Федеральное государственное автономное образовательное учреждение
высшего образования
«Санкт-Петербургский политехнический университет Петра Великого»

На правах рукописи

Беляевский Кирилл Олегович

МЕТОДЫ И АЛГОРИТМЫ ФОРМИРОВАНИЯ И ИСПОЛЬЗОВАНИЯ ОКТОДЕРЕВА ДЛЯ ОБРАБОТКИ ОБЛАКА ТОЧЕК ЛАЗЕРНОГО СКАНИРОВАНИЯ В ОГРАНИЧЕННОМ ОБЪЕМЕ ОПЕРАТИВНОЙ ПАМЯТИ

Специальность 05.13.01 – Системный анализ, управление и обработка информации (технические системы)

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
д.т.н, профессор
Мелехин Виктор Федорович

Санкт-Петербург – 2020

Оглавление

Введение

Актуальность темы диссертации. Одним из наиболее прогрессивных методов сбора цифровой пространственной информации на текущее время является лазерное сканирование (ЛС). Главными достоинствами устройств лазерного сканирования считаются высокая точность, возможность автоматизации процесса сбора данных, высокая разрешающая способность измерений. Результатом таких измерений является трехмерное облако точек, с высокой точностью отражающее геометрию объекта исследования. Современные лазерные сканеры могут быть компактными и мобильными, их точность и скорость работы на текущий момент выросла на порядок, а в некоторых случаях на несколько порядков.

Лазерные сканирующие системы позволяют производить миллионы измерений в секунду, а размеры получаемых облаков точек могут достигать нескольких сотен гигабайт, что предъявляет высокие требования к вычислительным ресурсам при обработке таких данных. В результате имеет место очевидное противоречие между быстрым развитием технологий ЛС и возможностями обработки результатов, на разрешение которого направлена настоящая работа.

Используемые стандартные методы обработки данных лазерного сканирования не позволяют в полной мере использовать возможности ЛС. На текущий момент существует и используется множество алгоритмов обработки облаков точек, например, предназначенных для построения полигональной поверхности или цифровой модели рельефа, сегментации по различным признакам, объединения нескольких облаков точек, сбора статистической информации, фильтрации, построения проекций и т.п. Обработка результатов лазерного сканирования в настоящее время зачастую связана с конкретным разработчиком/производителем оборудования для ЛС и программного обеспечения (ПО), предназначенного для этого.

Среди открытого ПО наблюдается нехватка инструментов для обработки облаков точек, чей размер превышает доступную оперативную память (ОП). Так как объемы доступной оперативной памяти могут варьироваться, и зачастую не способны вместить требуемое количество данных, а рост объемов оперативной памяти значительно отстает от роста объемов облаков точек, вызванного развитием технологий ЛС, задача организации обработки облака точек при ограниченном объеме оперативной памяти является актуальной.

В диссертационной работе представлены результаты разработки методов и алгоритмов решения задачи обработки больших облаков точек с использованием структурирования информации при помощи октодерева и различных подходов к выделению (аллокации) памяти, что позволяет разрешить противоречие между все возрастающим объемом данных ЛС и ограничений по оперативной памяти.

Степень разработанности темы исследования. Октодерево широко используется в различных областях, связанных с большими объемами трехмерных данных, начиная с момента представления его в начале 80-х годов. В середине 80-х были рассмотрены точки в качестве примитивов визуализации для моделей с замкнутыми поверхностями. На текущий момент получение сцен реального мира с использованием технологии трехмерного сканирования является стандартной техникой, область применения которой варьируется от сканирования механических деталей до сканирования памятников исторического наследия и городов. Среди выдающихся исследователей указанных направлений присутствуют Мигер Д., Левой М., Уитед Т., Русинкевич Ш., Цвикер М., Паули М., Нолл О., Гросс М., Виттер Д. С., Самет Х., Шибауер К., Шуц М.

Однако, на текущий момент, ввиду существенного роста объемов облаков точек, в системах обработки данных не хватает оперативной памяти, а при использовании внешней памяти (ВП) существенно снижается производительность. В существующих широко используемых открытых программах и библиотеках обработки облаков точек предполагается их размещение в оперативной памяти. При разработке средств повышения производительности при размещении облака точек во внешней памяти актуально предусмотреть использование этих средств со стандартными программами обработки. Таким образом, исследования в области формирования октодерева для обработки больших облаков точек, позволяющего сохранить приемлемую производительность обработки при росте размера облака, сократить нагрузку на файловую систему, а также обеспечить возможность интеграции с другими программными решениями, являются актуальным и перспективным направлением научных исследований.

