AABB) ячейки (достаточно 8 бит на каждую координатную компоненту).

Адаптивное разбиение. В [79] предложено адаптивное PIC на основе знакоопределённого октодерева (см. подраздел 2.3.1), которое для возможности выполнения CSG-операций помимо «серых» листовых узлов (пересекающих поверхность объекта) может содержать «чёрные» листовые узлы (расположенные полностью внутри объекта) и «белые» узлы (расположенные полностью снаружи объекта). Каждый серый листовой узел-ячейка содержит позицию острой вершины, лежащей на поверхности внутри ячейки, знаки (значения «внутри» и «снаружи») в углах ячейки, и время последнего изменения (для построения частичного контура при выполнении CSG-операций). Использование октодерева приводит к дублированию знаков (индексов материалов) в углах смежных ячеек.

Триангуляция PIC сводится к соединению вершин, позиции которых уже известны и хранятся явно, в отличие от всех других рассмотренных форматов объёмных данных. Для соединения вершин можно использовать любой ячеечный двойственный метод триангуляции (DC, DMC, Closest Point Contouring и т.д.). Поскольку двойственные методы триангуляции обладают особенностью inter-cell dependency (см. раздел 4.3), при генерации бесшовной сетки, соединяющей каждый блок ландшафта с его соседями, необходимо включать в триангуляцию прилегающие к нему приграничные ячейки с соседних блоков. Для избежания обращений к внешней памяти и

возможности распараллеливания процесса триангуляции на практике обычно применяется концепция «призрачных» ячеек (англ. “ghost cells”) [153], которая заключается в добавлении слоёв фиктивных ячеек для дублирования данных, принадлежащих другим блокам. (Призрачные ячейки физически расположены внутри соседнего блока.) Другими словами, для каждого блока в памяти хранятся, помимо собственных данных, также копии прилегающих к нему ячеек с соседних блоков. Образованные призрачными ячейками полигоны служат только для соблюдения граничных условий (исключения разрывов полигональной сетки и расчёта «гладких» вершинных нормалей) и не должны включаться в конечную полигональную сетку (см. подраздел 4.3.1). Для предотвращения разрывов достаточно перекрытия смежных блоков на слой толщиной в одну ячейку, для вычисления гладких вершинных нормалей необходимо перекрывать блоки на слой толщиной в две ячейки.

Редактирование. PIC позволяет эффективно выполнять булевы операции (CSG). Для необходимой части блока двойственным алгоритмом триангуляции (DC или DMC) строится фрагмент полигональной сетки и ищутся его пересечения с активными рёбрами ячеек для получения Эрмитовых данных. Затем над воксельной решёткой (точками) и Эрмитовыми данными (отрезками) выполняются CSG-операции, как описано в разделе 3.3, после чего вычисляются новые позиции острых вершин. В целом, даже с квантованными до восьми бит позициями вершин PIC обеспечивает довольно высокое качество реконструкции острых углов после выполнения над ландшафтом булевых операций (рисунок 3.11).

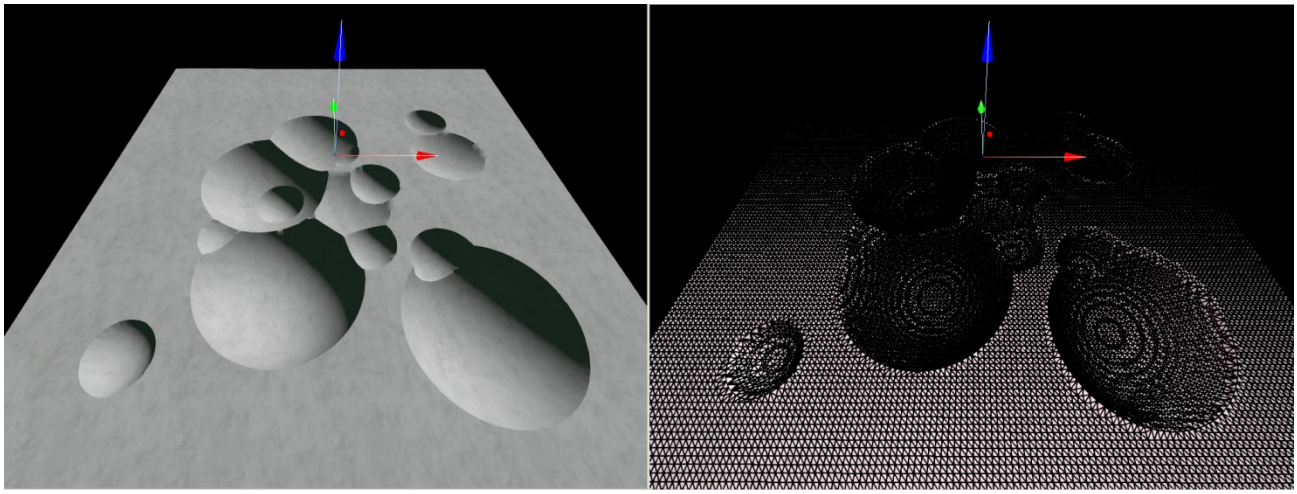


Рисунок 3.11 – Использование точечного представления для качественной реконструкции острых рёбер и углов после выполнения булевых операций и треугольная сетка поверхности.

PIC является единственным из предложенных форматов, поддерживающим операции сглаживания поверхности ландшафта (что может потребоваться, например, для сглаживания нежелательных острых углов и рёбер ландшафта или для улучшения качества треугольников в построенной сетки). Сглаживание поверхности выполняется путём сдвигания позиции вершины активной ячейки, затронутой операцией сглаживания, в центр масс вершин из соседних активных ячеек. Для получения хороших результатов для усреднения достаточно брать вершины тех соседних ячеек, которые имеют с текущей ячейкой общую активную грань (ячейка кубической формы может иметь до шести активных граней), как в алгоритме Surface Nets [30].

После выполнения операций редактирования перед триангуляцией блоков необходимо обновить призрачные ячейки всех затронутых блоков (англ. “halo exchange”), что усложняет реализацию и затрудняет распараллеливание системы.

Преимущества точечных представлений с неявной связностью:

1) высокая скорость триангуляции двойственными методами: позиции острых вершин внутри ячеек уже вычислены, и остаётся только их соединить;

2) полный контроль над позициями вершин в ячейках, что позволяет легко реализовать операцию сглаживания (smoothing) ландшафта и предоставляет возможность корректировки вершин для повышения качества построенной сетки; после выполнения операций редактирования при триангуляции не требуется заново вычислять позиции острых вершин в незатронутых ячейках блоках;

3) высокое качество восстановления острых рёбер и углов;

4) встроенная поддержка составных, мультиматериальных областей на регулярной решётке (в адаптивном варианте возникает дублирование данных);

5) возможность хранения дополнительных данных в острых вершинах (например, коэффициент гладкости, «затенение» точки (ambient occlusion));

6) снижение вычислительной сложности операций редактирования и триангуляции за счёт пропуска однородных частей пространства в адаптивном варианте PIC;

7) низкое потребление памяти в распакованном, несжатом виде (данный формат — самый компактный из остальных предложенных, например, в PIC для каждой активной ячейки достаточно хранить позицию острой вершины против трёх векторов (единичных нормалей) и трёх скаляров (направленных расстояний) в воксельной решётке с Эрмитовыми данными);

8) высокая степень сжатия (для сжатия позиций вершин в ячейках разумно использовать квантование и энтропийное кодирование с предсказанием).

Недостатки точечных представлений с неявной связностью:

1) данный формат может быть использован только с определённым двойственным методом триангуляции;

2) усугубление проблемы inter-cell dependency: при редактировании блока требуется подгружать данные из соседних блоков для избежания артефактов (разрывов сетки и резких переходов освещения) на границах блоков;

3) сложность операций редактирования, поскольку точки пересечения рёбер ячеек с поверхностью и нормали к поверхности не хранятся в явном виде;

4) низкая скорость создания: требуется нахождение острых вершин;

5) необходимость синхронизации приграничных данных («призрачных» ячеек) между смежными блоками, что значительно усложняет реализацию и затрудняет распараллеливание.

Комментарии. Среди рассмотренных форматов представления объёмов точечное представление является самым технологичным и обладает наиболее широкой функциональностью, однако создаёт зависимости между смежными блоками ландшафта (при редактировании блока могут потребоваться данные его соседей). На основе анализа свободно доступной документации [66] и видеоматериалов можно предположить, что точечное представление на регулярной решётке используется для хранения ландшафта в Voxel Farm [23].

3.6 Разработанный формат для компактного хранения уровней детализации ландшафта

Как было сказано ранее, воксели используются в качестве робастного представления, обеспечивающего целостность ландшафта в результате редактирования, а для визуализации ландшафта в настоящее время наиболее практично использовать растеризацию треугольной сетки, аппроксимирующей поверхность воксельного ландшафта. Для обеспечения эффективной работы системы для визуализации и редактирования ландшафтов к формату представления воксельных данных и к построенным треугольным сеткам выдвигаются различные требования, следовательно, целесообразно разделить объёмное представление ландшафта от отображения его поверхности. Это разделение имеет ряд преимуществ:

1) блоки воксельного ландшафта могут хранится в различных формах представления, а для их бесшовной триангуляции может использоваться единственный формат данных, при этом в процессе триангуляции не требуется подгружать воксельные данные смежных блоков;

2) для генерации упрощённых уровней детализации блоков не требуется использовать (редактируемые) воксельные данные блоков, что уменьшает потребление рабочей памяти;

3) если не требуется выполнять редактирование воксельного ландшафта, то для его визуализации достаточно хранить только данные, используемые для построения триангуляции;

4) данные для построения адаптивной триангуляции поверхности ландшафта могут быть максимально оптимизованы на этапе препроцессинга (например, для уменьшения количества отображаемых треугольников).

Для компактного хранения поверхности ландшафта с наличием острых углов и рёбер и различными материалами был разработан адаптивный вариант PIC, основанный на линейном представлении знакоопределённого октодерева, описанном в главе 2.

Новый формат данных получил название SLOG (Signed Linear Octree with Geometry) по аналогии с форматом SOG (Signed Octree with Geometry), использующимся в программе для «починки» моделей PolyMender [105]. SLOG представляет собой знакоопределённое линейное октодерево, в котором хранятся только листовые узлы-ячейки, пересекающие поверхность (см. подраздел 2.4.1).

Триангуляция. Поскольку в SLOG хранятся уже найденные позиции острых вершин поверхности, то, как и в PIC, процесс триангуляции сводится только к соединению вершин ячеек. Для непосредственной триангуляции SLOG хорошо подходит алгоритм [72], предложенный в подразделе 2.4.2.

Редактирование. SLOG является статичной структурой данных, не предназначенной для выполнения операций редактирования.

Упрощение. Для генерации упрощённых уровней детализации из SLOG может быть использован стандартный алгоритм упрощения знакоопределённого октодерева, описанный в подразделе 2.3.1. QEF-матрицы, используемые для определения позиций вершин и оценки ошибки при слиянии ячеек, могут быть получены путём триангуляции октодерева. Упрощение знакоопределённого линейного октодерева в формате SLOG происходит в два этапа:

1) инициализация QEF-матриц вершин с помощью триангуляции октодерева;

2) упрощение октодерева путём слияния его ячеек снизу вверх.

На первом этапе выполняется триангуляция линейного октодерева с помощью алгоритма из раздела 2.4, но создаётся только массив вершин, в котором каждой вершине **v** сопоставлена симметричная 4x4 матрица **Q** (см. подраздел 2.3.1). Инициализация матриц **Q** происходит таким же образом, как в алгоритмах упрощения полигональных сеток методами слияния вершин (vertex

clustering) [41, 42] и удаления рёбер (edge collapse) [43]. Изначально каждой вершине присваивается нулевая матрица **Q**. В процессе триангуляции для каждого созданного треугольника находится матрица $\mathbf{Q_T}$, которая позволяет вычислить квадрат расстояния $\mathbf{x}^{\mathrm{T}}\mathbf{Q_T}\mathbf{x}$ от плоскости треугольника $\mathbf{p}$ до произвольно заданной точки **x**. Плоскость треугольника может быть задана уравнением $\hat{\mathbf{n}}^{\mathrm{T}}\mathbf{v} + d = 0$, где $\hat{\mathbf{n}}^{\mathrm{T}}$ — нормаль к плоскости $\mathbf{p}$, а $\mathbf{v}$ — какая-либо точка, лежащая в плоскости треугольника (например, центр масс треугольника). Если взять $\mathbf{p} = \left[\hat{\mathbf{n}}_x\ \hat{\mathbf{n}}_y\ \hat{\mathbf{n}}_z\ d\right]^{\mathrm{T}}$, то $\mathbf{Q}_{\mathrm{T}} = \mathbf{p}\mathbf{p}^{\mathrm{T}}$. Полученная таким образом матрица $\mathbf{Q}_{\mathrm{T}}$ добавляется к матрицам **Q** каждой из трёх вершин, образующих треугольник. Чтобы результат упрощения октодерева не зависел от формы и количества треугольников, $\mathbf{Q}_{\mathrm{T}}$ умножается на площадь треугольника. Кроме того, для лучшей реконструкции острых углов в упрощённом октодереве желательно разбивать каждый четырёхугольник по острой диагонали (см. подраздел 2.4.3). Результатом выполнения первого этапа алгоритма упрощения является массив вершин, в котором каждой вершине **v** сопоставлена QEF-матрица **Q**, являющаяся суммой отмасштабированных по площади матриц $\mathbf{Q_T}$ треугольников, в которые входит вершина **v**.

На втором этапе сканируется отсортированный массив ячеек и производится слияние ячеек, имеющих общий родительский узел (см. подраздел 2.3.1).

Преимуществом данного алгоритма генерации упрощённых уровней детализации является то, что не требует никакой дополнительной информации, помимо знакоопределённого октодерева с более детального уровня детализации, и не хранит истории предыдущих упрощений (Memoryless Simplification)[106]. Результаты работы данного алгоритма упрощения можно увидеть на рисунке 3.12.

Сжатие. В SLOG хранятся только граничные, пересекающие поверхность ячейки, что является более эффективным с точки зрения расхода памяти, чем PIC [79]. Квантование сжимает координаты острых вершин в четыре раза без заметных потерь визуального качества, поскольку октодерево «сгущается» к поверхности объекта, а также исключает возможность появления видимых разрывов (обусловленных погрешностями вычислений при работе с числами с плавающей запятой) между смежными блоками воксельного ландшафта.

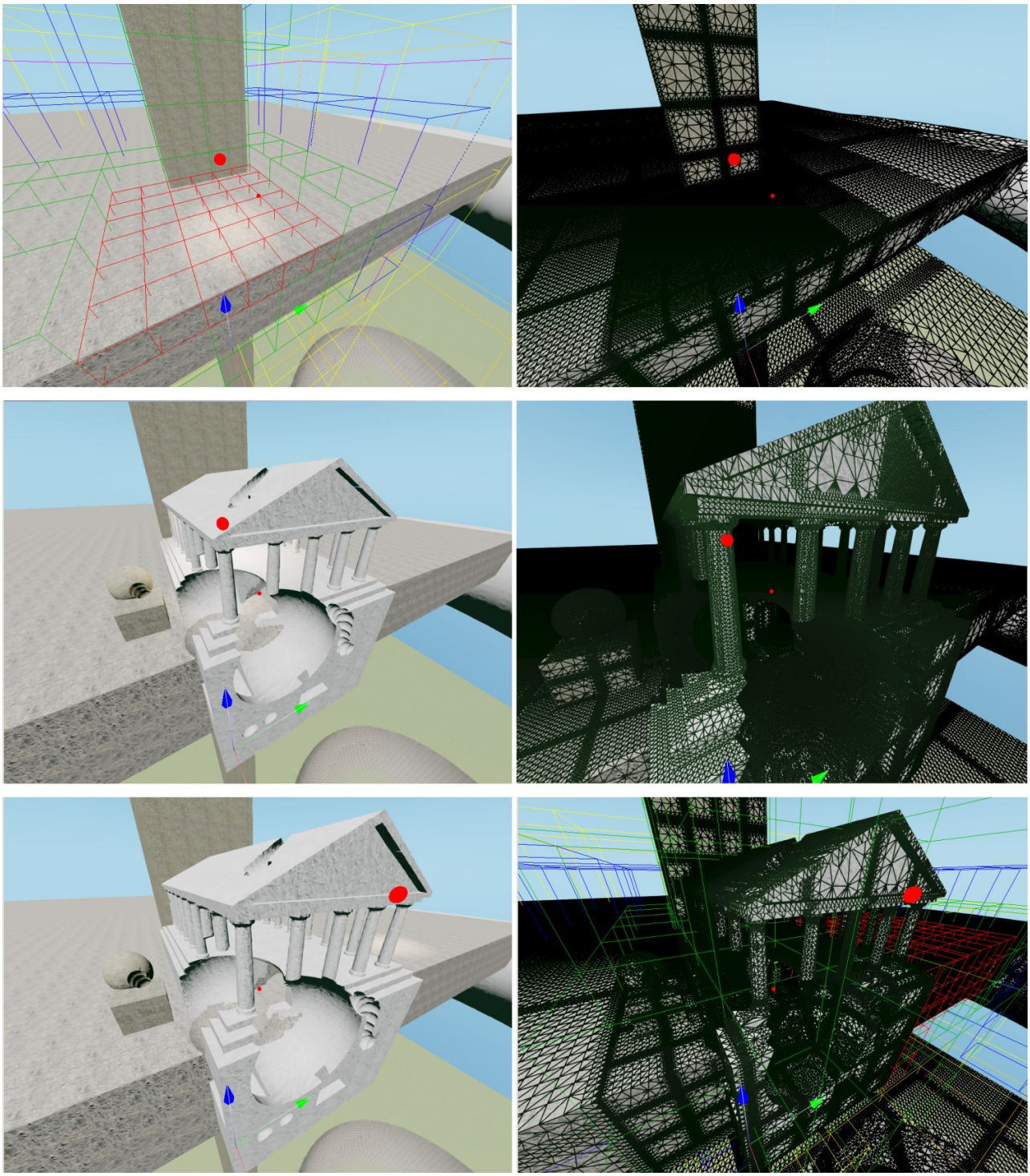


Рисунок 3.12 – Использование SLOG для генерации уровней детализации ландшафта. *Верхний ряд*: исходный ландшафт до начала редактирования. Разрешение каждого блока — 32^3 ячеек. *Средний ряд*: ландшафт на самом детальном уровне детализации после применения булевых операций. *Нижний ряд*: упрощённый ландшафт после того, как камера наблюдателя сместилась вправо, и для центральных блоков были созданы более грубые уровни детализации. Слева показана треугольная сетка для каждого случая.

Для дальнейшей экономии памяти, минимизации дублирования данных и повышения эффективности сжатия было принято решение хранить только один индекс материала для каждой ячейки (вместо восьми значений для углов ячейки). В текущей реализации выбирается сплошной материал (не обозначающий пустое пространство), который больше всего присутствует в ячейке (имеет наибольшее количество ненулевых индексов в углах ячейки). При упрощении октодерева запрещается слияние неоднородных ячеек, чтобы построенная сетка отображала внутреннее строение ландшафта. На рисунке 3.13 показан пример сетки, которая адаптируется к материалам подобластей, помимо поверхностных особенностей.

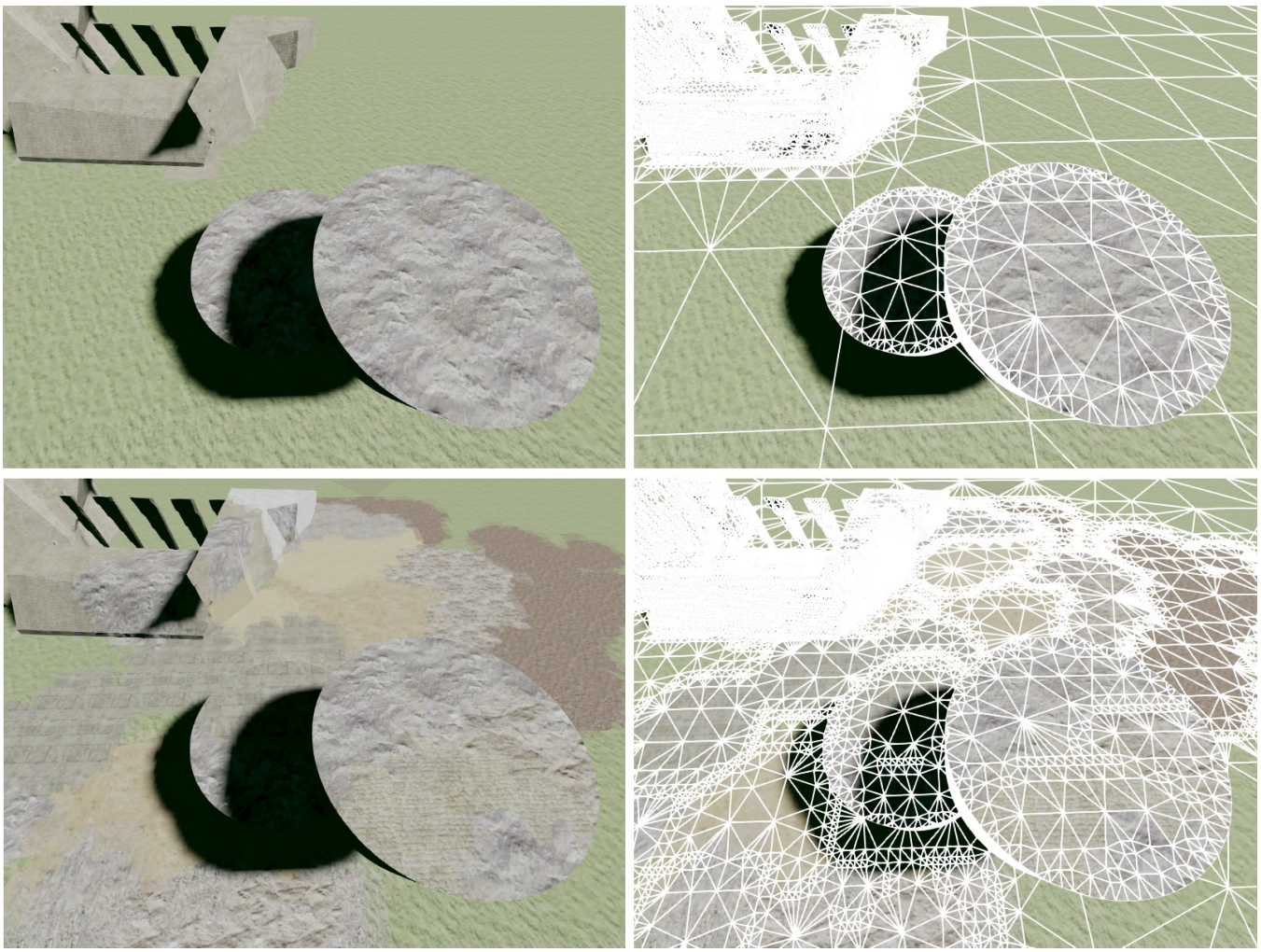


Рисунок 3.13 – Использование SLOG для триангуляции ландшафта с различными материалами. Вверху представлено изображение сцены с плоскими участками: октодерево сгущается вблизи поверхности ландшафта и адаптируется к кривизне поверхности. Внизу представлено изображение этой же сцены после «раскраски» различными материалами: октодерево также локально сгущается в неоднородных областях.

Сжатие SLOG для хранения во внешней памяти происходит в четыре этапа:

1) сортировка ячеек по их мортон-кодам;

2) сжатие и запись массива мортон-кодов;

3) запись позиций острых вершин и знаков;

4) сжатие и запись массива материалов ячеек.

На первом этапе ячейки октодерева сортируются по соответствующим кодам Мортона. Для сжатия кодов не имеет значения порядок сортировки, но рекомендуется сортировать в убывающем порядке для возможности «раннего выхода» из radix-сортировки в алгоритме триангуляции, предложенном в главе 2.

На втором этапе для сжатия отсортированных мортон-кодов применяется дельта-кодирование: вместо непосредственной записи числа в выходной поток записывается разница (дельта) относительно предыдущего числа. Массив мортон-кодов (адресов ячеек) содержит уникальные, неповторяющиеся элементы, поэтому дельты всегда больше нуля. Среди них много повторяющихся последовательностей из близких к нулю значений, особенно много единиц встречается на последних, наиболее детализированных уровнях линейного октодерева. Из-за соображений простоты и скорости сжатия/декодирования серии из нескольких дельта-значений пакуются в 32-битные машинные слова с помощью кодировки simple-9 [107].

На третьем этапе происходит сохранение квантованных позиций острых вершин и знаков ячеек (их вторичное сжатие в данной работе не реализовано).

На четвёртом этапе ячейки обходятся в сортированном порядке для построения цепочки из индексов материалов, которая затем сжимается с помощью RLE. При этом для повышения степени сжатия используется свойство пространственной когерентности точек, расположенных на Z-кривой.

Преимущества формата SLOG для хранения поверхности ландшафта:

1) ориентация на проблемную область;

2) высокая скорость триангуляции методами ADC и ADMC;

3) высокое качество восстановления острых рёбер и углов;

4) низкое потребление памяти в распакованном, несжатом виде;

5) высокая скорость сжатия и декомпрессии при небольшой степени сжатия.

Недостатки данного формата:

1) данный формат «заточен» под методы триангуляции ADC и ADMC;

2) необходимость в преобразовании из другого представления;

3) необходимость в перестроении после изменения исходных данных;

4) низкая скорость создания;

5) отсутствие возможности редактирования.

Комментарии. Для квазистатичного ландшафта в SLOG для каждого блока целесообразно хранить приграничные ячейки соседних блоков для избежания обращений к внешней памяти при бесшовной триангуляции (см. раздел 4.3). Если смежные блоки имеют одинаковые LoD, то разумно использовать максимально оптимизированную на этапе подготовки треугольную сетку блока. Это повышает производительность системы в общем случае (когда ландшафт почти не изменяется) за счёт её понижения в худшем случае (изменяется много блоков). Для вычисления непрерывных повершинных нормалей потребуется хранить приграничный слой толщиной в две ячейки, как в Voxel Farm [23,66].

3.7 Вокселизация полигональных моделей

Большой практический интерес представляет вокселизация произвольных полигональных моделей: преобразование объекта из непрерывного граничного представления в дискретное объёмное. Например, сконвертированные в объёмное представление модели могут использоваться для помещения искусственных элементов на процедурно-сгенерированный воксельный ландшафт.

Для вокселизации модели необходимо классифицировать каждый воксель по отношению к её поверхности (лежит ли центр данного вокселя внутри или снаружи объекта?). Для получения Эрмитовых данных, использующихся для реконструкции острых углов поверхности, требуется находить точки пересечения рёбер ячеек с границей объекта и нормали к поверхности в этих точках [40,31].

В данной работе вокселизация полигональных моделей для помещения на ландшафт может осуществляться с помощью лучевого представления (Ray-Rep) или BSP-деревьев. Из исходной полигональной модели на этапе препроцессинга строится Ray-Rep или BSP-дерево, которое затем используется для выполнения булевых операций с воксельным ландшафтом. Обе структуры данных позволяют эффективно определять значения «внутри»/«снаружи» для центров вокселей, находить пересечения рёбер ячеек с поверхностью и выполнять булевы операции.

Вокселизация моделей с помощью трассировки лучей. Для построения Ray-Rep поверхность объекта пересекается лучами, проходящими через центры вокселей. При трассировке закрытого объекта на каждом луче в Ray-Rep должно быть создано чётное количество точек

пересечения (сёрфелей) или нечётное количество «сплошных» сегментов (декселей) (см. раздел 3.4). На отдельно взятом луче воксель считается расположенным внутри поверхности, если его центр лежит внутри декселя, т.е., начиная от начала луча, перед центром вокселя находится нечётное количество точек пересечения (Parity Counting).

В результате конвертирования даже самых простых и абсолютно корректных моделей из-за численных погрешностей и вырожденных случаев могут возникнуть ошибки вокселизации. Например, при создании Ray-Rep из сеточной модели куба, состоящей из 24 треугольников (по два треугольника на каждой грани) луч может «попасть» в ребро, являющееся общим для двух смежных треугольников (ditriangle), и на луче будет зарегистрировано два пересечения с гранью куба вместо одного. В редких случаях луч также может «попасть» в вершину, входящую в несколько треугольников, или пройти очень близко и параллельно к какой-либо лицевой плоскости куба. Очевидно, что результат вокселизации с помощью лучевой трассировки зависит от ориентации исходной модели. В [90,92,108] описаны возможные способы разрешения таких проблемных ситуаций. В [109] предложен робастный алгоритм проверки пересечения треугольника с отрезком.

Для повышения надёжности результатов вокселизации лучи, проходящие через центры вокселей, бросаются вдоль трёх и более различных направлений [102]. В Triple Ray-Rep лучи бросаются вдоль трёх взаимно перпендикулярных направлений). Обычно результаты классификации каждого вокселя (значения «внутри»/«снаружи») совпадают на всех лучах, но из-за вычислительных погрешностей могут возникнуть расхождения. Неоднозначности классификации разрешаются «голосом большинства» (majority voting): воксель считается расположенным внутри объекта, если на большинстве лучей его центр классифицирован, как лежащий «внутри» объекта.

Для оптимизации поиска пересечений лучей с поверхностью массивных моделей, состоящих из большого количества треугольников, необходимо использовать ускоряющую структуру данных [108], которая позволит быстро отбрасывать треугольники, заведомо не пересекающие заданный луч. В качестве ускоряющих структур могут использоваться регулярные решётки (uniform grids) и иерархические структуры для организации объектов или разбиения пространства: вложенные решётки (nested grids), октодеревья (octrees), *k*D-деревья (*k*D-trees), иерархии ограничивающих объёмов (Bounding Volume Hierarchy, BVH), иерархии ограничивающих интервалов (Bounding Interval Hierarchy, BIH), BSP-деревья или деревья двоичного разбиения пространства (Binary Space Partitioning trees) и т.д. Решётки, октодеревья, *k*D-деревья и BSP-деревья являются структурами данных для разбиения пространства (space

partitioning) на (выпуклые) ячейки, и каждый полигон исходной модели может одновременно находиться в нескольких ячейках, которые он пересекает. BIH и BVH, напротив, являются способами иерархической организации объектов (data partitioning) на основе их ограничивающих оболочек, поэтому для их построения не требуется разбиения или многократного добавления полигонов исходной модели в несколько листовых узлов-ячеек, однако, в процессе лучевой трассировки полученные точки пересечения необходимо сортировать вдоль направления луча (поскольку подобласти пространства дочерних узлов могут пересекаться). В настоящей реализации BVH (AABB-дерево) занимает вдвое больше памяти, чем BIH (spatial *k*D-tree), но на практике разница между ними по времени лучевой трассировки полигональных моделей незначительна. Примеры BIH и BVH показаны на рисунках 3.14 и 3.15.

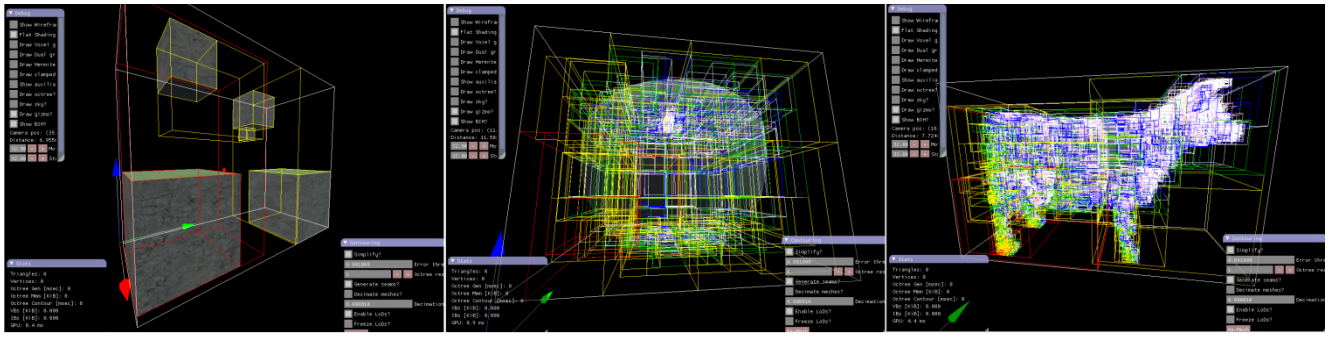


Рисунок 3.14 – Использование BVH в качестве ускоряющей структуры для трассировки лучей.

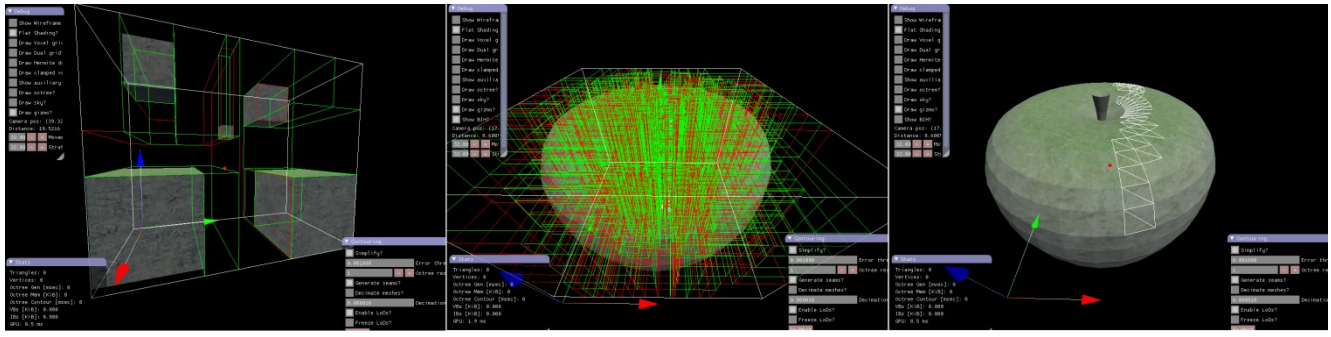


Рисунок 3.15 – Использование BIH в качестве ускоряющей структуры для трассировки лучей.

Для построения качественных (оптимизированных для трассировки лучей) BVH, *k*D- и BSP-деревьев на каждом шаге желательно выбирать плоскость разбиения в соответствии с эвристикой SAH (Surface Area Heuristic), впервые предложенной в работе [110] и формализованной в [111,112]. SAH основана на предположении, что вероятность пересечения случайного луча с (выпуклой) областью пространства пропорциональна площади поверхности этой области.

Вокселизация моделей с помощью BSP-дерева. BSP-дерево может быть использовано не только как ускоряющая структура, но и как форма представления сплошных объектов [113,114]. BSP-дерево представляет собой бинарное дерево, иерархически разбивающее пространство плоскостями: каждому внутреннему узлу дерева сопоставляется разделяющая плоскость, а каждому листовому узлу — (выпуклая) область пространства, ограниченная разделяющими плоскостями родительских узлов. В BSP-дереве для нахождения пересечений лучей с поверхностью модели не требуется хранить треугольники, а достаточно хранить только внутренние узлы и (уникальные) плоскости, образованные полигонами исходной модели (Polygon-aligned или auto-partitioning BSP tree). Разделяющая плоскость хранится для каждого внутреннего узла дерева (node-storing BSP tree), а листовые узлы, представляющие образованные разделяющими плоскостями выпуклые области пространства, помечаются флагом «внутри»/«снаружи» (solid leaf-labeled BSP tree). Хорошая практическая реализация подобного BSP-дерева с эвристикой SAH содержится в APEX SDK[9], входящего в состав PhysX от NVIDIA.

При лучевой трассировке BSP-дерева гарантируется, что полученные точки пересечения будут строго упорядоченными вдоль направления лучей. Основным недостатком BSP-дерева, помимо долгого времени построения (до нескольких минут), является высокая чувствительность к ошибкам округления (roundoff errors) и необходимость в численных допусках (plane epsilon). Ошибки накапливаются в результате разбиения полигонов при построении дерева и могут быть уменьшены применением робастной процедуры разбиения полигона плоскостью [108]. Во время трассировки лучей плоскости BSP-дерева следует считать более «толстыми», нежели при создании дерева, иначе могут быть пропущены некоторые пересечения [108] (рисунок 3.16).

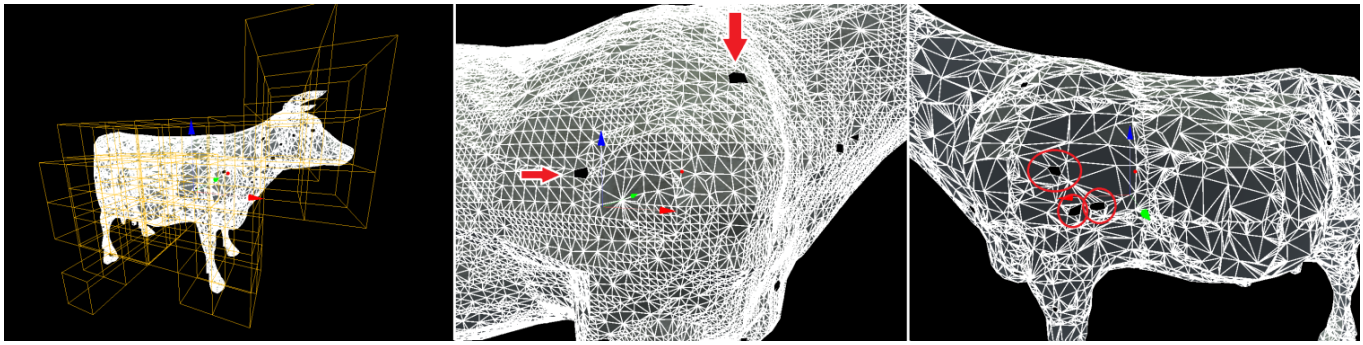


Рисунок 3.16. Результаты триангуляции объёмных данных, полученных трассировкой BSP-дерева: реконструированная поверхность содержит пропущенные участки из-за ошибок округления. (Полигональные сетки слева и справа были упрощены путём удаления рёбер [41].)

[9] https://github.com/NVIDIAGameWorks/PhysX-3.3/APEXSDK/shared/internal/src/authoring/ApexCSG.cpp

Несмотря на эти недостатки, BSP-деревья обладают множеством полезных свойств, и были выбраны в качестве непрерывного (в отличие от Ray-Rep) объёмного представления для помещения полигональных моделей на воксельный ландшафт, состоящий из блоков с различными разрешениями.

На этапе препроцессинга из полигональной модели строится BSP-дерево. Для помещения полигональной модели на ландшафт BSP-дерево масштабируется, поворачивается и переносится в заданное положение в глобальной, мировой системе координат. В текущей реализации плоскости дерева хранятся отдельно от его узлов в виде массива четырёхкомпонентных векторов, что позволяет быстро трансформировать плоскости с помощью SIMD инструкций: для этого достаточно перемножить каждую плоскость-вектор с матрицей $(\mathbf{M}^{-1})^{T}$, где **M** — матрица трансформации из локального пространства модели в мировые координаты.

Затем между воксельными данными и трансформированным BSP-деревом выполняются булевы операции объединения или разности, реализация которых зависит от используемого формата воксельных данных.

Комментарии. При недостаточном разрешении воксельной решётки реконструированная модель из-за пропуска деталей поверхности внутри ячеек и геометрического алиасинга будет содержать всевозможные дефекты (рисунок 3.17).

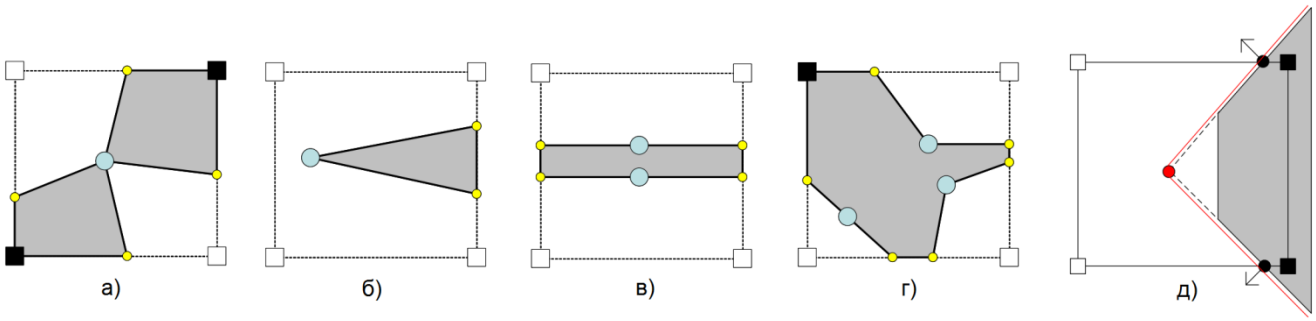


Рисунок 3.17. Ошибки реконструкции поверхности, возникающие при недостаточном разрешении решётки разбиения: а) создание сингулярной вершины; б-г) пропуск фрагментов поверхности, пересекающих ребра чётное количество раз; д) ошибочное создание острых углов. Изображения взяты из [93].

Для предотвращения этих дефектов необходимо, чтобы размер ячейки разбиения (или размер вокселя) более чем в два раза превышал толщину самой тонкой «стенки» в исходной модели [115]. Для уменьшения алиасинга можно строить Ray-Rep или BSP-дерево из огрублённой, упрощённой модели (для возможности их масштабирования необходимо подготовить набор упрощённых моделей в нескольких разрешениях), или использовать другие методы вокселизации

[54,105, 79,116], в которых позиция вершины каждой ячейки вычисляется с учётом плоскостей (и площадей) всех треугольников, пересекающих ячейку, в отличие от Ray-Rep и BSP-дерева, где используются только пересечения рёбер ячейки с поверхностью модели.

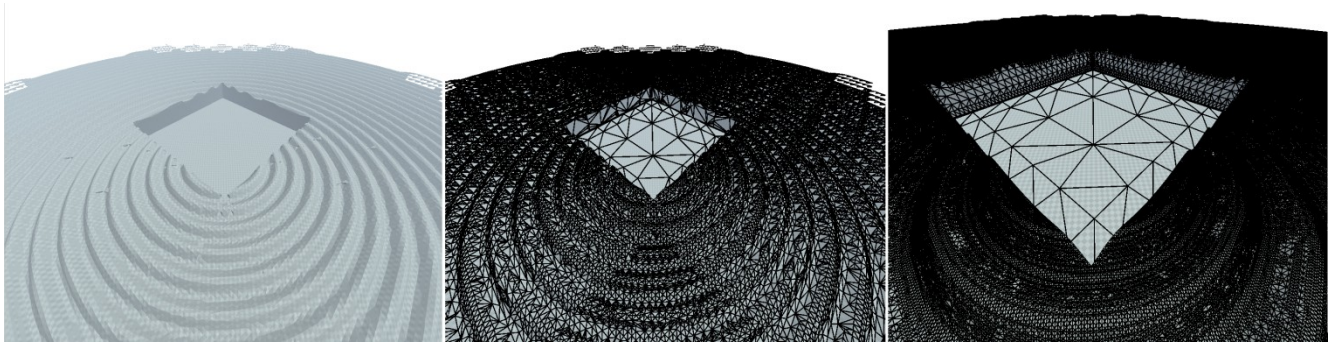
Стоит отметить, что BSP-дерево позволяет строить закрытое объёмное представление при наличии умеренных дефектов (отверстий и самопересечений) в исходной полигональной модели [117] без необходимости применения сложного программного обеспечения для «починки» моделей. Для получения корректного Ray-Rep, напротив, исходная полигональная сетка должна быть полностью закрытой и двусторонней, что не всегда соблюдается на практике (как, например, для большинства моделей, созданных руками художников), поэтому для устранения небольших дефектов и ошибок прибегают к специализированному математическому и программному обеспечению для «ремонта» моделей [105], рассмотрение которого выходит за рамки данной работы.

3.8 Результаты экспериментов

Эксперименты проводились на компьютере, оснащённом процессором Intel® Core™ i7-2600K @ 3.40 ГГц и 16 Гб оперативной памяти.

В первом эксперименте оценивались скорость выполнения булевых операций и качество восстановления острых рёбер и углов поверхности.

Тестовая сцена построена из неявного описания: разности концентрической синусоидальной волны, заданной уравнением $\sin\left(\sqrt{x^2 + y^2}\right) = z$, и куба. Сцена состоит из 8^3 блоков, каждый блок содержит 32^3 ячеек и может иметь до четырёх уровней детализации, которые получены упрощением октодерева, построенного из данных блока. Каждый блок хранит редактируемые воксельные данные в виде лучевого представления с нормалями (раздел 3.4), из которых для каждого уровня детализации строятся линейные октодеревья в формате SLOG (см. раздел 3.6). На рисунке 3.18 можно увидеть, как адаптивная триангуляция уменьшает количество треугольников на плоских участках поверхности в центре сцены. Бесшовная триангуляция сцены выполнялась методом, описанным в подразделе 4.3.1.



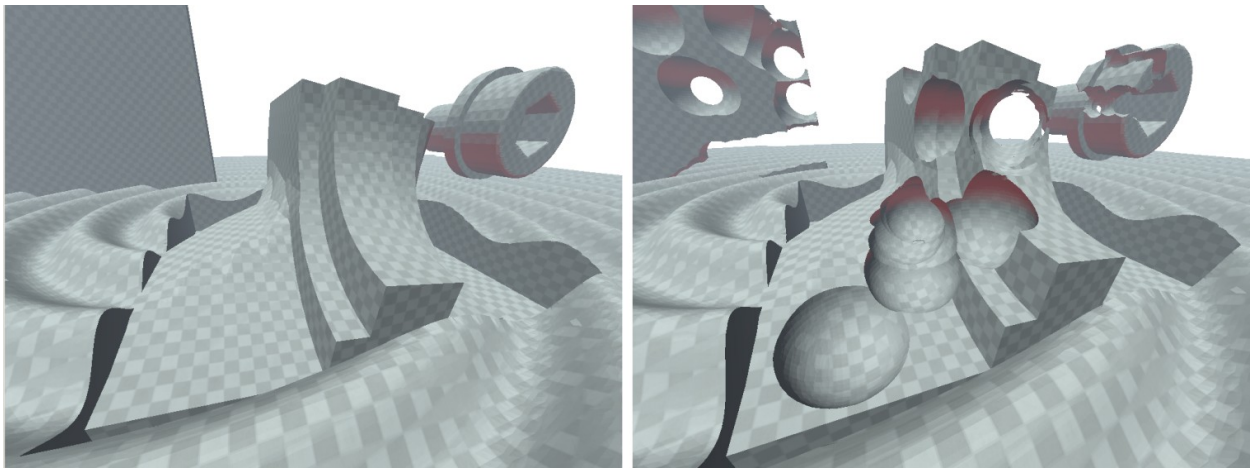
а) б) в)

Рисунок 3.18 – а) Тестовая сцена; б) визуализация треугольной сетки; в) визуализация сетки, полученной с помощью адаптивной триангуляции без использования уровней детализации.