Целью работы является снижение затрат времени на обработку облака точек при хранении во внешней памяти за счет новых методов, алгоритмов и программного обеспечения организации хранения и доступа к этой информации, снижающих потребление оперативной памяти. Для достижения цели в диссертационной работе поставлены и решены следующие **задачи**:

1. Системный анализ процессов формирования и использования октодерева по облаку точек ЛС при использовании внешней памяти, включающий анализ методов, алгоритмов и структур данных, применяемых для обработки в оперативной и внешней памяти, их декомпозицию на составные компоненты, исследование их взаимодействия между собой, центральным процессором, внешней и оперативной памятью с целью выделения компонентов, за счет новой организации которых можно уменьшить затраты времени.

2. Постановка задачи снижения затрат времени на обмен с внешней памятью, определение критерия и показателей для оценивания эффективности реализации вычислительного процесса обработки. Формулирование гипотезы об организации вычислительного процесса обработки и структур данных, позволяющей сократить временные затраты на использование внешней памяти, и формирование подзадач на последующее исследование и уточнение предположений, основанных на данной гипотезе.

3. Разработка новых методов организации вычислительного процесса обработки облака точек, основанных на выдвинутой гипотезе. Разработка компонентов, алгоритмов и структур данных таких систем, исследование взаимодействий между ними, системным ПО, оперативной и внешней памятью.

4. Планирование и проведение экспериментальных исследований с целью оценки эффективности предложенных методов для различных задач обработки облака точек и подтверждения правильности выдвинутой гипотезы. Определение показателей, характеризующих вычислительный процесс обработки облака точек при использовании внешней памяти, и разработка вычислительных экспериментов для получения этих показателей и сравнения с существующими реализациями. Разработка методики выбора параметров октодерева в зависимости от целевой направленности обработки. Исследование возможности применения предложенных решений в сторонних программных библиотеках обработки облаков точек, ориентированных на работу в оперативной памяти, и разработка соответствующих практических рекомендаций.

Объектом исследования является структура данных облака точек ЛС при использовании внешней памяти и процесс обработки этой информации.

Предметом исследования являются методы и алгоритмы формирования октодерева облаков точек при использовании внешней памяти и организация управления обработкой информации, обеспечивающая уменьшение затрат времени на обмены с внешней памятью.

Методология и методы диссертационного исследования. В качестве методической и теоретической основы в данном диссертационном исследовании используются методы системного анализа и обработки информации, теория множеств и отношений. При разработке архитектуры программного обеспечения применяется объектно-ориентированный подход.

Положения, выносимые на защиту. На основе проведенной работы и ее экспериментальной апробации на защиту выносятся следующие положения:

1. Концептуальные модели формирования октодерева и вычислительного процесса обработки облака точек во внешней памяти, предназначенные для выделения компонентов и этапов, модификацией которых с учетом особенностей структуры облака точек можно сократить затраты времени при обработке.

2. Гипотеза об уменьшении затрат времени на обращения к внешней памяти при обработке облака точек за счет изменения способов доступа, размещения и идентификации блоков данных октодерева во внешней памяти, позволяющих сократить количество файловых операций и создаваемых файлов, а также снизить количество задержек, обусловленных файловой системой. Сформированные на базе данной гипотезы предположения об организации вычислительного процесса позволили выделить этапы и компоненты обработки, которые стали предметом детального исследования, что позволило предложить новые методы обработки, являющиеся основными результатами.

3. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе асинхронной двухуровневой системы кеширования, использующий внешнюю память при превышении обрабатываемой информацией объема оперативной памяти и позволяющий сократить количество создаваемых файлов для снижения влияния обменов с внешней памятью на производительность.

4. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе механизма отображения памяти, использующий внешнюю память для хранения информации с возможностью прямого доступа и интерактивной модификации, а также позволяющий ускорить выполнение операций доступа к данным октодерева за счет кодирования с использованием целочисленной арифметики и сократить количество создаваемых файлов до одного для снижения влияния обменов с внешней памятью на производительность.

Научная новизна предлагаемой диссертации состоит в следующем:

1. Разработаны концептуальные модели организации обработки облака точек, формирования октодерева, компонентов вычислительного процесса обработки облака точек во внешней памяти, позволившие за счет декомпозиции этих процессов выделить операции, выполнение которых с использованием предложенных способов организации доступа к блокам данных дало возможность существенно ускорить обработку. Разработаны модели вычислительного процесса обработки облака точек, анализ которых позволил предложить ряд модификаций, повышающих эффективность процесса обработки.