На сцену были помещены CAD-объекты с поверхностными особенностями и плоскими и кривыми гранями: модели "Box", "Fandisk" и "Bolt" (рис. 3.14, а). Затем к сцене в интерактивном режиме были применены булевы операции. На рисунке 3.19, б) показаны результаты модификации сцены после вычитания сфер.

Можно увидеть, что полученная сетка содержит довольно качественную реконструкцию острых углов и рёбер поверхности. Среднее время между запросом на CSG-операцию и отображением результата для каждого блока составляло 8-12 мсек (включая изменение лучевого представления и регенерацию всех уровней детализации в формате SLOG). Если при редактировании было затронуто много блоков, то в процессе выполнения в фоновом потоке булевых операций и операций перестроения треугольной сетки в течение короткого времени становятся заметными швы между смежными блоками. Кроме того, из-за избыточности лучевого представления после операций редактирования возможно появление швов и разрывов на границах между смежными блоками.

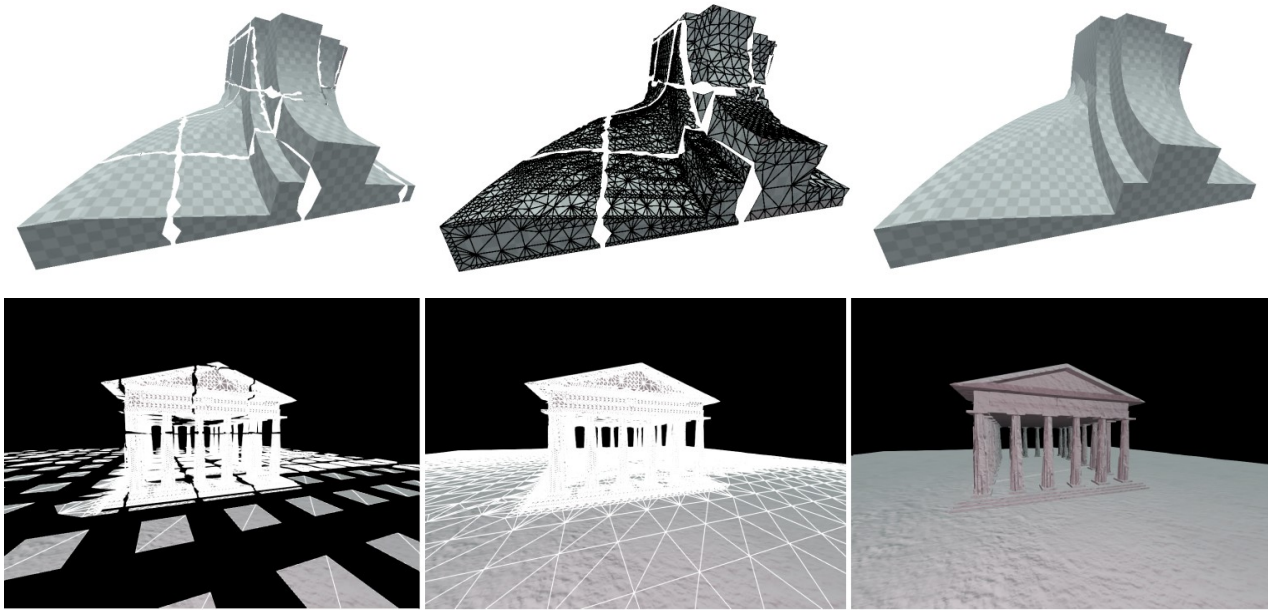
Поэтому в качестве основного формата для хранения редактируемых воксельных данных лучше использовать воксельную решётку с Эрмитовыми данными (трёхмерный массив материалов вокселей с точками пересечения и нормалями), которая обеспечивает такое же качество реконструкции особенностей поверхности, но обладает несколько меньшей скоростью выполнения CSG-операций. При этом для обеспечения интерактивности (при выполнении булевых операций и повторной триангуляции) оптимальный размер блока составил 32^3 ячейки.



а) б)

Рисунок 3.19 – Тестовая сцена а) до и б) после выполнения булевых операций вычитания.

На рисунке 3.20 показаны результаты бесшовной триангуляции тестовых сцен, содержащих острые рёбра и углы. Из полигональной модели вначале строилось лучевое представление. Затем для каждого блока сцены создавалось линейное октодерево в формате SLOG. Видно, что октодерево хорошо упрощается на плоских участках, параллельных координатным плоскостям: в нижнем ряду на рисунке 3.20 видно много блоков, которые содержат по четыре активные ячейки, что приводит к созданию сеток, состоящих из одного четырёхугольника.

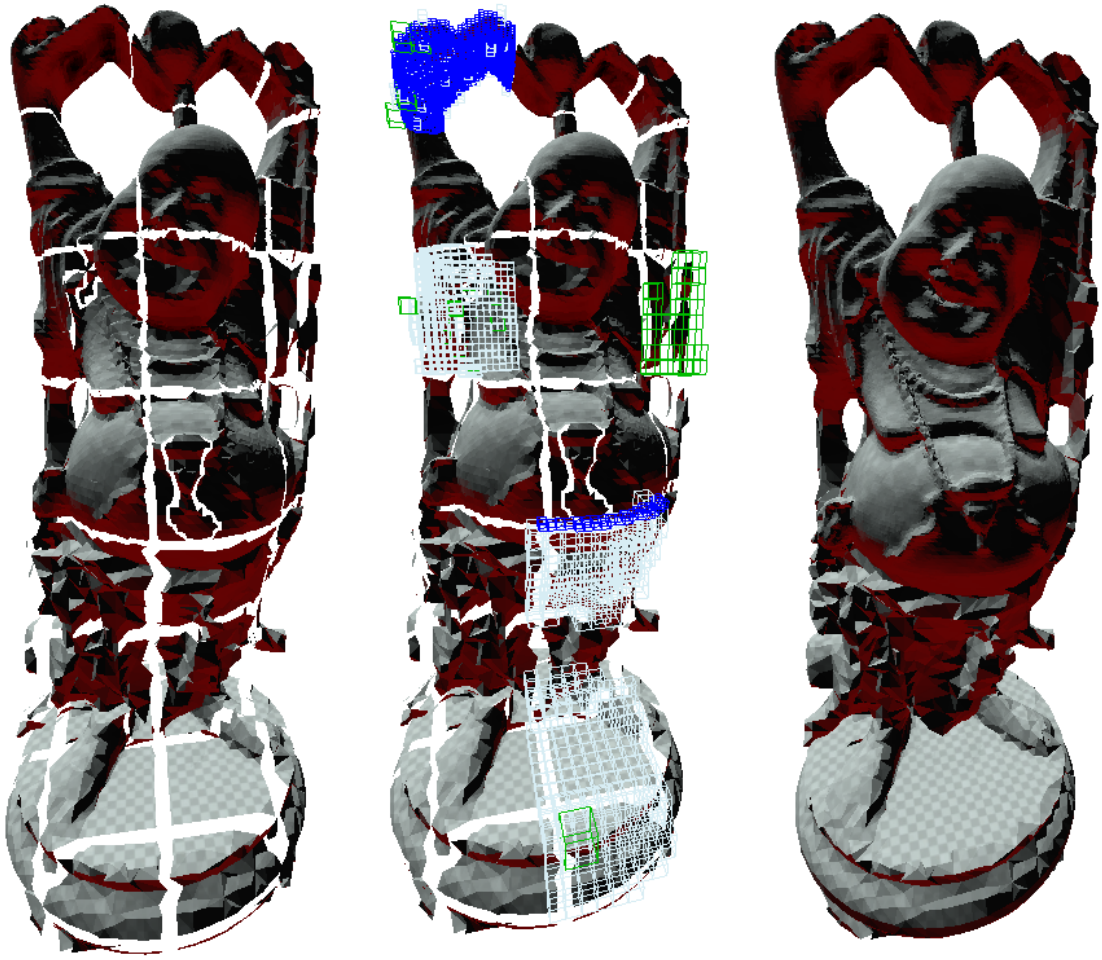


а) б) в)

Рисунок 3.20 – Бесшовная триангуляция моделей "Fandisk" (сверху) и "Parthenon" (снизу), разбитых на одинаковые блоки, с помощью октодеревьев: а) триангуляция блоков по отдельности; б) построенная бесшовная треугольная сетка; в) результат бесшовной триангуляции сцены.

SLOG (см. раздел 3.6) — унифицированное представление поверхностей, разработанное для компактного хранения и бесшовной внеядерной триангуляции воксельных ландшафтов с острыми углами и различными материалами. SLOG представляет собой линейное октодерево, оптимизированное для использования с предложенным в подразделе 2.4.2 алгоритмом триангуляции.

На рисунке 3.21 показаны разбиение модели "Happy Buddha", состоящей из 1087716 треугольников, на одинаковые кубические блоки с различными уровнями детализации, визуализация ячеек SLOG и результаты бесшовной внеядерной триангуляции.



а) б) в)

Рисунок 3.21 – Бесшовная триангуляция модели "Happy Buddha": а) разбиение модели на блоки с различными разрешениями; б) визуализация ячеек октодеревьев (показано несколько блоков); в) бесшовная треугольная сетка. Камера наблюдателя находится слева сверху от модели.

Главными недостатками SLOG являются нагрузка на CPU и необходимость обращения к данным смежных блоков при построении бесшовной триангуляции, а также необходимость в перестроении SLOG после редактирования блока.

Во втором эксперименте для исследования зависимости коэффициента сжатия SLOG от степени «разреженности» объекта к тестовым моделям (рисунок 3.22) применялись булевы операции объединения и разности со сферами.

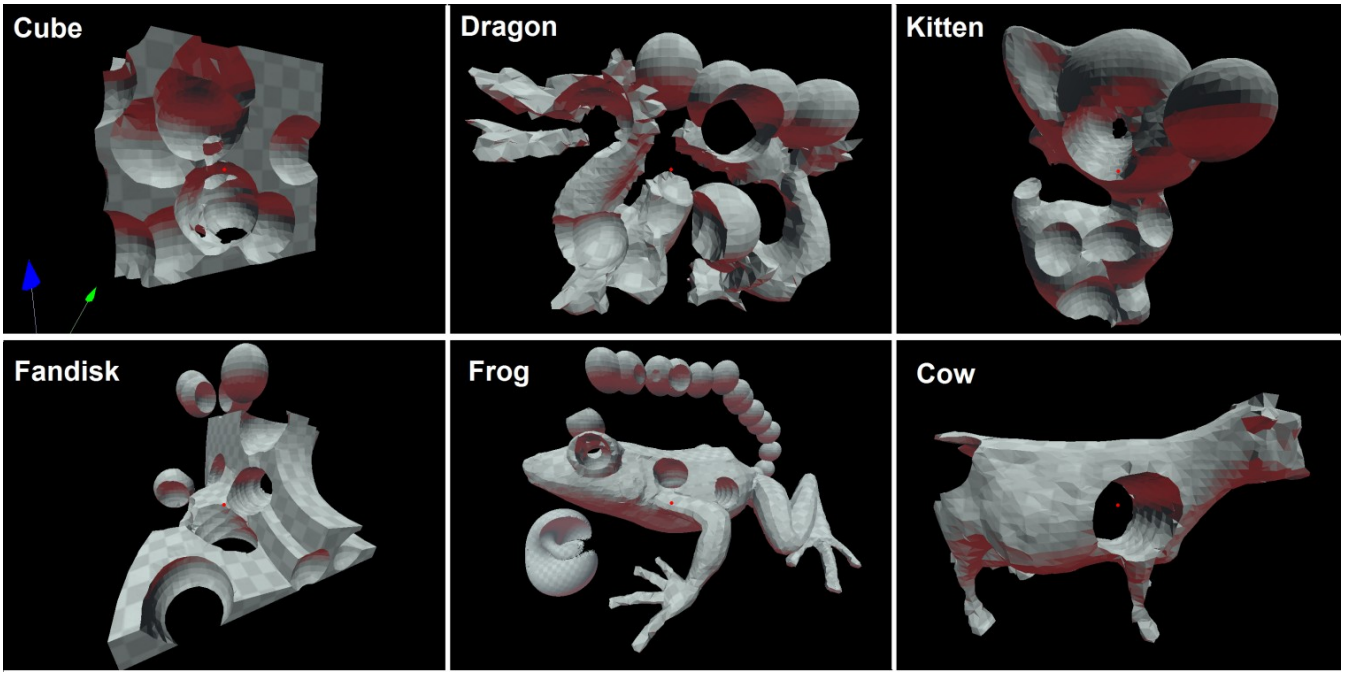


Рисунок 3.22 – Модели, используемые для тестирования степени сжатия данных в формате SLOG.

В текущей реализации SLOG каждая ячейка линейного октодерева занимает в памяти 16 байт (из них 4 байта занимает адрес (код Мортона) ячейки, 4 байта — квантованная позиция острой вершины и знаки углов, один байт — индекс доминантного материала ячейки, а оставшаяся часть отведена под служебную информацию и выравнивание). Адреса ячеек сжимаются с помощью simple-9 [186] в 4-6 раз, но коэффициент сжатия может быть гораздо выше при использовании кодов переменной длины. Позиции острых вершин «сжимаются» квантованием в четыре раза. Средний коэффициент сжатия SLOG составил 3.3. Все тестовые модели состоят из одного материала, поэтому «сплошные» цепочки материалов сжимаются с помощью RLE в среднем в 125 раз. В таблице 3.2 показаны результаты сжатия исходных моделей (без применения CSG-операций).

Таблица 3.2 – Результаты сжатия поверхностей, представленных в формате SLOG.

Название модели и разрешение воксельной решётки	**Количество ячеек линейного октодерева**	**Размер сжатых адресов ячеек (байт)**	**Размер сжатых индексов материалов ячеек (байт)**	**Общий размер сжатого октодерева (байт)**	**Общий коэффициент сжатия октодерева**
Cube (33^3)	1 862	1 604	16	9 084	3.280
Dragon (65^3)	9 015	6 376	72	42 524	3.392
Kitten (65^3)	9 185	6 896	72	43 724	3.361
Fandisk (65^3)	16 384	5 176	50	30 138	3.304
Frog (65^3)	8 242	5 708	66	38 758	3.402
Cow (65^3)	4 923	3 676	40	23 424	3.363

Вопреки ожиданиям, модель куба имеет наименьший коэффициент сжатия, хотя является самой простой и «регулярной» среди тестовых моделей. После выполнения булевых операций общий коэффициент сжатия поверхности куба, изображённой на рисунке 3.9, достиг величины 3.438, а затем снижался по мере роста доли пустого пространства (в результате применения операций вычитания).

3.9 Основные выводы по третьей главе

Исследованы способы представления и форматы хранения редактируемых объёмных данных с информацией для восстановления острых углов поверхности, которые могут быть использованы для моделирования воксельных ландшафтов с острыми углами.

Предложен компактный формат для хранения поверхности воксельного ландшафта в готовом для триангуляции виде, основанный на использовании линейных октодеревьев и названный SLOG (Signed Linear Octree with Geometry). Описано расширение формата для хранения поверхности составных областей. Предложен способ сжатия SLOG с помощью разностного и RLE-кодирования и показана эффективность этого способа. Общий коэффициент сжатия поверхности, представленной в формате SLOG, превысил 3.

4 Разработка методов и алгоритмов для визуализации воксельных ландшафтов с использованием методов синтеза виртуальной реальности

В предыдущих главах были предложены алгоритм для триангуляции изоповерхностей с восстановлением её острых углов и рёбер и форматы воксельных данных для редактирования и хранения отдельных блоков воксельного ландшафта.

В данной главе предлагаются решения для визуализации сверхбольших сцен, структуры данных которых могут не умещаться в оперативной памяти компьютера. Описаны методы для бесшовной триангуляции воксельных ландшафтов, состоящих из блоков с различными уровнями детализации.

4.1 Существующие подходы к визуализации ландшафтов с различными уровнями детализации

Рендеринг ландшафтов — хорошо исследованная проблема в компьютерной графике [118]. Для визуализации больших территорий в интерактивном режиме необходимо использовать методы рендеринга с применением различных уровней детализации (Levels of Detail, LoD) и динамической подкачки (streaming) данных с целью оптимального распределения ограниченных вычислительных ресурсов для уменьшения нагрузки на CPU и GPU до приемлемого уровня. В частности, при отрисовке ландшафта с помощью растеризации размер спроецированных на экран треугольников должен многократно превышать размеры пикселя, иначе резко снижается эффективность растеризации и возникает алиасинг. Самые первые и ранние LoD-схемы фокусировались на создании в каждом кадре минимальной триангуляции, которая бы наилучшим образом аппроксимировала набором треугольников исходный рельеф с точки зрения наблюдателя.

На практике ландшафт, как правило, разбивается на отдельные блоки, что позволяет подгружать (подкачивать), обрабатывать (в том числе и параллельно), сохранять и отрисовывать только необходимые его части. Блочная декомпозиция является практически единственной и эффективной возможностью работы с большими ландшафтами и в том или ином виде используется почти во всех опубликованных работах и существующих программных реализациях. Основной проблемой при этом является поддержание непрерывности и целостности ландшафта на границах между смежными блоками (см. раздел 4.3).

Для визуализации больших территорий (вплоть до гигантских ландшафтов планетарного масштаба, которые едва умещаются даже во внешней памяти) из исходной сцены строится *многомасштабное* (*multi-scale*) представление. Обычно это древовидная структура данных, корень которой содержит информацию для первоначального, грубого (с низким разрешением) приближения всей сцены, а дочерние узлы содержат либо уточняющую информацию, либо более детализированные аппроксимации сцены, вплоть до листовых узлов, которые хранят исходные данные ландшафта на самом детальном уровне детализации (с максимальным разрешением). Данная структура данных используется методами рендеринга с различными уровнями детализации или со многими разрешениями (multiresolution) для того, чтобы в процессе визуализации сократить количество обрабатываемых данных и время вычислений и быстро выбрать адекватную аппроксимацию исходных данных с минимальной потерей визуального качества.

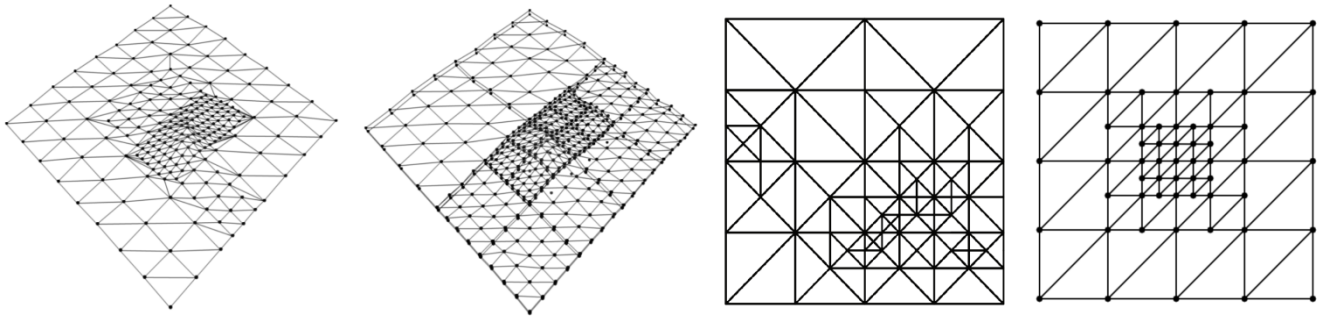
При использовании дискретного представления ландшафта с различными уровнями детализации возникает задача построения C^0-непрерывной триангуляции: на границах смежных блоков с различными LoD появляются разрывы (cracks) и T-стыки (T-junctions) полигональной сетки. Для построения C^0-непрерывной полигональной сетки необходимо, чтобы совпадали граничные вершины смежных блоков. Для предотвращения на границах между блоками резких переходов при интерполяции вершинных атрибутов (например, нормалей, текстурных координат, цвета) также требуется построение C^1-непрерывных сеток. В таких сетках нормаль каждой вершины вычислена с учётом всех полигонов, включающих данную вершину.

Другой проблемой является непрерывность триангуляции «во времени»: при переключении уровней детализации наблюдается резкая смена геометрии и её освещения (LoD popping), что ухудшает визуальное восприятие сцены.

Методы для многомасштабной визуализации карт высот. Наиболее полный обзор методов рендеринга карт высот с различными уровнями детализации приведён в [8]. Мы ограничимся обзором наиболее важных и практичных методов, которые могут быть адаптированы для рендеринга объёмных ландшафтов. Во всех рассмотренных методах вывод геометрии и смена дискретных LoD происходит для целых блоков (чанков, патчей, тайлов, батчей, кластеров или сегментов) ландшафта, а не отдельных треугольников. Каждый блок ландшафта, граничащий с блоками, которые имеют отличный от него LoD (уровень детализации или разрешение), будем называть *переходным* (*transitional*).

В методе *Geometrical MipMapping* (*GMM*) [119] ландшафт разбивается на квадратные блоки одинакового размера. Каждый блок представлен регулярной сеткой и может иметь несколько

уровней детализации, которые выбираются в зависимости от экранной погрешности ρ (screen-space error) (в пикселях экрана): спроецированной на экран геометрической ошибки δ (geometric error) блока, равной максимальному вертикальному отклонению упрощённой геометрии блока от исходного рельефа. (Если ρ меньше размера пикселя, то изображения на экране упрощённой и исходной сетки будут одинаковыми, а смена уровней детализации — полностью незаметной.) Разрывы между переходными блоками устраняются путём прореживания сетки на границе блока с более высоким разрешением (рисунок 4.1, а). Достоинствами GMM являются простота реализации, низкая нагрузка на CPU, возможность быстрого обновления уровней детализации после модификации исходных данных блоков, поддержка устаревшего графического оборудования. Однако, использование блоков одинакового и фиксированного размера делает GMM непрактичным для рендеринга больших территорий.



а) б) в) г)

Рисунок 4.1 – Примеры треугольных сеток, построенных для рендеринга карт высот в методах а) GMM, б) Chunked LoD, в) на основе конформных треугольных сеток и г) Geometry Clipmaps.

В методе *Chunked Level of Detail* (*Chunked LoD*) [120] на этапе подготовки строится многомасштабное представление ландшафта в виде квадродерева. Каждый узел квадродерева хранит упрощённую TIN и соответствующее значение геометрической ошибки δ, а четыре его потомка представляют ландшафт с вдвое бóльшим уровнем детализации (т.е. с примерно вдвое меньшим δ). Корень квадродерева имеет грубую сетку, покрывающую весь ландшафт. Разрывы между соседними чанками скрываются под «бортиками» — вертикальными полосами, высота которых должна превышать δ (рисунок 4.1, б). Достоинства Chunked LoD: использование минимальной триангуляции, адаптированной к особенностям поверхности, возможность визуализации ландшафтов гигантских размеров и быстрого, иерархического отсечения частей ландшафта по пирамиде видимости. Другими достоинствами являются простота реализации внеядерного рендеринга и возможность работы на устаревшем оборудовании. Недостатки Chunked LoD: длительный и трудоёмкий процесс подготовки, большой объём занимаемого во внешней

памяти пространства (из-за необходимости хранения полигональных сеток), необходимость перестройки всей LoD-иерархии при модификациях карты высот и возможность появления графических артефактов из-за использования «бортиков» (резкие переходы освещения, «растяжение» при текстурировании, артефакты при рендеринге карт теней и прозрачных частей ландшафта, таких как водные поверхности, и т.д.).

В методе *Batched Dynamic Adaptive Meshes* (*BDAM*) [121−123] ландшафт на стадии подготовки разбивается на *конформные* (conforming) треугольные сетки[10], образованные бинарным деревом прямоугольных равнобедренных треугольников (Hierarchy of Right Triangles, HRT). В HRT каждый треугольник может конформно соединяться через гипотенузу с треугольником, принадлежащим более грубому уровню детализации, а через катеты — с треугольниками, принадлежащими более детальному уровню (рисунок 4.1, в). В отличие от HRT/ROAM, в BDAM вместо отдельных треугольников используются оптимизированные для рендеринга нерегулярные треугольные сетки (в виде TIN). Достоинства BDAM: бесшовная адаптивная триангуляция, быстрый рендеринг, поддержка старого оборудования. Недостатки BDAM: длительный и трудоёмкий процесс подготовки, большой объём занимаемого на диске пространства из-за хранения полигональных сеток, скачкообразные переходы при смене уровней детализации (LoD popping).

В методе *геометрических карт отсечения* (*Geometry Clipmaps*) [124,125] ландшафт кэшируется и отрисовывается на GPU в виде вложенных регулярных сеток, расположенных вокруг камеры и передвигаемых вместе с ней (по мере движения камеры сетки пошагово обновляются) (рисунок 4.1, г). Данные рельефа инкрементально, с помощью тороидального доступа подгружаются из больших сжатых текстур, в мип-уровнях которых в различных разрешениях хранятся отфильтрованные карта высот и цвет ландшафта [126]. Координаты вершин для сеток различных разрешений вычисляются в вершинном шейдере на основе соответствующих им мип-уровней карты высот (графическое оборудование должно предоставлять возможность чтение текстуры в вершинном шейдере (Vertex Texture Fetch, VTF)). Разрывы и Т-стыки на границах между соседними сетками с отличающимися (в два раза) разрешениями закрываются полосками вырожденных треугольников. Преимущества метода Geometry Clipmaps: минимальная нагрузка на CPU, предсказуемая производительность (частота кадров, количество треугольников, потребление памяти и т.д.), быстрый препроцессинг, низкое потребление памяти, возможность визуализации карт высот практически неограниченного размера, отсутствие LoD popping. Главным недостатком

[10] В конформных сетках любые два элемента (например, два треугольника, тетраэдра, гексаэдра) либо не имеют общих точек, либо имеют ровно одну общую вершину, либо одно общее ребро, либо одну общую грань.

Geometry Clipmaps является избыточность триангуляции, которая построена на основе регулярных квадратных решёток и зависит только от позиции камеры. Это приводит к более высокому количеству треугольников и зачастую меньшей «разрешающей способности» по сравнению с методами, использующими адаптивную триангуляцию.

В методе *Continuous Distance-Dependent LoD* (*CDLOD*) [127] используется квадродерево, как в Chunked LoD, однако разрешения соседних узлов не могут отличаться более чем в два раза. Для отрисовки каждого блока используется единственная регулярная сетка (аналогично Geometry Clipmaps), разрешение которой можно изменять во время рендеринга (например, разрешение сетки можно снизить для рендеринга грубых теней или отражений ландшафта). Для бесшовного соединения смежных блоков с различными разрешениями используется *геоморфинг* (*geomorphing*) [128−132]: интерполяция атрибутов (например, позиций и нормалей) вершин, соответствующих соседним уровням детализации, в зависимости от расстояния до камеры и с соблюдением граничных условий. При этом в CDLOD не возникает разрывов, поскольку для вычисления позиций граничных вершин в различных разрешениях используются одни и те же горизонтальная сетка и карта высот. Геоморфинг также решает проблему плавной, незаметной смены уровней детализации, но налагает дополнительные ограничения: разрешения смежных блоков могут отличаться только в два раза, а графическое оборудование должно предоставлять возможность VTF, как в методе вложенных регулярных сеток. В остальном, метод CDLOD унаследовал основные достоинства и недостатки методов Chunked LoD и Geometry Clipmaps.

Среди недавних исследований отечественных авторов можно отметить работы [134,135], посвящённые созданию и эффективному сжатию многомасштабного представления карты высот (на основе квадродерева) и высокопроизводительному построению адаптивной триангуляции ландшафта.

Последние работы (например, [136,137]) фокусируются на максимальном переносе нагрузки с CPU на GPU и достижении непрерывной смены LoD. Данные методы позволяют отрисовывать весь ландшафт одним или несколькими батчами и зачастую обладают большей простотой реализации по сравнению с методами, основанными на поддержании иерархической структуры разбиения пространства. Однако их эффективное применение для рендеринга объёмных ландшафтов, которые, в отличие от карт высот, могут включать трёхмерные поверхности с любой топологией, является затруднительным. (Тем не менее, в демосцене существуют реализации, отрисовывающие воксельные ландшафты с помощью единственного индексного буфера и вырожденных треугольников.)

Методы многомасштабной визуализации объёмных ландшафтов. В отличие от 2.5D ландшафтов на основе карт высот, проблема интерактивной визуализации объёмных ландшафтов была исследована сравнительно мало.

В [138] описана технология *Vector field displacement* (*VFD*), используемая в видеоигре Halo Wars для отображения ландшафтов с нависающими ледяными глыбами. В VFD карта высот, помимо скалярных значений высоты, также содержит вектор горизонтального смещения для каждой точки карты высот.

Для генерации «бесшовной» треугольной сетки между соседними блоками воксельного ландшафта, разрешение (уровень детализации) которых отличается в два раза, был разработан модифицированный алгоритм марширующих кубиков — *Transvoxel Algorithm* (*TVA*) [10,13]. В TVA между блоками ландшафта с отличающимися уровнями детализации вставляются «переходные» ячейки (transition cells), принадлежащие блоку с более грубым уровнем детализации (см. рис. 4.2, а, б). Для триангуляции «переходных» ячеек используется расширенная таблица марширующих кубиков (MC) [28] с 512 возможными конфигурациями (для девяти вершин переходной ячейки), аналогично подходу в [140] (см. рис. 4.2, в).

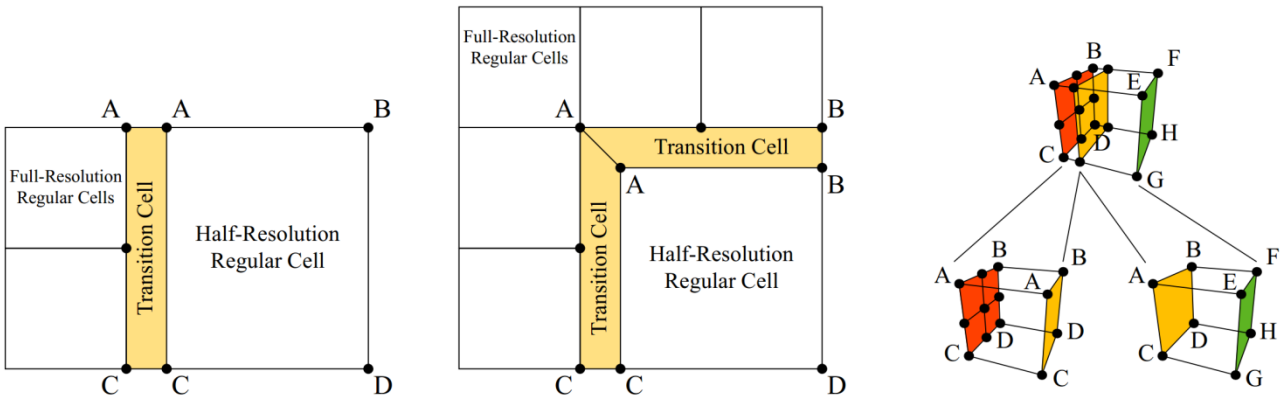


Рисунок 4.2 – Иллюстрация работы алгоритма Transvoxel [10,13] в 2D. Расположение переходных ячеек (выделены жёлтым цветом), когда блок низкого разрешения (half-resolution) граничит а) с одним блоком более высокого разрешения (full-resolution) и б) несколькими блоками. в) Топология переходной ячейки в 3D: левая часть триангулируется с помощью модифицированного MC с 512 конфигурациями, правая часть — стандартным MC [28].

Во избежание частых обновлений вершинных и индексных буферов выбор позиций вершин происходит на GPU в вершинном шейдере. Для визуализации больших ландшафтов блоки организованы в октодерево. Каждый блок, или узел октодерева, хранит «отфильтрованные»

(mipmapped) воксельные данные, представляющие более грубый уровень детализации его дочерних узлов. Другими словами, все блоки в октодереве имеют одинаковое разрешение, а уровень детализации каждого блока определяется его уровнем в октодереве.

В работах [14,15,11,140] предложен метод визуализации воксельных данных с различными уровнями детализации на основе *вложенных кубов отсечения* (*Nested Clip Boxes*), который является расширением метода вложенных регулярных сеток (Geometry Clipmaps), используемого для карт высот: камеру окружают несколько вложенных друг в друга кубов одинакового разрешения (состоящих из одинакового количества ячеек), представляющих различные LoD. Каждый следующий куб в два раза больше предыдущего (покрывает в восемь раз больший объём), а точка наблюдения находится во внутреннем, наименьшем кубе. Для обеспечения 2:1 разницы между соседними уровнями детализации вложенные кубы не должны касаться друг друга. Таким образом, только наиболее детальный LoD отрисовывается как целый куб, остальные LoD представлены в виде кубов с вырезанной сердцевиной. При движении камеры внешние кубы требуется обновлять в два раза реже, чем внутренние. По сравнению с методами, основанными на поддержании иерархической структуры разбиения пространства, данный метод позволяет визуализировать сцены практически неограниченного размера без какой-либо подготовки. В целом, метод вложенных кубов отсечения наследует достоинства и недостатки метода вложенных регулярных сеток.

В работе [16] предложен гибрид карты высот и воксельного ландшафта: ландшафт представлен набором вертикальных колонок, содержащих индексы материалов. [18] является расширением [16], добавляющим разбиение ландшафта (с помощью октодерева), генерацию уровней детализации и бесшовную триангуляцию «гладкого» ландшафта методом DC на GPU. В [19] для качественной полигонизации воксельного ландшафта предложен метод для интерактивной генерации четырёхугольных сеток методом DMC на GPU, однако не рассмотрены проблемы рендеринга больших ландшафтов с различными уровнями детализации и бесшовной триангуляцией. В [20,21] ландшафт разбивается на конформные тетраэдральные сетки (трёхмерный аналог BDAM [121−123]), в которых каждый тетраэдр разбивается на четыре скошенные кубические решётки, которые затем триангулируются с помощью MC. В [22] предлагается 2.5D гибрид между картами высот и вокселями: октодерево, грани ячеек которого являются картами высот. Для рендеринга бесшовной треугольной сетки с различными LoD используются патчи Безье и геоморфинг.

В последние годы начинают применяться гибридные подходы, сочетающие растеризацию и «бросание» лучей (ray casting) или «бросание снежков» (splatting) для различных частей ландшафта [141,142,143,11,89]. Перечень основных работ по интерактивной визуализации воксельных ландшафтов путём растеризации треугольных сеток приведён в таблице 4.1.

Таблица 4.1 – Основые публикации по интерактивной визуализации воксельных ландшафтов.

Название публикации	**Краткое описание работы**
Voxel-Based Terrain for Real-Time Virtual Simulations [2010]	(Modified) Marching Cubes (Transvoxel), «бесшовная» триангуляция «гладких» воксельных ландшафтов
Visualization of Large Isosurfaces Based on Nested Clipboxes [2011]	Визуализация «гладких» воксельных ландшафтов с различными уровнями детализации
Real-time Rendering of Stack-based Terrains [2011]	Dual Contouring на GPU, бесшовная триангуляция больших «гладких» воксельных ландшафтов
Generating Smooth High-Quality Isosurfaces for Interactive Modeling and Visualization of Complex Terrains [2012]	Dual Marching Cubes на GPU, возможность восстановления острых рёбер и углов ландшафта, но не рассмотрены проблемы визуализации больших ландшафтов с различными уровнями детализации
Level of Detail for Real-Time Volumetric Terrain Rendering [2013]	Иерархия тетраэдров (Longest Edge Bisection, LEB), бесшовная стыковка блоков «гладкого» ландшафта с различными уровнями детализации (LEB образуют конформную триангуляцию), Marching Cubes на GPU
Real-Time Isosurface Extraction With View-Dependent Level of Detail and Applications [2014]	Развитие предыдущей работы: Marching Cubes на GPU со сглаживанием сетки (Laplasian smoothing) с целью улучшения её качества (формы треугольников)

На время написания работы автору неизвестно о каких-либо публикациях непосредственно по триангуляции воксельных ландшафтов с восстановлением острых рёбер и углов поверхности, что могло бы применяться для отображения зданий и крупных искусственно созданных объектов на ландшафте.

4.2 Предложенный подход к визуализации больших воксельных ландшафтов

4.2.1 Многомасштабное представление воксельного ландшафта

На самом нижнем и детальном уровне воксельный ландшафт представлен в виде трёхмерного массива одинаковых по размеру блоков кубической формы. Каждый блок может

хранить данные в любом из форматов, предложенных в третьей главе. Все форматы являются дискретным представлением сплошных трёхмерных объектов, поэтому «разрешающая способность» воксельного ландшафта ограничена разрешением его блоков. Для обхода этого ограничения блоки могут хранить данные в форме «непрерывного» граничного представления (например, B-Rep в виде треугольных или тетраэдральных сеток, которые при редактировании конвертируются в объёмное представление и затем обратно) или неявного, функционального представления (F-Rep). Например, F-Rep в виде функций расстояния со знаком (SDF) часто используются в демосцене для предоставления практически неограниченного разрешения и уровней детализации.

Выбор разрешения блока. Разрешение каждого блока (количество содержащихся в нём ячеек или вокселей) является компромиссом между скоростью обработки, объёмом занимаемой рабочей памяти и степенью сжатия для его долговременного хранения в базе данных. Например, более высокое разрешение блока позволяет передать более мелкие детали ландшафта и, как правило, увеличивает коэффициент сжатия, но приводит к существенному увеличению вычислительной сложности и гораздо более высокому объёму рабочей памяти. На практике чаще всего используются блоки с разрешениями до 64^3 ячеек. В C4 Engine [10,13] каждый чанк состоит из 16^3 ячеек (или 17^3 вокселей) и занимает несколько десятков килобайт в распакованном виде. В Upvoid Engine [24] чанки имеют разрешение 32^3 ячеек. В Voxel Farm [23] сцена разбита на блоки размером 40^3 ячейки, что ещё позволяет использовать 16-битные индексы ячеек/вершин при триангуляции каждого блока алгоритмом Uniform Dual Contouring [31]. Проведенные эксперименты показывают, что для обеспечения интерактивности при редактировании ландшафта оптимальное разрешение блока составляет от 32^3 до 64^3 ячейки, в зависимости от формата воксельных данных и алгоритма триангуляции.

Для возможности визуализации больших сцен воксельный ландшафт организован в виде октодерева — иерархии регулярных кубических решёток, в которых размер каждого блока равен $L_0 \cdot 2^{LoD}$, где L_0 — наименьший возможный размер блока (на наиболее нижнем и детальном уровне), LoD — индекс уровня детализации блока (нуль соответствует самому детальному LoD). Для адресации каждого блока используются 64-битные идентификаторы *ChunkID*, содержащие в четырёх младших битах LoD-индекс и в остальных битах — три 20-битные координаты блока в кубической решётке с соответствующим LoD. Если размер каждой ячейки (или вокселя) в мировых координатах равен 0.5 м, а каждый блок состоит из 32^3 ячеек и имеет размер $L_0 = 16$ м, то данная

схема позволяет адресовать ландшафты размером $16 \cdot 2^{20} \approx 1.7 \cdot 10^7$ м, что сопоставимо с диаметром Земли ($\sim 12.7 \cdot 10^6$ м).

Хранение и загрузка воксельных данных. В силу устройства иерархии памяти современных компьютеров доставка несжатых данных из внешнего хранилища (дискового или сетевого пространства) в оперативную память обычно происходит гораздо медленнее, чем чтение и распаковка сжатых данных. Поэтому воксельные данные целесообразно хранить во внешней памяти в сжатом виде. На практике степень сжатия воксельных данных достигает нескольких десятков раз.

Изначально каждый блок воксельного ландшафта хранился на диске в виде отдельного файла, что приводило к значительным накладным расходам на операции с мелкими файлами (каждый блок в сжатом виде занимает несколько десятков или сотен килобайт) и субоптимальному использованию дискового пространства из-за фрагментации. Поэтому было принято решение использовать «монолитную» базу данных, оптимизированную для работы с бинарными блобами. В качестве такой базы данных была выбрана LMDB[11], поскольку она обладает наиболее высокой, среди альтернатив, скоростью чтения за счёт отображения файлов в адресное пространство процесса, не требует ручной синхронизации и осуществляет подкачку данных используя нативные, системные механизмы кэширования. Кроме того, LMDB не выполняет сжатия при записи, что может слегка увеличить размер уже сжатых воксельных данных.

В настоящей реализации для хранения ландшафта в различных разрешениях используются четыре базы данных: *VoxelsDB* — для хранения воксельных данных блоков на самом детальном LoD (*редактируемые* данные в любом из форматов, описанных в третьей главе), *MeshesDB* содержит аппроксимацию исходных воксельных данных на различных уровнях детализации в виде треугольных сеток (треугольные сетки для рендеринга имеют только не полностью пустые и не полностью сплошные блоки), *SeamsDB* хранит линейные октодеревья в формате SLOG для бесшовной триангуляции и генерации уровней детализации, в *IndexDB* хранятся метаданные для каждого блока ландшафта: флаги (редактируемость, «однородность» блока, наличие SLOG и треугольной сетки), индекс «сплошного» материала (если блок однородный) и время последнего изменения. Теоретически, *MeshesDB*, *SeamsDB* и *IndexDB* могут быть восстановлены на основе данных в *VoxelsDB*. Для оптимального расположения файлов во внешней памяти (для ускорения операций чтения) целесообразно упорядочить записи вдоль кривой Гильберта или Мортона, что

[11] https://symas.com/products/lightning-memory-mapped-database/

увеличит степень локальности доступа при подкачке данных, однако LMDB не предоставляет подобной возможности.

4.2.2 Визуализация ландшафта с различными уровнями детализации

В ходе исследования были опробованы следующие подходы к визуализации воксельных ландшафтов с различными уровнями детализации:

1) разбиение ландшафта на блоки одинакового размера, каждый из которых может иметь несколько уровней детализации (как в методе GMM [119]);

2) разбиение ландшафта на блоки различных размеров с помощью иерархической структуры данных (по аналогии с методами для карт высот, использующими квадродеревья: Chunked LoD [120], CDLOD [127] и др.);

3) визуализация ландшафта с помощью вложенных регулярных сеток (как в методе Geometry Clipmaps [124] и его адаптации для трёхмерного случая — Nested Clip Boxes [14, 15, 11, с. 45–71]).

Первый подход непригоден для визуализации больших территорий, потому что общее количество блоков, имеющих одинаковые размеры и разрешения, растёт в кубической степени при увеличении линейных размеров отображаемой сцены.

В настоящее время для визуализации «больших» воксельных ландшафтов наиболее распространён второй подход, заключающийся в организации сцены с помощью древовидной, иерархической структуры данных, в которой каждый каждый листовой узел соответствует блоку ландшафта. Наиболее распространены такие структуры данных, как октодерево [10,13,22,24,145], квадродерево из вертикальных воксельных «колонок» [18] и иерархии тетраэдров [20,21].

На основе проведенного анализа в качестве LoD-схем для визуализации «больших» ландшафтов были выбраны подходы с использованием технологии вложенных регулярных сеток и организации ландшафта с помощью октодерева.

Главными достоинствами метода вложенных регулярных сеток являются лёгкость реализации (из-за простой и регулярной структуры) и возможность визуализации гигантских сцен с минимальными накладными расходами на поддержание LoD-иерархии, которые возникают при использовании октодеревьев с большим количеством уровней вложенности. Главный недостаток — «жёсткая» и фиксированная структура, затрудняющая реализацию прогрессивного рендеринга, когда изначально грубые части ландшафта постепенно детализируются по мере подкачки или

генерации новых данных. В данной работе был реализован метод вложенных кубов отсечения [14,15,11]. В отличие от оригинальных работ, каждый куб, представляющий часть сцены с определённым уровнем детализации, содержит трёхмерный массив блоков ландшафта, для триангуляции которых используется алгоритм дуальных контуров (рисунок 4.3).

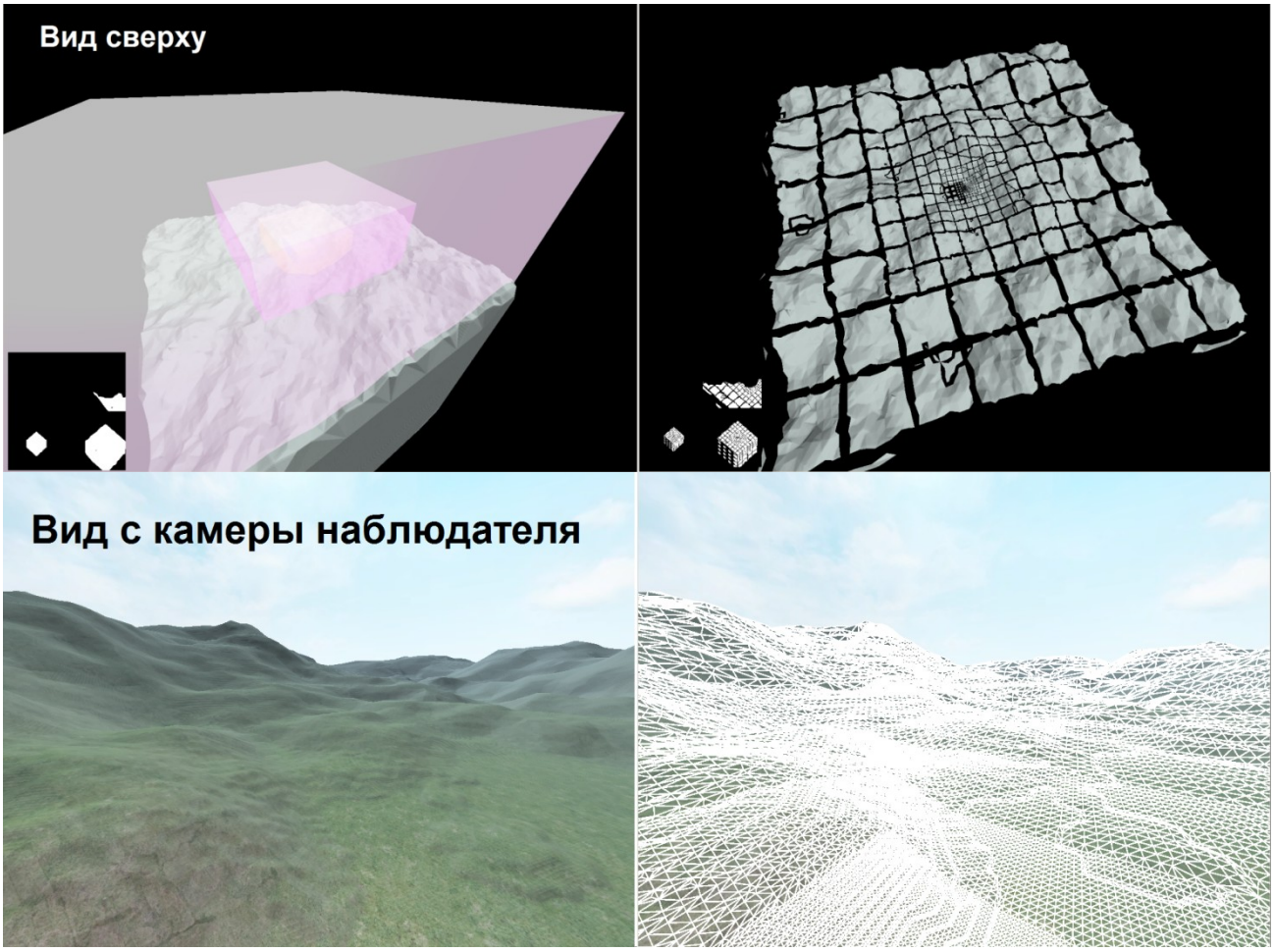


Рисунок 4.3 – Визуализация ландшафта с различными уровнями детализации (камера находится в его центре). *Сверху*: визуализация ландшафта с высоты птичьего полёта (справа показаны его отдельные блоки). *Снизу*: изображение с точки зрения наблюдателя (справа показана сетка). Детализация ландшафта уменьшается с увеличением расстояния до камеры. Размеры блоков увеличиваются в геометрической прогрессии в зависимости от расстояния до камеры, а размеры спроецированных на экран треугольников остаются примерно одинаковыми.

В отличие от вложенных кубов отсечения, октодерево является довольно гибкой структурой разбиения пространства и может быть использовано с любыми LoD-метриками. Как и в большинстве систем визуализации ландшафтов, в данной работе применяются *сбалансированные*

(*balanced*) или *ограниченные* (*restricted*) октодеревья, в которых размеры двух смежных блоков не могут отличаться более чем в два раза, а к переходной грани блока может прилегать максимум до четырёх смежных блоков, находящихся на более глубоком уровне измельчения[12]. Данное ограничение позволяет разрабатывать простые и эффективные алгоритмы бесшовной адаптивной триангуляции. Условие сбалансированности октодерева должно гарантироваться LoD-метрикой (octree refinement metric), которая определяет желаемый уровень детализации каждого узла октодерева.

В данной работе используется стандартная LoD-метрика [127,147,150], которая основана на ближайшем расстоянии между узлом и камерой и не учитывает экранной погрешности аппроксимации: узел необходимо измельчить (разбить на восемь дочерних узлов), если выполнено условие

$$d < k \cdot L,$$

где d — расстояние между позицией $\mathbf{c}$ камеры и ближайшей к ней точкой $\mathbf{p}$ на охватывающей оболочке (ограничивающем кубе или сфере) узла, L— размер узла (длина ребра ограничивающего куба или диаметр описанной сферы), а k должен быть строго больше единицы для получения сбалансированного октодерева.

В качестве меры близости были исследованы следующие нормы:

1) октаэдрическая (L_1-норма): $d = \|\mathbf{c} - \mathbf{p}\|_1$, $\|\mathbf{x}\|_1 = \sum_{i=1}^{3} |\mathbf{x}_i|$.

2) эвклидова (L_2-норма): $d = \|\mathbf{c} - \mathbf{p}\|_2$, $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{3} |\mathbf{x}_i^2|}$.

3) кубическая (L_∞-норма): $d = \|\mathbf{c} - \mathbf{p}\|_\infty$, $\|\mathbf{x}\|_\infty = \max_{1 \le i \le 3} |\mathbf{x}_i|$.

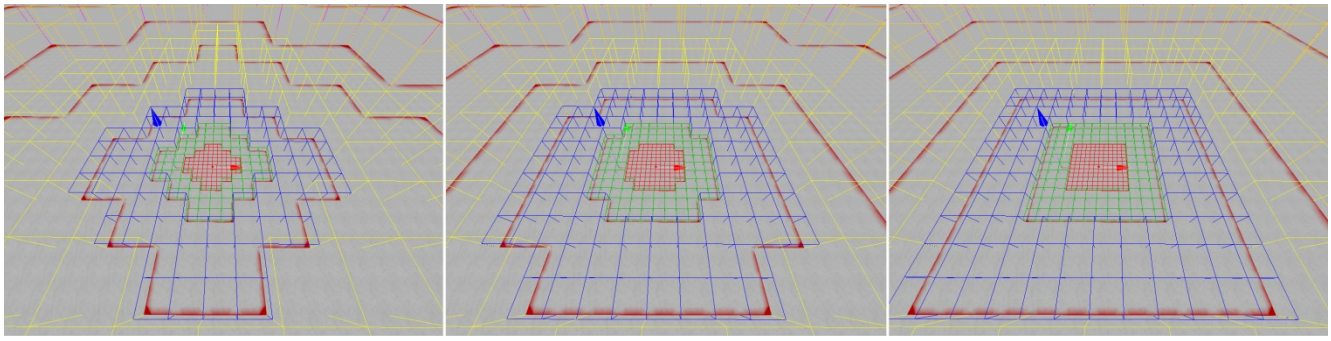
Октаэдрическая норма является наиболее дешёвой с вычислительной точки зрения, но приводит к неоптимальному выбору уровней детализации узлов и избыточному расщеплению/слиянию узлов октодерева при движении камеры.

Метрика на основе эвклидовой нормы обеспечивает наиболее близкий к оптимальному выбор уровней детализации (степень измельчения узла зависит от его эвклидова расстояния до камеры), но приводит к избыточному перестроению узлов октодерева при движении камеры и является наиболее дорогостоящей с вычислительной точки зрения (из-за операции квадратного корня).

Метрика на основе кубической нормы, или метрика Чебышёва, занимает промежуточное значение по качеству выбора уровней детализации (поскольку она приводит к избыточному

[12] В общем случае, если в d-мерном октодереве размеры смежных блоков могут отличаться в k раз, то к переходной грани блока может прилегать $2^{k(d-1)}$ блоков, находящихся на более глубоком уровне разбиения.

измельчению октодерева в углах кубической области с одинаковым LoD) и вычислительной стоимости. Основным преимуществом данной метрики является минимальное количество узлов октодерева, которое требуется перестраивать при движении камеры. В отличие от эвклидовой метрики, для вычисления расстояний может использоваться целочисленная арифметика на CPU, которая не имеет проблемы потери точности на больших сценах. Данная метрика также реализуется эффективней на GPU (через min/max, без операции квадратного корня) для вычисления коэффициента интерполяции для геоморфинга в вершинном шейдере [147] Таким образом, для экономии вычислений и по геометрическому содержанию задачи в качестве критерия для измельчения (адаптации) октодерева предпочтительнее чебышева метрика $\| \cdot \|_\infty$. Октодерево на основе данной метрики эквивалентно набору вложенных кубов (как в методе геометрических карт отсечения для карт высот) (рисунок 4.4, в).



а) б) в)

Рисунок 4.4 – Разбиение плоскости с помощью октодерева и использованием для выбора уровней детализации а) октаэдрической (L^1), б) Эвклидовой (L^2) и в) кубической (L^{max}) метрик. Красным цветом обозначены области перехода от детального к более грубому уровню детализации, цветными линиями — границы блоков с одинаковым размером и уровнем детализации. Камера находится в центре сцены. При движении камеры симметрия может слегка нарушаться.

Данная LoD-метрика используется в большинстве систем для визуализации воксельных ландшафтов, в том числе в Voxel Farm [23], Upvoid Engine [24], Ultimate Terrains (uTerrains) [25], TerrainEngine [26], TerraVol [27].

Нерекурсивный алгоритм адаптации октодерева для выбора уровней детализации. В традиционном подходе, при использовании октодерева в качестве LoD-иерархии, выбор уровня детализации каждого блока ландшафта происходит рекурсивно, путём обхода октодерева сверху вниз, начиная с его корня. Это позволяет одновременно с выбором уровней детализации иерархически отсекать невидимые части ландшафта по пирамиде видимости (hierarchical view

frustum culling) — если при обходе октодерева оболочка очередного узла не пересекает пирамиду видимости, то данный узел и всё, что внутри, может быть отброшено.

Однако, данный способ плохо масштабируется с увеличением размеров сцены и ростом количества возможных уровней детализации, которые могут быть видимыми одновременно. Например, если размер блока ландшафта на самом детальном LoD в мировых координатах составляет 16 м, то для отображения ландшафта размером с планету Земля потребуется октодерево с глубиной не менее 16. Если виртуальный наблюдатель стоит на поверхности планеты, то алгоритм выбора LoD будет вынужден в каждом кадре обходить все уровни октодерева, спускаясь от корня до листовых узлов. При таком количестве уровней вложенности становятся ощутимыми накладные расходы на рекурсивный спуск и обход всей LoD-иерархии, проверки видимости каждого узла, вычисление его желаемого уровня детализации и принятие решения о разбиении или слиянии листовых узлов [148]. Поэтому даже в приложениях для визуализации больших карт высот редко можно встретить более чем 12 уровней детализации.

Кроме того, для бесшовной триангуляции ландшафта требуется выполнять операцию поиска ближайших соседей (какие блоки прилегают к данному блоку (или листовому узлу)?), которая в традиционных октодеревьях на указателях реализуется громоздко и неоптимально.

В силу вышеуказанных причин Voxel Farm [23], а также видеоигры Space Engineers и Planet Nomads [149] с воксельными ландшафтами планетарного масштаба, в качестве LoD-иерархии используют октодеревья на основе хэш-таблиц и нерекурсивные алгоритмы выбора LoD. В данной работе был разработан нерекурсивный алгоритм выбора LoD, который является более простым в реализации, чем аналогичные алгоритмы, и требует меньше рабочей памяти.

В качестве LoD-иерархии алгоритм использует линейное октодерево, в котором листовые узлы (представляющие блоки ландшафта) и их адреса хранятся в отдельных массивах, отсортированных по адресам. Адреса узлов — это мортон-коды (без бита глубины), которые составляются простым последовательным чередованием бит дискретных координат узла. Такая структура линейного октодерева также хорошо подходит для поиска ближайших соседей, что требуется при бесшовной триангуляции ландшафта.

Перед началом визуализации создаётся корневой узел октодерева, другими словами, в самый последний массив уровней детализации добавляется адрес и указатель на новый узел.

В процессе визуализации в каждом кадре обновляется только один уровень детализации.

В главном цикле алгоритма адаптации LoD-иерархии рассматривается каждый листовой узел октодерева на текущем уровне детализации L_i (который подлежит обновлению). Для каждого

листового узла *C* с адресом *Mcurrent* на L_i проверяется, требуется ли разбить данный узел на 8 дочерних, чтобы увеличить уровень детализации до L_{i-1} (нуль — индекс самого детального уровня детализации).

Одновременно для узла *C* строится адрес *Mparent* соответствующего родительского узла (этот узел ещё не существует) и проверяется, может ли узел с адресом *Mparent* существовать на L_{i+1}, не нарушая LoD-критерия. Если это условие выполняется, то необходимо выполнить слияние дочерних узлов (в число которых входит и узел *C*) на L_i и создать новый узел с адресом *Mparent* на следующем (более грубом) уровне детализации L_{i+1}.

На рисунке 4.5 приведёна принципиальная блок-схема предложенного восходящего алгоритма выбора уровней детализации для каждого узла линейного октодерева:

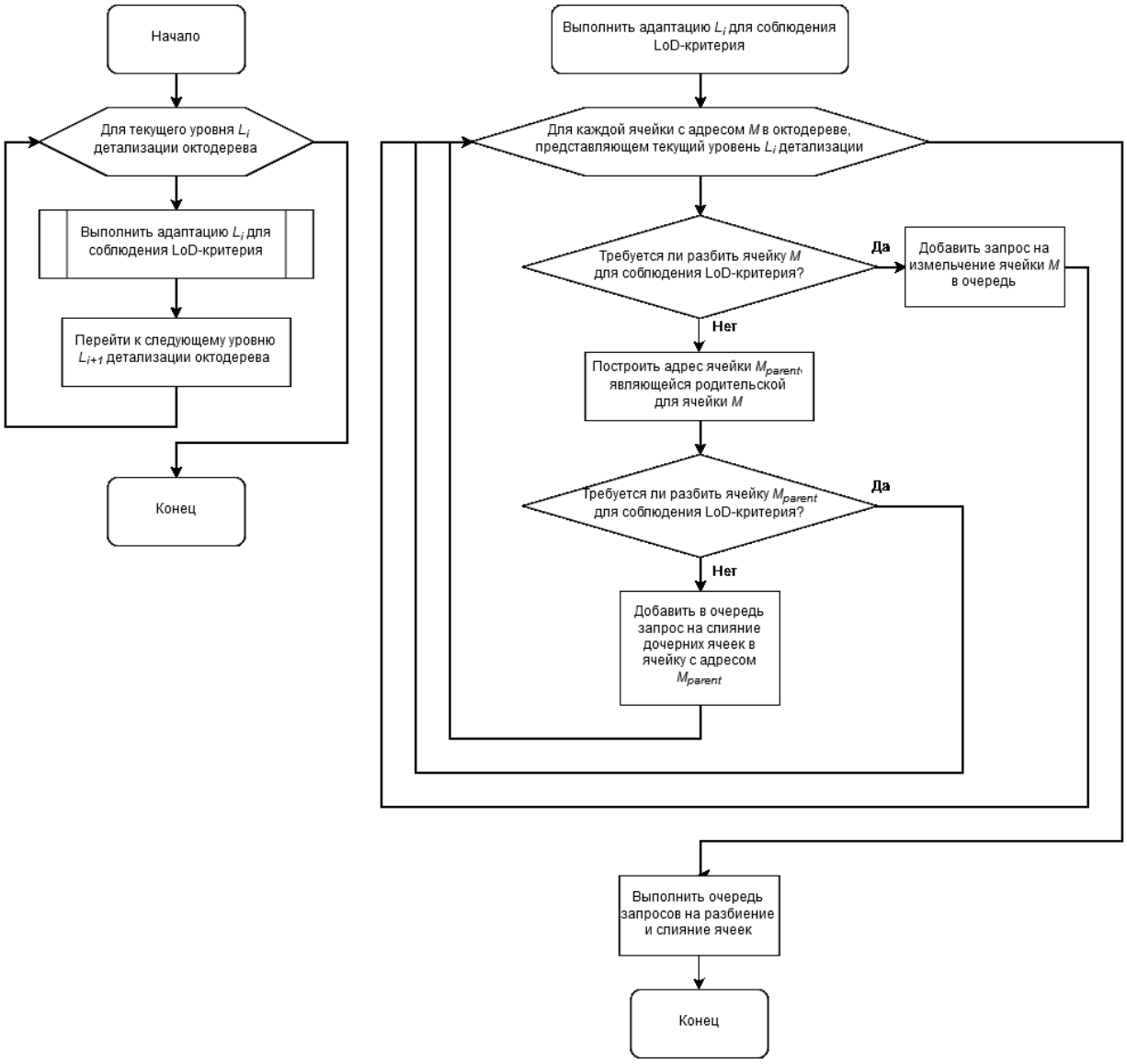


Рисунок 4.5 – Блок-схема предложенного алгоритма выбора уровней детализации октодерева.

Таким образом, в отличие от традиционного рекурсивного спуска по октодереву, в предложенном алгоритме для адаптации LoD-иерархии в каждом кадре происходит обновление только одного уровня детализации. Поэтому вычислительная сложность нового алгоритма не зависит от глубины октодерева, а пропорциональна среднему количеству узлов на одном уровне детализации.

В процессе визуализации в каждом кадре обновляется только один LoD. Для корректной перестройки LoD после редактирования ландшафта необходимо первыми обрабатывать более детальные LoD.

В Приложении В описан псевдокод алгоритма адаптации выбора уровней детализации.

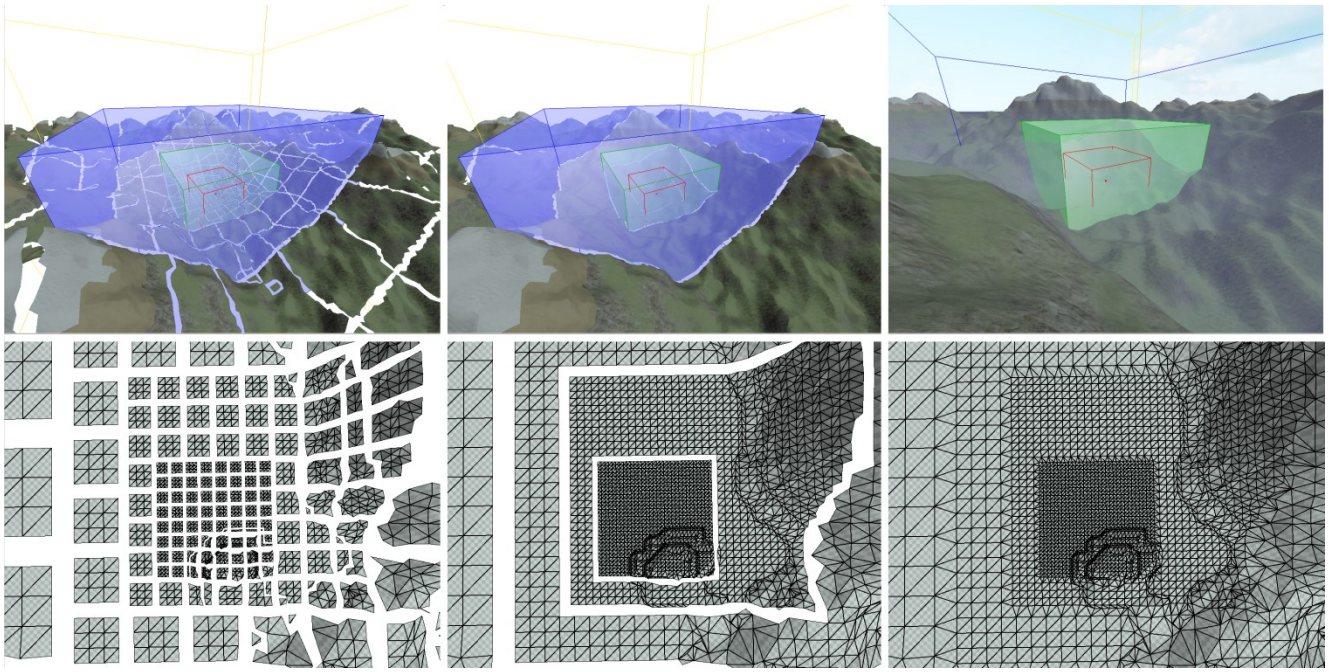
4.3 Бесшовная триангуляция воксельного ландшафта

Поскольку ландшафт обычно не умещается в оперативную память целиком, для его триангуляции используются алгоритмы с внешней памятью (external memory algorithms) или так называемые «внеядерные» (out-of-core) алгоритмы, которые позволяют обрабатывать сверхбольшие поверхности по частям и работать в оперативной памяти только с одной треугольной сеткой для небольшого участка ландшафта.

В двойственных методах триангуляции для создания каждого полигона требуется соединить вершины нескольких смежных ячеек (inter-cell dependency) (см. Приложение А), поэтому результатом применения двойственного алгоритма триангуляции к каждому блоку воксельного ландшафта по отдельности, без включения ячеек из соседних блоков, является треугольная сетка, содержащая разрывы на границе данного блока со смежными к нему блоками (см. рис. 4.6, а, б).

Данная проблема отсутствует при использовании прямых (MC [28], MT [29], [139], [10]) и гибридных (EMC [40], [152], CMS [33]) методов триангуляции, в которых вершины полигональной сетки создаются на рёбрах и гранях ячеек, а границы смежных блоков представляют собой конформные триангуляции.

В обоих случаях остаётся классическая проблема бесшовной триангуляции воксельного ландшафта, состоящего из блоков с различными разрешениями и размерами (задача сшивки различных уровней детализации). На границах между переходными блоками возникают разрывы (cracks, gaps), перекрытия и Т-стыки. В настоящее время применяются следующие подходы к решению данной проблемы:



а) б) в)

Рисунок 4.6 – Визуализация ландшафта с помощью вложенных регулярных сеток: а) триангуляция каждого блока по отдельности методом дуальных контуров приводит к появлению разрывов; б) разрывы между блоками одинакового размера устранены, но остались пробелы между блоками с отличающимися (в два раза) размерами и разрешениями; в) после применения предложенных в данном разделе способов все разрывы сетки полностью устранены. Для большей наглядности разрешение каждого блока ландшафта выбрано 4^3 ячейки.

1) «честное» построение бесшовной триангуляции для блоков с различными LoD с использованием для заполнения разрывов того же алгоритма, который применяется и для триангуляции отдельных блоков ландшафта (например, DC [145,23]), или его модификацию, если используемый алгоритм триангуляции не поддерживает адаптивность (например, модификация MC [28] для LoD [10,13]);

2) использование прогрессивных сеток с геоморфингом (Progressive Meshes, PM), в которых каждая вершина «знает» свои позиции на текущем и следующем уровнях детализации блока, и позиции граничных вершин блока устанавливаются в соответствии с LoD соседних блоков (очевидно, при этом уровни детализации смежных блоков не могут отличаться более чем в два раза) [130,132,137,150];

3) «сшивка» границ блоков: каждая граничная вершина переходного блока стягивается к ближайшей вершине блока с более грубым LoD или передвигается в середину ближайшего ребра грубого блока, образуя T-стыки [14,15,11];

4) расширение области триангуляции переходных блоков на несколько ячеек для генерации полигональных сеток внахлёст, чтобы спрятать разрывы между переходными блоками (этот способ применяется в uTerrains [25] и до 2012 года использовался в Voxel Farm [23], согласно комментариям в блоге автора);

5) создание на границах переходных блоков полос из треугольников, перпендикулярных нормалям к поверхности, которые должны спрятать разрывы полигональной сетки между смежными блоками [144] (по аналогии с «юбками» или «бортиками», использующимися в методе Chunked LoD [120]);

6) заполнение разрывов в пространстве экрана (image-space, screen-space): отрисовка переходных блоков с текущим и более грубым уровнями детализации и альфа-блендингом между ними [12,17] и/или заполнением «пустых» пикселей в процессе постпроцессинга [151].

Из вышеперечисленных подходов только первые два являются решениями, которые могут гарантировать создание бесшовных, водонепроницаемых сеток без наложений, пересечений и T-стыков, и поэтому подходят для качественной визуализации ландшафтов с наличием тонких стенок, острых рёбер и углов. Остальные способы приводят к появлению заметных графических артефактов и налагают ограничения на степень упрощения блоков. (На практике многие из этих дефектов незаметны при отрисовке грубых, «гладких» воксельных ландшафтов.) В следующих подразделах предложены методы, реализующие первые два подхода к бесшовной триангуляции смежных блоков с различными LoD: честное построение («stitching») на CPU бесшовной треугольной сетки с дискретными уровнями детализации (Discrete Levels of Detail, DLoD) и геоморфинг на GPU с непрерывным уровнем детализации (Continuous Level of Detail, CLoD).

4.3.1 Бесшовное соединение смежных блоков одинакового размера и уровня детализации

Как было сказано ранее, из-за особенности inter-cell dependency (см. Приложение A), присущей двойственным методам триангуляции, при триангуляции каждого блока ландшафта по отдельности между его смежными блоками появляются разрывы, даже если все смежные блоки имеют одинаковые размеры и разрешения (или одинаковый LoD) (рисунок 4.7, a).

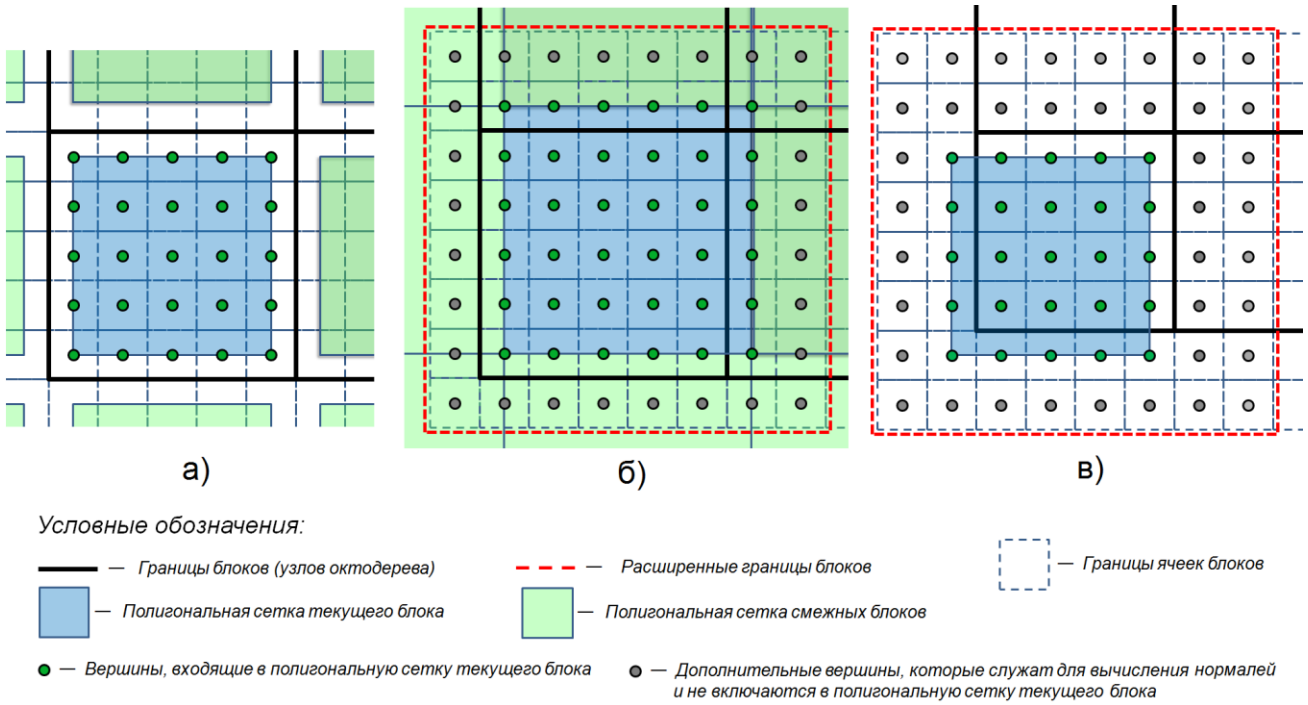


Рисунок 4.7 – а) Триангуляция каждого блока по отдельности приводит к появлению разрывов между смежными блоками (разрешение каждого блока: 5^3 ячеек). б) Генерация бесшовной сетки путём расширения каждого блока на две ячейки за максимальную границу и на одну ячейку за минимальную границу (разрешение каждого блока с учётом границ: 8^3 ячеек). в) Расширение области триангуляции каждого блока на две ячейки для симметрии. В случаях а) и б) триангуляция каждого блока может выполняться параллельно и независимо от его соседей.

Бесшовное соединение смежных блоков перекрытием. Для генерации бесшовной сетки достаточно расширить область триангуляции каждого блока так, чтобы области смежных блоков перекрывались на одну ячейку (рисунок 4.7, б).

Данный способ особенно часто применяется для триангуляции процедурно сгенерированных воксельных ландшафтов [18,150], в которых воксельные данные для построения сетки всегда могут быть всегда вычислены локально, без необходимости обращения к данным соседних блоков. Для закрытия разрывов и создания C^0-непрерывной сетки достаточно перекрытия блоков в одну ячейку [97], а для генерации C^1-непрерывной сетки необходимо перекрывать блоки на толщину в две ячейки. Во втором случае образованные «призрачными» ячейками полигоны служат только для расчёта гладких вершинных нормалей и должны быть исключены из финальной сетки. Для генерации иерархических уровней детализации удобней симметричная схема, показанная на рисунке 4.7, в. Исходя из документации и открытых заголовочных файлов из SDK

[66] можно предположить, что подобная схема используется в Voxel Farm [23], где каждый блок окаймляется дополнительным слоем в две ячейки (BLOCK_MARGIN = 2).

Для предотвращения повторного создания одинаковых полигонов [97,145] необходимо запрещать создание полигонов, все вершины которых образованы ячейками, лежащими в области триангуляции соседнего блока (рисунок 4.8), иначе в области перекрытия смежных блоков возникнет мерцание, связанное с z-fighting.

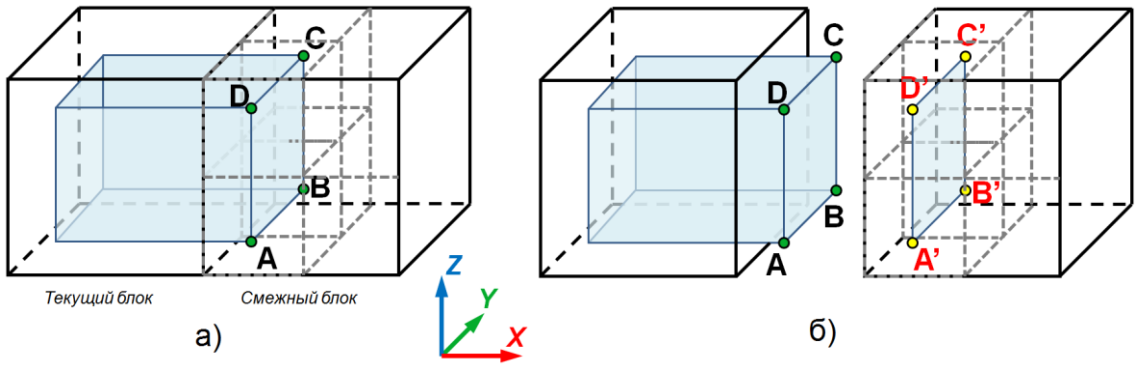


Рисунок 4.8 – Проблема повторного создания полигонов. а) Текущий блок перекрывает смежный к нему блок на одну ячейку для создания бесшовного соединения. б) Триангуляция смежного блока приводит к повторному созданию четырёхугольника *ABCD* для ребра вдоль оси *X*.

Достоинством данного способа является простота реализации, поскольку не требуется модифицировать исходный алгоритм триангуляции.

Ограничения данного подхода: смежные блоки должны иметь одинаковые размеры и разрешения, их границы не могут быть упрощены, а приграничные данные смежных блоков должны полностью совпадать после выполнения операций редактирования, чтобы гарантировать создание согласованной, конформной поверхностной сетки; ограничивающие параллелепипеды (AABB) блоков не подходят для отсечения по пирамиде видимости (view frustum culling), поскольку полигональная сетка каждого блока может выходить за его пределы.

Бесшовное соединение путём адаптивной триангуляции. Адаптивные двойственные методы триангуляции, такие как ADC, ADMC и CMS, позволяют создать бесшовную треугольную сетку для группы из нескольких смежных блоков ландшафта (не обязательно имеющих одинаковое разрешение или LoD). Для этого достаточно выполнить алгоритм триангуляции над октодеревом, построенным из ячеек, входящих во все блоки в данной группе.

По аналогии с предыдущим подходом, основанным на перекрытии блоков (см. рис. 4.7, б), для заполнения разрывов удобно использовать стандартную схему [145,13,134], в которой каждый

блок «отвечает» за бесшовное соединение со своими максимальными соседями (расположенными «выше» по координатным осям). На рис. 4.9 показана схема заполнения разрывов для отдельно взятого блока ландшафта в двухмерном случае, когда каждый блок имеет три максимальных соседа (разрешение каждого блока было выбрано 8^3 ячеек для большей наглядности).

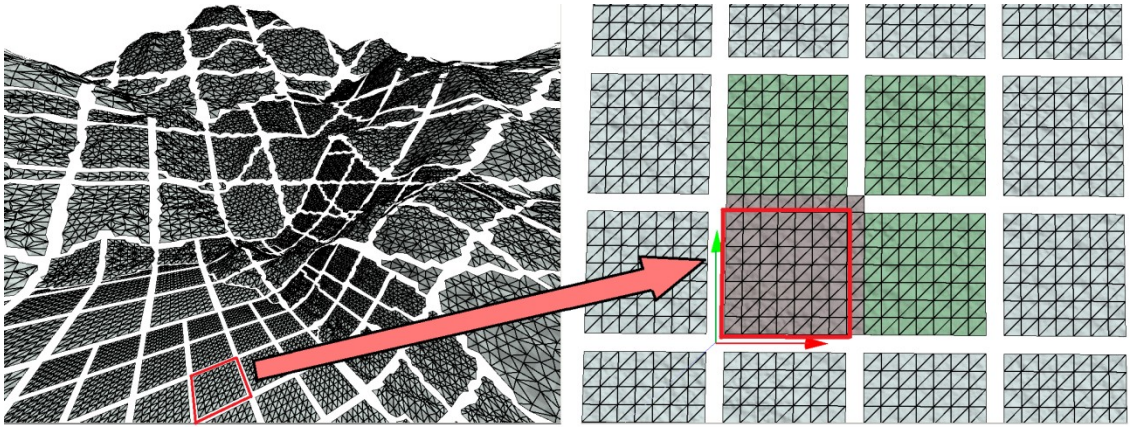
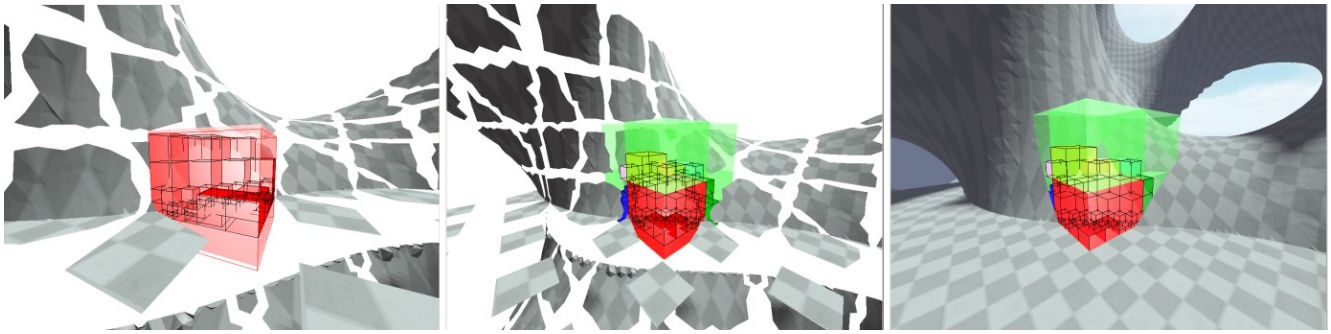


Рисунок 4.9 – Схема бесшовного соединения блока ландшафта (выделен красным) с его максимальными соседями (зелёные) в 2D. Шов принадлежит блоку в нулевом квадранте.

Для бесшовного соединения каждого блока с его максимальными соседями необходимо построить октодерево вдвое большего размера, в нулевом октанте которого расположены ячейки текущего блока, а в других — смежные к нему ячейки с соседних блоков (шов принадлежит блоку в нулевом октанте), а затем выполнить алгоритм адаптивной триангуляции над полученным октодеревом [146]. На рис. 4.10 показан процесс заполнения разрывов между смежными блоками в трёхмерном случае, когда каждый блок имеет до семи максимальных соседей.



а) б) в)

Рисунок 4.10 – Процесс бесшовной триангуляции в 3D: а) визуализация ячеек октодерева блока; б) добавление в октодерево ячеек с соседних блоков, прилегающих к граням данного блока; в) триангуляция полученного октодерева для бесшовного соединения блока с его соседями.

В данной работе для максимальной эффективности используются линейные октодеревья, хранящиеся в формате SLOG в готовом для триангуляции виде (см. раздел 3.6). Это позволяет избежать динамического выделения памяти под новые внутренние узлы октодерева и свести процесс построения октодерева для группы из смежных блоков к составлению и сортировке массива ячеек. Вместо манипуляций с указателями для вставки ячейки в заданный октант достаточно вставить трёхбитовый номер октанта в адрес ячейки сразу после бита глубины. Триангуляция линейного октодерева сводится к соединению вершин его ячеек.

Как и в способе с перекрытием блоков, для создания C^1-непрерывной сетки из соседних блоков необходимо собрать слой толщиной в две ячейки, а также запрещать создание полигонов, образованных ячейками, принадлежащими соседним блокам [145,49,58], иначе последующая триангуляция соседних блоков повторно создаст треугольники в области швов, что приведёт к появлению таких графических артефактов, как z-fighting (см. рисунок 4.8).

На рисунке 4.11 показан результат применения предложенного метода для бесшовной триангуляции воксельного ландшафта, составленного из блоков с разрешениями от 8^3 до 64^3 ячеек. На ландшафт (поверхность «синусоидальные волны», заданную уравнением $\sin(x) + \sin(y) = z$) была помещена модель акрополиса. Затем к ландшафту применялись операции «раскраски» материалами и булевы операции объединения и разности с моделями цилиндров, сфер и кубов.

Основным достоинством данного метода бесшовного соединения является то, что разрешения смежных блоков не обязательно должны быть одинаковыми, и могут отличаться на любое значение, кратное степени двойки. Блоки с высокими разрешениями могут хранить важные части сцены с наличием мелких деталей, а «грубый» ландшафт может быть представлен блоками низкого разрешения. Другим преимуществом данного подхода к бесшовной триангуляции является свобода в степени упрощения границ блоков, поскольку ячейки смежных блоков триангулируются как единое октодерево. Это обеспечивает построение минимальной адаптивной триангуляции (с созданием минимального количества полигонов для заполнения разрывов между блоками) с сохранением острых углов поверхности и без необходимости дублировать приграничные данные между соседними блоками. Кроме того, сжатые линейные октодеревья в формате SLOG занимают небольшой объём дискового пространства.

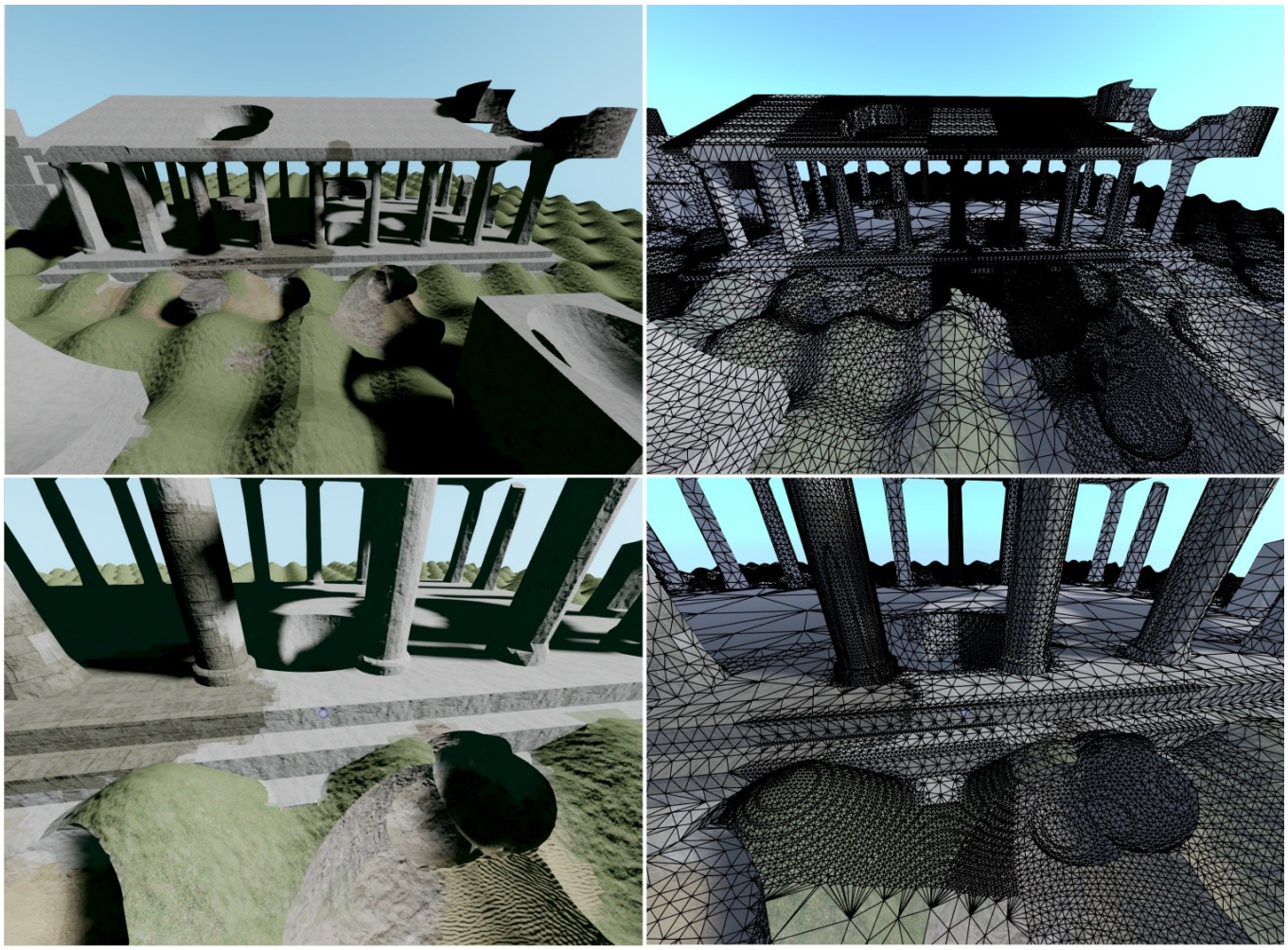


Рисунок 4.11 – Бесшовное соединение блоков с различными разрешениями с помощью адаптивной триангуляции. Каждый блок представлен линейным октодеревом в виде SLOG. Для построения бесшовной треугольной сетки октодеревья смежных блоков триангулируются как единое целое.

Главные недостатки данного подхода к бесшовной триангуляции блоков:

– Создание зависимостей между смежными блоками ландшафта, поскольку для бесшовной триангуляции каждого блока необходимо обращаться к данным его максимальных соседей. Это существенно снижает производительность и ограничивает возможности параллелизации: каждый раз при модификации блока необходимо перестроить треугольные сетки всех его минимальных соседей для сохранения бесшовного соединения.

– Для отсечения по пирамиде видимости ограничивающий параллелепипед (AABB) должен быть вычислен из построенной треугольной сетки блока, которая может выходить за пределы блока.

– В местах соединения блоков с отличающимися разрешениями возможно создание треугольников плохой формы и вершин с высокой валентностью.

Для полного устранения зависимостей при триангуляции блоков, помимо подхода с перекрытием блоков (см. рисунок 4.7, б), можно использовать идеи из CMS [33] и [152]: создавать дополнительные вершины на рёбрах и, если необходимо, острые вершины на гранях приграничных ячеек, отслеживая особенности поверхности на гранях с помощью нормалей. В обоих случаях для обеспечения согласованности границ необходимо дублировать приграничные данные между смежными блоками и запретить упрощение границ блоков, при этом все смежные блоки должны иметь одинаковое разрешение.

На основе вышеуказанных идей был разработан простой способ, полностью устраняющий зависимости между смежными блоками (inter-chunk dependency) при построении бесшовной триангуляции алгоритмом дуальных контуров (Dual Contouring, DC) [31].

Модифицированный алгоритм дуальных контуров. Предложенный подход к бесшовной триангуляции воксельного ландшафта основывается на двух идеях:

1) Дублирование данных на границах между смежными блоками позволяет устранить зависимости по данным между этими блоками при создании бесшовной триангуляции. В частности, дублирование нормалей к поверхности на границах блоков позволяет избежать необходимость создания при триангуляции блока расширенной треугольной сетки, треугольники которой используются только для расчёта «гладких» нормалей вершин сетки блока, как показано на рисунке 4.7, б, в.

2) Проблема inter-cell dependency в оригинальном алгоритме дуальных контуров, приводящая к зависимостям между смежными блоками ландшафта при построении бесшовной сетки, может быть решена путём создания и включения в триангуляцию дополнительных вершин, расположенных точно на гранях и рёбрах блока, как в алгоритме кубических марширующих квадратов (CMS) [33,65]. При этом для включения дополнительных граничных вершин в треугольную сетку не требуется модифицировать исходный алгоритм триангуляции — для граничных вершин создаются ячейки, которые примыкают к граням и рёбрам блока снаружи и также включаются в триангуляцию алгоритмом дуальных контуров.

Предложенный подход к бесшовной триангуляции предполагает наличие точек пересечения с поверхностью и нормалей к поверхности на активных рёбрах ячеек, которые должны дублироваться на границах между смежными блоками (для возможности создания C^1-непрерывной

полигональной сетки), как показано на рисунке 4.12. Для этой цели подходит такой формат хранения блоков, как воксельная решётка с Эрмитовыми данными (раздел 3.3).