2. В результате анализа организации данных и вычислительного процесса предложена модель иерархической структуры данных октодерева, использующаяся при обработке облака точек в оперативной или внешней памяти и позволяющая ускорить выполнение операций доступа к данным октодерева, а также отличающаяся возможностью масштабирования для работы с данными произвольной размерности и возможностью динамического расширения структуры октодерева без увеличения его глубины.

3. Разработаны два метода, позволяющие добиться снижения временных затрат на использование внешней памяти при ограничении потребления оперативной памяти в процессе обработки облака точек, основанные на выдвинутой гипотезе о изменении способов доступа, размещения и идентификации блоков данных октодерева во внешней памяти. Первый метод использует асинхронную систему кеширования, модифицированную процедуру формирования октодерева и отличается возможностью присоединения узлов октодерева к общему файлу. Как следствие, он характеризуется сокращением количества создаваемых в процессе формирования файлов и уменьшением затрачиваемого времени на обмен данными. Второй метод использует механизм отображения памяти совместно с алгоритмом динамической аллокации, а также применяет целочисленную иерархическую модель октодерева для ускорения доступа к данным. Отличается возможностью прямого доступа к данным во внешней памяти, сокращением количества создаваемых в процессе формирования файлов до одного и, как следствие, уменьшением затрачиваемого времени на обмен данными.

4. Предложен способ обработки больших облаков точек в сторонних библиотеках, ориентированных на работу в оперативной памяти, позволяющий помимо ограниченной оперативной памяти использовать внешнюю память, обеспечивая прямой доступ к данным за счет механизмов отображения памяти и динамической аллокации. Получены экспериментальные результаты, подтверждающие возможность такой обработки без существенного падения производительности в сравнении с обработкой информации только в основной памяти и доказывающие возможность работы с облаком точек, превышающим объемы оперативной памяти.

Теоретическая значимость работы. Разработаны концептуальные модели организации обработки облака точек, формирования октодерева и компонентов вычислительного процесса обработки облака точек во внешней памяти, позволившие провести анализ процесса обработки и выделить этапы и компоненты, модификация которых с учетом структуры данных и новых способов кодирования доступа позволила повысить производительность при использовании внешней памяти для октодерева.

Практическая ценность работы заключается в создании программной системы, предназначенной для ограничения потребления оперативной памяти в процессе обработки облаков точек и подтверждающей теоретические результаты работы. Предложенные подходы, методы и алгоритмы позволяют кратно снизить потребление оперативной памяти при выполнении типовых операций обработки облаков точек, сохраняя при этом приемлемую производительность.

Обоснованность и достоверность предложенных методов и алгоритмов подтверждается согласованностью теоретических положений и результатов, полученных при практической реализации предложенных методов и алгоритмов, апробацией основных теоретических положений диссертации в печатных трудах и докладах, а также положительными результатами внедрения основных положений диссертации.

Реализация результатов работы. Внедрение представленных алгоритмов и методов было произведено в рамках проекта ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014 - 2020 годы» по теме: «Исследование и разработка алгоритмов и программных средств по обработке, хранению и визуализации данных лазерного сканирования и фотосъемки» (Уникальный идентификатор проекта RFMEFI58417X0025) индустриальным партнером ООО «Экоскан» (Соглашение о предоставлении субсидии от 03.10.2017 г. № 14.584.21.0025). Представленные в работе методы и алгоритмы используются в качестве алгоритмического и методического обеспечения для разработки программного комплекса, предназначенного для обработки, хранения и визуализации данных лазерного сканирования.

Апробация результатов работы. Результаты исследований докладывались на следующих конференциях: 19th International Conference on Computational Science and its Applications, Saint-Petersburg, 2019; XXI Международная конференция по мягким вычислениям и измерениям, Санкт-Петербург, 2018г.

Решение апробировано компанией ООО «ЭкоСкан» в программном комплексе для обработки, хранения и визуализации данных лазерного сканирования и фотосъемки. Практическим результатом использования диссертационной работы является использование предложенного подхода для снижения загрузки оперативной памяти в процессе работы с облаками точек. Это позволяет выполнять обработку облаков точек лазерных отражений, размер которых значительно превышает доступные объемы оперативной памяти.

Зарегистрированы следующие результаты интеллектуальной деятельности (РИД): «Программа для создания растровых проекций», №2018665901, 11.12.2018; «Программа для сортировки и фильтрации облака точек», №2018666456, 17.12.2018; «Программное средство построения проекций облака точек, полученных от мобильного комплекса лазерного сканирования», № 2015617817, 22.07.2015.