Расширенный алгоритм дуальных контуров выполняется в два этапа, второй этап ничем не отличается от исходного алгоритма (см. подраздел 1.3.2):

1) Создание вершины для каждой активной ячейки.

2) Создание четырёхугольников для активных рёбер каждой ячейки.

На первом этапе, помимо вершин для внутренних активных ячеек блока, создаются дополнительные вершины на гранях и рёбрах блока (см. рисунок 4.13).

Для вершины $\mathcal{V}_\mathcal{E}$ полигональной сетки, созданной на ребре блока, позиции и нормали берутся из Эрмитовых данных (точек пересечения активных рёбер ячеек с изоповерхностью и единичных нормалей к поверхности в этих точках).

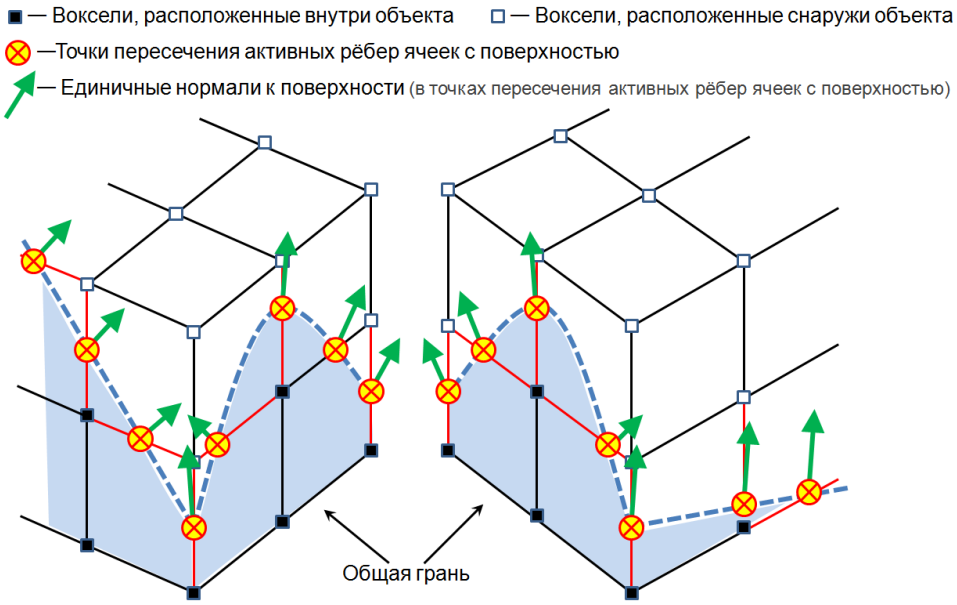
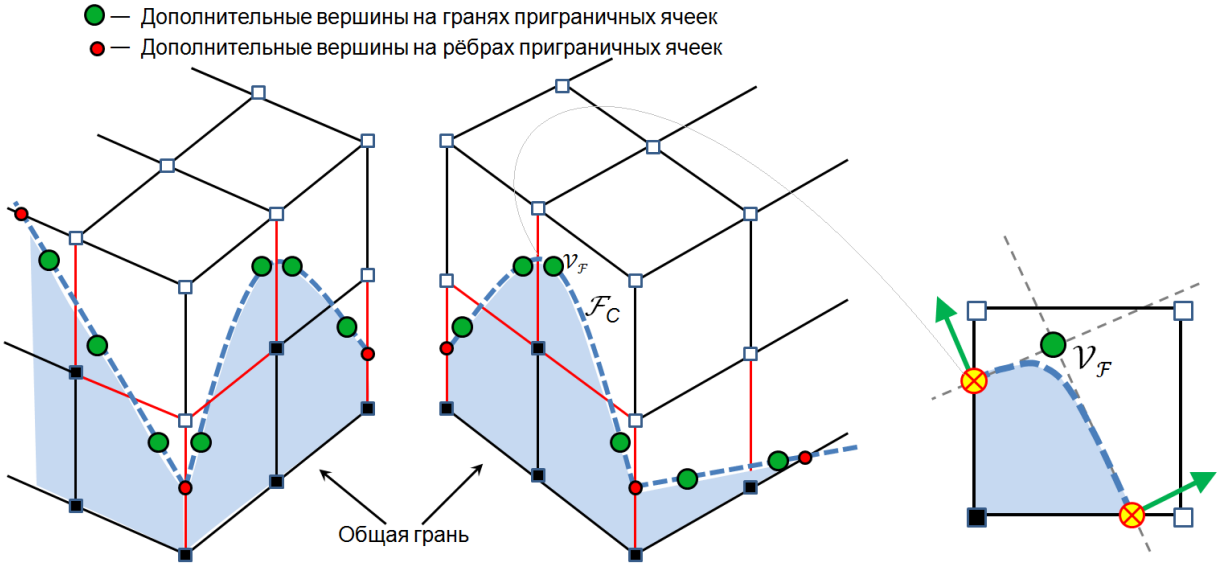


Рисунок 4.12 – Дублирование приграничных вокселей (материалов) и Эрмитовых данных на границе между двумя смежными блоками ландшафта. (Разрешение каждого блока — 2^3 ячейки.)

Для вершин сетки, созданных на гранях блока, позиции и нормали могут быть оценены локально. Каждая вершина $\mathcal{V}_\mathcal{F}$, созданная на грани блока, будет также лежать на грани $\mathcal{F}_C$ соответствующей приграничной ячейки блока (т.е. в квадратной ячейке $\mathcal{F}_C$) (рисунок 4.13, а).



а) б)

Рисунок 4.13 – а) Создание дополнительных вершин на гранях и рёбрах блока.
б) Использование нормалей для отслеживания особенностей изоповерхности на грани ячейки.

Для нахождения позиции и нормали новой граничной вершины $\mathcal{V}_{\mathcal{F}}$ используется информация на граничных рёбрах $\mathcal{F}_C$, как в алгоритме CMS [33,65] : поиск позиции острой вершины $\mathcal{V}_{\mathcal{F}}$ на грани $\mathcal{F}_C$ сводится к решению системы линейных уравнений (1.2) (рисунок 4.13, б), как и в случае с локализацией острых вершин внутри ячеек. Вершина $\mathcal{V}_{\mathcal{F}}$ может быть также помещена в центроид (усреднённую позицию) точек пересечения рёбер $\mathcal{F}_C$ с изоповерхностью, как в алгоритме поверхностных сеток (Surface Nets) [30]. Первый способ позволяет восстанавливать поверхности с острыми углами, второй способ подходит для реконструкции гладких поверхностей.

Созданные на гранях и рёбрах блока вершины $\mathcal{V}_{\mathcal{F}}$ и $\mathcal{V}_{\mathcal{E}}$, наряду с вершинами внутренних ячеек блока, также включаются в процесс триангуляции для создания бесшовного соединения с соседними блоками (см. рисунок 4.14).

Для дополнительных приграничных вершин создаются ячейки, которые «окаймляют» блок снаружи. Для каждой вершины $\mathcal{V}_{\mathcal{E}}$ полигональной сетки, расположенной на ребре $\mathcal{E}$ блока, создаётся наружная ячейка, которая касается ребра $\mathcal{E}$. Для каждой вершины $\mathcal{V}_{\mathcal{F}}$ полигональной сетки, расположенной на грани $\mathcal{F}$ блока, создаётся наружная ячейка, которая граничит с гранью $\mathcal{F}$ и гранью $\mathcal{F}_C$ соответствующей приграничной ячейки блока. Активные рёбра наружных ячеек также участвуют в создании четырёхугольников при триангуляции алгоритмом дуальных контуров. При этом не требуется создавать и включать в процесс триангуляции дополнительные угловые

вершины и наружные ячейки, которые бы касались углов блока, поскольку в методе дуальных контуров полигоны создаются только для рёбер ячеек.

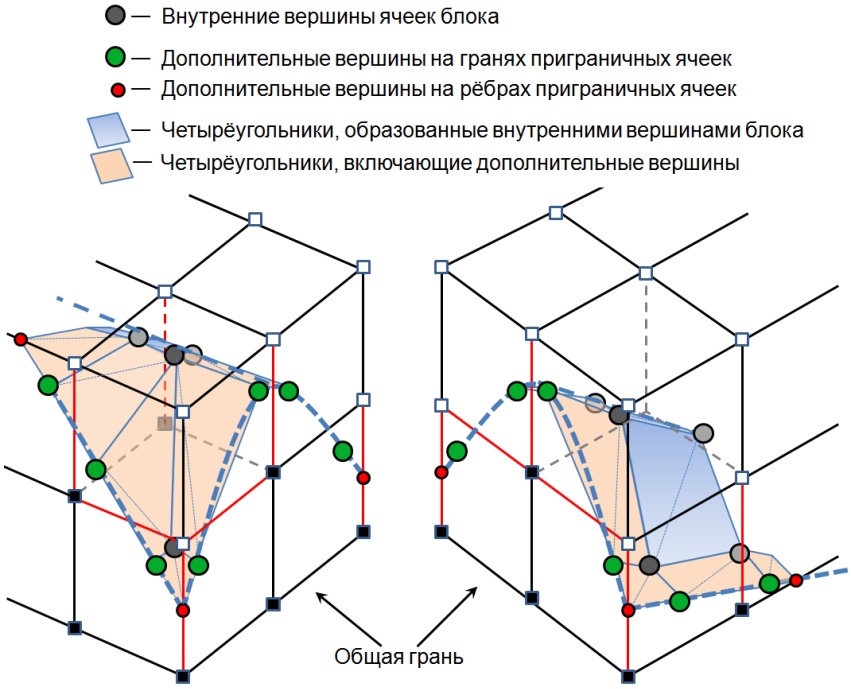


Рисунок 4.14 – Включение в триангуляцию дополнительных вершин на гранях и рёбрах блока.

На рисунке 4.15 иллюстрируются общая схема и пример работы предложенного расширенного алгоритма дуальных контуров в двумерном случае.

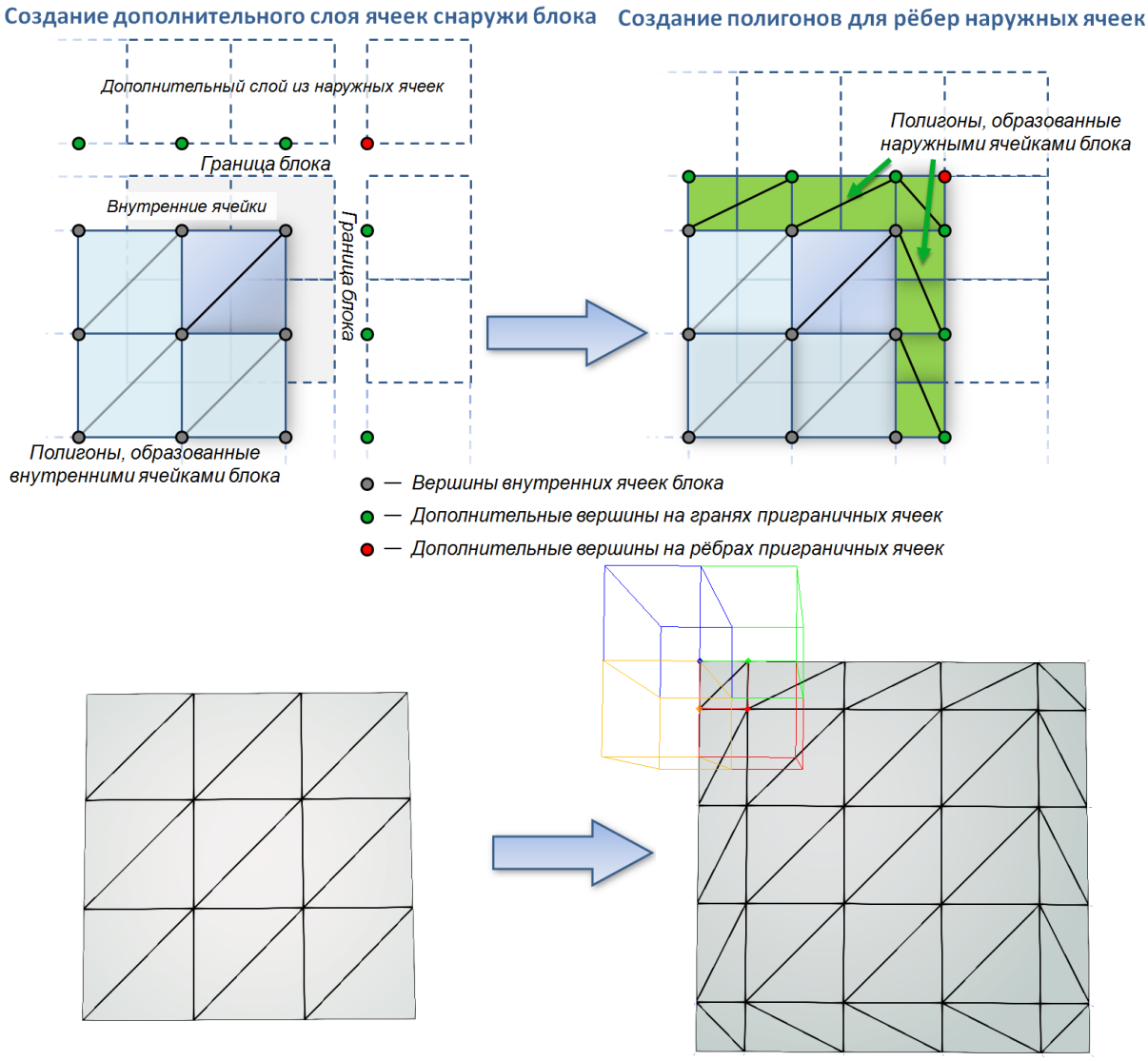


Рисунок 4.15 – *Сверху*: схема работы расширенного алгоритма дуальных контуров на регулярной кубической решётке (показана только угловая часть блока). Блок «окаймляется» дополнительным слоем ячеек, вершины которых лежат на границе блока. Активные рёбра каждой созданной ячейки также включаются в процесс триангуляции. *Снизу*: результаты триангуляции блока с разрешением 4^3 ячейки, содержащего фрагмент горизонтальной плоскости, с помощью стандартного и расширенного алгоритмов. (Цветом выделены четыре ячейки, образующие четырёхугольник в верхнем левом углу сетки).

На рисунке 4.16 приведены примеры треугольных сеток с открытой границей, полученные триангуляцией изоповерхностей, пересекающих область кубической формы. Можно заметить, как расширенный алгоритм дуальных контуров создаёт дополнительные вершины на границе области.

При этом предложенный алгоритм гарантирует, что границы смежных блоков будут представлять собой конформные треугольные сетки, которые будут бесшовно «пристыковываться» друг к другу.

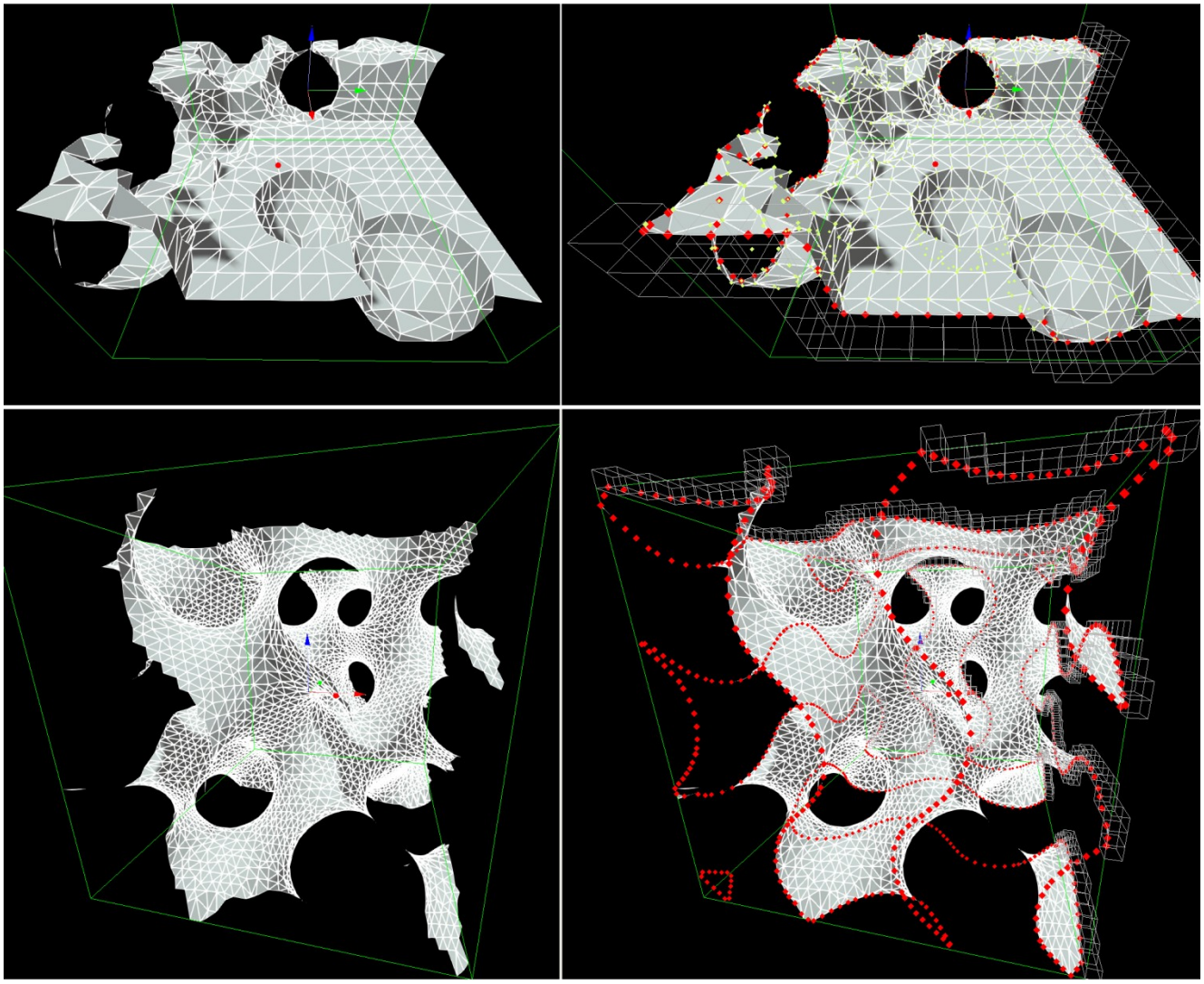
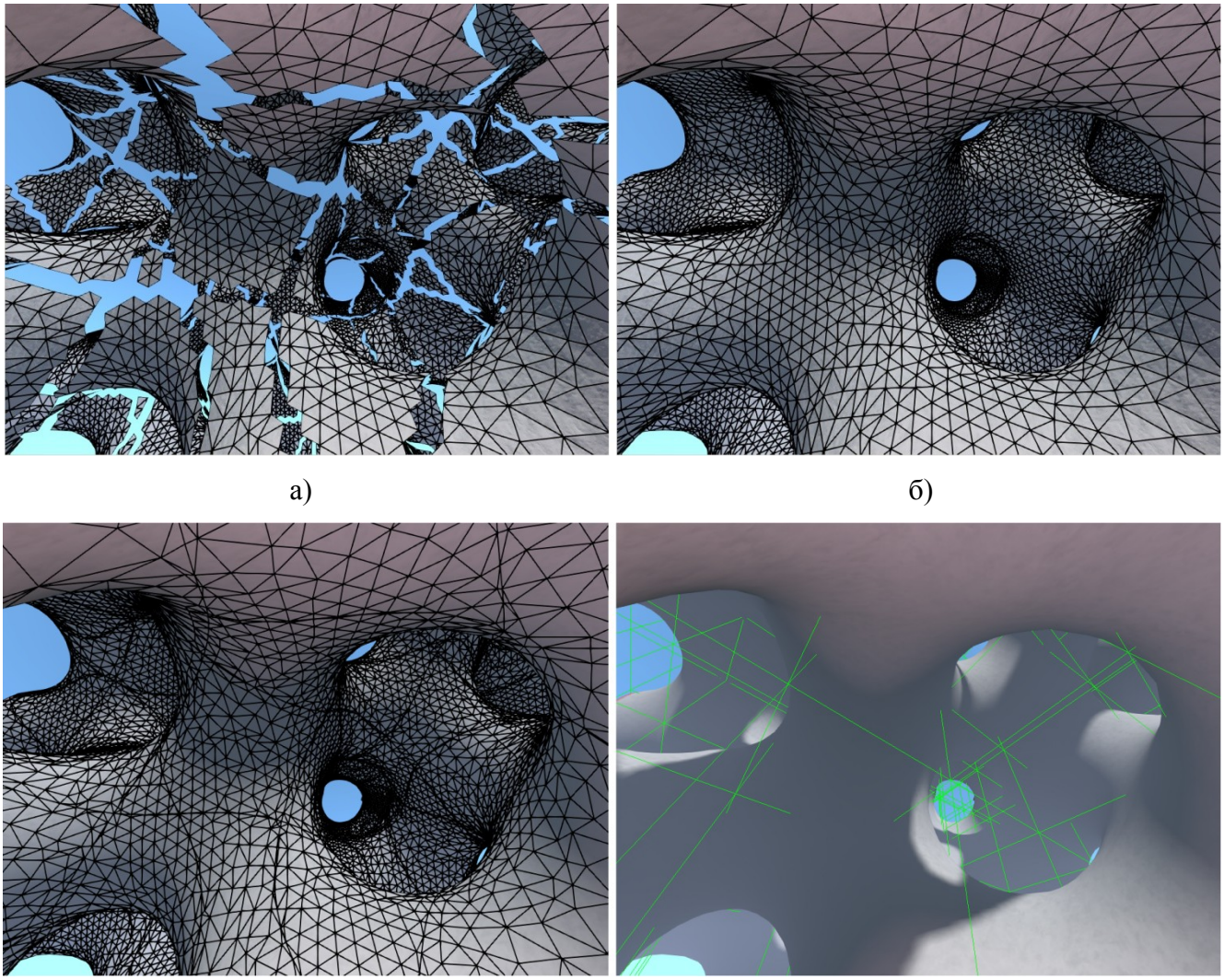


Рисунок 4.16 – *Слева*: триангуляция фрагмента изоповерхности с помощью стандартного алгоритма дуальных контуров. *Справа*: триангуляция того же фрагмента изоповерхности с помощью предложенного расширенного алгоритма дуальных контуров. Красными точками отмечены вершины дополнительных наружных ячеек, которые располагаются на гранях и рёбрах блока.

Это позволяет триангулировать каждый блок воксельного ландшафта полностью независимо от его соседей, без необходимости обращения к данным смежных блоков для заполнения разрывов, при этом построенные треугольные сетки полностью лежат внутри ограничивающих оболочек (параллелепипедов) (AABB) соответствующих блоков воксельного ландшафта.

На рисунке 4.17 приводится сравнение результатов триангуляции изоповерхности гироида ($\cos(x)\cdot\sin(y) + \cos(y)\cdot\sin(z) + \cos(z)\cdot\sin(x) = 0$) предложенными методами: построением бесшовного соединения путём адаптивной триангуляции октодеревьев, созданных из ячеек смежных блоков, и использованием расширенного алгоритма триангуляции.



а) б)

в) г)

Рисунок 4.17 – Бесшовная триангуляция поверхности гироида, разбитой на блоки с разрешением 8^3 ячеек различными методами. а) Результат триангуляции блоков по отдельности методом дуальных контуров. б) Бесшовная треугольная сетка, полученная перекрытием смежных блоков (как показано на рисунке 4.6, б).
в) Результат триангуляции с помощью модифицированного алгоритма дуальных контуров.
г) Изображение сцены с обозначением границ (AABB) блоков (выделены зелёным цветом).

Видно, что в обоих случаях треугольные сетки практически идентичны. В первом случае все блоки триангулируются как единое целое. Во втором случае расширенный алгоритм дуальных контуров создаёт бесшовную треугольную сетку за счёт создания треугольников плохой формы в местах соединения смежных блоков.

Преимущества данного способа бесшовной триангуляции:

– При триангуляции каждого блока не требуется обращаться к данным его соседей для генерации бесшовного соединения со смежными блоками.

– Ограничивающий параллелепипед (AABB) блока может использоваться для консервативного отсечения по пирамиде видимости, поскольку все вершины построенной сетки блока расположены полностью внутри его AABB.

– Разрешение каждого блока может быть любым, и не обязательно должно равняться степени двойки или быть кратным двум.

Ограничения данного подхода (помимо недостатков, унаследованных от алгоритма дуальных контуров на регулярной решётке, см. раздел 2.2):

– Смежные блоки должны иметь одинаковое разрешение, а их границы не могут быть упрощены, чтобы гарантировать создание конформной триангуляции.

– Необходимость в дублировании приграничных данных между смежными блоками: вокселей (индексов материалов), позиций точек пересечения и нормалей к поверхности на рёбрах приграничных ячеек; фактически, для хранения ландшафта может использоваться только воксельная решётка с Эрмитовыми данными.

– Создание треугольников плохой формы в области соединения между блоками.

4.3.2 Бесшовное соединение блоков с различными размерами и уровнями детализации

Предлагаемые методы могут быть применены для бесшовной триангуляции воксельного ландшафта, в котором в качестве LoD-схемы используется метод вложенных кубов отсечения или сбалансированное октодерево на основе метрики Чебышева (см. подраздел 4.2.2).

Бесшовное соединение переходных блоков с помощью адаптивной триангуляции. Самый простой подход к бесшовному соединению блоков различных размеров и разрешений является расширением предыдущей схемы: для группы смежных переходных блоков строится октодерево, размер которого в два раза превышает размер наибольшего блока в группе (или в

четыре раза превышает размер наименьшего блока), и затем над построенным октодеревом выполняется алгоритм адаптивной триангуляции [145].

При использовании метода вложенных кубов отсечения каждый блок «отвечает» за создание шва со своими максимальными соседями. При этом между блоками с отличающимися (в два раза) размерами и разрешениями (на границе между вложенными кубами) можно выделить два типа L-образных переходных областей (рисунок 4.18).

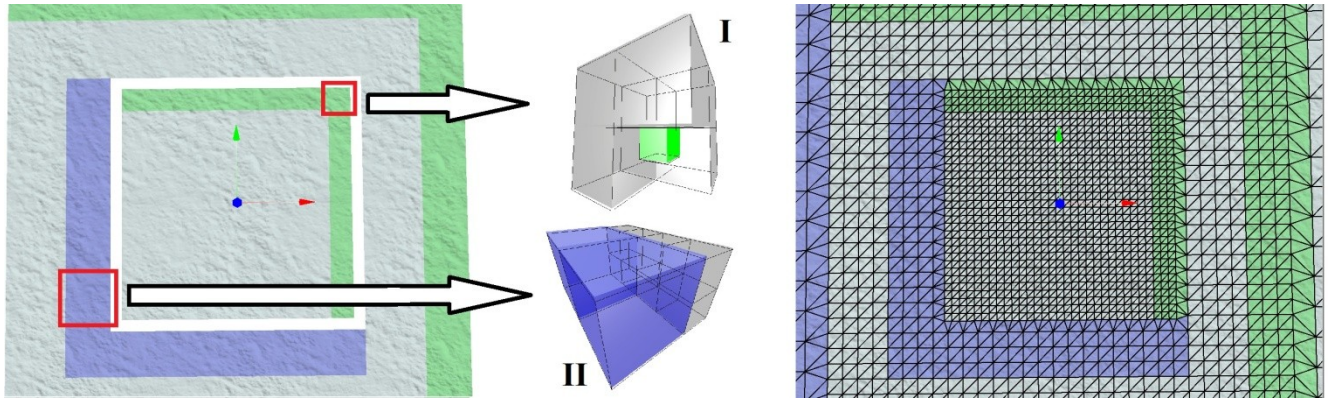


Рисунок 4.18 – Два вида L-образных переходных областей (выделены зелёным и синим цветами) между вложенными кубами отсечения с различными уровнями детализации (вид сверху на плоскость).

В первом случае максимальные соседи текущего блока имеют одинаковый или вдвое больший размер (если они принадлежат «наружному» кубу). Для бесшовной стыковки блока с его семью максимальными соседями строится октодерево, в четыре раза превышающее по размеру текущий блок.

В втором случае максимальные соседи текущего блока имеют одинаковый или вдвое меньший размер (если они принадлежат «внутреннему» кубу), поэтому блок может иметь более семи максимальных соседей, а размер построенного октодерева будет вдвое превышать размер текущего блока (см. рисунок 4.19).

В обоих случаях программная реализация формирования линейных октодеревьев является весьма громоздкой, но гораздо менее трудоёмкой и более эффективной, чем при использовании традиционных октодеревьев на указателях[13].

[13] Традиционную реализацию, описанную в [145], можно найти на GitHub: https://github.com/nickgildea/leven

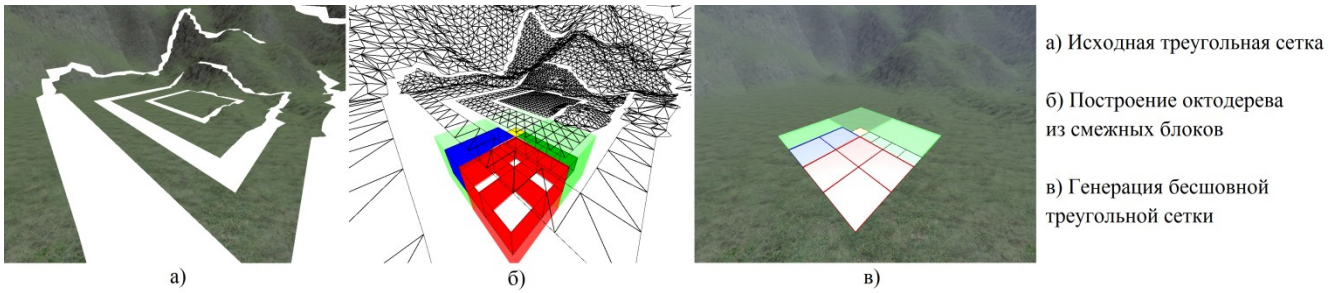


Рисунок 4.19 – Процесс заполнения разрывов в переходных областях второго типа. Шов принадлежит блоку в нулевом октанте (выделен красным цветом).

Вышеописанный подход к бесшовной триангуляции ландшафта обладает простотой реализации, но малопригоден на практике: при движении камеры данная схема приводит к избыточному перестроению (remeshing) блоков для сохранения бесшовного соединения.

В текущей реализации для построения линейных октодеревьев используются заранее предрасчитанные таблицы, содержащие битовые префиксы, которыми нужно предварять коды Мортона каждой ячейки для вставки ячейки в нужную ветвь октодерева. (Октодерево целесообразно строить только для «сшивки» переходных блоков, а блоки с одинаковым LoD (с одинаковым размером и разрешением) целесообразно «соединять» перекрытием, как показано на рисунке 4.7, б, в).

Данный подход к бесшовной триангуляции ландшафта, составленного из блоков различных размеров, унаследовал недостатки исходного метода: высокая нагрузка на CPU и создание зависимостей между смежными блоками ландшафта, которые значительно снижают производительность и ограничивают возможности распараллеливания. Кроме того, при смене уровней детализации блоков наблюдаются резкие скачки, что ухудшает визуальное восприятие сцены.

Для исправления вышеуказанных недостатков в ходе исследования был разработан способ бесшовного соединения блоков, основанный на использовании предложенного в предыдущем подразделе расширенного алгоритма дуальных контуров и геоморфинга, полностью устраняющий зависимости между блоками.

Закрытие разрывов путём геоморфинга. Предложенный в предыдущем подразделе расширенный алгоритм дуальных контуров создаёт бесшовную треугольную сетку, если блоки ландшафта имеют одинаковые размеры и разрешения. Если ландшафт содержит блоки с различными уровнями детализации, то между переходными блоками возникают разрывы (рисунок 4.20).

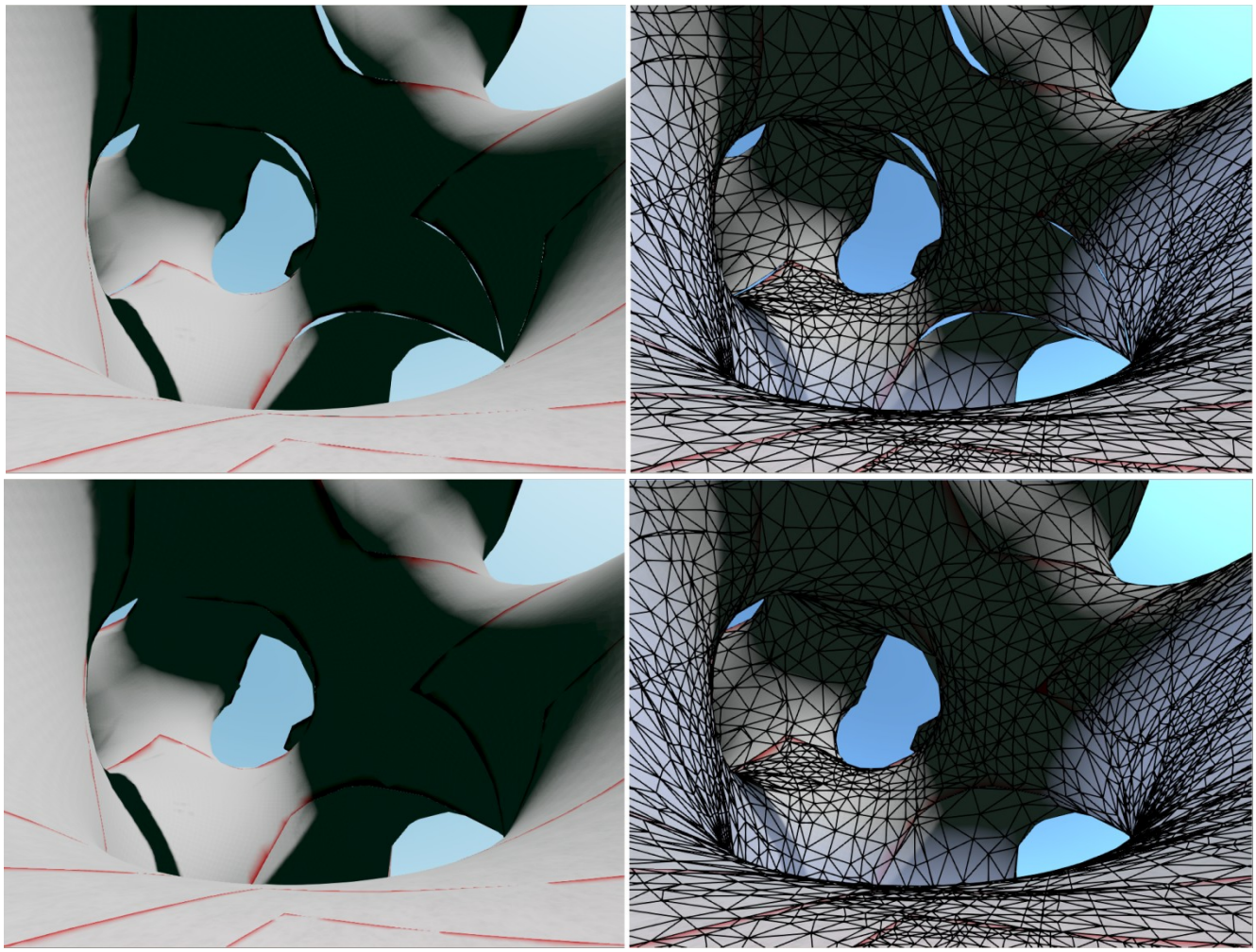


Рисунок 4.20 – *Сверху*: триангуляция гироида расширенным алгоритмом дуальных контуров приводит к появлению разрывов сетки между смежными блоками с различными уровнями детализации. *Снизу*: все разрывы и Т-стыки полностью устранены с помощью геоморфинга, при этом триангуляция каждого блока выполнялась полностью независимо от его соседей. Сцена представлена восемью уровнями детализации, разрешение каждого блока составляет 4^3 ячейки. Красным цветом обозначены области перехода к более грубому уровню детализации.

Для закрытия разрывов между блоками с различными LoD было решено использовать геоморфинг на GPU в вершинном шейдере, чтобы полностью исключить вмешательство CPU. *Геоморфинг* (*geo-morphing*, *geometrical morphing*) [128−132] — процесс плавного изменения уровня детализации полигональной сетки путём постепенного перемещения каждой её вершины из текущей позиции в соответствующую ей позицию на другом уровне детализации. При увеличении уровня детализации происходит разбиение вершин полигональной сетки, а при уменьшении — слияние вершин. Когда все вершины сетки находятся в позиции, соответствующей более грубому

LoD, сетка может быть незаметно подменена более грубой версией с меньшим количеством вершин и полигонов, и наоборот.

Помимо позиций вершин должны также интерполироваться нормали, текстурные координаты, цвета и т.д. Коэффициент интерполяции $k_{morph} \in [0, 1]$ обычно расчитывается на основе расстояния от вершины до камеры. Таким образом, у каждой вершины должны быть известны её атрибуты на двух смежных уровнях детализации. Полигональные сетки, обладающие информацией для двух и более LoD, называются *прогрессивными* (*progressive meshes*, *PM*) [128−130]. В процессе визуализации для предотвращения разрывов между переходными блоками достаточно ставить граничные вершины каждого блока в позицию в соответствии с уровнем детализации соседнего блока [133]. Очевидно, что уровни детализации соседних блоков не должны отличаться более чем в два раза.

Прогрессивные сетки для геоморфинга могут быть легко построены для рендеринга процедурно-сгенерированного ландшафта. На этапе препроцессинга при триангуляции каждого блока разрешением *n* ячеек для каждой вершины треугольной сетки вычисляются её позиция и нормаль на текущем $\mathcal{L}_{fine}$ и следующем, более грубом уровне $\mathcal{L}_{coarse}$ детализации. Для вычисления позиций и нормалей вершин на $\mathcal{L}_{coarse}$ достаточно сгенерировать и выполнить триангуляцию воксельных данных на решётке с вдвое бóльшим шагом (т.е. с вдвое меньшим разрешением в *n/2* ячеек) (subsampling) [150]. Затем необходимо сопоставить каждую вершину $\mathcal{V}_{\text{fine}}$ сетки текущего уровня детализации с соответствующей вершиной $\mathcal{V}_{\text{coarse}}$ сетки более грубого уровня детализации, если такая существует. Если изоповерхность является гладкой, то для сопоставления $\mathcal{V}_{\text{fine}} \leftrightarrow \mathcal{V}_{\text{coarse}}$ достаточно сгенерировать только вершинный буфер на более грубом уровне $\mathcal{L}_{coarse}$ детализации. Если изоповерхность содержит острые углы и рёбра, то необходимо построить сетку на $\mathcal{L}_{coarse}$, дублировать острые вершины и присвоить им нормали смежных граней, для каждой $\mathcal{V}_{\text{fine}}$ выбрать $\mathcal{V}_{\text{coarse}}$ с наименьшим углом отклонения нормали.

В процессе рендеринга для каждой вершины $\mathcal{V}$ полигональной сетки вершинный шейдер линейно интерполирует её атрибуты между $\mathcal{V}_{\text{fine}}$ и $\mathcal{V}_{\text{coarse}}$:

$$\mathcal{V} = \text{lerp}(\mathcal{V}_{\text{fine}}, \mathcal{V}_{\text{coarse}}, k_{morph}) = \mathcal{V}_{\text{fine}} \cdot (1 - k_{morph}) + \mathcal{V}_{\text{coarse}} \cdot k_{morph},$$

где $\text{lerp}(\mathbf{x}, \mathbf{y}, \alpha)$ — это стандартная (встроенная в язык шейдеров) функция, которая возвращает линейно-интерполированное значение между x и y:

$$\text{lerp}(\mathbf{x}, \mathbf{y}, \alpha) = \mathbf{x} + (\mathbf{y} - \mathbf{x}) \cdot \alpha.$$

Для избежания разрывов между переходными блоками при интерполяции приграничных вершин значение k_{morph} нужно ограничивать в соответствии с LoD соседних блоков: если вершина

$\mathcal{V}$ граничит с блоком более грубого LoD, то для избежания разрывов необходимо принять $k_{morph} = 1$. При использовании LoD-метрики на основе кубической (L_∞) нормы (метрики Чебышёва) (см. подраздел 4.2.2) для вычисления k_{morph} в вершинном шейдере может быть использована формула из библиотеки Proland (**pro**cedural **land**scape) [147], которая обеспечивает плавное «перетекание» уровней детализации и автоматически гарантирует отсутствие разрывов сетки на границах между переходными блоками:

$$k_{morph} = \text{clamp}\left(\frac{\frac{d}{L} - (k+1)}{k-1}, 0, 1\right),$$

где d — расстояние в L_∞-норме между позициями вершины и камеры, L — размер блока (длина ребра его ограничивающего куба), k — коэффициент, влияющий на измельчение октодерева (k всегда должен быть больше единицы для получения сбалансированного октодерева), а $\text{clamp}(x, a, b)$ — это стандартная (встроенная) шейдерная функция, которая ограничивает значение аргумента диапазоном $[a, b]$:

$$\text{clamp}(x, a, b) = \max(\min(x, a), b).$$

На риснуке 4.21 проиллюстрирован процесс плавного изменения отображаемой треугольной сетки для незаметной смены уровней детализации при движении камеры наблюдателя (его позиция отмечена красной точкой) справа налево.

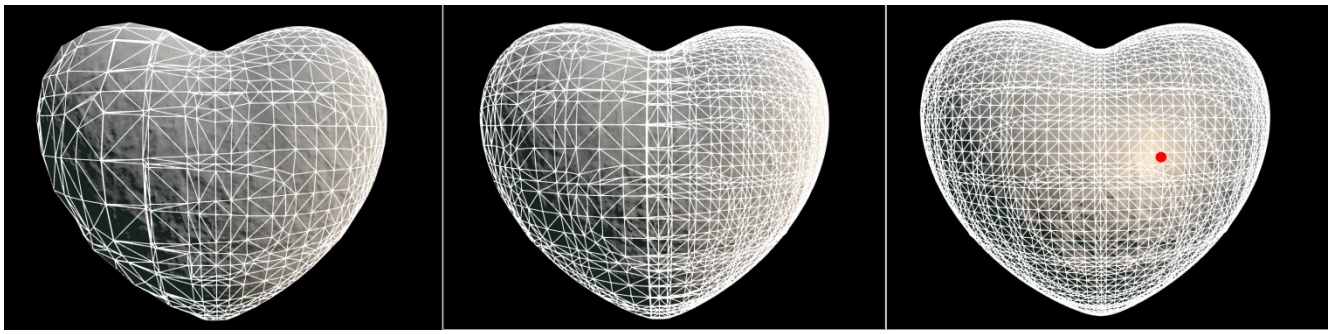


Рисунок 4.21 – Использование геоморфинга для плавной смены уровней детализации на примере изоповерхности «Сердце»: $(2x^2 + y^2 + z^2 - 1)^6 - (0.1x^2 + y^2)z^3 = 0$.

Преимущества данного подхода для визуализации ландшафта (помимо достоинств, унаследованных от расширенного алгоритма дуальных контуров):

– Низкая нагрузка на CPU и минимальное количество обращений к внешней памяти: при триангуляции каждого блока не требуется обращаться к данным его соседей для создания бесшовного соединения, поскольку каждый блок содержит всю информацию, необходимую для рендеринга бесшовной сетки и плавного переключения уровней детализации. Если в качестве LoD-схемы используется октодерево, то отпадает необходимость находить соседние узлы октодерева, а

сбалансированность октодерева, отсутствие разрывов и плавная смена уровней детализации гарантируются LoD-метрикой. При смене уровня детализации блока нет необходимости перестраивать (remesh) его полигональную сетку (и сетки его соседей для сохранения бесшовного соединения), а затем загружать их на GPU — геометрические данные могут быть максимально оптимизированы и закэшированы на диске.

– Плавная, почти незаметная смена уровней детализации.

– Универсальность: геоморфинг может быть использован с любыми алгоритмами упрощения полигональных сеток, в которых можно сопоставить вершины сетки до и после её упрощения. При этом сетка не обязательно должна быть закрытой и развёртываемой с двусторонней топологией (2-manifold).

Недостатки данного подхода (помимо недостатков, унаследованных от расширенного алгоритма дуальных контуров на регулярной решётке):

– Увеличение объёма памяти, занимаемого полигональными сетками, из-за необходимости хранения вершинных данных для двух смежных уровней детализации.

– Увеличение нагрузки на GPU из-за усложнения вершинного шейдера: использование геоморфинга приводит к увеличению количества инструкций, комбинаций über-шейдеров и повышению стоимости рендеринга теней.

– При перестройке уровней детализации после редактирования ландшафта необходимо пересохранять полигональные сетки дочерних блоков, чтобы они затем смогли плавно перетекать в более грубую сетку родительского блока.

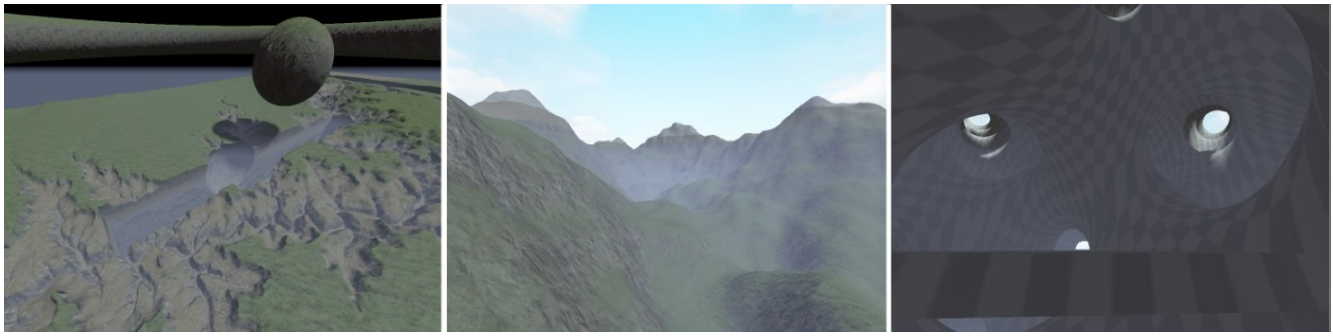
– Сложности с адаптивной триангуляцией / упрощением сетки и плавным блендингом материалов между уровнями детализации.

– Полученная бесшовная треугольная сетка имеет визуальный характер и «существует» только на GPU в момент отрисовки, поэтому она не может быть использована на CPU для обнаружения столкновений (collision detection) или для отбраковки невидимых (заслонённых) объектов (software occlusion culling).

4.4 Результаты экспериментов

Тесты эффективности предложенных подходов для многомасштабной визуализации воксельных ландшафтов были выполнены на компьютере, оснащённом процессором Intel® Core™ i7-2600K @ 3.40 ГГц, 16 Гб оперативной памяти и графическим процессором среднего класса

AMD Radeon 6950. Все тесты производительности выполнялись при разрешении экрана 1920×1200 пикселей. В экспериментах визуализировались воксельные ландшафты, полученные из карт высот или сгенерированные из функций расстояния со знаком (SDF) (рисунок 4.22).



а) б) в)

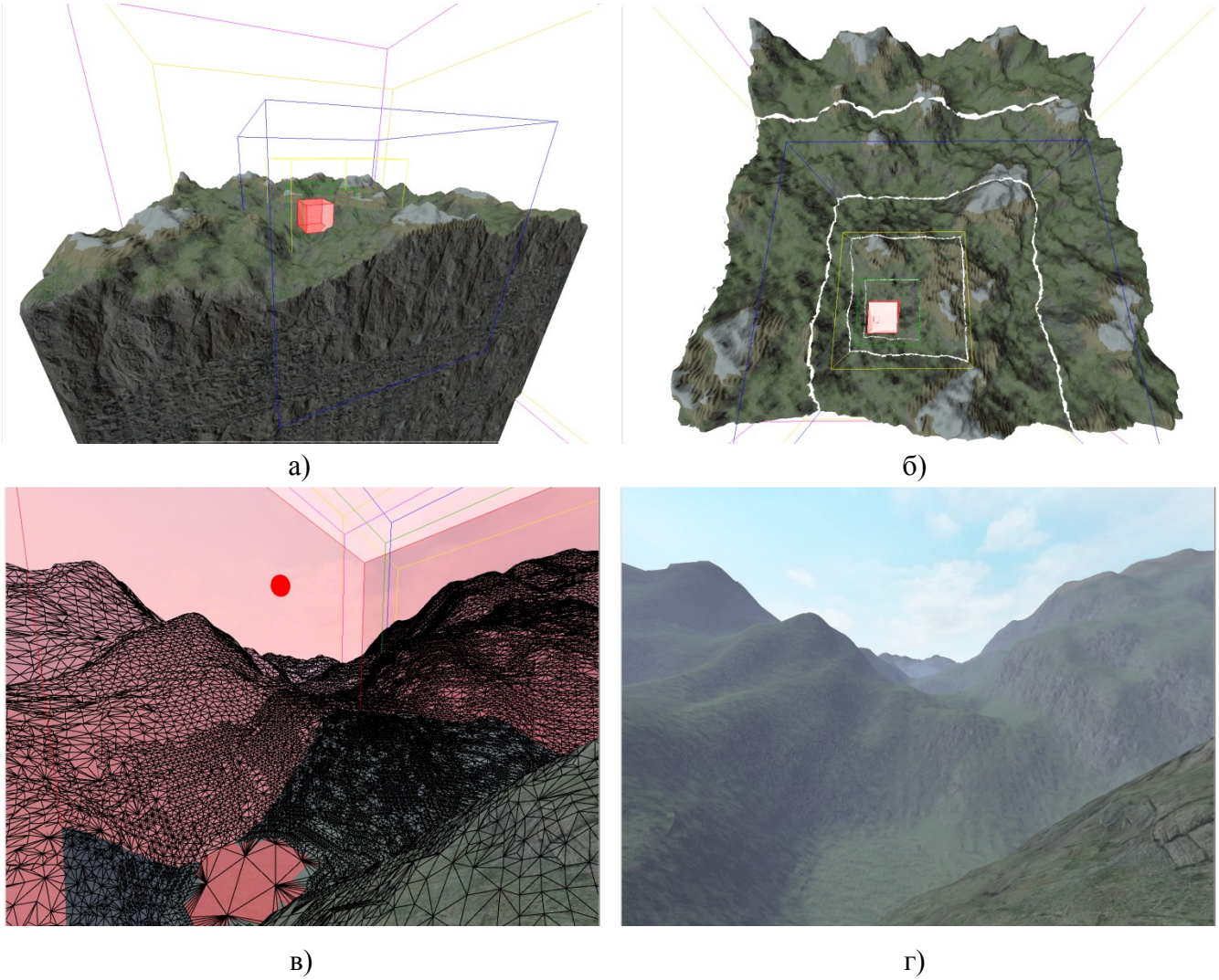
Рисунок 4.22 – Примеры тестовых сцен, использованных в экспериментах: а) карта высот Grand Canyon с тором и сферой; б) карта высот из Urho3D; в) синтетическая изоповерхность.

Бесшовное соединение с помощью адаптивной триангуляции. В первом эксперименте в качестве LoD-схемы использовался метод вложенных регулярных сеток, а бесшовное соединение блоков осуществлялось с помщью адаптивной триангуляции. Блоки ландшафта в формате SLOG (см. раздел 3.6) генерировались из исходных объёмных данных без какой-либо подготовки, путём сэмплирования (выборки) SDF или карты высот с фиксированным шагом, соответствующим уровню детализации блока. Поскольку процедурная генерация ландшафта из аналитического описания и подготовка блоков в формате SLOG занимает много времени, в качестве готовых источников данных использовались карты высот. (Карта высот может рассматриваться, как источник объёмных данных, поскольку для неё определены значения «внутри»/«снаружи», а Эрмитовы данные могут быть найдены путём пересечением рёбер ячеек с этой картой высот.) Материал (глина, трава, земля, песок, снег) каждой вершины полигональной сетки (или ячейки в SLOG) устанавливался в зависимости от её «высоты» (значения координаты Z). При рендеринге ландшафта между смежными вершинами не выполнялся блендинг материалов, поэтому заметны резкие переходы между участками поверхности с различными материалами.

На рисунке 4.23 показаны результаты визуализации карты высот, входяшей в состав дистрибутива библиотеки для разработки видеоигр Urho 3D[14], с шестью уровнями детализации, что соответствует реальному размеру примерно 5×5 км. Можно заметить, как адаптивная

[14] https://urho3d.github.io/

триангуляция учитывает локальные особенности рельефа и уменьшает количество треугольников на плоских участках ландшафта.



а) б)

в) г)

Рисунок 4.23 – Бесшовная триангуляция ландшафта на примере карты высот из Urho3D. а) Вид на ландшафт сбоку, красным цветом выделен наименьший куб, в котором находится камера наблюдателя; б) вид на ландшафт сверху, показаны границы между вложенными кубами отсечения; в) триангуляция с точки зрения наблюдателя; г) финальное изображение. Сцена представлена шестью кубами отсечения, каждый куб содержит 8^3 блоков, каждый блок — 16^3 ячеек. Разрешение карты высот — 1024×1024, весь ландшафт целиком состоит из 1 072 025 Δ.

На рисунке 4.24 представлены результаты визуализации карты высот Grand Canyon[15], которая представляет собой восстановленный по данным спутникового сканирования реальный участок местности в Аризоне (США).

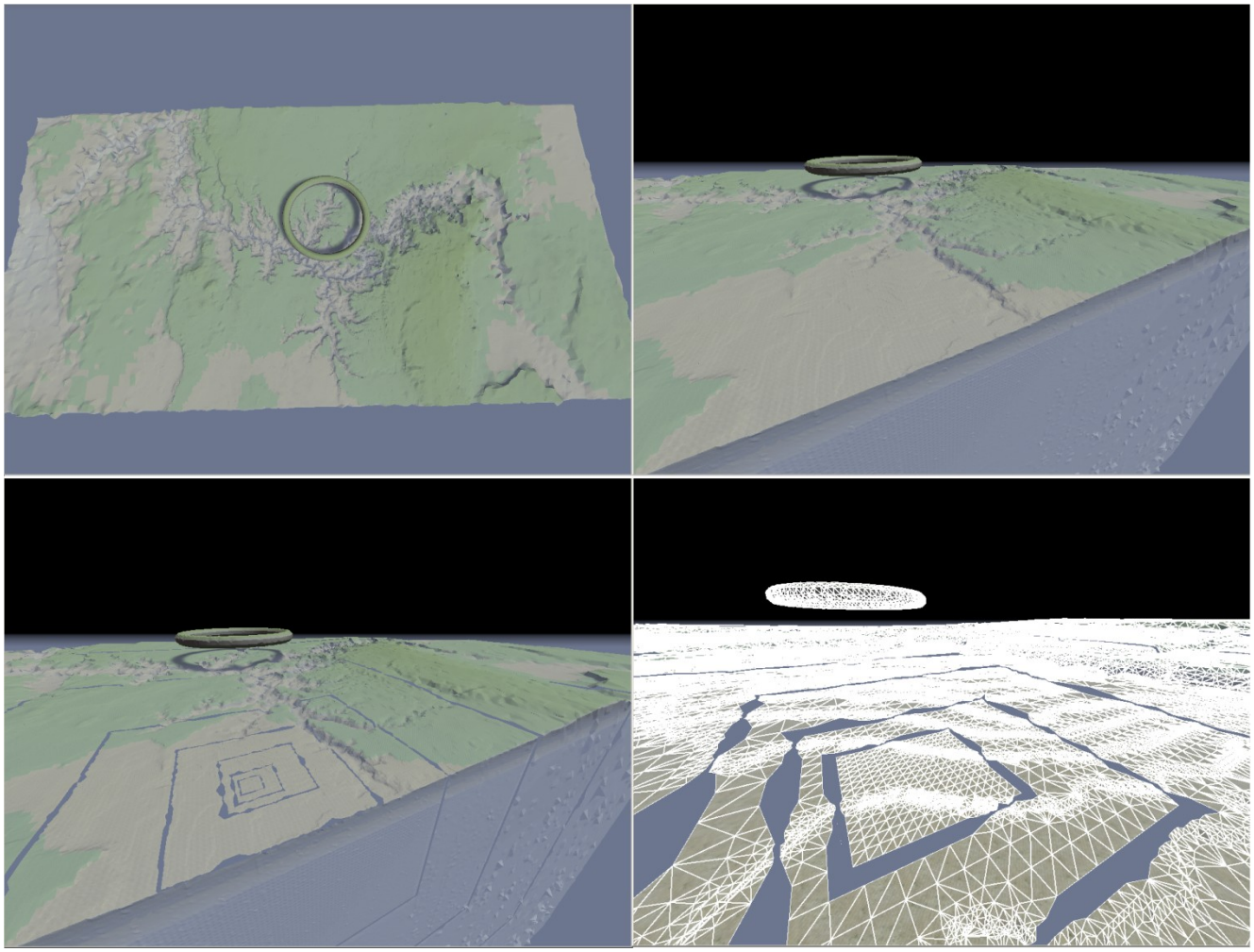


Рисунок 4.24 – Бесшовная триангуляция ландшафта на примере карты высот Grand Canyon. Сверху показан вид ландшафта с разных ракурсов. Снизу показаны разбиение на блоки и треугольная сетка. Сцена представлена восемью кубами отсечения, каждый куб содержит 8^3 блоков, каждый блок — 16^3 ячеек. Разрешение карты высот — 4097×2049, ландшафт состоит из ~1.5 млн. Δ.

С неподвижной камерой и отключенными тенями средняя частота смены кадров при отрисовке ландшафта составляла более сотни кадров в секунду. Со включенными тенями производительность снижалась до 9−11 кадров в секунду (поскольку создавалось более четырёх тысяч батчей из-за неоптимизированной реализации каскадных теней). При движении камеры по ландшафту возникали видимые задержки, временно наблюдались разрывы между сетками

[15] http://www.cc.gatech.edu/projects/large_models/gcanyon.html

смежных блоков ландшафта и «проседание» частоты кадров из-за процедурной генерации блоков ландшафта и построения бесшовной треугольной сетки, а также наблюдалась резкая, скачкообразная смена уровней детализации.

Таким образом, данный подход позволяет отрисовывать большие открытые пространства, однако предложенная схема бесшовной триангуляции существенно нагружает CPU, что ограничивает её практическую применимость. В реальных приложениях ВР, помимо рендеринга ландшафта, CPU будет занят множеством других задач: выполнение логики приложения, определение и обработка столкновений, физическое моделирование и т.д. Поэтому целесообразнее использовать более ориентированные на GPU методы, которые создают менее точную адаптивную триангуляцию, но максимально разгружают CPU.

Визуализация с помощью расширенного алгоритма дуальных контуров и геоморфинга. Во втором эксперименте в качестве LoD-схемы использовалось октодерево, триангуляция выполнялась с помощью модифицированного алгоритма дуальных контуров, исключение разрывов и плавная смена уровней детализации осуществлялись путём геоморфинга. Каждый блок ландшафта после создания хранился в виде треугольной сетки в готовом для рендеринга виде.

На рисунке 4.25 показан пример визуализации изоповерхности бутылки Клейна с 12 уровнями детализации и разрешением каждого блока в 32^3 ячейки. При размерах каждой ячейки в один метр размер сцены составил бы 131 км. Благодаря минимальному обращению к внешней памяти обеспечивалось передвижение камеры наблюдателя со скоростью 2500 метров в секунду без ощутимых задержек, при этом скорость визуализации не падала ниже 92 кадров в секунду.

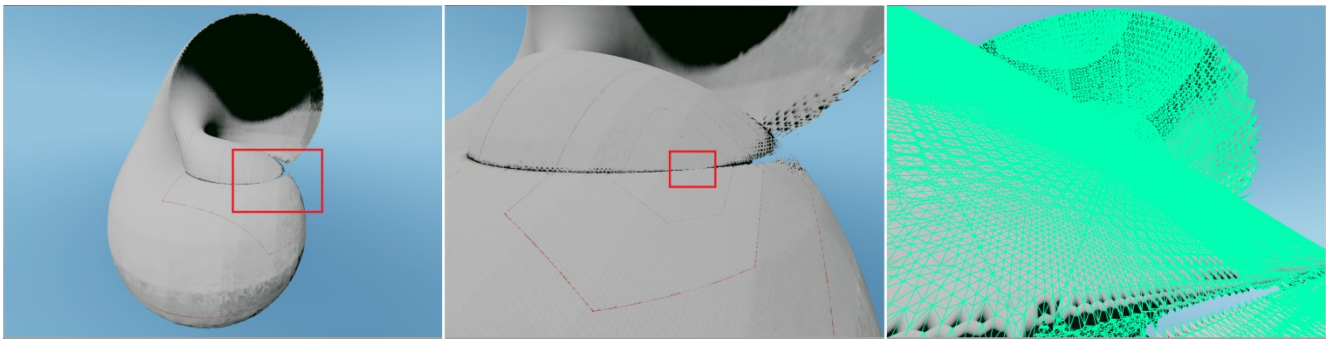


Рисунок 4.25 – Визуализация изоповерхности бутылки Клейна в различных разрешениях. Внутри выделенной области видны артефакты, появляющиеся из-за того, что в неоднозначных ячейках (где поверхность пересекает саму себя) алгоритм дуальных контуров создаёт по одной вершине.

На рис. 4.26 изображена синтетическая изоповерхность с острыми углами и рёбрами, с 8 уровнями детализации и разрешением каждого блока в 16^3 ячейки, при этом средняя частота кадров составила 134 кадра в секунду.

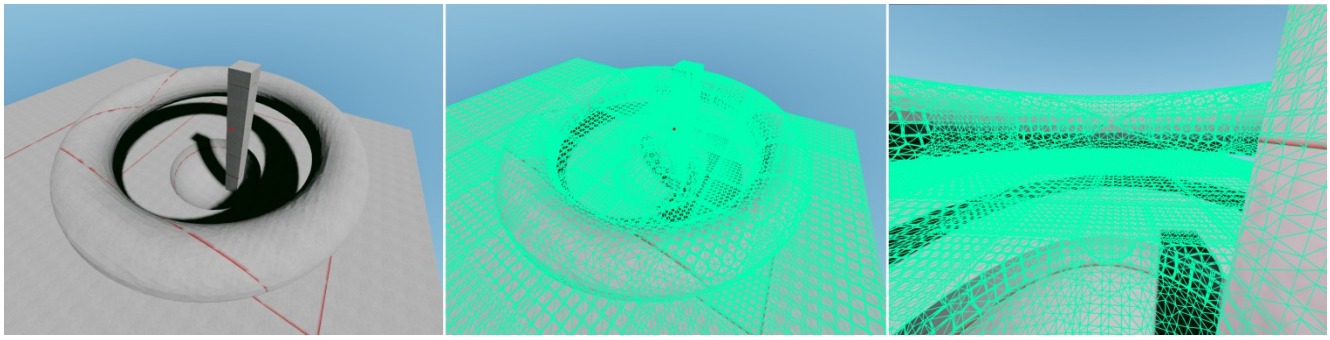


Рисунок 4.26 – Пример визуализации изоповерхности с острыми углами (используется «плоское» затенение).

На рисунке 4.27 показаны результаты визуализации карты высот Grand Canyon с 12 уровнями детализации и разрешением каждого блока в 32^3 ячейки, что при размерах каждой ячейки в один метр соответствует ландшафту размером 33 км. На рисунке в центре отрисовывалось более 800 тысяч треугольников с частотой более 100 кадров в секунду. Скорость визуализации может быть ещё увеличена, если применять геоморфинг только для рендеринга переходных блоков.

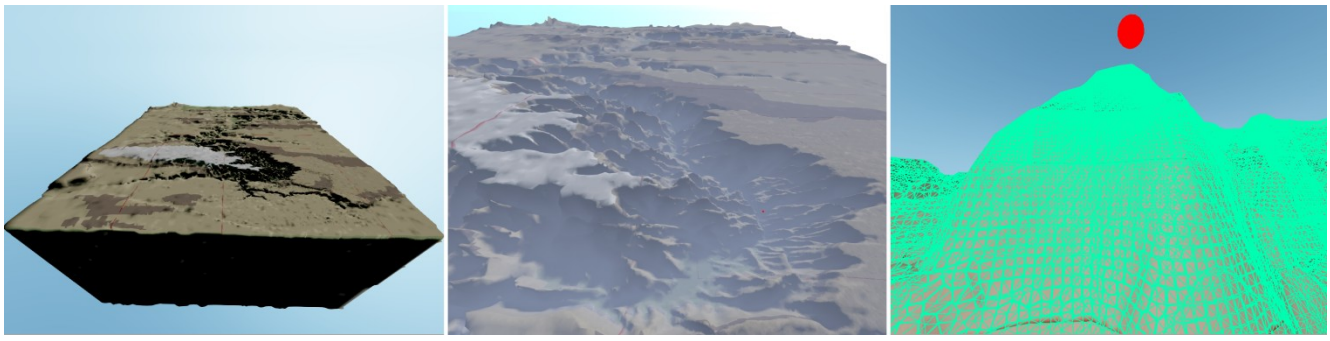


Рисунок 4.27 – Пример визуализации воксельного ландшафта, построенного из карты высот Grand Canyon.

На рисунках 4.28, 4.29 и 4.30 показаны примеры других гладких изоповерхностей.

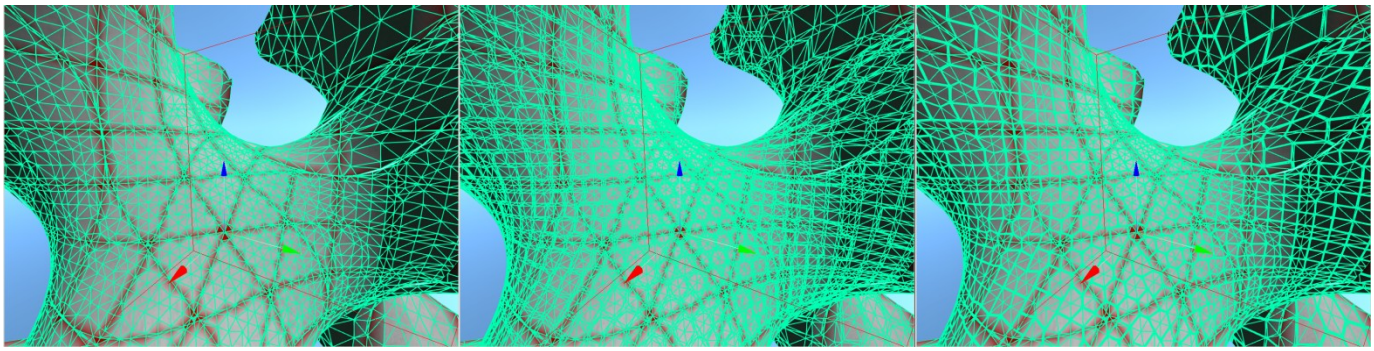


Рисунок 4.28 – Визуализация изоповерхности гироида с плавной сменой уровней детализации.

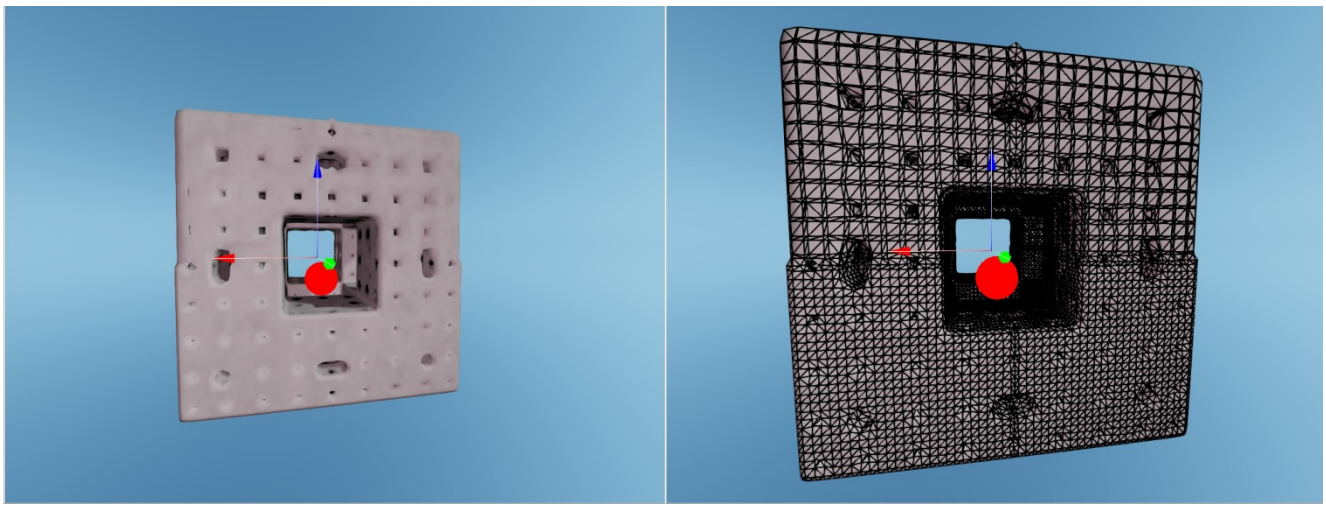


Рисунок 4.29 – Визуализация «мягкой» губки Менгера с блендингом между различными уровнями детализации.

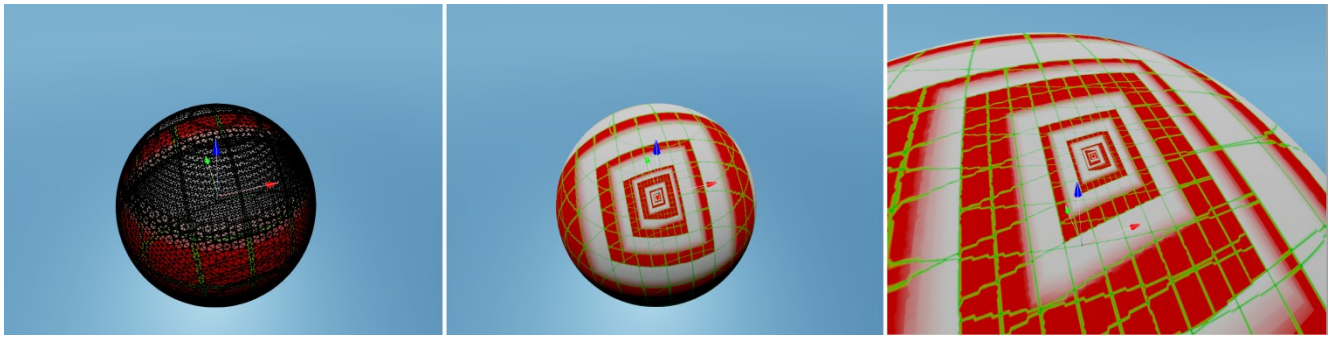


Рисунок 4.30 – Визуализация сферы с 15 уровнями детализации.

На рисунке 4.31 показан пример изменяемого ландшафта с 10 уровнями детализации и разрешением каждого блока в 32^3 ячейки. После редактирования общий размер ландшафта на диске (в сжатом виде) увеличился с 1.5 до 1.7 Gb.

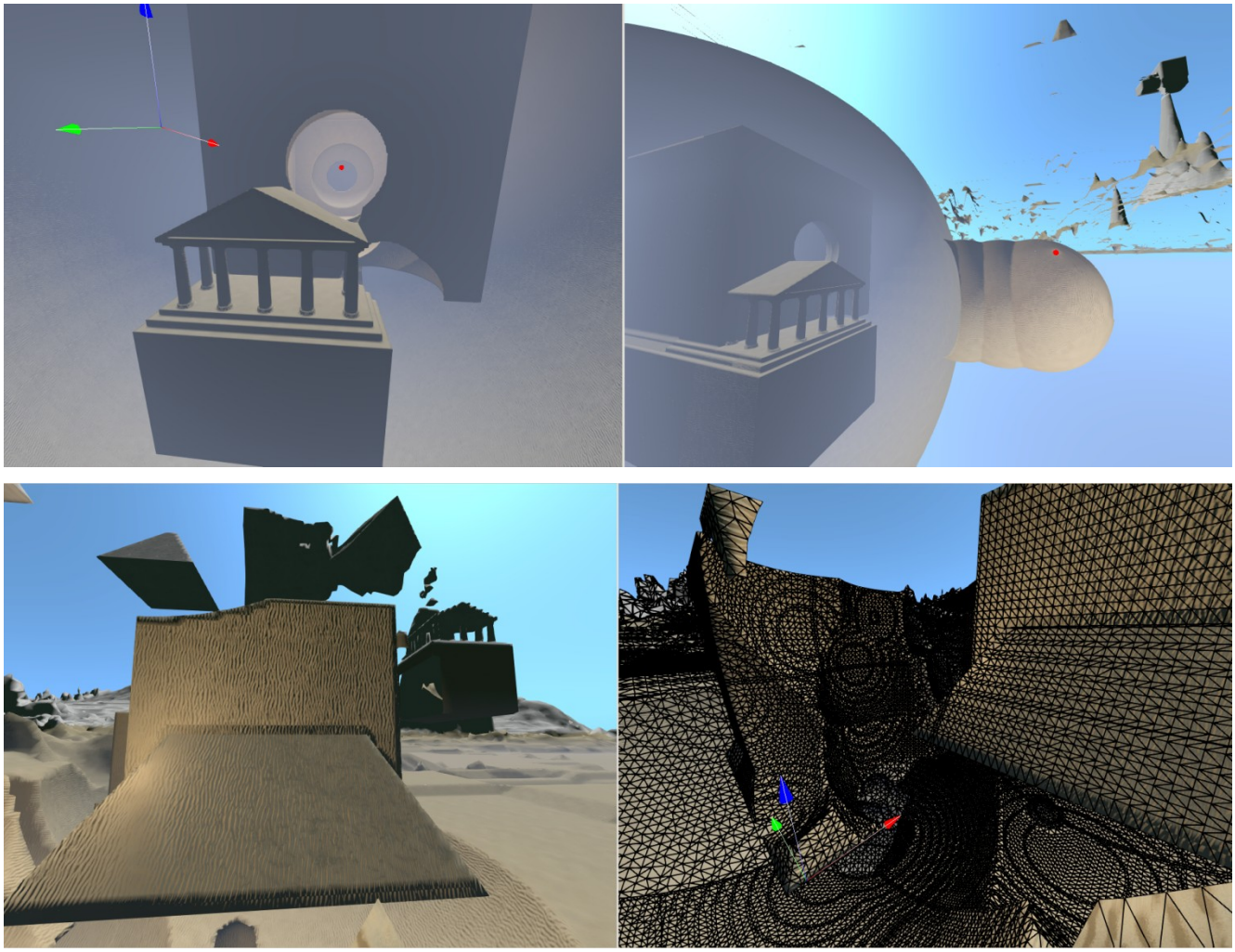


Рисунок 4.31 – Результаты моделирования и визуализации воксельного ландшафта с помощью модифицированного алгоритма дуальных контуров и геоморфинга.

По сравнению с технологией визуализации ландшафтов, основанной на бесшовном соединении блоков путём адаптивной триангуляции, геоморфинг гораздо лучше подходит для интерактивной визуализации ландшафта с высокой скоростью перемещения наблюдателя, что встречается в таких приложениях, как геоинформационные системы, аэрокосмические симуляторы, и видеоигры, где требуется быстрый рендеринг ландшафтов планетарных масштабов.

4.5 Основные выводы по четвёртой главе

Разработаны, реализованы и протестированы алгоритмы и методы для многомасштабного представления и визуализации сверхбольших воксельных ландшафтов с различными уровнями детализации.

Разработан метод для бесшовной триангуляции воксельного ландшафта, состоящего из блоков с различными уровнями детализации, с помощью предложенного алгоритма адаптивной триангуляции (подраздел 2.4.2) и линейных октодеревьев в формате SLOG (раздел 3.6). Метод обеспечивает возможность «стыковки» блоков с различными разрешениями, построение минимальной адаптивной триангуляции, высокое качество восстановления острых углов и рёбер поверхности, минимальное количество занимаемого на диске пространства, отсутствие дублирования данных между смежными блоками. Главным недостатком разработанного метода бесшовной триангуляции является создание зависимостей между смежными блоками ландшафта (для бесшовной триангуляции каждого блока необходимо каждый раз обращаться к данным его максимальных соседей), что снижает производительность и ограничивает возможности распараллеливания.

Разработан модифицированный алгоритм дуальных контуров, основные идеи которого заключаются в дублировании приграничных данных на границах смежных блоков и в создании и включении в процесс триангуляции дополнительных вершин, расположенных точно на гранях и рёбрах блока, как в алгоритме кубических марширующих квадратов (CMS). Это позволяет выполнять триангуляцию каждого блока полностью параллельно и независимо от соседей. Описано расширение алгоритма для визуализации изоповерхностей с непрерывным уровнем детализации с помощью геоморфинга на GPU в вершинном шейдере. Проведены эксперименты, доказывающие эффективность предложенных методов.

5 Программная реализация методов и алгоритмов визуализации воксельных ландшафтов в системах виртуальной реальности

5.1 Описание программного комплекса

Разработанный программный комплекс ориентирован на моделирование и визуализацию воксельных ландшафтов в интерактивном режиме в системах ВР.

Инструменты разработки. Программный комплекс реализован на языке программирования C++. В качестве IDE использовались Microsoft© Visual Studio 9-15 (2008-2017). Сборка проекта автоматизирована с помощью системы Premake (использует Lua для описания конфигурации сборки). Для автоматической генерации файлов документации используется Doxygen.

Аппаратное обеспечение. Программный комплекс разрабатывался с учётом современных многоядерных процессоров и распараллелен с помощью многопоточности и путём использования SIMD инструкций (SSE2 − SSE4.2).

Для достижения максимальной производительности рекомендуется использовать 64-битные экземпляры библиотек и исполняемых файлов, но при необходимости комплекс может быть собран в 32-битном режиме. Для работы демонстрационных программ необходим графический процессор с полной аппаратной поддержкой графических интерфейсов Direct3D 11 или OpenGL 4.

Внешние зависимости. Демонстрационные программы используют для своей работы различные сторонние библиотеки, перечисленные в таблице 5.1.

Для грубого профилирования и поиска ошибок зацикливания использовался профайлер VerySleepy, для более точного определения узких мест программы — Brofiler. Для отладки и профилирования участков кода, интенсивно работающего с динамической памятью, использовались библиотека DebugHeap и инструмент MemTrace от Insomniac Games.

Таблица 5.1 – Список используемых внешних библиотек.

Название библиотеки	**Задачи, решаемые библиотекой**
SDL	Создание окон, обработка событий ввода/вывода, управление курсором мыши и т.д
Microsoft DirectX SDK (June 2010)	SDK для работы с графическим API Direct3D
assimp	Загрузка полигональных моделей из различных форматов (в компиляторе ресурсов и демонстрационных программах)
Dear ImGui	Отрисовка графического интерфейса для отображения и изменения параметров работающей программы
Eigen	Решение оптимизационных задач для нахождения оптимальных позиций острых вершин в двойственных методах триангуляции
stb	Загрузка карт высот в форматах PNG, TGA, BMP (для тестирования комплекса); stb также содержит генератор тайлов Вонга, упаковщик атласов, реализацию шума Перлина и множество других полезных функций
liblmdb (LMDB)	Запись, хранение и чтение больших объёмов бинарных данных путём проецирования файла DB в адресное пространство процесса
LZ4	Сжатие воксельных данных (индексов материалов)
double-conversion	Быстрая конвертация чисел с плавающей точкой двойной точности в текстовые строки (при сериализации в текстовые форматы)
Lua	Привязка и выполнение команд пользователя, введённых с помощью интерактивной консоли
dlmalloc	Организация «кучи» для работы с динамической памятью в Lua
SQLite	Работа с базой данных ресурсов (ассетов) на этапе их компиляции
efsw	Отслеживание изменений в файловой системе для автоматической перекомпиляции изменённых исходных файлов ресурсов в системе сборки ресурсов, и перезагрузки ресурсов во время работы программы

Архитектура. Программный комплекс использует модульную архитектуру.

На рисунке 5.1 показана общая структура созданного программного комплекса и диаграмма зависимостей между его основными модулями.

По умолчанию каждый модуль представлен в виде отдельной статически подключаемой библиотеки. В файле premake5.lua задаётся конфигурация и настройки сборки: подключение внешних библиотек, настройки профилирования, опции препроцессора, параметры компиляции и

т.д. В финальной сборке отключен механизм исключений C++, вместо этого функции возвращают коды ошибок.

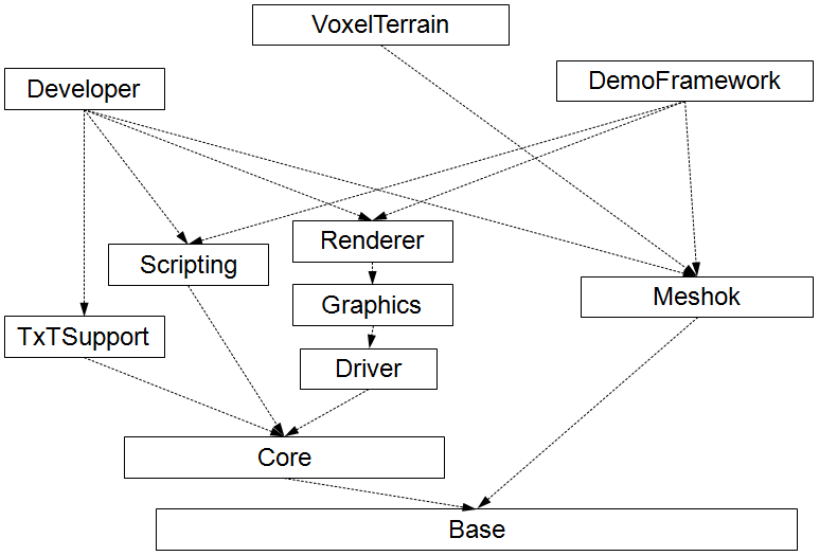


Рисунок 5.1 – Основные модули разработанного программного комплекса и связи между ними.

Описание основных модулей. Основу программного комплекса составляет модуль Base, в котором содержатся специфичные для текущих платформы и компилятора макросы и функции, определения базовых типов, утилиты для отладки, утилиты для работы с памятью, интерфейсы для логгирования, шаблоны алгоритмов сортировки и поиска, а также различные структуры данных (массивы, связанные списки, очередь, циклический буфер, хэш таблица, множество битов, обобщённые множества), «умные» указатели, утилиты для работы со строками, цветами, GUID и FourCC, делегаты и реализация рефлексии, более подробно описанная в разделе 5.2. Кроме этого, в модуле Base расположена математическая библиотека MiniMath, в состав которой входят определения, макросы и функции для работы с целыми и вещественными числами, генераторы случайных чисел, функции хэширования и квантования, интерполяции, перевода индексов 1D ↔ 3D и работы с интервалами, различные типы ограничивающих объёмов и функции определения пересечений. В качестве базовых типов линейной алгебры MiniMath предоставляет вектора и матрицы. Vector4 — тип четырёхкомпонентного вектора, являющийся переопределением типа __m128, который может быть загружен в векторные регистры для использования с SIMD инструкциями. С помощью шаблонов Tuple2‹›, Tuple4‹›, Tuple4‹› определены «невекторизованные» векторные типы V2f, V3f, V4f, Int2, Int3, Int4, UInt2, UInt3 и т.д. M44f и M33f — 4×4 и 3×3 вещественные матрицы, M34f — «упакованная» матрица (с тремя единицами в последнем ряду/колонке). В данной работе используется правосторонняя система координат, в

которой ось Z направлена вверх, что соответствует обозначениям в подавляющем большинстве математических текстов, но отличается от принятых в OpenGL и Direct3D соглашений.

Модуль Core содержит «менеджер» памяти (для каждого типа выполняемых операций отведены специальные «кучи»), объектную модель, «менеджер» ресурсов, очередь заданий с приоритетами для выполнения в фоновом потоке, планировщик задач для их параллельного исполнения в рабочих потоках, функции для (де-)сериализации в бинарном формате и в виде дампа памяти, а также интерфейсы для обработки оконных событий и привязки к ним действий. Для уменьшения операций выделения и освобождения динамической памяти каждый поток имеет собственные «кучу» и буфер для форматирования строк.

Модуль Driver содержит функции для создания окон и обработки оконных событий, которые реализованы на основе SDL.

Модуль Graphics содержит абстракцию низкоуровневого графического API, названную LLGL (Low Level Graphics Layer/Library), которая подробно описана в разделе 5.3.

Модуль Renderer содержит высокоуровневую графическую библиотеку, содержащую готовые функции для отображения объектов виртуального мира.

Модуль Scripting содержит функциональность для работы со встраиваемым (скриптовым) языком Lua.

Модуль Meshok содержит функциональность для работы с полигональными сетками: загрузка полигональных моделей из различных форматов с помощью Assimp и конверсия в различные форматы вершин/индексов, сохранение моделей в формат Wavefront .obj для отладки, иерархии для ускорения поиска пересечений лучей с моделями (октодерево, BVH, BIH, *k*D-дерево, BSP-дерево). Кроме этого, в Meshok содержится библиотека для работы с кубическими решётками (фрагменты которой приведены в Приложении Б), библиотека знакоопределённых полей расстояния (SDF), трассировщик лучей, различные солверы для нахождения позиций острых вершин в двойственных методах триангуляции, методы триангуляции (MC, DC, DMC и их адаптивные варианты и модификации), реализации предложенных во предыдущих главах алгоритмов триангуляции (адаптивные алгоритмы триангуляции на основе линейных октодеревьев для использования с методами ADC/ADMC, расширенный алгоритм дуальных контуров), алгоритмы упрощения и прореживания (десимации) треугольных сеток на основе операций объединения вершин [154,43] и удаления рёбер [41,42], функции для работы с лучевым представлением (см. раздел 3.4), генераторы шума, функции для построения пространственных индексов и работы с кривыми заполнения пространства (кривые Мортона и Пеано-Гильберта).

Модуль DemoFramework содержит каркас демонстрационного приложения, функции для генерации пользовательского интерфейса на основе рефлексии и метаданных и его отрисовки с помощью ImGui, различные классы видовых камер.

Модуль Developer содержит функциональность, требующуюся только на этапе разработки приложения: подготовка различных данных и отслеживание изменений в файловой системе для перекомпиляции и перезагрузки ресурсов.

Модуль TxTSupport реализует текстовый формат представления данных, аналогичный JSON и названный SON (Simple Object Notation), а также функции для (де-)сериализации объектов в формат SON с помощью рефлексии и работу с текстовыми конфигурационными файлами.

Модуль VoxelTerrain осуществляет работу с воксельными ландшафтами и зависит от модулей Base и Meshok. Остальные модули требуются только для работы демонстрационных программ. В модуле VoxelTerrain содержится базовая функциональность по созданию, сохранению, загрузке, редактированию и триангуляции воксельного ландшафта, построенная на основе функций, реализованных в модуле Meshok. VoxelTerrain использует различные C++-интерфейсы, позволяющие пользователю расширить функциональность модуля без необходимости модификикации программного кода. Интерфейсы были спроектированы для удобства использования и, по возможности, таким образом, чтобы издержки на вызовы виртуальных методов классов были незначительны по сравнению со временем, затрачиваемым на выполнение операции.

Новые ландшафты могут быть сгенерированы из неявного описания посредством функций расстояния (SDF) и их комбинаций, которые представлены интерфейсом Isosurface в пространстве имён SDF в модуле Meshok. От интерфейса Isosurface наследуются классы, представляющие различные объекты (плоскость, сфера, параллелепипед, конус, цилиндр, тор, синусоидальные волны, гироид, изоповерхности «седло» и «сердце», бутылка Клейна и т.д.), операции переноса и поворота, теоретико-множественные операции (булевы операции из таблицы 3.1, операция гладкого сопряжения), а также класс HeightMap, представляющий знакоопределённое поле расстояний карты высот. Для отладки алгоритмов триангуляции реализована возможность визуализации неявно-заданных поверхностей в интерактивном режиме методом бросания лучей (ray casting) и трассировки сфер (sphere tracing) (рисунок 5.2).

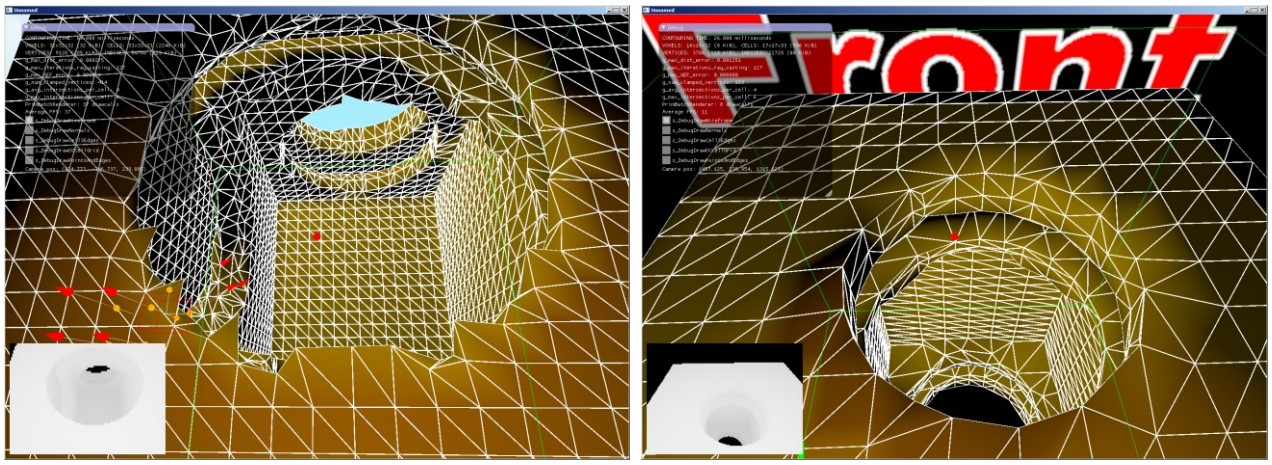


Рисунок 5.2 – Визуальная отладка алгоритма триангуляции объёмных данных в виде SDF. В нижнем левом углу показан результат лучевой трассировки неявной заданной поверхности на CPU.

Для триангуляции воксельных данных используются предложенные во второй главе двойственные методы, реализованные в модуле Meshok. Интерфейс AVolumeSampler служит для определения знаков (значений «внутри»/«снаружи») вершин ячеек и нахождения Эрмитовых данных: точек пересечения активных рёбер ячеек с изоповерхностью, а также единичных нормалей к поверхности в этих точках. От AVolumeSampler наследуются классы для получения Эрмитовых данных из функции расстояния со знаком (посредством интерфейса Isosurface), из воксельной решётки с Эрмитовыми данными, лучевого представления с нормалями и BSP-дерева (для вокселизации полигональных моделей).

Для нахождения позиций острых вершин все двойственные методы триангуляции используют интерфейс QEF_Solver, входными данными для которого являются Эрмитовы данные (список позиций и нормалей), AABB ячейки, значение допустимой погрешности и максимальное количество итераций солвера. На выходе солвер возвращает оптимальную позицию двойственной вершины (минимизирующую сумму квадратов расстояний до касательных плоскостей к изоповерхности, образованных Эрмитовыми данными на рёбрах ячейки), норму невязки (показывающую величину ошибки) и тип вершины (лежит ли вершина в остром углу, на остром ребре или на гладкой поверхности?).

В Meshok реализованы следующие типы солверов («решателей»):

1) QEF_Solver_Bloxel помещает вершину в центр ячейки, что может быть использовано для рендеринга «блочных» (bloxel, boxel, cuberille) воксельных ландшафтов, как в Minecraft;

2) QEF_Solver_Simple помещает вершину в центр масс точек пересечения, как в методе поверхностных сеток (Surface Nets) [30], и может использоваться для реконструкции гладких поверхностей;

3) QEF_Solver_ParticleBased реализует простой итеративный метод [57], в котором вершина ячейки рассматривается, как материальная точка, на которую действуют нелинейные упругие силы; обычно приводит к созданию острых рёбер с неровными краями;

4) QEF_Solver_Direct_ATA решает задачу наименьших квадратов путём решения нормальных уравнений, что очень дёшево с вычислительной точки зрения, но совершенно не подходит для триангуляции зашумленных данных;

5) QEF_Solver_QR решает задачу наименьших квадратов методом QR-разложения, используя реализацию оригинальных авторов [31,46];

6) QEF_Solver_SVD_Eigen решает задачу наименьших квадратов методом сингулярного разложения с помощью математической библиотеки Eigen; обеспечивает наиболее качественную реконструкцию острых вершин и рёбер.

Для сохранения и загрузки воксельных данных служит интерфейс AVoxelDatabase. От него наследуются классы Database_RAM, Database_Disk и Database_LMDB, реализующие хранилище воксельных данных как файл в оперативной памяти, множество отдельных файлов или монолитный файл на жёстком диске.

Редактируемые воксельные данные каждого блока могут храниться в любом из форматов, описанных в третьей главе:

1) Знакоопределённое поле расстояний с градиентами (Point-Sampled Signed Distance Field), представленное классом PSDF;

2) Воксельная решётка с Эрмитовыми данными (Voxel Grid with Hermite data), представленная классом VoxelGrid_with_HermiteData;

3) Лучевое представление с нормалями (Triple Ray-Rep with Normals), представленное классом RayRep;

4) Точечное представление с неявной связностью (Points with Implicit Connectivity), представленное классом Voxels_with_Points.

Разрешение каждого блока не фиксировано на этапе компиляции, а задаётся при создании нового ландшафта (и хранится в метаданных каждого блока), что позволяет экспериментировать с различными разрешениями воксельной решётки. (Максимальное разрешение блока составляет 512^3 ячеек, потому что 30-битные коды Мортона ограничивают разрешение октодерева 1024 ячейками,

а верхний уровень октодерева служит для бесшовного соединения восьми смежных блоков.) Булевы операции над воксельными данными (например, вычитание сфер и параллелепипедов, объединение с полигональной моделью) также реализованы единым способом — через общий интерфейс AVolumeSampler. Это значительно снижает производительность, поскольку компилятор не может оптимизировать операции перевода 3D↔1D индексов и «встроить» (inline) виртуальные функции, но повышает расширяемость, модульность и поддерживаемость кода.

В комплексе реализованы следующие операции редактирования воксельных ландшафтов:

- «вычитание» из ландшафта полигональной модели или неявно-заданной области (путём выполнения булевых операций разности);
- заполнение («закраска») указанной области заданным материалом (новый материал применяется только к сплошным участкам ландшафта);
- «добавление» к ландшафту полигональной модели или неявно-заданной области с установкой заданного материала в сплошных областях ландшафта (с помощью выполнения булевых операций объединения);
- «добавление» к ландшафту полигональной модели или неявно-заданной области без перезаписи материалов сплошных областей ландшафта.

Также реализована возможность масштабирования и изменения положения объектов при помещении их на ландшафт посредством мыши.

Для работы с воксельными ландшафтами в комплексе реализованы классы ChunkedWorld, ClipMapWorld, и OctreeWorld, которые наследуются от WorldBase. В ChunkedWorld камеру окружает трёхмерный массив из кубических блоков одинакового размера. Каждый блок может иметь до четырёх уровней детализации (по аналогии с Geometrical MipMapping, использующимся для рендеринга карт высот). При движении камеры трёхмерная «матрица» блоков обновляется с помощью тороидального доступа, данные старых блоков выгружаются, а данные новых блоков подгружаются (или генерируются и сохраняются в базе данных). Для визуализации больших ландшафтов подходят ClipMapWorld и OctreeWorld, в которых сцена соответственно представлена набором вложенных кубов или организована в октодерево. Разработанный комплекс поддерживает до 16 уровней детализации, однако на больших сценах теряется точность чисел типа float, что выражается в появлении ребристых поверхностей, «грубых» нормалей и артефактов, связанных с перекрытием полигонов (Z-fighting).

Интерфейс AJobSystem служит для регистрации независимых между собой задач для выполнения в рабочих потоках и содержит функции AddJob() и WaitForAll(), с помощью которых

пользователь может добавить новую задачу или подождать завершения всех добавленных задач. В каждой задаче хранятся буфер данных и указатель на функцию. Операции процедурной генерации и редактирования воксельного ландшафта реализованы как отдельные задачи. В JobSystem_Threaded задачи помещаются в очередь, из которой рабочие потоки в произвольном и непредсказуемом порядке выбирают задачи для исполнения. Для избежания операций с динамической памятью очередь имеет фиксированный размер, поэтому при её переполнении задачи начинают выполняться в главном потоке. Для отладки задач рекомендуется использовать класс JobSystem_Serial, в котором задачи выполняются последовательно в главном потоке приложения.

5.2 Распараллеливание при помощи многопоточности

Для обеспечения интерактивности и повышения производительности все операции создания, редактирования и триангуляции блоков целесообразно выполнять в отдельных потоках исполнения, параллельных основному потоку и не блокирующих его. Многопоточность не только повышает производительность, но и становится необходимым условием плавной и быстрой работы всей системы.

В текущей реализации при старте программы создаётся несколько рабочих потоков и один «фоновый» поток. Каждый поток имеет собственную, локальную «кучу» (heap) для оптимизации работы с динамической памятью. Для операций, предусматривающих выделение и освобождение памяти в разных потоках, используется глобальный аллокатор на основе кольцевого буфера (ring buffer).

В рабочих потоках выполняются обычно короткие по продолжительности, независимые друг от друга и интенсивные с вычислительной точки зрения задачи, такие как процедурная генерация воксельных данных, булевы операции, а также задачи, связанные с подготовкой текущего кадра. Фоновый поток служит для выполнения длительных по продолжительности (занимающих по времени более одного кадра) задач, таких как фоновая загрузка и сохранение данных и триангуляция изменившихся блоков (если между ними существуют зависимости).

Параллельная триангуляция независимых блоков. Предложенный в разделе 4.3 расширенный алгоритм дуальных контуров позволяет обрабатывать каждый блок воксельного ландшафта отдельно, независимо от соседних блоков. Для каждого затронутого при редактировании блока создаётся задача для параллельного исполнения в рабочем потоке. В задаче

загружаются воксельные данные блока, выполняются операции редактирования и триангуляции и затем происходит сохранение блока в базе данных. После этого идентификатор блока и построенная треугольная сетка передаются в главный поток для замены старой сетки блока. После выполнения булевых операций вычитания блок может стать полностью пустым, тогда для него не будет создано новой треугольной сетки.

Благодаря распараллеливанию процессов редактирования и триангуляции ландшафта достигается почти мгновенное отображение изменений.

Параллельная триангуляция блоков с зависимостями. Предложенный способ бесшовного соединения с помощью адаптивной триангуляции создаёт зависимости между смежными блоками ландшафта, которые ограничивают возможности распараллеливания. В силу особенности двойственных методов триангуляции для сохранения бесшовного соединения изменившегося блока с соседними блоками необходимо также перестроить треугольные сетки некоторых смежных к нему блоков (см. раздел 4.3).

Задачи триангуляции с зависимостями между блоками необходимо либо последовательно выполнять в фоновом потоке, либо отложить до конца кадра, расставить зависимости (как блоки будут соединяться друг с другом и каким блокам будут принадлежать швы?) и запустить выполнение задач в рабочих потоках (новые задачи не могут быть созданы, пока не завершатся текущие).

В первом случае при модификации каждого блока его уникальный адрес и адреса его минимальных соседей добавляются в массив *Dirty_Chunks*. В каждом кадре *Dirty_Chunks* сортируется по координатам блоков в порядке возрастания, из отсортированного массива удаляются дубликаты, и в очередь задач в фоновом потоке помещаются запросы на триангуляцию блоков из *Dirty_Chunks*. В фоновом потоке выполняется последовательная триангуляция блоков, вычисляются повершинные нормали и ограничивающие параллелепипеды (AABB) созданных треугольных сеток. Выполненные задачи триангуляции «финализируются» в главном потоке, создавая или обновляя динамические вершинные и индексные буферы, используемые для рендеринга ландшафта.

При небольших модификациях ландшафта и/или медленной скорости передвижения наблюдателя последовательная триангуляция блоков не является критичной проблемой, поскольку триангуляция каждого блока занимает не более 2-3 миллисекунд. После модификации большого количества блоков и/или при высокой скорости движения камеры наблюдается процесс их

последовательной «перестройки» для закрытия швов, и в течение короткого времени становятся видимыми пробелы и разрывы между смежными блоками.

Для обеспечения возможности распараллеливания процесса триангуляции блоков необходимо дублировать данные приграничных ячеек, использовать методы триангуляции, свободные от inter-cell dependency (например, MC и CMS), или комбинацию прямых и двойственных методов для триангуляции ячеек на границах блоков, как в предложенном ранее расширенном алгоритме дуальных контуров (см. подраздел 4.3.1). В Upvoid Engine [24] для триангуляции ландшафта используется алгоритм CMS [33], что позволяет обрабатывать чанки полностью отдельно и независимо друг от друга.

5.3 Реализация механизма рефлексии встроенными средствами языка C++

При разработке приложений часто требуется написание больших объемов однотипного кода для обработки данных на основе информации об их типах. Например, сохранение, загрузка и передача данных в различных форматах по каналам связи (сериализация и десериализация), привязка свойств и методов классов к скриптовым языкам, написание редактора данных. Одним из методов решения подобных задач является кодогенерация — автоматическое создание кода, ориентированного на решение конкретной проблемы. Другим способом является применение рефлексии или интроспекции — механизма получения информации (метаданных) о структуре программы и использующихся в ней типах данных. В сравнении с кодогенерацией использование рефлексии, как правило, требует меньших трудозатрат на реализацию и не замедляет сборки программы.

На момент написания работы в языке C++ все ещё отсутствует полноценная поддержка рефлексии, поэтому на практике рефлексию в C++ реализовывают с помощью таких способов, как разбор исходного кода и автоматическая генерация программного кода с реализацией необходимого функционала, извлечение метаинформации из сгенерированных компилятором файлов, ручное описание метаданных во внешних файлах на DSL, или объявление метаданных в исходном коде программы с помощью специальной разметки. За исключением метода ручной разметки встроенными средствами языка, такие способы усложняют процесс сборки проекта и обычно являются трудоёмкими в реализации. Количество рутинных операций при ручном описании метаданных может быть сведено к минимуму при использовании таких средств языка C++, как шаблоны и макросы. В процессе исследования была разработана собственная реализация

механизмов рефлексии встроенными средствами языка [156], построенная на шаблонах и макросах и обладающая следующими достоинствами:

– минимальный объём кода для ручного объявления метаданных;

– минимальный объём кода в заголовочных файлах;

– отсутствие операций динамических выделений памяти;

– минимальное потребление памяти и высокая производительность;

– высокая скорость динамической идентификации типа данных (RTTI);

– не использует встроенного в C++ механизма RTTI;

– возможность отключения рефлексии в финальной, «релизной» сборке;

– возможность рефлексии сторонних POD-структур без модификации кода.

Текущая реализация рефлексии имеет следующие ограничения:

– не поддерживаются сторонние сложные типы данных (STL-контейнеры);

– не поддерживаются битовые поля;

– не поддерживаются множественное и виртуальное наследование;

– не поддерживаются динамически конструируемые типы.

Данные ограничения не являются существенными в данной работе, поскольку в коде используются только контейнеры собственной разработки (из STL применяются только шаблонные алгоритмы), битовые поля заменяются именованными флагами, и используется ограниченное подмножество языка C++.

Реализация рефлексии занимает более 4600 строк кода, не считая кода в шаблонных контейнерах, поэтому рассмотреть её полностью не представляется возможным. Класс MetaType — базовый класс для всех типов метаданных, содержит имя метатипа (указатель на статическую строку), его уникальный идентификатор (32-битный хэш от имени типа, статически вычисленный на этапе компиляции), его размер и выравнивание. От MetaType наследуются классы BuiltInType (для встроенных, примитивных типов данных: char, I32, U32, float, double, bool и т.д.), MetaClass (для структур и классов), MetaPointer (для любых указателей), MetaArray (для любых массивов), MetaEnum (для перечислений), MetaFlags (для битовых флагов на основе перечисления), MetaString (для текстовых строк), ClassID_Type (для уникальных идентификаторов классов, которые сериализуются в текстовые форматы как строки, а в бинарные — как 32-битные хэши), AssetID_Type (для уникальных идентификаторов ресурсов).

Метаданные хранятся в статических массивах и инициализируются на этапе компиляции. Тип каждого поля в структурах определяется с помощью частичной специализации шаблонов.

Структуры и классы, не содержащие виртуальных методов, должны наследоваться от базового класса CStruct. Структуры и классы с виртуальными методами должны наследоваться от базового класса AObject, предоставляющего функции для быстрой RTTI.

В данной работе рефлексия используется для (де-)сериализации данных в различные форматы, автоматической генерации пользовательского интерфейса и отладки программы на основе верификации содержимого памяти. Сериализация POD-структур является тривиальной задачей, поскольку в программе известны смещение и размер каждого поля в структуре. Для сериализации множеств структур и объектов, содержащих указатели друг на друга (граф «живых» объектов), необходима разработка объектной модели, определяющей семантику владения и контроль жизни объектов. Например, в C++ объекты могут «жить» в статической, динамической и стековой памяти, иметь различные стратегии управления. Для корректной и эффективной сериализации и загрузки множества объектов со взаимными ссылками неполиморфные объекты одинакового типа сгруппированы в массивы, представленные классом ObjectList, которые, в свою очередь, организованы в связанный список внутри Clump. Для оптимизации работы с динамической памятью и повышения локальности доступа удалённые объекты помещаются в свободные списки (address-ordered free lists) для повторного использования. Память, выделенная для хранения объектов, освобождается только тогда, когда уничтожается их родительский Clump.

С помощью рефлексии реализована возможность (де-)сериализации данных в кастомный текстовый формат (в виде SON), в кастомный бинарный формат и в виде дампа памяти (memory image saving, in-place loading).

5.4 Описание графического ядра

В основе подсистемы визуализации в демонстрационных приложениях лежит собственное графическое ядро («Testbed»), обеспечивающее решение всех необходимых задач. Разработанный программный комплекс, в частности, поддерживает текстурные массивы (texture arrays) — удобное средство для реализации системы материалов для рендеринга воксельных ландшафтов [12,10].

Абстрагирование низкоуровнего графического интерфейса. Поскольку персональные компьютеры отличаются большим многообразием аппаратного обеспечения, то работа с графическим оборудованием происходит посредством драйвера соответствующего устройства и прикладных программных интерфейсов (Application Programming Interface, API), которые предоставляют стандартные графические интерфейсы (Graphics API, GAPI), такие как Direct3D и

OpenGL, независимые от версии установленной операционной системы, конкретного производителя и модели видеопроцессора. На практике разработчики, как правило, создают собственные слои абстракции, «сглаживающие» разницу между различными API и упрощающие написание кода приложения. На наш взгляд, хорошим примером абстрагирования GAPI является кроссплатформенная графическая библиотека bgfx[16], которая активно разрабатывается с 2012 года. На момент написания работы, в bgfx весьма эффективно реализована поддержка Direct3D 9, Direct3D 11, Direct3D 12, Metal, OpenGL 2.1, OpenGL 3.1+, OpenGL ES 2, OpenGL ES 3.1, WebGL 1.0 и WebGL 2.0. При этом весь наружный интерфейс библиотеки размещён в одном небольшом заголовочном файле, который является самодостаточным и остаётся без изменений при сборке клиентского приложения на всех целевых платформах. Однако, со времени публикации обзора [157] в bgfx до сих пор отсутствует поддержка текстурных массивов (возможно, для поддержки совместимости со старыми графическими API).

В данной работе, аналогично bgfx, был выбран достаточно низкий уровень абстракции над графическим API, оперирующий такими базовыми понятиями, как графические ресурсы (буферы, текстуры, шейдеры, программы), объекты состояний рендеринга и форматы вершин, для доступа к которым пользователь использует «непрозрачные» (opaque) дескрипторы (16- и 8-битные индексы или указатели, «завёрнутые» в структуры для обеспечения типовой безопасности):

```
// Resource handles
mxDECLARE_8BIT_HANDLE( HDepthStencilState );
mxDECLARE_8BIT_HANDLE( HRasterizerState );
mxDECLARE_16BIT_HANDLE( HSamplerState );
mxDECLARE_8BIT_HANDLE( HBlendState );
mxDECLARE_8BIT_HANDLE( HInputLayout );
mxDECLARE_8BIT_HANDLE( HColorTarget );
mxDECLARE_8BIT_HANDLE( HDepthTarget );
mxDECLARE_16BIT_HANDLE( HTexture );
mxDECLARE_16BIT_HANDLE( HBuffer );// generic buffer handle ( e.g. vertex, index, uniform )
mxDECLARE_16BIT_HANDLE( HShader );// handle to a shader object ( e.g. vertex, pixel )
mxDECLARE_16BIT_HANDLE( HProgram );// handle to a program object
// shader resource is everything that can be sampled in a shader
mxDECLARE_16BIT_HANDLE( HShaderInput );
// Direct3D's Unordered Access Views or OpenGL's Indirect Buffers
mxDECLARE_16BIT_HANDLE( HShaderOutput );
```

Дескрипторы занимают меньше места, чем указатели, что используется для составления компактных ключей для сортировки по различным критериям, и позволяют реализовать командный буфер, формат которого не зависит от GAPI.

В командный буфер записываются действия, которые нужно выполнить в текущем кадре: создание, обновление, привязка и удаление ресурсов, отрисовка геометрии или вызов

[16] https://github.com/bkaradzic/bgfx

вычислительного шейдера, а также установка маркеров для профилирования и отладки. Для построения каждого кадра изображения пользователь формирует списки команд, каждый из которых заканчивается командой отрисовки. Все команды имеют переменную длину, а каждая команда начинается заголовком, в котором хранится её тип. Каждая команда отрисовки, или запрос на отрисовку (draw call), содержит все основные данные, необходимые для отрисовки геометрии: набор дескрипторов (состояний рендеринга, вершинных и индексного буферов, шейдерной программы и её параметров), диапазон и формат вершинных и индексного буферов, топология примитивов и различные флаги. Ниже приведено описание команды для отрисовки геометрии:

```
/// NOTE: all commands must be at least 4-byte aligned
struct Cmd_Base {
	U32	header; //!< lowest 'CMD_NUM_BITS' bits = command type
};

/// non-programmable (configurable) render states
union RenderStates {
  struct {
    HBlendState		blendState;
    HRasterizerState	rasterizerState;
    HDepthStencilState	depthStencilState;
    U8			stencilReference;
  };
  U32  asInt;
};// 4 bytes

/// contains (almost) everything needed for issuing a draw call
struct Cmd_Draw : Cmd_Base {
  RenderState		states;//!< non-programmable, configurable (FFP) states
  U32			inputs;//!< vertex format, index buffer format, primitive topology
  HProgram		program; //!< {VS+HS+DS+GS+PS+CS} shader bundle
  HBuffer		IB;
  HBuffer		VB[ LLGL_MAX_VERTEX_STREAMS ];
  U32		baseVertex;	//!< index of the first vertex
  U32		vertexCount;//!< number of vertices
  U32		startIndex;	//!< offset of the first index
  U32		indexCount;	//!< number of indices
};
```

Для установки шейдерных параметров (константных буферов, сэмплеров, текстур и буферов для записи) в командный буфер записываются команды в виде машинных слов, составленных из типа команды (например, CMD_SET_CBUFFER, CMD_SET_SAMPLER, CMD_SET_UAV), дескриптора соответствующего графического ресурса (например, HBuffer, HSamplerState, HShaderInput и HShaderOutput) и номера регистра, к которому требуется привязать ресурс. HShaderInput служит для привязки различных ресурсов для чтения в шейдере и содержит тип ресурса (буфера, текстуры, цели рендеринга, буфера глубины) и его индекс, полученный из дескриптора ресурса. HShaderOutput служит для привязки буферов для записи в вычислительном шейдере (compute shader) и содержит тип ресурса (буфера, текстуры или цели

рендеринга) и его индекс. Ниже приведены функции создания и распаковки идентификаторов HShaderInput и HShaderOutput:

```
enum EShaderResourceType {
  SRT_Buffer,
  SRT_Texture,
  SRT_ColorSurface,
  SRT_DepthSurface,
  SRT_UnorderedAccess,
  SRT_NumBits = 3
};
/// In front-end:
static inline HShaderInput MakeResourceHandle( EShaderResourceType type, U16 index ) {
  const HShaderInput handle = { (index << SRT_NumBits) | type };
  return handle;
}
static inline HShaderOutput MakeUAVHandle( EShaderResourceType type, U16 index ) {
  const HShaderOutput handle = { (index << SRT_NumBits) | type };
  return handle;
}

/// In Direct3D 11 rendering back-end:
extern BackendD3D11* bed;

static inline ID3D11ShaderResourceView* GetResourceByHandle( HShaderInput handle ) {
  mxASSERT( handle.IsValid() );
  ID3D11ShaderResourceView* pSRV = nil;
  const U32 type = handle.id & ((1u << SRT_NumBits)-1u);
  const U32 index = ((U32)handle.id >> SRT_NumBits);
  if( type == SRT_Buffer ) {
    pSRV = bed->buffers[ index ].m_SRV;
  } else if( type == SRT_Texture ) {
    pSRV = bed->textures[ index ].m_SRV;
  } else if( type == SRT_ColorSurface ) {
    pSRV = bed->colorTargets[ index ].m_SRV;
  } else if( type == SRT_DepthSurface ) {
    pSRV = bed->depthTargets[ index ].m_SRV;
  }
  mxASSERT_PTR(pSRV);
  return pSRV;
}

static inline ID3D11UnorderedAccessView* GetUAVByHandle( HShaderOutput handle ) {
  mxASSERT( handle.IsValid() );
  ID3D11UnorderedAccessView* pUAV = nil;
  const UINT type = handle.id & ((1 << SRT_NumBits)-1);
  const UINT index = ((UINT)handle.id >> SRT_NumBits);
  if( type == SRT_Buffer ) {
    pUAV = bed->buffers[ index ].m_UAV;
  } else if( type == SRT_Texture ) {
    pUAV = bed->textures[ index ].m_UAV;
  } else if( type == SRT_ColorSurface ) {
    pUAV = bed->colorTargets[ index ].m_UAV;
  }
  mxASSERT_PTR(pUAV);
  return pUAV;
}
```

Для обновления ресурсов (буферов, текстур) служат специальные команды, содержащие указатель на память, из которой в ресурс будут копироваться новые данные, область ресурса для

записи (например, TextureUpdateRegion) и указатель на функцию для освобождения памяти, который выставляется в нуль, если память была выделена в командном буфере. Например, данные для установки шейдерных констант, размер которых невелик (обычно несколько сотен килобайт, и никогда не превышает 64 KiB) записываются в командный буфер непосредственно после команды обновления соответствующего константного буфера (по выровненному вперёд на 16 байт адресу). Большие по объёму данные (например, обновление динамических вершинных и индексных буферов, текстур) передаются по ссылке, и после обновления ресурса высвобождаются. Для временного хранения таких данных используется специальный аллокатор в виде циклического буфера.

Командный буфер представлен классом RenderContext, который позволяет сортировать «цепочки» команд по различным параметрам (глобальный порядок отрисовки, шейдерная программа, номер материала, расстояние до камеры и т.д.):

```
/// Render [Context|Queue] or Command [Buffer|List|Queue]:
/// a (sortable) collection of rendering operations that make up a full frame.
/// Separate command lists can be built in different threads.
struct RenderContext : NonCopyable {
  /// Marks the beginning of a series of commands.
  U32  GetCurrent();  // returns the current 'write/put' pointer
  /// returns a pointer to the space allocated in the command buffer
  ERet Allocate(
    void **_buffer,
    const U32 _size,
    const U32 _alignMask = 15 // 16-byte aligned by default
  );
  /// Inserts a command chain.
  ERet SubmitCommands(
    const U32 _start, //!< the offset of the first command
    const U64 _sortKey
  );
public_internal:
  /// represents a command chain
  struct Command {
    U64 sortKey;//!< sortKey key (scene pass, program, material hash, user-defined parameters)
    U32 start;  //!< byte offset of the command data in the command buffer
    U32 size;   //!< size of data allocated within the command buffer
  };
  TArray< Command >   m_keys; //!< keys for sorting commands
  char *       m_start;    //!< command buffer data
  U32          m_current;  //!< current write pointer, always increasing and at least 4-byte aligned
  U32          m_allocated;//!< high watermark
  char *       m_nameAndPadding;
};
```

Списки команд могут быть предоставлены пользователем в любом порядке, поскольку на низком уровне они будут отсортированы для выполнения в правильном порядке и наиболее оптимальным способом. Таким образом, от пользователя скрыта машина состояний графического API. Это значительно облегчает отладку, исключает возможность совершения ошибок, связанных с

неправильной установкой состояний рендеринга (state leaking), а также позволяет легко реализовать формирование запросов на отрисовку в нескольких потоках.

Графическое ядро разделено на фронт-энд (Front-End) и специфичный для каждого GAPI бэк-энд (Back-End), аналогично bgfx. Фронт-энд предоставляет компактный интерфейс для работы с графическим оборудованием. Вызовы его функций очень дешёвы, поскольку их задачей является только запись данных в командный буфер для бэк-энда. Бэк-энд сортирует списки команд и транслирует команды в вызовы низкоуровневого графического API, отсекая ненужные вызовы:

```
	// in BackendD3D11::Render() :
	D3D11State    currentState;  // initialized with default settings (as if
ID3D11DeviceContext::ClearState() were called)
	ID3D11DeviceContext * deviceContext;
	for( U32 iCommandList = 0; iCommandList < numCommandLists; iCommandList++ )
	{
	  const RenderContext::Command& commandList = sortedCommandLists[ iCommandList ];
	  const char* commandListStart = commandBufferStart + commandList.start;
	  const char* commandListEnd = commandListStart + commandList.size;
	  const char* currentCommand = commandListStart;
	  while( currentCommand < commandListEnd ) {
	    const U32 commandHeader = *(U32*) currentCommand;
	    const U32 commandType = ( commandHeader & ((1u << CMD_NUM_BITS) - 1u) );
	    switch( commandType ) {
	    case CMD_DRAW : {
	        const Cmd_Draw& cmd_Draw = *(Cmd_Draw*) currentCommand;
	        Execute_Draw( cmd_Draw, currentState, deviceContext );
	        currentCommand += sizeof(Cmd_Draw);
	      } break;
	    case CMD_SET_CBUFFER : {
	        U32 slot;
	        const HBuffer handle = { Cmd_BindResource::DecodeHeader( commandHeader, slot ) };
	        mxASSERT(slot < LLGL_MAX_BOUND_UNIFORM_BUFFERS);
	        if( currentState.shader.CBs[ slot ] != handle ) {
	          currentState.shader.dirtyCBs |= (1u << slot);
	          currentState.shader.CBs[ slot ] = handle;
	        }
	        currentCommand += sizeof(Cmd_BindResource); // sizeof(Cmd_BindResource) == 4 bytes
	      } break;
	    case CMD_UPDATE_BUFFER : {
	        const Cmd_UpdateBuffer& cmd_UpdateBuffer = *(Cmd_UpdateBuffer*) currentCommand;
	        const HBuffer bufferHandle = { commandHeader >> 16u };
	        UpdateBuffer( bufferHandle, cmd_UpdateBuffer.data, cmd_UpdateBuffer.size );
	        currentCommand += sizeof(Cmd_UpdateBuffer);
	        if( cmd_UpdateBuffer.deallocator ) {
	          cmd_UpdateBuffer.deallocator->Deallocate( cmd_UpdateBuffer.data );
	        } else {
	          currentCommand = mxALIGN16( currentCommand );
	          currentCommand += mxALIGN16( cmd_UpdateBuffer.size );
	        }
	      } break;
	      /* ... handle other commands ... */
	      mxSWITCH_DEFAULT_UNREACHABLE;
	    }//switch
	  }//For each command.
	}//For each command list.
	ResetState( deviceContext ); // mirrors D3D11State::Reset()
	swapChain->Present();
```

Для повышения производительности сортировка и выполнение команд происходят в отдельном потоке RenderThread параллельно главному потоку исполнения (передача данных потоку рендеринга происходит с помощью двойной буферизации). В рамках данной работы в основном используется графический интерфейс Direct3D 11 [158], но реализован и бэк-энд с поддержкой OpenGL 4.2+ [159].

Высокоуровневая графическая библиотека. На основе вышеописанного слоя абстракции GAPI построена графическая библиотека более высокого уровня, оперирующая такими понятиями, как камера, полигональная сетка («меш»), модель («инстанс меша») и материал. Для пространственного разбиения сцены используются регулярные решётки. Для быстрого отсечения невидимых объектов по пирамиде видимости ограничивающие оболочки (AABB) моделей и объектов, отбрасывающих тени, хранятся в отдельных массивах в SoA-формате. При этом одновременно проверяется четыре AABB на пересечение с пирамидой видимости:

```
/// Test 4 AABBs against the frustum.
/// Returns a 4-vector mask where 0xFFFFFFFF entries indicate which boxes intersect (or are inside) the frustum.
static Vector4 Test_4_AABBs_SSE2( const Frustum_SoA& _frustum,
  Vec4Arg0 centers_x, Vec4Arg0 centers_y, Vec4Arg0 centers_z,
  Vec4Arg4 extents_x, Vec4Arg5 extents_y, Vec4Arg5 extents_z )
{
  // Assume all 4 boxes are inside (or intersect) the frustum.
  Vector4       all_inside = g_MM_AllMask;       // {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF};
  //OPTIMIZE?: test the near plane with early exit (branch), the 4 remaining ones - branchless unrolled
  for( unsigned iPlane = 0; iPlane < 5; iPlane++ )
  {
    // Splat out the plane's normals and distances:
    const Vector4 plane_xxxx = _frustum.planes[ iPlane ].xxxx;
    const Vector4 plane_yyyy = _frustum.planes[ iPlane ].yyyy;
    const Vector4 plane_zzzz = _frustum.planes[ iPlane ].zzzz;
    const Vector4 plane_dddd = _frustum.planes[ iPlane ].dddd;
    const Vector4 plane_abs_xxxx = _frustum.abs_normals[ iPlane ].xxxx;
    const Vector4 plane_abs_yyyy = _frustum.abs_normals[ iPlane ].yyyy;
    const Vector4 plane_abs_zzzz = _frustum.abs_normals[ iPlane ].zzzz;
    const Vector4 tmp0 = V4_ADD( V4_MUL( centers_x, plane_xxxx ), V4_MUL( centers_y, plane_yyyy ) );
    const Vector4 tmp1 = V4_MAD( centers_z, plane_zzzz, plane_dddd );

    // Distances from the 4 boxes' centers to the planes:
    const Vector4 distances_to_planes = V4_ADD( tmp0, tmp1 );
    const Vector4 tmp2 = V4_ADD( V4_MUL( extents_x, plane_abs_xxxx ), V4_MUL( extents_y, plane_abs_yyyy ) );
    // Effective radii (half-size along the planes' normals) for the 4 boxes (always positive).
    const Vector4 effective_radii = V4_MAD( extents_z, plane_abs_zzzz, tmp2 );

    // This test assumes frustum planes face outward.
    //bool box_intersects_frustum = distance_from_center_to_plane <= effective_radius;
    const Vector4 inside_mask = V4_CMPLE( distances_to_plane, effective_radii );

    all_inside = V4_AND( all_inside, inside_mask );
  }//For each frustum plane.

  return all_inside;
}
```

Построение списка видимых объектов для главной камеры и добавление батчей на отрисовку сцены выполняется одновременно с рендерингом теней.

Поскольку реалистичный рендеринг не является ключевой целью данной работы, в высокоуровневой графической библиотеке была реализована лишь базовая функциональность,

требующаяся для визуализации «разрушаемых» воксельных ландшафтов: система материалов с поддержкой текстурных массивов (texture arrays), пул динамических вершинных и индексных буферов, отложенное освещение и затенение (deferred shading) с простой моделью освещения, базовая реализация каскадных теневых карт (Cascaded Shadow Maps), возможность отрисовки геометрии в каркасном режиме (wireframe mode), а также отрисовки отладочной геометрии (цветных линий и полигонов). Для гибкой настройки конвейер рендеринга может быть загружен из конфигурационного файла, который описывается в текстовом формате. Для ускорения процесса разработки все ресурсы (текстуры, шейдеры, материалы, модели) могут быть перезагружены «на лету». Специальное консольное приложение AssetCompiler осуществляет подготовку данных для их дальнейшего использования графической библиотекой, при этом ошибки при компиляции ресурсов сигнализируются звуковым сигналом.

5.5 Описание демонстрационных программ

Для экспериментального тестирования разработанных алгоритмов и методов был создан набор демонстрационных приложений. Все программы используют вышеописанную графическую библиотеку.

Приложение для тестирования интерактивных методов триангуляции

Для экспериментальной оценки эффективности различных методов триангуляции было разработано специальное приложение, содержащее детальные программные реализации следующих двойственных методов триангуляции:

1) (Uniform) Dual Contouring (DC) [31] — стандартный двойственный метод триангуляции на регулярной кубической решётке;

2) (Uniform) Dual Marching Cubes (DMC) [56] — аналогичен DC, но почти всегда генерирует развёртываемые полигональные сетки за счёт создания в каждой активной ячейке от одной до четырёх вершин;

3) (Standard) Adaptive Dual Contouring (ADC) [31,46] — оригинальный ADC на основе классических октодеревьев;

4) Adaptive Dual Contouring of Linear Octrees [72] — реализация предлагаемого подхода с триангуляцией линейных октодеревьев;

5) Adaptive Dual Marching Cubes (ADMC) [49] — реализация наиболее близкого к предложенному метода, реализующего адаптивный DMC на основе линейных октодеревьев;

6) Adaptive Dual Contouring of Linear Octrees with 2-manifold correction — реализация предложенного подхода с триангуляцией линейных октодеревьев, почти всегда генерирующего развёртываемые сетки (см. подраздел 2.4.3);

7) (Uniform) Dual Contouring with Boundary — расширенный алгоритм DC на регулярных решётках, решает проблему зависимостей между смежными блоками при построении бесшовной триангуляции созданием дополнительных вершин на рёбрах и гранях приграничных ячеек каждого блока, как в алгоритме CMS [33].

В приложении триангулируется лучевое представление (см. раздел 3.4), построенное из произвольных полигональных моделей или по неявной функции расстояния со знаком. Пример лучевого представления показан на рис. 5.3.

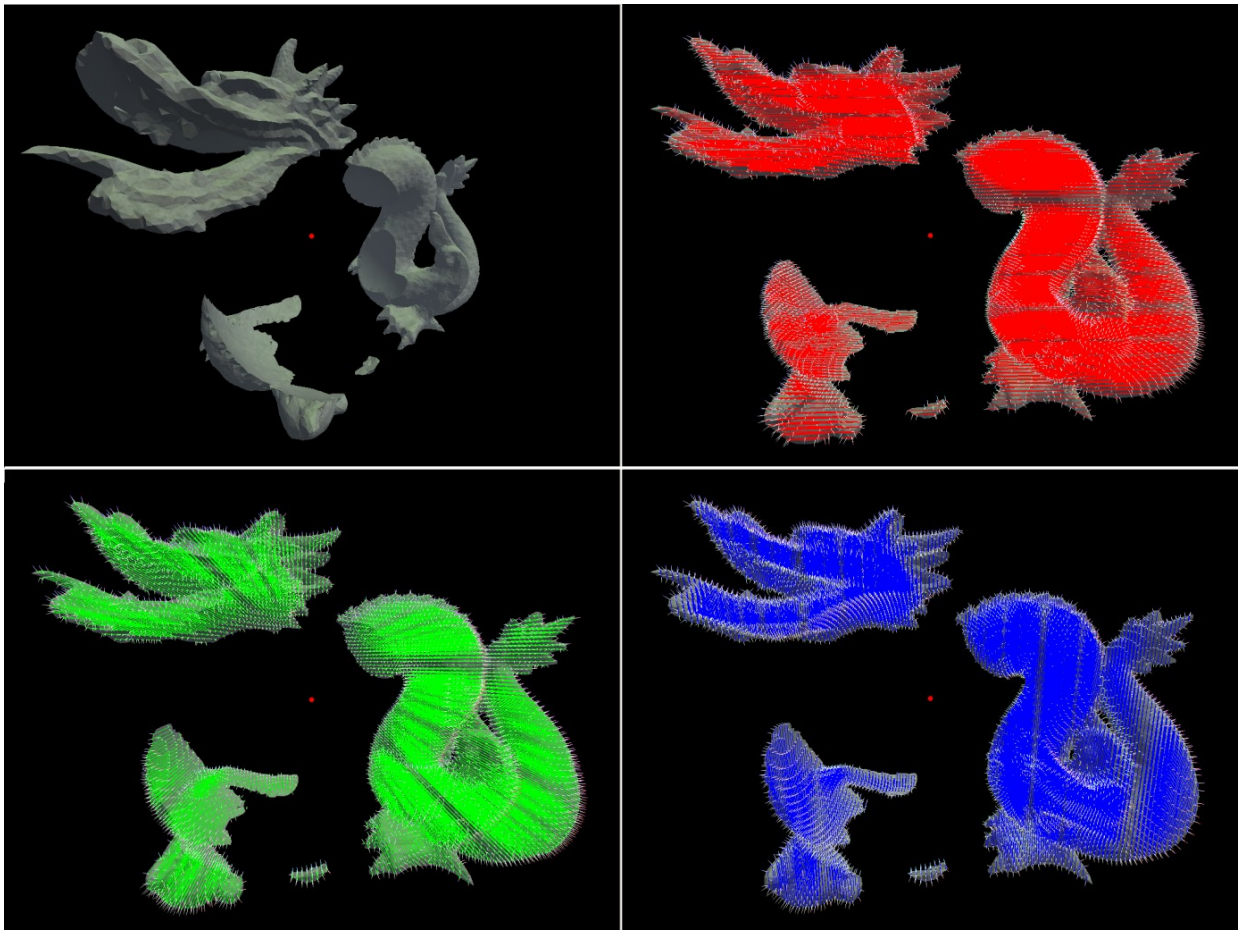


Рисунок 5.3 – Результат триангуляции и визуализация лучевого представления с разрешением 64^3 ячеек, полученного из модели Stanford Dragon, к которому были применены булевы операции вычитания.

Все реализации методов триангуляции на регулярных решётках используют оптимизацию, в которой в каждый момент времени в памяти хранятся только текущий и предыдущий слои ячеек. В адаптивных методах триангуляции октодерево упрощается до достижения заданной погрешности, без выполнения проверок сохранения исходной топологии. Реализации лучевого представления и методов триангуляции находятся в модуле Meshok.

Пользовательский интерфейс реализован с помощью библиотеки Dear ImGui[17], и бóльшая его часть генерируется автоматически с помощью рефлексии. Пользователю дается возможность выбрать любой из перечисленных алгоритм триангуляции и тонко настраивать его параметры.

Приложение обеспечивает выполнение следующих функций:

– построение лучевого представления по неявно заданной функции расстояния со знаком (SDF) или из загруженной с диска полигональной модели;

– сохранение в файл и загрузка построенного лучевого представления;

– выполнение над лучевым представлением и «кисти» (brush) в форме сферы, управляемой мышью, булевых операций разности и объединения;

– включение/выключение и настройка размеров «кисти», «прицела» и гизмо;

– триангуляция лучевого представления различными алгоритмами;

– генерация уровней детализации с помощью адаптивной триангуляции;

– упрощение построенной треугольной сетки путём удаления рёбер;

– тонкая настройка параметров алгоритма триангуляции: минимальный размер листового узла октодерева, максимальная глубина октодерева, стратегия нахождения острых вершин (стандартный, блочный, гладкий), стратегия ограничения позиций острых вершин (допускается ли выход вершин за границы ячейки и пороговое значение отклонения острой вершины от границ ячейки);

– возможность упрощения (редуцирования) построенной треугольной сетки методом удаления рёбер [41,42];

– измерение времени исполнения и объёма потребляемой памяти алгоритма в процессе триангуляции, а также свойств построенной треугольной сетки;

– визуализация лучевого представления с различными опциями: с отображением лучевых сегментов, позиций точек пересечения и нормалей к поверхности в этих точках;

[17] https://github.com/ocornut/imgui

– визуализация построенной треугольной сетки в нескольких режимах: с затенением, каркасном и полупрозрачном режиме (рис. 5.4).

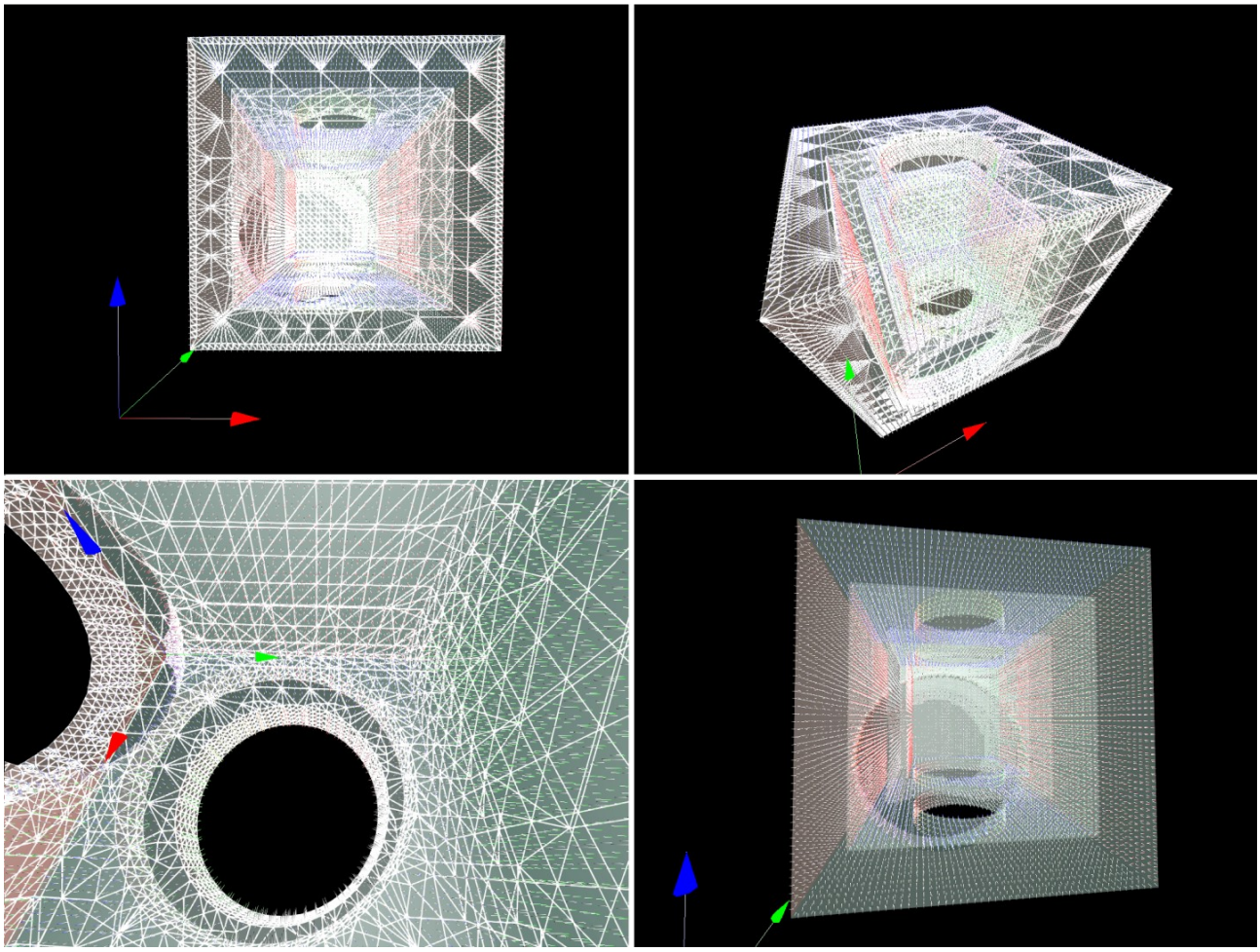


Рисунок 5.4 – Визуализация результатов адаптивной триангуляции неявно заданной поверхности в каркасном режиме и со включенной полупрозрачностью.

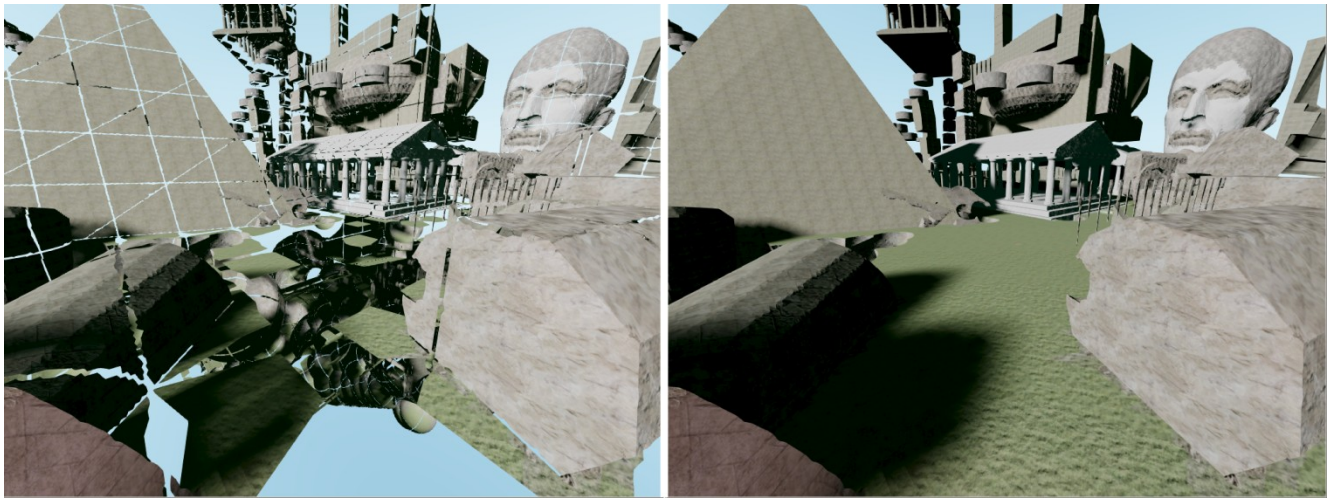
Приложения для тестирования методов моделирования и визуализации воксельных ландшафтов в интерактивном режиме

Для экспериментальной оценки эффективности разработанных методов представления, моделирования и визуализации воксельных ландшафтов в интерактивном режиме было проведено множество экспериментов.

На рисунке 5.5 показаны результаты редактирования воксельных ландшафтов, разбитых на одинаковые кубические блоки, как в методе GMM для карт высот. Ландшафт был построен из неявно-заданной функции, после чего к ландшафту применялись булевы операции и операции раскрашивания различными материалами.

Рисунок 5.5 – Примеры ландшафтов, смоделированных в интерактивном режиме.

На рисунке 5.6 показаны результаты бесшовной триангуляции тестовой сцены, содержащей острые рёбра и углы. С помощью булевых операций на сцену были помещены стандартные тестовые модели: пирамида, акрополис, Bolt, Fandisk, Max Planck и многие другие. Для триангуляции каждого блока использовался формат SLOG (см. раздел 3.6), разрешение каждого блока составило 32^3 ячеек, размер сцены — 16^3 блоков, общий объём воксельных данных на диске — ≈ 22 мегабайта, средний коэффициент сжатия воксельных данных превысил 26.



а) б)

Рисунок 5.6 – Бесшовная триангуляция разбитой на одинаковые блоки сложной сцены с наличием острых рёбер и углов:а) триангуляция блоков по отдельности приводит к появлению разрывов между смежными блоками;
б) результат бесшовной триангуляции предложенным в четвёртой главе методом.

На рисунке 5.7 показаны результаты моделирования и визуализации большого ландшафта, который использует октодерево в качестве иерархии уровней детализации.

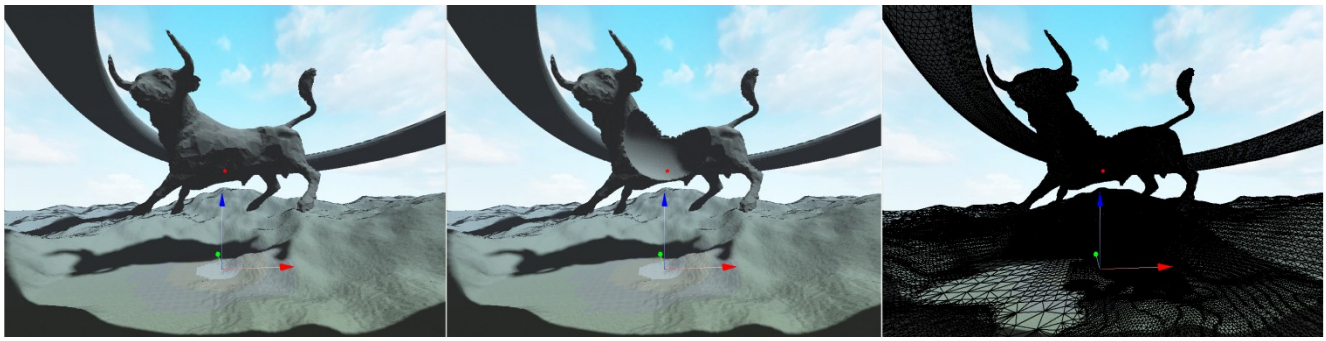


Рисунок 5.7 – Визуализация ландшафта с помощью метода вложенных кубов отсечения и разработанного метода бесшовной триангуляции.

На рисунке 5.8 показан пример ландшафта, который использует описанный в разделе 4.3 метод бесшовной триангуляции для сшивки дискретных уровней детализации.

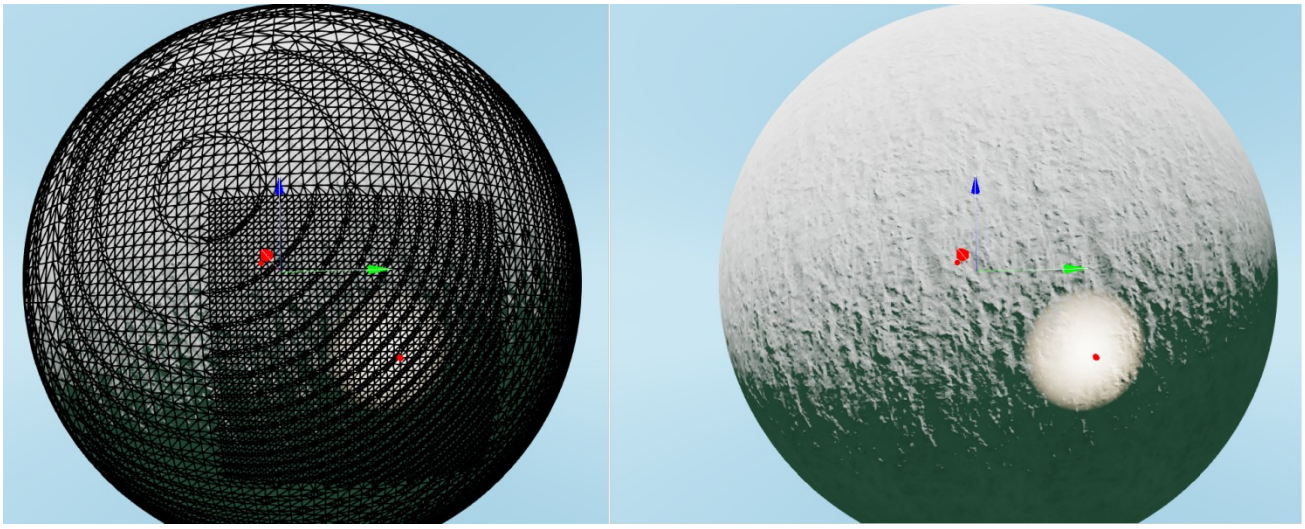


Рисунок 5.8 – Бесшовная триангуляция сферы с 3 уровнями детализации.

5.6 Основные выводы по пятой главе

Разработан программный комплекс, который обеспечивает визуализацию воксельных ландшафтов и взаимодействие с ними в виртуальном окружении на современном оборудовании.

Программный комплекс включает в себя графическую библиотеку, набор демонстрационных программ и инструменты для компиляции ресурсов.

Заключение

Главным результатом диссертационной работы является разработка совокупности алгоритмов и методов, направленных на совершенствование процесса проектирования систем виртуальной реальности за счёт расширения функциональных возможностей САПР и повышение эффективности процессов моделирования и визуализации воксельных ландшафтов.

Предложенный алгоритм триангуляции позволяет строить бесшовные треугольные сетки для интерактивной визуализации воксельных ландшафтов с различными уровнями детализации. Кроме того, используемый метод триангуляции способен восстанавливать острые рёбра и конические вершины поверхности, что может быть использовано для визуализации искусственных элементов ландшафта, таких как здания и технические объекты. Приложениями разработанных методов и алгоритмов могут выступать системы архитектурного проектирования, системы визуализации для тренажерных комплексов, системы виртуальной реальности, видеоигры с разрушаемым окружением и большими открытыми пространствами.

Разработан программный комплекс, который обеспечивает визуализацию воксельных ландшафтов и взаимодействие с ними в виртуальном окружении на современном оборудовании. Получено свидетельство о государственной регистрации программы для ЭВМ [160].

В диссертации были получены следующие основные результаты:

1) осуществлён системный анализ современных работ по моделированию и визуализации воксельных ландшафтов в интерактивном режиме в САПР ВР, выделены достоинства и недостатки существующих решений;

2) разработан алгоритм адаптивной триангуляции на основе линейных октодеревьев, обладающий более низким потреблением памяти, генерирующий когерентные треугольные сетки (со значением ACMR < 0.75) и позволяющий выполнять триангуляцию сверхбольших воксельных ландшафтов по частям;

3) исследованы способы представления и форматы хранения воксельных данных, подходящие для моделирования воксельных ландшафтов с острыми углами и обладающие различным спектром возможностей;

4) разработан метод для компактного хранения уровней детализации воксельного ландшафта с поддержкой его острых углов, различных материалов, возможностью генерации уровней детализации, высокой скоростью адаптивной триангуляции, сжатия и декомпрессии; в экспериментах средний коэффициент сжатия превысил 3;

5) разработан новый алгоритм для адаптации иерархии уровней детализации к позиции камеры наблюдателя, являющийся более простым в реализации и обладающий более низкой вычислительной сложностью по сравнению с традиционными рекурсивными алгоритмами;

6) разработан метод для бесшовной триангуляции сверхбольших воксельных ландшафтов, состоящих из блоков с различными уровнями детализации, который использует предложенные алгоритм адаптивной триангуляции и метод компактного хранения изоповерхности ландшафта;

7) разработан алгоритм расширенных дуальных контуров, позволяющий выполнять редактирование и триангуляцию каждого блока независимо от его соседей, в том числе с использованием параллельных вычислений;

8) разработан метод для визуализации сверхбольших изоповерхностей с помощью расширенного алгоритма дуальных контуров и геоморфинга, позволяющий визуализировать гигантские изоповерхности с плавной сменой уровней детализации без какой-либо предварительной подготовки;

9) спроектирован, реализован и апробирован прототип системы, реализующей разработанные алгоритмы и методы для интерактивных моделирования и визуализации воксельных ландшафтов в системах ВР;

10) экспериментально показана высокая эффективность разработанных алгоритмов и методов для моделирования и визуализации воксельных ландшафтов, в том числе показаны их высокое быстродействие и небольшие требования по размерам оперативной памяти.

В целом, разработанные методы, алгоритмы и программный комплекс позволяют усовершенствовать процесс проектирования систем ВР на этапах моделирования и визуализации воксельных ландшафтов расширением функциональных возможностей САПР и повышения эффективности процессов хранения и обработки проектной информации. Таким образом, в ходе выполнения диссертационной работы достигнута цель исследования и решены поставленные задачи.

В качестве перспектив дальнейшей работы можно выделить использование более совершенных способов представления и форматов хранения объёмных данных и алгоритмов их триангуляции для лучшей реконструкции мелких деталей, острых углов и рёбер поверхности, разработку методов генерации и незаметной смены уровней детализации ландшафта, распараллеливание и перенос некоторых задач на GPU, и поддержку более широкого класса операций геометрического моделирования в САПР систем виртуальной реальности.

Список используемых сокращений и условных обозначений

ВР — виртуальная реальность

САПР – система автоматизированного проектирования, система автоматизации проектных работ

API — Application Programming Interface (прикладной программный интерфейс)

TIN — Triangulated Irregular Network (нерегулярная треугольная сетка)

MC — Marching Cubes (маршрирующие, шагающие или движущиеся кубики)

EMC — Extended Marching Cubes (расширенные маршрирующие кубики)

DC — Dual Contouring (метод дуальных или двойственных контуров)

DMC — Dual Marching Cubes (дуальные или двойственные маршрирующие кубы)

CMS — Cubical Marching Squares (кубические маршрирующие квадраты)

SDF — Signed Distance Field (знакоопределённое поле расстояний)

CSG — Constructive Solid Geometry (конструктивная сплошная, твердотельная или блочная геометрия)

RLE — Run-Length Encoding (кодирование длин серий)

LoD — Level of Detail (уровень детализации)

DLoD — Discrete Levels of Detail (дискретные уровни детализации)

CLoD — Continuous Level of Detail (непрерывный уровень детализации)

AABB — Axis-Aligned Bounding Box (ограничивающий бокс или параллелепипед, выровненный по осям координат)

SIMD – Single Instruction, Multiple Data (одна инструкция, множество данных)

SSE – Streaming SIMD Extensions (потоковые SIMD-расширения)

CPU — Central Processing Unit (центральный процессор)

GPU — Graphics (Graphical) Processing Unit (графический процессор)

3D — Three-dimensional (трёхмерный)

2D — Two-dimensional (двухмерный или двумерный)

Список литературы

1. Авербух, В.Л. Развитие подходов к разработке специализированных систем компьютерной визуализации / В.Л. Авербух, М.О. Бахтерев, П.А. Васёв, Д.В. Манаков, И.С. Стародубцев // Труды Международной научной конференции GraphiCon 2015. – Протвино, 2015. – С. 17-21.

2. Авербух, Н.В. К вопросу о феномене присутствия в системах компьютерной визуализации на баз е виртуальной реальности // Труды Международной научной конференции GraphiCon 2018. – Томск, 2018. – С. 74-76.

3. Захарова, А.А. Роль компьютерной графики при проектировании основной образовательной программы по направлению «дизайн» / А. А. Захарова, Е. В. Вехтер, Ю. С. Ризен // ГРАФИКОН'2016 труды 26-й Международной научной конференции. — Нижний Новгород, 2016 . — С. 518-522.

4. Захарова, А.А. Структурный подход к визуализации данных / А.А. Захарова, А.В. Шкляр // Материалы XX Юбилейной Международной конференции по вычислительной механике и современным прикладным программным системам (ВМСППС'2017). — Москва, 2017. — С. 609-611.

5. Захарова, А.А. Построение многокомпонентных визуальных 3D-моделей с использованием разнородных источников информации на примере создания геологических моделей / А.А. Захарова, А.В. Шкляр // Известия Томского политехнического университета, Т. 320, №. 5. — Томск, 2012 — С. 73-79.

6. Захарова, А.А. Основные принципы построения визуальных моделей данных на примере интерактивных систем трехмерной визуализации / А.А. Захарова, А.В. Шкляр // Научная визуализация, № 2. — 2014. — С. 62-73.

7. Klyachin, V.A. Mathematical model of 3D maps and design of information system for its control / V.A. Klyachin, E.G. Grigorieva // Journal of Computational and Engineering Mathematics, Vol. 3, № 4. — 2016. — pp. 51–58.

8. Luebke, D. Level of Detail for 3D Graphics / D. Luebke, M. Reddy, J.D. Cohen, A. Varshney, B. Watson, R. Huebner // Elsevier Science Inc. — New York, NY, USA, 2002. — 390 P.

9. Скворцов, А.В. Алгоритмы построения и анализа триангуляции / А.В. Скворцов, Н.С. Мирза // Изд-во Том. ун-та. – Томск, 2006. – 168 с.

10. Lengyel, E. Voxel-Based Terrain for Real-Time Virtual Simulations // PhD diss., University of California at Davis. — 2010. — 95 P.

11. Forstmann, S. Research on Improving Methods for Visualizing Common Elements in Video Game Applications // PhD diss., Waseda University. — 2013. — 168 P.

12. Geiss, R. Generating complex procedural terrains using the GPU // GPU Gems 3, Addison-Wesley Professional. — 2007. — P. 7–37.

13. Lengyel, E. Transition Cells for Dynamic Multiresolution Marching Cubes // Journal of Graphics, GPU, and Game Tools, Vol. 15, Iss. 2. — 2010. — P. 99–122.

14. Forstmann, S., Ohya, J. Visualization of Large Isosurfaces Based on Nested Clipboxes / S. Forstmann, J. Ohya // Siggraph 2005, Article No. 126. . — 2005.

15. Forstmann, S., Ohya, J. Visualizing Large Procedural Volumetric Terrains Using Nested Clip-Boxes // GITS Research Bulletin 2011. — 2011.

16. Peytavie, A. Arches: a Framework for Modeling Complex Terrains / A. Peytavie et al // Computer Graphics Forum, Vol. 28, Iss. 2. — 2009. — P. 457–467.

17. Rosa, M. Destructible Volumetric Terrain // GPU Pro, A K Peters. — 2010. — P. 597–609.

18. Löffler, F. Real-time rendering of stack-based terrains / F. Löffler, A. Müller, H. Schumann // Proceedings of 16th international workshop on Vision, Modeling, and Visualization (VMV), Geneve, CH: Eurographics. — Berlin, 2011. — P. 161–168.

19. Löffler, F. Generating smooth high-quality isosurfaces for interactive modeling and visualization of complex terrains / F. Löffler, H. Schumann // Proceedings of 17th international workshop on Vision , Modeling, and Visualization (VMV), Geneve, CH: Eurographics. — Magdeburg, 2012. — P. 79–86.

20. Scholz, M. Level of Detail for Real-Time Volumetric Terrain Rendering / M. Scholz, J. Bender, C. Dachsbacher // VMV 2013: Vision, Modeling & Visualization. — 2013. — P. 211–218.

21. Scholz, M. Real-Time Isosurface Extraction With View-Dependent Level of Detail and Applications / M. Scholz, J. Bender, C. Dachsbacher // Computer Graphics Forum, Vol. 34, Iss. 1. — 2015. — P. 103-115.

22. Koca, Ç. A hybrid representation for modeling, interactive editing, and real-time visualization of terrains with volumetric features / Ç. Koca, U. Güdükbay // International Journal of Geographical

Information Science, Vol. 28, Iss. 9. — 2014. — P. 1821–1847.

23. Voxel Farm [Электронный ресурс]. — 2016. — Режим доступа: http://voxelfarm.com/

24. Upvoid [Электронный ресурс]. — 2016. — Режим доступа: https://upvoid.com/

25. Ultimate Terrains [Электронный ресурс]. — 2016. — Режим доступа: http://uterrains.com/

26. TerrainEngine [Электронный ресурс]. — 2016. — Режим доступа: https://www.terrainengine.io/

27. TerraVol [Электронный ресурс]. — 2016. — Режим доступа: http://terravol.blogspot.ru/

28. Lorensen, W. Marching Cubes: a high resolution 3D surface construction algorithm / W. Lorensen, H. Cline // Computer Graphics (SIGGRAPH 87 Proceedings). — 1987. — P. 163–169.

29. Doi, A. An efficient method of triangulating equivalued surfaces by using tetrahedral cells / A. Doi, A. Koide // IEICE Transactions Communication E74, 1. — 1991. — P. 214–224.

30. Gibson, S. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data // Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention, MICCAI 1998. — 1998. — P. 888–898.

31. Ju, T. Dual Contouring of Hermite Data / T. Ju, F. Losasso, S. Schaefer, J. Warren // ACM Transactions on Graphics, Vol. 21, Iss. 3. — 2002. — P. 339–346.

32. The Technology Of The Tomorrow Children [Электронный ресурс] . — 2016. — Режим доступа: http://fumufumu.q-games.com/archives/TheTechnologyOfTomorrowsChildrenFinal.pdf

33. Ho, C.-C. Cubical marching squares: Adaptive feature preserving surface extraction from volume data / C.-C. Ho, F.-C. Wu, B.-Y. Chen, Y.-Y. Chuang, M. Ouhyoung // EUROGRAPHICS 2005, 24, 3. — 2005. — P. 537–545.

34. de Araújo, B. R. A Survey on Implicit Surface Polygonization / B. R. de Araújo, D.S. Lopes, P. Jepp, J.A. Jorge, B. Wyvill // ACM Computing Surveys, Vol. 47, Iss. 4. — 2015. — P. 1–39.

35. Wenger, R. Isosurfaces: Geometry, Topology, and Algorithms // A K Peters/CRC Press Reference. — 2013. — 488 P.

36. Широкий, А.А. Триангуляция Делоне многомерных поверхностей / А.А. Широкий, В.А. Клячин // Вестник Самарского государственного университета, 4 (78). — 2010. — С. 51–55.

37. Шакаев, В.Д. Методы полигонизации изоповерхностей с реконструкцией острых рёбер и углов

/ В. Д. Шакаев, О. А. Шабалина // Изв. ВолгГТУ. Серия "Актуальные проблемы управления, вычислительной техники и информатики в технических системах". Вып. 2 : межвуз. сб. науч. ст. / ВолгГТУ. – Волгоград, 2016. – № 11. – С. 155-160.

38. Engel, K., et al. Real-time volume graphics // A K Peters, Ltd. — 2006. — 506 P.

39. Botsch, M. Geometric Modeling Based on Triangle Meshes / M. Botsch, M. Pauly, C. Rossl, C. Bischoff, L. Kobbelt // EUROGRAPHICS 2006, ACM SIGGRAPH 2006 Courses (SIGGRAPH '06), New York, USA, Article 1. — 2006. — 106 P.

40. Kobbelt, L. Feature sensitive surface extraction from volume data / L. Kobbelt, M. Botsch, U. Schwanecke, P. Seidel // Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001. — 2001. — P. 57–66.

41. Garland, M. Surface simplification using quadric error metrics / M. Garland, P. Heckbert // Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1997. — 1997. — P. 209–216.

42. Garland, M. Quadric-Based Polygonal Surface Simplification // PhD diss., Carnegie Mellon University. — 1999. — 200 P.

43. Lindstrom, P. Out-of-core simplification of large polygonal models // Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2000. — 2000. — P. 259–262.

44. Ho, C.-C., Ouhyoung M. Adaptive Extended Marching Cubes [Электронный ресурс]. — 2004. — Режим доступа: ntur.lib.ntu.edu.tw/retrieve/170788/06.pdf

45. Perry, R.N. Kizamu: A system for sculpting digital characters / R.N. Perry, S.F. Frisken // Proceedings of SIGGRAPH 2001, ACM Press / ACM SIGGRAPH. — 2001. — P. 47–56.

46. Dual Contouring : "The secret sauce" // Technical Report TR 02-408, Dept. of Computer Science, Rice University [Электронный ресурс]. — 2002. — Режим доступа: https://people.eecs.berkeley.edu/~jrs/meshpapers/SchaeferWarren2.pdf

47. Ummenhofer, B. Global, Dense Multiscale Reconstruction for a Billion Points / B. Ummenhofer, T. Brox // IEEE International Conference on Computer Vision (ICCV). — 2015. — P. 1341–1349.

48. Lobello, R.U. Multi-resolution Dual Contouring from Volumetric Data / R.U. Lobello, F. Dupont, F. Denis // GRAPP/IVAPP. — 2012. — P. 163–168.

49. Lobello, R.U. Out-of-core adaptive iso-surface extraction from binary volume data / R.U. Lobello, F. Dupont, F. Denis // Graphical Models 76 (6). — 2014. — P. 593–608.

50. Varadhan, G. Feature-sensitive subdivision and iso-surface reconstruction / G. Varadhan, S. Krishnan, Y. Kim, D. Manocha // IEEE Visualization 2003. — 2003. — P. 99–106.

51. Bischoff, S. Automatic restoration of polygon models / S. Bischoff, D. Pavic, L. Kobbelt // CM Transactions on Graphics, Vol. 24, No. 4. — 2005. — P. 1332–1352.

52. Zhang, N. Dual contouring with topology-preserving simplification using enhanced cell representation / N. Zhang, W. Hong, A. Kaufman // IEEE Proceedings of the conference on Visualization ’04. — 2004, Washington, DC, USA. — P. 505–512.

53. Ju, T. Manifold Dual Contouring / T. Ju, S. Schaefer, J. Warren // IEEE Transactions on Visualization and Computer Graphics 13. — 2007. — P. 610–619.

54. Greß, A. Efficient representation and extraction of 2-manifold isosurfaces using kD-trees / A. Greß, R. Klein // Graphical Models, Vol. 66, No. 6. — 2004. — P. 370–397.

55. Rashid, T. 2-manifold surface meshing using dual contouring with tetrahedral decomposition / T. Rashid, S. Sultana, M.A. Audette // Advances in Engineering Software 102. — 2016. — P. 83–96.

56. Nielson, G.M. Dual Marching Cubes // IEEE visualization’04. — 2004. — P. 489–96.

57. Schmitz, L. Efficient and High Quality Contouring of Isosurfaces on Uniform Grids / L. Schmitz, C. Dietrich, J. Comba // Computer Graphics and Image Processing. — 2009. — P. 64–71.

58. Lobello, R.U. Génération de maillages adaptatifs à partir de données volumiques de grande taille // Thèse de Doctorat en Informatique, Université Lumière Lyon 2. — 2013. — 152 P.

59. Bhattacharya, A. Constructing Isosurfaces with Sharp Features from Scalar Data - Tech Report / A. Bhattacharya, R. Wenger // Technical Report, The Ohio State University, 2014. [Электронный ресурс] / Режим доступа: ftp://ftp.cse.ohio-state.edu/pub/tech-report/2014/TR01.pdf

60. Bhattacharya, A. SHREC: SHarp REConstruction of isosurfaces / Bhattacharya, R. Vasko, R. Wenger // Technical Report TR: OSU-CISRC-11/15-TR22, The Ohio State University, 2015. [Электронный ресурс] / Режим доступа: ftp://ftp.cse.ohio-state.edu/pub/tech-report/2015/TR22.pdf

61. Nielson, G.M. Dual Marching Tetrahedra: Contouring in the Tetrahedronal Environment // Proceedings of the 4th International Symposium on Advances in Visual Computing (ISVC '08). — 2008.

— P. 183–194.

62. Schaefer, S. Dual marching cubes: primal contouring of dual grids / S. Schaefer, J. Warren // Computer Graphics Forum, 24(2). — 2005. — P. 195–201.

63. Manson, J. Isosurfaces over simplicial partitions of multiresolution grids / J. Manson, S. Schaefer // Computer Graphics Forum (Proceedings of Eurographics), 29(2). — 2010. — P. 377–385.

64. Lewiner, T. Fast generation of pointerless octree duals / T. Lewiner, V. Mello, A. Peixoto, S. Pesco, H. Lopes // Symposium on Geometry Processing 2010, Computer Graphics Forum, Vol. 29. — 2010. — P. 1661–1669.

65. Чернышенко, А. Технология построения адаптируемых многогранных сеток и численное решение эллиптических уравнений 2-го порядка в трехмерных областях и на поверхностях: кандидатская диссертация. ИВМ РАН. Москва, 2013.

66. Voxel Farm SDK Documentation [Электронный ресурс]. — 2015. — Режим доступа: http://www.voxelfarm.com/doc.html

67. Gargantini, I. Linear octrees for fast processing of three-dimensional objects // Computer Graphics and Image Processing, Vol. 4, No. 20. — 1982. — P. 365–374.

68. Meagher, D. Octree encoding: A new technique for the representation, manipulation and display of arbitrary three dimensional objects by computer // Tech. Rep. IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, N.Y. — 1980.

69. Samet, H. Applications of Spatial Data Structures // Addison Wesley, Reading, MA. – 1990.

70. Morton, G.M. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd. — 1966, Ottawa, Ontario.

71. Perry, R.N. Simple and Efficient Traversal Methods for Quadtrees and Octrees / R.N. Perry, S.F. Frisken // Journal of Graphics Tools, Vol.7, Iss.3. — 2002. — P. 1–11.

72. Shakaev, V. Polygonizing Volumetric Terrains with Sharp Features // Proc. of the 26th International Conference on Computer Graphics and Vision «GraphiCon 2016». – Nizhny Novgorod, Sept. 19-23, 2016. – P. 364-368.

73. Hoppe, H. Optimization of mesh locality for transparent vertex caching // ACM Transactions on Computer Graphics (Proceedings of SIGGRAPH 1999). — 1999. — P. 269–276.

74. Vo, H.T. Simple and Efficient Mesh Layout with Space-Filling Curves / H.T. Vo, C.T. Silva, L.F. Scheidegger, V. Pascucci // Journal of Graphics Tools, Vol. 16, Iss. 1. — 2012. — P. 25–39.

75. Stocco, L.J. lnteger Dilation and Contraction for Quadtrees and Octrees / L.J. Stocco, G. Schrack // IEEE Communications, Computers and Signal Processing. — 1995. — P. 426–428.

76. Raman, R. Converting to and from Dilated Integers / R. Raman, D.S. Wise // IEEE Transactions on Computers, Vol. 57, No. 4. — 2008. — P. 567–573.

77. Zhang, N. Multiresolution volume simplification and polygonization / N. Zhang, A. Kaufman // Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics (VG '03). — 2003. — P. 87–94.

78. Шакаев, В.Д. Способы представления воксельного ландшафта при проектировании систем виртуальной реальности / В. Д. Шакаев, А. Г. Кравец // Моделирование, оптимизация и информационные технологии. – Воронеж, 2019. – Т. 7, № 1.

79. Zhang, N. CSG Operations on Point Models with Implicit Connectivity / N. Zhang, H. Qu, A. Kaufman // Computer Graphics International 2005. — 2005. — P. 87–93.

80. Wang, C.C.L. Dual Contouring on Uniform Grids: An Approach Based on Convex-Concave Analysis [Электронный ресурс]. — 2011. — Режим доступа: http://www.mae.cuhk.edu.hk/~cwang/pubs/TRIntersectionFreeDC.pdf

81. Ju, T. Intersection-free contouring on an octree grid / T. Ju, T. Udeshi // Proceedings of the 14th Pacific Conference on Computer Graphics and Applications, IEEE Computer Society. — 2006.

82. Liu, S. Fast Intersection-free Offset Surface Generation from Freeform Models with Triangular Meshes / S. Liu, C.C.L. Wang // IEEE Transactions on Automation Science and Engineering, Vol. 8, Iss. 2. — 2011. — P. 347–360.

83. Carr, H. Artifacts caused by simplicial subdivision / H. Carr, T. Moller, J. Snoeyink // IEEE Transaction on Visualization and Computer Graphics, Vol. 12, Iss. 2. — 2006. — P. 231–242.

84. Knoll, A.M. Ray Tracing Implicit Surfaces for Interactive Visualization // PhD diss., The University of Utah. — 2009. — 161 P.

85. Hoffmann, C. Robustness in geometric computations // Journal of Computing and Information Science in Engineering, 1. — 2001. — P. 143–156.

86. Frisken, S.F. Designing with distance fields / S.F. Frisken, R.N. Perry // Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI '05), IEEE Computer Society. — 2005, Washington, DC, USA. – P. 58–59.

87. Толок, А.В. Функционально-воксельные вычисления на компьютере. / Труды 26-й Международной научной конференции GraphiCon2016. — Нижний Новгород, 2016 . — С. 1–8.

88. Фрязинов, О.В. Методы и алгоритмы дискретизации неявно заданных неоднородных геометрических объектов: кандидатская диссертация. — Москва, 2004. — 147 с.

89. Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for ‘Dreams PS4’, a Geometrically Dense, Painterly UGC Game / Evans A. [Электронный ресурс]. — 2015.— Режим доступа: media.lolrus.mediamolecule.com/AlexEvans_SIGGRAPH-2015-sml.pdf

90. Menon, J. More powerful solid modeling through ray representations / J. Menon, R. Marisa, J. Zagajac // IEEE Computer Graphics and Applications, 14. — 1994. — P. 22–35.

91. Rocchini, C. Marching Intersections: an Efficient Resampling Algorithm / C. Rocchini, P. Cignoni, F. Ganovelli, C. Montani, P. Pingi, R. Scopigno // Shape Modeling International, IEEE Computer Society. — 2001. – P. 296–305.

92. Benouamer, M.O. Bridging the Gap between CSG and Brep via a Triple Ray Representation / M.O. Benouamer, D. Michelucci // Proceedings of the fourth ACM symposium on Solid modeling and applications (SMA '97), ACM. — 1997, New York, NY, USA. — P. 68–79.

93. Wang, C.C.L. Layered Depth-Normal Images: a Sparse Implicit Representation of Solid Models / C.C.L. Wang, Y. Chen // Technical Report, The Chinese University of Hong Kong. — 2007.

94. Chen, Y. Layer Depth-Normal Images for Complex Geometries: Part One — Accurate Modeling and Adaptive Sampling / Y. Chen, C.C.L. Wang // ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Volume 3: 28th Computers and Information in Engineering Conference, Parts A and B. — 2008. — P. 717–728.

95. Chen, Y. Layered Depth-Normal Images for Complex Geometries: Part Two — Manifold-Preserved Adaptive Contouring / Y. Chen, C.C.L. Wang // ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Volume 3: 28th Computers and Information in Engineering Conference, Parts A and B. — 2008. — P. 729–739.

96. Wang, C.C.L. Solid modeling of polyhedral objects by Layered Depth-Normal Images on the GPU /

C.C.L. Wang, Y.-S. Leung, Y. Chen // Computer-Aided Design, Vol. 42, Iss. 6. — 2010. — P. 535–544.

97. Zhao, H. Parallel and efficient Boolean on polygonal solids / H. Zhao, C.C.L. Wang, Y. Chen, X. Jin // The Visual Computer, Vol. 27, Iss. 6–8. — 2011. — P. 507–517.

98. Wang, C.C.L. Computing on rays: A parallel approach for surface mesh modeling from multi-material volumetric data // Computers in Industry, Vol. 62, Iss. 7. — 2011. — P. 660–671.

99. Wang, C.C.L. Regulating complex geometries using layered depth-normal images for rapid prototyping and manufacturing / C.C.L. Wang, Y. Chen // Rapid Prototyping Journal, Vol. 19, Iss. 4. — 2013. — P. 253–268.

100. Kwok, T.-H. Geometric Analysis and Computation Using Layered Depth-Normal Images for Three-Dimensional Microfabrication / T.-H. Kwok, Y. Chen, C.C.L. Wang // Three-Dimensional Microfabrication Using Two-photon Polymerization (Micro and Nano Technologies), William Andrew Publishing. — 2016. — P. 119–147.

101. Ho, C.-C. Detail sculpting using cubical marching squares / C.-C. Ho, C.-H. Tu, M. Ouhyoung // Proceedings of the 2005 international conference on Augmented tele-existence (ICAT '05). — 2005, NY, USA. — P. 10–15.

102. Nooruddin, F. Simplification and Repair of Polygonal Models Using Volumetric Techniques / F. Nooruddin, G. Turk // ACM Transactions on Visualization and Computer Graphics, Vol. 9, Iss. 2. — 2003. — P. 191–205.

103. Lefebvre, S. IceSL: A GPU Accelerated CSG Modeler and Slicer // AEFA'13, 18th European Forum on Additive Manufacturing. — 2013, Paris, France.

104. Feng, P. A Dual Method for Constructing Multi-material Solids from Ray-Reps / P. Feng, J. Warren // International Symposium on Visual Computing (ISVC 2012), Lecture Notes in Computer Science, vol 7431. – 2012. — P. 92–103.

105. Ju, T. Robust repair of polygonal models // ACM Transactions on Graphics, Vol. 23, No. 3. — 2004. — P. 888–895.

106. Lindstrom, P. Fast and Memory Efficient Polygonal Simplification / P. Lindstrom, G. Turk // IEEE Proceedings on Visualization. — 1998. — P. 279–286.

107. Vo, N.A. Inverted index compression using word-aligned binary codes / N.A. Vo, M. Alistair // Information Retrieval, Vol. 8, Iss. 1. — 2005. — P. 151–166.

108. Ericson, C. Real-Time Collision Detection // Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. – 2004. – 594 P.

109. Данилов, А. Технология построения неструктурированных сеток и монотонная дискретизация уравнения диффузии: кандидатская диссертация. ИВМ РАН. Москва, 2010.

110. Goldsmith, J. Automatic Creation of Object Hierarchies for Ray Tracing / J. Goldsmith, J. Salmon // IEEE Computer Graphics & Applications 7, 5 (May). — 1987. — P. 14–20.

111. MacDonald, D.J. Heuristics for Ray Tracing Using Space Subdivision / D.J. MacDonald, K.S. Booth // Visual Computer, 6, 3. — 1990. — P. 153–166.

112. Havran, V. Heuristic ray shooting algorithms // PhD. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague. — 2000.

113. Naylor, B.F. A Tutorial on Binary Space Partitioning Trees // ACM SIGGRAPH 94. — 1994.

114. Петрухин, А.В. Обнаружение столкновений движущихся твёрдых невыпуклых объектов в процессе компьютерной 2D и 3D анимации / А.В. Петрухин, В.Д. Шакаев // Изв. ВолгГТУ. Серия "Актуальные проблемы управления, вычислительной техники и информатики в технических системах". Вып. 12 : межвуз. сб. науч. ст. / ВолгГТУ. – Волгоград, 2011. – № 11. – С. 63-65.

115. Yngve, G. Robust Creation of Implicit Surfaces from Polygonal Meshes / G. Yngve, G. Turk // IEEE Transactions on Visualization and Computer Graphics, Vol. 8, Iss. 4. — 2002. — P. 346–359.

116. Chen, Y. An accurate sampling-based method for approximating geometry // Computer-Aided Design, Vol. 39, Iss. 11. — 2007. — P. 975–986.

117. Fryazinov, O. BSP-fields: An exact representation of polygonal objects by differentiable scalar fields based on binary space partitioning / O. Fryazinov, A. Pasko, V. Adzhiev // Computer-Aided Design, Vol. 43, Iss. 3. — 2011. — P. 265–277.

118. Virtual Terrain Project [Электронный ресурс]. — 2016. — Режим доступа: http://vterrain.org/

119. H. de Boer, Willem. Fast Terrain Rendering Using Geometrical MipMapping [Электронный ресурс]. — 2000. — Режим доступа: https://www.flipcode.com/archives/article_geomipmaps.pdf

120. Ulrich, T. Rendering Massive Terrains using Chunked Level of Detail Control // ACM SIGGRAPH Course Notes “Super-size it! Scaling up to Massive Virtual Worlds”. — 2002.

121. Cignoni, P. BDAM − Batched Dynamic Adaptive Meshes for High Performance Terrain

Visualization / P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno // Computer Graphics Forum, Vol. 22, Iss. 3. — 2003. — P. 505–514.

122. Cignoni, P. Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM) / P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno // IEEE Visualization. — 2003. — P. 147–154.

123. Gobbetti, E. C-BDAM – Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering / E. Gobbetti, F. Marton, P. Cignoni, M. Di. Benedetto, F. Ganovelli // Computer Graphics Forum, Vol. 25, Iss. 3. — 2006. — P. 333–342.

124. Losasso, F. Geometry Clipmaps: Terrain Rendering using Nested Regular Grids / F. Losasso, H. Hoppe // ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004), Vol. 23, Iss. 3. — 2004. — P. 769–776.

125. Asirvatham, A. Terrain rendering using GPU-based geometry clipmaps / A. Asirvatham, H. Hoppe //, H. Hoppe // // GPU Gems 2. — 2005. — P. 27–46.

126. Tanner, C.C. The Clipmap: A Virtual Mipmap / C.C. Tanner, C.J. Migdal, M.T. Jones // SIGGRAPH’98 Proc. 25th Annu. Conf. Comput. Graph. Interact. Tech. — 1998. — P. 151–158.

127. Strugar, F. Continuous distance-dependent level of detail for rendering heightmaps // Journal of Graphics, GPU, and Game Tools 14, vol. 4. — 2009. — P. 57–74.

128. Hoppe, H. 1996. Progressive meshes // ACM Transactions on Computer Graphics (Proceedings of SIGGRAPH 1996). — 1996. — P. 99–108.

129. Hoppe, H. View-dependent refinement of progressive meshes // ACM Transactions on Computer Graphics (Proceedings of SIGGRAPH’97). — 1996. — P. 189–198.

130. Hoppe, H. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering // IEEE Visualization 1998. — 1998. — P. 35–42.

131. Fergusson, R. Continuous terrain level of detail for visual simulation / R. Fergusson, R. Economy, A. Kelly // ACM Symposium on Interactive 3D Graphics. — 2001. — P. 111–120.

132. Sander, P.V. Progressive Buffers: View-dependent Geometry and Texture LOD Rendering / P.V. Sander, J.L. Mitchell // Proceedings of the third Eurographics symposium on Geometry processing (SGP '05), Article 129. — 2010. — P. 129–138.

133. Shakaev, V. View-Dependent Level of Detail for Real-Time Rendering of Large Isosurfaces / V.

Shakaev, Н.П. Садовникова, Д.С. Парыгин // Creativity in Intelligent Technologies and Data Science. Second Conference, CIT&DS 2017 (Volgograd, Russia, September 12-14, 2017) : Proceedings / ed. by A. Kravets, M. Shcherbakov, M. Kultsova, Peter Groumpos ; Volgograd State Technical University [et al.]. – [Germany] : Springer International Publishing AG, 2017. – P. 501-516. – (Ser. Communications in Computer and Information Science ; Vol. 754).

134. Юсов, Е.А. Эффективное кодирование адаптивной триангуляции рельефа в контексте иерархического вейвлет-сжатия сетки высот / Е.А. Юсов, В.Е. Турлапов // Вестник Нижегородского университета им. Н.И. Лобачевского, № 5. — 2010. — С. 209–219.

135. Yusov, E. High-performance terrain rendering using hardware tessellation // E. Yusov, M. Shevtsov // Journal of WSCG ISSN 1213-6972. — 2011. — Vol. 19, No. 3. — P. 85–92.

136. DirectX 11 Terrain Tessellation // NVidia SDK 2011. — 2011.

137. Pangerl, D. Dynamic GPU Terrain // GPU Pro 6. — 2016.

138. McAnlis, C. Halo wars: the terrain of next gen. // Game developers conference 2009, March. San Francisco, CA: IGDA. URL: http://www.gdcvault.com/play/1277/HALO-WARS-The-Terrain-of

139. Ju, T. Convex contouring of volumetric data / T. Ju, S. Schaefer, J. Warren // Visual Computing, Vol. 19, Iss. 7-8. — 2003. — P. 513–525.

140. Bönning, R. Interactive sculpturing and visualization of unbounded voxel volumes / R. Bönning, H. Müller // SM’02, Vol. 19, No. 3. — 2002. — P. 85–92.

141. Gobbetti, E. Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms / E. Gobbetti, F. Marton // ACM SIGGRAPH 2005 Papers (SIGGRAPH '05) . — 2005. — P. 878–885.

142. Dick, C. GPU-Aware Hybrid Terrain Rendering / C. Dick, J. Krüger, R. Westermann // Proceedings of IADIS Computer Graphics, Visualization, Computer Vision and Image Processing 2010. — 2010. — P. 3–10.

143. Treib, M. Interactive Editing of GigaSample Terrain Fields / M. Treib, F. Reichl, S. Auer, R. Westermann // Computer Graphics Forum (Proc. Eurographics), Vol. 31, No. 2. — 2012. — P. 383–392.

144. Volume GFX: Realtime Volume Rendering Aimed at Terrain [Электронный ресурс]. — 2014. — Режим доступа: https://www.volume-gfx.com/

145. Guildea, N. Dual Contouring: Seams & LOD for Chunked Terrain [Электронный ресурс]. — 2014. — Режим доступа: http://ngildea.blogspot.ru/2014/09/dual-contouring-chunked-terrain.html

146. Шакаев, В.Д. Метод бесшовной стыковки блоков воксельного ландшафта // Инновационные, информационные и коммуникационные технологии. ИНФО–2016: сб. тр. XIII междунар. науч.-практ. конф. (г. Сочи, 1-10 окт. 2016 г.) / ред-кол.: С.У. Увайсов (гл. ред.), И.А. Иванов (отв. ред.) [и др.]; Ассоциация выпускников и сотрудников ВВИА им. проф. Жуковского [и др.]. – Москва, 2016. – С. 356-358.

147. Proland: A C++/OpenGL library for the real-time realistic rendering of very large and detailed 3D natural scenes on GPU [Электронный ресурс]. — 2016. — Режим доступа: http://proland.inrialpes.fr

148. Whigham, J. Goodbye Octrees [Электронный ресурс]. — 2013. — http://johnwhigham.blogspot.ru/2013/06/goodbye-octrees.html

149. Maslovsky, D. Bending Unity to Carry Spherical Voxel Planets in Planet Nomads [Электронный ресурс]. — 2016. — Режим доступа: https://www.gamasutra.com/blogs/DanielMaslovsky/20161018/283317/Bending_Unity_to_Carry_Spherical_Voxel_Planets_in_Planet_Nomads.php

150. Voxels and Seamless LOD Transitions [Электронный ресурс]. — 2016. — Режим доступа: https://dexyfex.com/2016/07/14/voxels-and-seamless-lod-transitions/

151. Cozzi, P. A WebGL Globe Rendering Pipeline / P. Cozzi, D. Bagnell // GPU Pro 4. — 2013. — P. 39–48.

152. Kazakov, M. Fast isosurface polygonization with embedded sharp features extraction / M. Kazakov, A. Pasko, V. Adzhiev // Technical Report HCIS-2003-04, Hosei University, Tokyo, Japan. — 2003. — 11 P.

153. Kjolstad, F.B. Ghost Cell Pattern / F.B. Kjolstad, M. Snir // Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10), Article 4. — 2010, New York, USA. — 9 P.

154. Rossignac, J. Multi-resolution 3D approximations for rendering complex scenes / J. Rossignac, P. Borrell // Modeling in Computer Graphics. — 1993. — P. 455–465.

155. Melax, S. A simple, fast, and effective polygon reduction algorithm // Game Developer Magazine, No. 11. — 1998. — P. 44–49.

156. Shakaev, V. C++ reflection library for game engines / Shakaev V., Shabalina O., Kamaev V.

Proceedings IADIS International Conference Applied Computing, 23 – 25 October 2013. – Fort Worth, Texas, USA. – P. 237-240.

157. Шакаев, В.Д. Абстрагирование низкоуровневых API при создании кроссплатформенных графических приложений / В. Д. Шакаев, О. А. Шабалина // Вестник компьютерных и информационных технологий (индекс в Web of Science): ежемесячный научно-технический и производственный журнал. – 2014. – N 8. – С.11-16.

158. Shakaev, V. Interactive Graphics Applications Development: An Effect Framework for Directx 11 / V. Shakaev, O. Shabalina, V. Kamaev // World Applied Sciences Journal 24 (Information Technologies in Modern Industry, Education & Society), IDOSI Publications (индексируется в Scopus). – 2013. – P. 165-170.

159. Шакаев, В.Д. Low-level API agnostic rendering interface for bridging OpenGL and Direct3D / В.Д. Шакаев, О.А. Шабалина, В.А. Камаев, S. Chickerur // Innovation Information Technologies: mater. of the 3rd Int. scien.-pract. conf. (Praque, April 21-25, 2014). Part 1 / МИЭМ ВШЭ, Рос. центр науки и культуры в Праге. – М., 2014. – С. 203-211.

160. Свид. о гос. регистрации программы для ЭВМ № 2015610924 от 21 янв. 2015 г. РФ, МПК (нет). Система управления процессом обучения / В.Д. Шакаев, О.А. Шабалина; ВолгГТУ. – 2015.

Приложение А

Ключевые понятия предметной области

Во многих областях науки и техники для описания трёхмерных объектов со сложной внутренней структурой используются функции вида $F(x, y, z) : \mathbb{R}^3 \longmapsto \mathbb{R}$. Например, в каждой точке области скалярная функция F может определять такие свойства рассматриваемой модели, как расстояние до поверхности тела или его плотность, концентрацию вещества, напряжённость поля, давление, температуру. Такое неявное описание может быть построено на основе геометрической модели тела или задано аналитически. При этом, как правило, ищется такая скалярная функция F, чтобы граница тела Ω оказалась бы её нулевой изоповерхностью:

$$\partial\Omega = \{ (x,y,z) \mid F(x,y,z) = 0 \}.$$

Изоповерхностью (isosurface) называется поверхность, проходящая через точки с постоянными значениями исследуемой скалярной функции. Процесс построения поверхностного, граничного представления (boundary representation, B-Rep), кусочно-линейно аппроксимирующего изоповерхность полигональной сеткой, называется *полигонизацией* (или, соответственно, *триангуляцией*, если строится треугольная сетка поверхности), *извлечением изоповерхности (isosurface extraction, isosurfacing)* или *построением контура* (*контурирование, contouring)*. Полученная при этом полигональная (многогранная) сетка приближает гладкие поверхности со вторым порядком точности, а поверхности с острыми углами (где не определена нормаль к поверхности) — с первым порядком точности.

Развёртываемые поверхности (2-manifold surfaces) являются двумерными многообразиями, погруженными в трёхмерное пространство. Развёртываемые или двусторонние полигональные (или многогранные) сетки ограничивают замкнутые объёмы (для которых однозначно определены понятия «внутри» и «снаружи»), не касаются себя, не имеют самопересечений, сингулярных вершин и рёбер, и обладают топологией объектов, которые могут существовать в реальном мире. В развёртываемых полигональных сетках все грани ориентированы наружу, а каждое ребро принадлежит только двум граням (на каждое ребро опирается не более чем два полигона).

Многие операции (например, вычисление нормалей к поверхности и её кривизны, сглаживание (smoothing), булевы операции (CSG), параметризация полигональной сетки $(x,y,z) \leftrightarrow (u,v)$ при создании UV-развёртки для текстурирования, декомпозиция объекта на тетраэдры для метода конечных элементов) определены только для двусторонних развёртываемых полигональных сеток, что является критичным для большинства приложений CAD/CAM/CAE.

(Кроме того, многие алгоритмы обработки сеток можно реализовать гораздо проще и эффективнее, если заранее известно, что сетки будут развёртываемыми.)

Под низкокачественными полигональными сетками будем понимать сетки, содержащие полигоны плохой формы, например, вырожденные (degenerated) или почти вырожденные (sliver, skinny) полигоны. Например, «плохая» треугольная сетка содержит треугольники с малой площадью или с большим соотношением углов или сторон, вырожденные треугольники и т.п.

В рамках данной работы рассматриваются интерактивные ячеечные (cube-based) методы полигонизации изоповерхностей, в которых полигональная сетка строится на основе данных, полученных путём пересечения поверхности с рёбрами (непересекающихся и выпуклых) ячеек решётки разбиения пространства.

В ячеечных методах триангуляции под клеткой или *ячейкой (cell)* понимается минимальный элемент дискретизации пространства (например, куб, если для разбиения области используется регулярная шестиугольная решётка или октодерево). *Вершина ячейки (cell vertex)* является *отрицательной (negative)*, если значение исследуемой скалярной функции в ней меньше, чем на изоповерхности, и *положительной (positive)* в противном случае. *Знакоопределённая решётка (signed grid)* представляет собой сетку разбиения пространства, для каждой ячейки которой известны знаки всех её вершин. В случае составных областей вершинам решётки разбиения вместо знака характеристической функции области присваивается индекс подобласти или *материал*. *Активное ребро (active edge)* — это ребро ячейки, вершины которого имеют разные знаки (т.е. активное ребро пересекает изоповерхность нечётное число раз). *Активная ячейка (active cell) или граничная ячейка (boundary cell)* — это ячейка, пересекающая изоповерхность (т.е. все вершины активной ячейки не могут быть одновременно положительными или отрицательными). *Особенные ячейки* — это активные ячейки, содержащие *особенности поверхности (surface features)*: острые рёбра и конические вершины. *Особые точки* поверхности — это граничные точки, в которых не определена нормаль к поверхности. *Острое ребро (crease edge)* поверхности — это ребро, двугранный угол[18] (dihedral edge angle) которого меньше некоторой пороговой величины θ (*feature angle*) (например, меньше 140 градусов).

Адаптивные методы (*adaptive, multiresolution*) триангуляции используют неравномерную сегментацию пространства, фокусируясь на неоднородных областях пространства и отбрасывая «неинтересные» части. При этом для разбиения пространства используются *адаптивные решётки*

[18] Например, в развёртываемой многогранной (или полигональной) сетке двугранный угол на ребре — это угол между нормалями двух смежных многогранников (или полигонов), которые опираются на данное ребро.

(adaptive grids) и иерархические структуры, такие как *октодеревья (octrees)* [69] (известные также, как октарные, октальные, октотомические или восьмеричные деревья), *kD-деревья (kD-trees)* и иерархии тетраэдров, во всех которых ячейками являются листовые узлы. По аналогии со знакоопределёнными решётками октодерево, у ячеек которого известны знаки вершин, называется *знакоопределённым октодеревом (signed octree)*. По сравнению с методами на регулярных сетках в адаптивных методах размеры ячеек варьируются в зависимости от свойств рассматриваемой области (например, от неоднородности её состава и кривизны поверхности), что позволяет снизить объём потребляемой памяти и получать более оптимальные поверхностные сетки, «сгущающиеся» в интересных областях.

В *прямых методах (primal methods)* триангуляции аппроксимация изоповерхности внутри каждой активной ячейки создаётся списком полигонов, вершины которых лежат на активных рёбрах ячейки. Прямые методы допускают табличную реализацию, обладают высокой скоростью исполнения и способны генерировать развёртываемые полигональные сетки с двусторонней топологией. Поскольку каждая ячейка триангулируется независимо от других, то прямые методы легко параллелизуются. Недостатками прямых методов являются невозможность передачи особенностей поверхности, находящихся внутри ячейки, сложная реализация для адаптивных схем разбиения пространства и высокая вероятность создания низкокачественных треугольных сеток. Примерами прямых методов триангуляции являются алгоритмы марширующих кубиков (Marching Cubes, MC) [28] и марширующих тетраэдров (Marching Tetrahedra, MT) [29].

Двойственные или дуальные методы (dual methods) триангуляции изоповерхностей генерируют полигональные сетки, двойственные по отношению к сеткам, получаемым прямыми методами: для каждой активной ячейки создаётся одна или несколько вершин, а для каждого активного ребра — полигон, образованный вершинами смежных ячеек, включающих это ребро. Классическими примерами дуальных или двойственных методов являются метод поверхностных сеток (Surface Nets) [30], алгоритм дуальных (или двойственных) контуров (Dual Contouring, DC) [31] и алгоритм дуальных (или двойственных) марширующих кубиков (Dual Marching Cubes, DMC) [56]. На рис. 1.4 и 1.5 показано сравнение работы прямых методов триангуляции (на примере алгоритма марширующих кубиков) и двойственных методов (на примере алгоритма дуальных контуров или поверхностных сеток).

Поскольку в двойственных методах вершины не закреплены на рёбрах ячейки, а могут свободно «плавать» внутри ячейки, то эти методы обычно создают более качественные полигональные сетки, чем прямые методы. Помещая вершины в особенные точки на

изоповерхности, двойственные методы могут воспроизводить острые углы и рёбра фрагмента изоповерхности внутри ячеек. Но при этом возникает проблема устранения самопересечений: полигоны, созданные для активных рёбер соседних ячеек, могут пересекаться друг с другом. Помимо этого, двойственные методы не всегда генерируют поверхностные сетки, которые являются двумерными многообразиями. Характерной особенностью двойственных методов триангуляции является зависимость ячеек друг от друга (*inter-cell dependency*): для получения каждого примитива полигональной сетки (например, чётырёугольника или треугольника) необходимо рассмотреть несколько смежных ячеек (соединить их вершины), т.е. отдельно взятая активная ячейка не может быть триангулирована по отдельности, без обращения к данным соседних ячеек. Эта особенность приводит к тому, что для бесшовного соединения каждого блока воксельного ландшафта со смежными к нему блоками необходимо обращаться к ячейкам, расположенным внутри смежных блоков (см. раздел 4.3). Однако эта особенность обусловливает тривиальное расширение двойственных методов до адаптивных версий.

Реконструкция острых углов изоповерхности в двойственных методах триангуляции. Для аппроксимации особенностей поверхности в двойственных методах триангуляции традиционно используются точные координаты точек пересечения активных рёбер ячейки с поверхностью и единичные нормали к поверхности в этих точках: *Эрмитовы данные* (*Hermite data*) [31,46] (рисунок А.1).

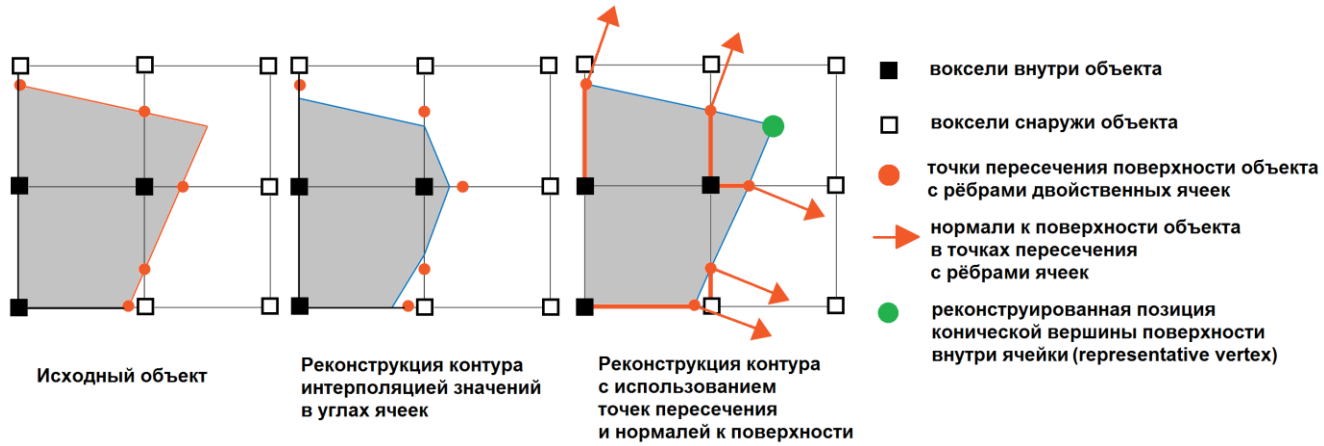


Рисунок А.1 – Использование нормалей в точках пересечения активных рёбер ячейки с контуром поверхности позволяет «предсказать» позицию острого угла поверхности.

При этом на каждом ребре, как правило, не должно быть более одной точки пересечения, а двойственные методы, в которых для каждой активной ячейки создаётся только по одной вершине, не гарантируют создание развёртываемых полигональных сеток при достаточном малом разрешении решётки разбиения.

Приложение Б

Программная реализация алгоритма триангуляции

В данном приложении приведены реализация предложенного в подразделе 2.4.2 алгоритма триангуляции, а также некоторые необходимые для его работы вспомогательные функции, на языке C++.

1. Используемые обозначения вершин и рёбер куба

В работе рассматриваются ячеечные алгоритмы триангуляции, в которых для дискретизации пространства используются кубические решётки. Функции для работы с кубическими решётками очень часто вызываются во внутренних циклах алгоритмов триангуляции, поэтому большое значение имеет эффективность их реализации. Далее приведены используемые обозначения вершин и рёбер куба и соответствующие функции.

```
/*
          Z                       UP  NORTH
          |  /Y                   |  /
          | /                     | /
          |/_____X          WEST--O--EAST
         (0,0,0)                 /|
                                / |
                           SOUTH  DOWN

LABELING OF VERTICES, EDGES AND FACES:

Vertex enumeration:         Using locational (Morton) codes (X - lowest bit, Y - middle, Z - highest):
        6___________7                   110_________111
       /|          /                     /|          /
      / |         /|                    / |         /|
     /  |        / |                   /  |        / |
    4------------5  |                100-----------101 |
    |   2________|__3                 |  010______|__011
    |   /        |  /                  |   /       |  /
    |  /         | /                   |  /        | /
    | /          |/                    | /         |/
    0/__________1                   000/__________001

(see Gray code, Hamming distance, De Bruijn sequence)
NOTE: two vertices are connected by an edge, if their indices differ by one and only one bit.

Cube edge enumeration:
(edges are split into 3 groups by axes X,Y,Z - this allows: edge_axis = edge_index / 4;
(numbered using right-hand rule):
        ______2______
       /|           /|
     5 |11       6  |
     /  |         /  |10
    |------3-----|   |
    |   |_____1__|___|
    |   /        |   /
   8|  4        9|  7
    | /          | /
    |/___________|/
          0
```

```
Face orientation (normal) - apply right-hand rule to the edge loop.
*/
static const U32 CUBE_EDGES_AXIS_X = (BIT(0) | BIT(1) | BIT(2) | BIT(3));
static const U32 CUBE_EDGES_AXIS_Y = (BIT(4) | BIT(5) | BIT(6) | BIT(7));
static const U32 CUBE_EDGES_AXIS_Z = (BIT(8) | BIT(9) | BIT(10) | BIT(11));

/// maps the given cube edge to the diagonally opposite one, e.g. 0 -> 2, 3 -> 1, 10 -> 8, etc.
static mxFORCEINLINE UINT DiagonallyOppositeEdge( UINT iEdge ) {
  return ((iEdge + 2) % 4) + (iEdge & ~3);
}

/// Returns the index of the edge on the given edge-adjacent cube 'iAdjCube',
/// where iAdjCube is the index [0..3] of the cube around the edge, listed in circular order.
/// E.g. for a cube edge 2 the returned values will be 2, 3, 0, 1;
/// for a cube edge 8 the returned values will be 8, 9, 10, 11;
/// for a cube edge 6 the returned values will be 6, 5, 4, 7.
static mxFORCEINLINE UINT EdgeOnNextAdjacentCube( UINT iEdge, UINT iAdjCube ) {
  const UINT iEdgeOnNeighbor = (iEdge + iAdjCube) % 4 + (iEdge & ~3);
  return iEdgeOnNeighbor;
}

/// removes the first set least significant bit from the given bitmask
mxFORCEINLINE U32 ExtractNextBit( U32 & _mask ) {
  DWORD nextBitIdx;
  _BitScanReverse( &nextBitIdx, _mask ); // find the LSB
  _mask &= ~(1u << nextBitIdx);  // remove the set bit from the mask
  return nextBitIdx;
}
// offsets of adjacent cells for each edge of the cube (the current cell is located in the 0th octant)
const Int3 NEIGHBORS_SHARING_EDGE[12][3] = {
  Int3( 0, -1,  0 ), Int3( 0,  -1, -1 ), Int3( 0,  0, -1 ),      // Edge 0 (X axis)
  Int3( 0,  0, -1 ), Int3( 0,   1, -1 ), Int3( 0,  1,  0 ),      // Edge 1 (X axis)
  Int3( 0,  1,  0 ), Int3( 0,   1,  1 ), Int3( 0,  0,  1 ),      // Edge 2 (X axis)
  Int3( 0,  0,  1 ), Int3( 0,  -1,  1 ), Int3( 0, -1,  0 ),      // Edge 3 (X axis)
  Int3( 0,  0, -1 ), Int3( -1,  0, -1 ), Int3( -1, 0,  0 ),      // Edge 4 (Y axis)
  Int3(-1,  0,  0 ), Int3( -1,  0,  1 ), Int3(  0, 0,  1 ),      // Edge 5 (Y axis)
  Int3( 0,  0,  1 ), Int3(  1,  0,  1 ), Int3(  1, 0,  0 ),      // Edge 6 (Y axis)
  Int3( 1,  0,  0 ), Int3(  1,  0, -1 ), Int3(  0, 0, -1 ),      // Edge 7 (Y axis)
  Int3(-1,  0,  0 ), Int3( -1, -1,  0 ), Int3(  0,-1,  0 ),      // Edge 8 (Z axis)
  Int3( 0, -1,  0 ), Int3(  1, -1,  0 ), Int3(  1, 0,  0 ),      // Edge 9 (Z axis)
  Int3( 1,  0,  0 ), Int3(  1,  1,  0 ), Int3(  0, 1,  0 ),      // Edge 10 (Z axis)
  Int3( 0,  1,  0 ), Int3( -1,  1,  0 ), Int3( -1, 0,  0 ),      // Edge 11 (Z axis)
};
// A branch saved - a cycle gained!
#define vxNONTRIVIAL_CELL( CORNER_MASK )     (U8((CORNER_MASK) + 1) > 1)
```

2. Функции для работы с кодами Мортона

Далее приведены функции для работы с 30-битными кодами Мортона, которые используются в предложенном алгоритме триангуляции.

```
/// Morton code/key, used as a cell's locational code or address in linear octrees.
typedef U32 Morton32;

const Morton32 MORTON32_OCTREE_ROOT_CODE = 1;    //!< The locational code of the root node = 1
/// Maximum octree depth/refinement level (XYZ: 3 coords of 10 bits each which fits into a 32-bit integer).
enum { MORTON32_MAX_OCTREE_LEVELS = ( sizeof(Morton32) * BITS_IN_BYTE ) / 3u };   //limited by the number of bits in
Morton32
const U32 MORTON32_MAX_RESOLUTION = (1u << MORTON32_MAX_OCTREE_LEVELS);   //!< max octree resolution

/// These masks allow to manipulate/inspect coordinates without unpacking Morton codes.
const U64 ISOLATE_X = 0x9249249249249249; //!< 0b1001...01001001
const U64 ISOLATE_Y = 0x2492492492492492; //!< 0b0010...10010010
const U64 ISOLATE_Z = 0x4924924924924924; //!< 0b0100...00100100

const U64 ISOLATE_XY = ISOLATE_X | ISOLATE_Y;
```

```
const U64 ISOLATE_XZ = ISOLATE_X | ISOLATE_Z;
const U64 ISOLATE_YZ = ISOLATE_Y | ISOLATE_Z;

/// Dilate bits along the 32-bit unsigned integer.
inline U32 Morton32_SpreadBits3( U32 x ) {
  x &= 0x3ff;    // zero out the upper 20 bits
  x = (x | x << 16) & 0x30000ff; // 0b______11 ________ ________ 11111111
  x = (x | x << 8) & 0x300f00f;  // 0b______11 ________ 1111____ ____1111
  x = (x | x << 4) & 0x30c30c3;   // 0b______11 ____11__ __11____ 11____11
  x = (x | x << 2) & 0x9249249;   // 0b____1__1 __1__1__ 1__1__1_ _1__1__1
  return x;
}//widen/dilate/spread/interleave/interlace

/// Contract bits along the 32-bit unsigned integer.
inline U32 Morton32_CompactBits3( U32 x ) {
  x &= 0x09249249; // 0b...01001001
  x = (x ^ (x >>  2)) & 0x030c30c3;
  x = (x ^ (x >>  4)) & 0x0300f00f;
  x = (x ^ (x >>  8)) & 0xff0000ff;
  x = (x ^ (x >> 16)) & 0x000003ff;
  return x;
}//shrink/contract/compact/deinterleave

/// Dilated integer addition.
inline Morton32 Morton32_Add( const Morton32 a, const Morton32 b ) {
  const Morton32 xxx = ( a | ISOLATE_YZ ) + ( b & ISOLATE_X );
  const Morton32 yyy = ( a | ISOLATE_XZ ) + ( b & ISOLATE_Y );
  const Morton32 zzz = ( a | ISOLATE_XY ) + ( b & ISOLATE_Z );
  return (xxx & ISOLATE_X) | (yyy & ISOLATE_Y) | (zzz & ISOLATE_Z); // masked merge bits
}

inline const Int3 Morton32_Decode( const Morton32 _code ) {
  return Int3(
    Morton32_CompactBits3(_code >> 0u),
    Morton32_CompactBits3(_code >> 1u),
    Morton32_CompactBits3(_code >> 2u)
  );
}

mxFORCEINLINE U32 Morton32_GetDepthBit( const Morton32 _code ) {
  mxASSERT2(_code, "Invalid (zero) Morton code! At least one bit must be set!");
  DWORD msb;
  // Find the first set bit (1) from most significant bit (MSB) to least significant bit (LSB).
  _BitScanReverse( &msb, _code );
  return msb;
}

/// returns cell coordinates (XYZ) and depth (W)
inline const Int4 Morton32_DecodeOctreeCode( const Morton32 _cellAddress ) {
  const U32 cellDepth = Morton32_GetCellDepth( _cellAddress );
  mxASSERT( cellDepth < MORTON32_MAX_OCTREE_LEVELS );
  const U32 depthMask = (1u << (cellDepth * 3u));// the index of the depth bit
  const U32 getBitsBelowDepthBit = depthMask - 1u;       // mask to extract bits below the depth bit
  const Morton32 cellAddressWithoutDepth = _cellAddress & getBitsBelowDepthBit;
  const Int3 cellCoords = Morton32_Decode( cellAddressWithoutDepth );
  return Int4::Set( cellCoords, cellDepth );
}

/// Calculates the bounds of an octree cell given its Morton code and the bounds of the whole octree.
inline const AABBf AABB_From_Morton( const Morton32 _address, const AABBf& _rootBounds ) {
  const Int4 cellCoords = Morton32_DecodeOctreeCode( _address );
  const U32 gridRes = (1u << cellCoords.w);      // grid resolution at this depth
  const V3f cellSize = _rootBounds.FullSize() / (float)gridRes;
  AABBf cellBounds;
  cellBounds.mins = V3f::Mul( cellSize, V3f::FromXYZ( cellCoords ) ) + _rootBounds.mins;
  cellBounds.maxs = cellBounds.mins + cellSize;
  return cellBounds;
}
```

3. Программная реализация предложенного алгоритма триангуляции линейных октодеревьев

Далее приведёна простейшая работающая реализация предложенного во второй главе восходящего нерекурсивного алгоритма адаптивной триангуляции. На вход алгоритма подаётся линейное знакоопределённое октодерево, которое может быть получено из стандартного октодерева (см. подраздел 2.3.1) или построено «на лету». Выходными данными является четырёхугольная сетка.

В данной реализации координаты вершин ячеек квантуются до восьми бит, мортон-коды и ячейки хранятся в отдельных массивах (m_keys и m_cells соответственно), для нахождения ячеек используется бинарный поиск (функции FindCellIndex и LowerBound), для предотвращения «холостого» поиска используются маски активных рёбер (edgeMasks), адреса для поиска смежных ячеек строятся с помощью функции Morton32_Add без перевода мортон-кодов в координаты и обратно, для сортировки массивов мортон-кодов и ячеек по убыванию мортон-кодов используется поразрядная сортировка с возможностью раннего выхода (функция RadixSort32_3Pass_DescendingOrder). Для дальнейшего ускорения поиска ячеек целесообразно завести таблицу со смещениями и количеством ячеек на каждом уровне октодерева или организовать хранение ячеек в отдельных массивах (отдельный массив для каждого уровня октодерева).

```
struct LinearOctree
{
  struct Cell
  {
    U32         data;   //!< quantized position and signs
  public:
    mxFORCEINLINE void Initialize( const U8 _signs, const V3f& _xyz01 ) {
      this->data =
        (_signs << 0)|
        (TQuantize<8>::EncodeUNorm( _xyz01.x ) << 8u )|
        (TQuantize<8>::EncodeUNorm( _xyz01.y ) << 16u)|
        (TQuantize<8>::EncodeUNorm( _xyz01.z ) << 24u);
    }
    /// returns the normalized position of the representative vertex of this cell
    mxFORCEINLINE const V3f UnpackPosition() const {
      return V3f(
        TQuantize<8>::DecodeUNorm( (data >> 8u ) & 0xFF ),
        TQuantize<8>::DecodeUNorm( (data >> 16u) & 0xFF ),
        TQuantize<8>::DecodeUNorm( (data >> 24u) & 0xFF )
        );
    }
    /// returns the signs at the eight corners of this cell (set bits == corners inside the solid)
    mxFORCEINLINE const UINT GetCornerSigns() const {
      return this->data & 0xFF;  // extract the lowest 8 bits
    }
  };
public:
  DynamicArray< Morton32 >    m_keys;    //!< locational codes (addresses) of each cell
  DynamicArray< Cell >        m_cells;//!< only active cells (i.e. intersecting the surface) are stored
```

```
  /// side effects: reorders Morton codes and cells
  ERet Contour(
    const ContourOptions& _options,      //!< settings
    const AABBf& _octreeBounds,  //!< Scene bounds for de-quantizing vertex positions.
    Meshok::TriMesh &_mesh, //!< output mesh
    AHeap & _scratch  //!< scratchpad memory
  );
};

/// Assumes that the keys are sorted in decreasing order.
static U32 LowerBound(
  const U32 _value,      //!< searched item
  const U32* _keys,      //!< sorted in decreasing order
  const U32 _right       //!< right bound (exclusive)
  )
{
  U32 offset = 0;
  U32 length = _right;
  U32 middle = length;
  while( middle ) {
    middle = length / 2u;
    if( _keys[ middle + offset ] >= _value ) {
      offset += middle;  // item in [mid..right)
    }// else - the searched item is in [0..mid)
    length -= middle;
  }
  return offset;
}

enum { CELL_NOT_FOUND = 0 };     // we can use zero to denote an invalid index, because _startIndex is always > 0

static U32 FindCellIndex(
  const Morton32* _keys, //!< cell addresses, sorted in decreasing order
  const U32 _startIndex,
  const U32 _totalCount,
  Morton32 _key
  )
{
  mxASSERT(_startIndex > 0 && _startIndex < _totalCount);
  while( _key > 1u )     // while not the root node (which has Morton code == 1)
  {
#if 0   // Use linear search.
    for( UINT i = _startIndex; i < _totalCount; i++ ) {
      if( _keys[i] == _key ) {
        return i;
      }
    }
#else   // Use binary search.
    const U32 i = LowerBound( _key, _keys + _startIndex, _totalCount - _startIndex ) + _startIndex;
    if( _keys[i] == _key ) {
      return i;
    }
#endif
    _key = _key >> 3u;   // search one level higher
  }
  return CELL_NOT_FOUND;
}

/// These are offsets of edge-adjacent cells in dilated form.
/// We can subtract Morton codes using add-with-wrap/overflow.
static const Morton32 s_AdjacentCellOffsets[12][3] = {
  { 0x92492492, 0xb6db6db6, 0x24924924 },        //  0 (X axis)
  { 0x24924924, 0x24924926, 0x2        },        //  1 (X axis)
  { 0x2,        0x6,        0x4        },        //  2 (X axis)
  { 0x4,        0x92492496, 0x92492492 },        //  3 (X axis)
  { 0x24924924, 0x6db6db6d, 0x49249249 },        //  4 (Y axis)
  { 0x49249249, 0x4924924d, 0x4        },        //  5 (Y axis)
  { 0x4,        0x5,        0x1        },        //  6 (Y axis)
  { 0x1,        0x24924925, 0x24924924 },        //  7 (Y axis)
  { 0x49249249, 0xdb6db6db, 0x92492492 },        //  8 (Z axis)
```

```
  { 0x92492492, 0x92492493, 0x1        },        //  9 (Z axis)
  { 0x1,        0x3,        0x2        },        // 10 (Z axis)
  { 0x2,        0x4924924b, 0x49249249 },        // 11 (Z axis)
};
#if 0 // How the above table was derived:
for( UINT iCubeEdge = 0; iCubeEdge < 12; iCubeEdge++ ) {
  for( UINT iAdjCell = 0; iAdjCell < 3; iAdjCell++ ) {
    const Int3 adjCellOffset = NEIGHBORS_SHARING_EDGE[ iCubeEdge ][ iAdjCell ];
    //NOTE: we can subtract dilated Morton codes by unsigned-add-with-overflow
    const U32 xmask = adjCellOffset.x == 1 ? (1u << 0u) : adjCellOffset.x == 0 ? 0 : ISOLATE_X;
    const U32 ymask = adjCellOffset.y == 1 ? (1u << 1u) : adjCellOffset.y == 0 ? 0 : ISOLATE_Y;
    const U32 zmask = adjCellOffset.z == 1 ? (1u << 2u) : adjCellOffset.z == 0 ? 0 : ISOLATE_Z;
    const U32 dilatedOffset = zmask | ymask | xmask;
    s_AdjacentCellOffsets [ iCubeEdge ][ iAdjCell ] = dilatedOffset;
    DBGOUT( "0x%x,\n", dilatedOffset );
  }
}
#endif

ERet LinearOctree::Contour(
  const ContourOptions& _options,        //!< contouring settings
  const AABBf& _octreeBounds,    //!< Scene bounds for de-quantizing vertex positions.
  Meshok::TriMesh &_mesh, //!< output mesh
  AHeap & _scratch  //!< scratchpad memory
  )
{
  mxASSERT(!_mesh.IsValid());
  const UINT numCells = m_keys.Num();

  Morton32 * __restrict  keys = m_keys.Raw();
  Cell * __restrict      cells = m_cells.Raw();

  // 1. Sort both Morton codes and cells by decreasing Morton codes (cell's addresses).
  // After sorting, the smallest cells (deepest in the hierarchy) will be placed at the beginning.
  {
    Morton32 *   tempKeys;
    mxTRY_ALLOC_SCOPED( tempKeys, numCells, _scratch );
    Cell *       tempCells;
    mxTRY_ALLOC_SCOPED( tempCells, numCells, _scratch );
    //NOTE: the array is usually (nearly) sorted so it's beneficial to use sorting with early exit.
    RadixSort32_3Pass_DescendingOrder( keys, tempKeys, cells, tempCells, numCells );
  }
  // Create edge masks (a 12-bit edge mask stores, for each cell, the set of currently active edges
  U16 * edgeMasks;       // (if edge mask == 0, the cell won't create any quads)
  mxTRY_ALLOC_SCOPED( edgeMasks, numCells, _scratch );

  // 2. Create vertices.
  mxDO(_mesh.vertex_positions.SetNum( numCells ));
  for( UINT iCell = 0; iCell < numCells; iCell++ )
  {
    const Cell& cell = cells[ iCell ];

    // Initialize edge masks.
    const U32 cornerSigns = cell.GetCornerSigns();
    edgeMasks[ iCell ] = CUBE_EDGES_LUT.edgeMask[ cornerSigns ];

    const Morton32 address = keys[ iCell ];
    // Unpack the vertex position.
    const AABBf cellAABB = AABBf_From_Morton( address, _octreeBounds );
    const V3f vertexPosition = AABBf_GetOriginalPosition( cellAABB, cell.UnpackPosition() );
    _mesh.vertex_positions[ iCell ].xyz = vertexPosition;
  }//For each cell cell.

  // 3. Create quads.
  // For each cell, in the sorted order (from smallest to largest == by decreasing Morton codes):
  for( UINT iCurrentCell = 0; iCurrentCell < numCells - 1; iCurrentCell++ )
  {
    const Cell& currentCell = cells[ iCurrentCell ];
    U32 edgeMask = edgeMasks[ iCurrentCell ];
    if( !edgeMask ) {
      continue; // skip, because all the intersected edges in the cell have been visited
```

```
    }
    const Morton32 cellAddress = keys[ iCurrentCell ];
    const U32 cellDepthBit = Morton32_GetDepthBit( cellAddress ); // get position of the depth bit
    const U32 cellDepthBitMask = (1u << cellDepthBit);            // a mask with only the depth bit set
    const U32 getBitsBelowDepthBit = cellDepthBitMask - 1;        // mask to extract bits below the depth bit
    const Morton32 addressWithoutDepth = cellAddress & getBitsBelowDepthBit;
    const U32 getBitsAboveDepthBit = ~getBitsBelowDepthBit;       // mask for validating Morton codes

    const UINT cornerSigns = currentCell.GetCornerSigns();

    // Create quads for each active edge of the current cell.
L_NextEdge:
    while( edgeMask )
    {
      const U32 iCurrentEdge = ExtractNextBit( edgeMask ); // index of the next active edge of the cell, [0..12)

      U32       adjacentCells[3];       //!< indices of the other three cells sharing the edge
      UINT numAdjacentCells = 0;

      for( UINT iAdjacentCell = 0; iAdjacentCell < 3; iAdjacentCell++ )
      {
        const Morton32 adjCellOffset = s_AdjacentCellOffsets[ iCurrentEdge ][ iAdjacentCell ];
        const Morton32 adjCellAddress = Morton32_Add( addressWithoutDepth, adjCellOffset );
        // Check for overflow:
        // after addition, Morton codes can 'wrap' around, connecting cells across the octree boundary and causing
stretched polys.
        if( adjCellAddress & getBitsAboveDepthBit ) {
          // the neighbor's address is out-of-bounds (which means the neighbor lies deeper in the octree than this
cell)
          goto L_NextEdge; // cannot happen, because all cells are sorted by decreasing addresses
        }
        const Morton32 adjCellSearchKey = adjCellAddress | cellDepthBitMask;      // search for a neighbor of the
same size or greater
        //NOTE: could accelerate the search for the cell's siblings using a small LUT
        const U32 adjCellIndex = FindCellIndex( keys, iCurrentCell + 1/*skip this cell and all smaller*/, numCells,
adjCellSearchKey );
        if( adjCellIndex != CELL_NOT_FOUND ) {
          adjacentCells[ numAdjacentCells++ ] = adjCellIndex;  // found an edge-adjacent cell
          // Mark the adjacent edge as processed.
          const UINT iCubeEdgeOnAdjCell = EdgeOnNextAdjacentCube( iCurrentEdge, iAdjacentCell + 1 );
          edgeMasks[ adjCellIndex ] &= ~(1u << iCubeEdgeOnAdjCell);       // remove this edge from the edgemask of
the adjacent cell
        } else {
          goto L_NextEdge;       // edge-adjacent cell not found - no quads should be formed for this edge
        }
      }//For each edge-adjacent cell.

      //NOTE: the four cells sharing the edge may contain a duplicate cell (corresponding to the largest cell)
      if( numAdjacentCells == 3 )
      {
        // determine orientation of the quad from the signs of the edge's endpoints
        // signs must be different => only one point should be examined
        const UINT iPointA = CUBE_EDGE[ iCurrentEdge ][0];

        if( cornerSigns & BIT(iPointA) ) {
          // The first corner is inside the surface, the second is outside
          ADC::AddQuad( iCurrentCell, adjacentCells[0], adjacentCells[1], adjacentCells[2], _mesh );
        } else {
          // The second corner is inside the surface, the first is outside
          ADC::AddQuad( adjacentCells[2], adjacentCells[1], adjacentCells[0], iCurrentCell, _mesh );
        }
      }//If the edge is shared by 4 cells.
    }//For each active edge of the current cell.

    // All the intersected edges in this cell have been processed.
    edgeMasks[ iCurrentCell ] = 0;       // Mark this cell as processed.
  }//For each cell.

  return ALL_OK;
}
```

```
// Based on bx::radixSort() by Branimir Karadzic, https://github.com/bkaradzic/bx

#define BX_RADIXSORT_BITS 11
#define BX_RADIXSORT_HISTOGRAM_SIZE (1<<BX_RADIXSORT_BITS)
#define BX_RADIXSORT_BIT_MASK (BX_RADIXSORT_HISTOGRAM_SIZE-1)

template< typename Ty >
void RadixSort32_3Pass_DescendingOrder(
  uint32_t* __restrict _keys, uint32_t* __restrict _tempKeys,
  Ty* __restrict _values, Ty* __restrict _tempValues,
  uint32_t _size )
{
  uint32_t* __restrict keys = _keys;
  uint32_t* __restrict tempKeys = _tempKeys;
  Ty* __restrict values = _values;
  Ty* __restrict tempValues = _tempValues;

  uint32_t histogram[BX_RADIXSORT_HISTOGRAM_SIZE];
  uint16_t shift = 0;
  uint32_t pass = 0;
  for (; pass < 3; ++pass)
  {
    memset(histogram, 0, sizeof(histogram));

    bool alreadySorted = true;
    {
      uint32_t key = keys[0];
      uint32_t prevKey = key;
      for (uint32_t ii = 0; ii < _size; ++ii, prevKey = key)
      {
        key = keys[ii];
        uint16_t index = (key>>shift)&BX_RADIXSORT_BIT_MASK;
        ++histogram[index];
        alreadySorted &= (prevKey >= key);
      }
    }
    if (alreadySorted)
    {
      goto L_done;
    }

    uint32_t offset = 0;
    for( int ii = BX_RADIXSORT_HISTOGRAM_SIZE - 1; ii >= 0; --ii )
    {
      uint32_t count = histogram[ii];
      histogram[ii] = offset;
      offset += count;
    }
    for (uint32_t ii = 0; ii < _size; ++ii)
    {
      uint32_t key = keys[ii];
      uint16_t index = (key>>shift)&BX_RADIXSORT_BIT_MASK;
      uint32_t dest = histogram[index]++;
      tempKeys[dest] = key;
      tempValues[dest] = values[ii];
    }

    uint32_t* swapKeys = tempKeys;
    tempKeys = keys;
    keys = swapKeys;

    Ty* swapValues = tempValues;
    tempValues = values;
    values = swapValues;

    shift += BX_RADIXSORT_BITS;
  }

L_done:
  if (0 != (pass&1) )
  {
```

```
    // Odd number of passes needs to do copy to the destination.
    memcpy(_keys, _tempKeys, _size*sizeof(uint32_t) );
    for (uint32_t ii = 0; ii < _size; ++ii)
    {
      _values[ii] = _tempValues[ii];
    }
  }
}
```

Ниже приведён псевдокод разработанного алгоритма триангуляции:

```
// Алгоритм триангуляции знакоопределённого линейного октодерева:
void contour( SignedLinearOctree octree /*input*/, TriangleMesh mesh /*output*/ )
{
  // Отсортировать все ячейки по кодам Мортона в убывающем порядке
  sort_cells_by_decreasing_addresses( octree.cells )
  // В силу конструкции мортон-кодов (позиция старшего бита зависит от глубины залегания
  // ячейки в октодереве) ячейки наименьшего размера (расположенные на максимальной
  // глубине разбиения) окажутся в самом начале отсортированного массива.

  // Для каждой ячейки:
  for each cell in octree.cells {
    // создать соответствующую вершину полигональной сетки
    mesh.create_vertex( from: cell.representative_vertex_position )
  }
  // Для каждой ячейки, кроме последней:
  for( int i = 0; i < octree.cells.count - 1; i ++ )
  {
    const Cell cell = octree.cells[i ]  //!< текущая ячейка
    // Получить маску активных рёбер ячейки из знаков углов ячейки:
    const int active_edges_mask = intersecting_cube_edges_LUT[ cell.corner_signs ]
    // Рассмотреть каждое активное ребро ячейки:
    for each edge in active_edges_mask
    {
      // Этот массив хранит индексы найденных соседних ячеек:
      [int]  neighbor_indices;  // данный массив может содержать до 3 элементов
      // Для данного активного ребра ячейки "cell" найти в октодереве остальные три ячейки
      // равного или большего размера, включающие данное ребро и смежные к "cell":
      for( int j = 0; j < 3; j++ ) {
        // Начать поиск со следующей за "cell" ячейки:
        const int neighbor_index = find_cell( edge_adjacent_to: cell.address, at: j
                                              , in: octree, starting_from: cell_index )
        // Если нашли ячейку по такому адресу (равного или большего размера, чем текущая ячейка):
        if( is_valid_cell( neighbor_index, in: octree ) ) {
          neighbor_indices.append( neighbor_index )
        }
      }//For each edge-adjacent neighbor.

      // Если найдены остальные три смежные к "cell" ячейки, которые включают ребро "edge",
      // то создать четырёхугольник, соединяющий вершины "cell" и найденных смежных к ней ячеек.
      if neighbor_indices.count == 3 {
        // Ориентация четырёхугольника выбирается согласно знакам на концах ребра "edge".
        // Для получения треугольной сетки четырёхугольник разбивается на два треугольника.
        // Вырожденные треугольники (с совпадающими индексами двух вершин) должны быть удалены.
        create_quad( i, neighbor_indices[0], neighbor_indices[1], neighbor_indices[2], mesh )
      }
    }//For each active edge of the cell.
  }//For each cell.
}
```

Приложение В

Псевдокод алгоритма адаптации выбора уровней детализации

```
// 0 - индекс самого детального уровня детализации
MAX_LODs = 16; // число уровней детализации == максимальная глубина октодерева
using ChunkID = U64; // уникальный идентификатор блока (64-битное число)
using Morton64 = U64; // код Мортона, полученный перемешиванием бит координат из ChunkID

// Представляет листовой узел в октодереве (которое является иерархией уровней детализации).
struct Node {
  Mesh* mesh;  // memory-resident data
};
// Представляет собой отдельный уровень детализации (все узлы/блоки октодерева на этом уровне).
struct LoD {
  Array< Morton64 > node_keys;       // адреса узлов в отсортированном порядке
  Array< Node* >    node_refs;       // указатели на узлы, синхронизированы с массивом выше
  int               node_count;      // число узлов на данном уровне детализации
}
/// Представляет собой LoD-иерархию.
struct Octree {
  int           next_LoD_to_update;  // индекс уровня детализации, который нужно обновить
  LoD           lods[ MAX_LODs ];    // массив с уровнями детализации
  Pool< Node >  nodes;       // память для хранения узлов
}

/// Алгоритм адаптации иерархии уровней детализации:
function update_next_level_of_detail( Octree octree, const Observer eye )
{
  // Индекс текущего уровня детализации, который нужно обновить:
  const int current_LoD_index = octree.next_LoD_to_update;
  // На каждом шаге обновляется только один уровень детализации (и сначала более детальные LoDы):
  octree.next_LoD_to_update = (octree.next_LoD_to_update + 1) % MAX_LODs;
  const LoD& lod = octree.lods[ current_LoD_index ]; // текущий уровень детализации
  if !lod.node_count {
    return;  // на текущем уровне детализации пока не создано узлов/блоков
  }
  const int coarser_LoD_index = current_LoD_index + 1; // индекс более грубого LoD
  // Данные для слияния дочерних узлов в родительский узел.
  Morton64 last_parent_key = 0;// адрес родительского узла (поставлен в инвалидное значение)
  int current_run_length = 0;// количество дочерних узлов у текущего родительского узла
  int first_child_index = 0; // индекс первого потомка текущего родительского узла
  // Для каждого листового узла на текущем уровне детализации (который нужно обновить):
  for( int i = 0; i < lod.node_count; i++ ) {
    const Morton64 node_key = lod.node_keys[ i ];
    const ChunkID chunk_id = node_key_to_chunk_id( node_key, current_LoD_index );
    Node * chunk = lod.node_refs[ i ];
    // Проверить, следует ли разбить данный узел на 8 дочерних:
    if is_too_coarse( chunk_id, relative_to: eye ) {
      if current_LoD_index > 0 { // 0 == индекс самого нижнего и детального уровня детализации
        request_split( chunk_id, chunk ); // разбить узел, т.е. увеличить уровень детализации
      } else {
        // достигнут максимальный уровень детализации, дальнейшее разбиение узла невозможно
      }
    } else  { // Проверить, могут ли узлы быть слиты в родительский узел.
      // Составить адрес родительского узла:
      const Morton64 parent_node_key = ( node_key >> 3 ); // shift out the lowest octant
      if parent_node_key == last_parent_key {
        current_run_length++; //
```

```
      } else {
        // Прошли все дочерние узлы текущего родительского узла. Попытаться их объединить.
        if current_run_length == 8 {
          const ChunkID parent_id = node_key_to_chunk_id( last_parent_key, coarser_LoD_index );
          try_merge_children( parent_id, first_child_index, eye );
        }
        // Подготовиться к слиянию дочерних узлов следующего родительского узла.
        last_parent_key = parent_node_key;
        current_run_length = 1;
        first_child_index = i;
      }
    }
  }//For each world chunk.
  // Проверить, требуется ли слияние оставшихся узлов.
  if current_run_length == 8 {
    const ChunkID parent_id = node_key_to_chunk_id( last_parent_key, coarser_LoD_index );
    try_merge_children( parent_id, first_child_index, eye );
  }
}
/// Оценивает LoD-критерий для (ещё не созданного) узла с заданным идентификатором,
/// и запрашивает слияние дочерних для него узлов, если LoD-критерий выполняется.
function try_merge_children( const ChunkID parent_id, int first_child_index, const Observer eye)
{
  if !is_too_coarse( parent_id, relative_to: eye ) {      // Если достаточный уровень детализации
    request_merge_children( parent_id, first_child_index ); // слить потомки в более грубый узел
  }
}
```

Приложение Г

Свидетельство о государственной регистрации программы для ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2018612122

«Система моделирования и визуализации воксельных ландшафтов»

Правообладатель: ***Шакаев Вячеслав Дмитриевич (RU)***

Автор: ***Шакаев Вячеслав Дмитриевич (RU)***

Заявка № **2017663957**
Дата поступления **28 декабря 2017 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***13 февраля 2018 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Г.П. Ивлиев