Публикации. Автором опубликовано по теме диссертации 7 печатных работ, среди них 2 работы в рецензируемых журналах из перечня ВАК и 2 работы, включенные в систему цитирования Scopus, а также зарегистрировано 7 Результатов Интеллектуальной Деятельности.

Структура и объем работы. Диссертационная работа объемом в 204 машинописных страницы состоит из введения, 4 разделов, списка литературы из 98 наименований, списка сокращений и условных обозначений, 78 иллюстраций, 12 таблиц, 14 листингов, 1 приложения и предметного указателя.

В первой главе проводится анализ принципов работы систем лазерного сканирования, а также процессов получения, обработки и предобработки данных лазерного сканирования. Проводится обзор существующих методов обработки облаков точек в ограниченном объеме потребляемой оперативной памяти. Рассматриваются структуры разбиения пространства и их применение для организации обработки облака точек с использованием вторичных систем хранения данных. Формируется гипотеза об уменьшении затрат времени на обращения к внешней памяти при обработке облака точек и организации взаимодействия между оперативной и внешней памятью. Производится постановка задачи снижения затрат времени на обмен с внешней памятью, а также определение показателей для оценивания эффективности реализации вычислительного процесса обработки.

Во второй главе производится анализ и уточнение различных вариантов реализации вычислительного процесса обработки и механизмов взаимодействия оперативной и внешней памяти, предложенных в первой главе. Рассматривается организация данных в октодереве и механизмы взаимодействия оперативной и внешней памяти. Предложены структуры данных и алгоритмы, предназначенные для построения октодерева, позволяющего выполнять обработку больших облаков точек с использованием вторичных систем хранения. Рассматриваются модели октодерева на основе арифметики с плавающей точкой и целочисленной арифметики. Предложена иерархическая модель с использованием целочисленной арифметики.

В третьей главе рассматривается реализация вычислительного процесса обработки на основе системы кеширования и механизма отображения памяти. Предложен алгоритм построения октодерева с использованием системы кеширования и возможностью объединения заполненных узлов в общий файл. Для обеспечения возможности использования механизма отображения памяти в октодереве предложена реализация алгоритма динамической аллокации на отображаемой памяти. Для обеспечения возможности обработки больших облаков точек сторонними библиотеками предложен способ обработки больших облаков точек путем внедрения системы аллокации отображаемой памяти в сторонние библиотеки для linux систем.

Четвертая глава посвящена экспериментальному исследованию предложенных октодеревьев, а также алгоритмов обработки облаков точек с использованием предложенных октодеревьев. Предложенные алгоритмы сравниваются с существующими решениями, рассматриваются их недостатки и ограничения.

Глава 1. Анализ проблемы обработки облаков точек в условиях ограничений по оперативной памяти

В главе проводится анализ принципов работы систем лазерного сканирования, а также процессов получения, обработки и предобработки данных лазерного сканирования. Проводится обзор существующих методов обработки облаков точек в ограниченном объеме потребляемой оперативной памяти. Рассматриваются структуры разбиения пространства и их применение для организации обработки облака точек с использованием вторичных систем хранения данных. Формируется гипотеза об уменьшении затрат времени на обращения к внешней памяти при обработке облака точек и организации взаимодействия между оперативной и внешней памятью. Производится постановка задачи снижения затрат времени на обмен с внешней памятью, а также определение показателей для оценивания эффективности реализации вычислительного процесса обработки.

В данной главе выполняются следующие задачи диссертации:

1. Системный анализ процессов формирования и использования октодерева по облаку точек ЛС при использовании внешней памяти, включающий анализ методов, алгоритмов и структур данных, применяемых для обработки в оперативной и внешней памяти, их декомпозицию на составные компоненты, исследование их взаимодействия между собой, центральным процессором, внешней и оперативной памятью с целью выделения компонентов, за счет новой организации которых можно уменьшить затраты времени.

2. Постановка задачи снижения затрат времени на обмен с внешней памятью, определение критерия и показателей для оценивания эффективности реализации вычислительного процесса обработки. Формулирование гипотезы об организации вычислительного процесса обработки и структур данных, позволяющей сократить временные затраты на использование внешней памяти, и формирование подзадач на последующее исследование и уточнение предположений, основанных на данной гипотезе.

1.1. Понятие и характеристики данных лазерного сканирования

Лазерный сканер представляет собой систему дистанционного зондирования [1], позволяющую выполнять измерения расстояний до поверхности объекта сканирования [2] с высокой скоростью и высокой точностью. Большинство современных моделей лазерных сканеров позволяют окрашивать облако точек, используя для этого встроенную цифровую фотокамеру [3]. На текущий момент выделяют три вида лазерного сканирования: наземное лазерное сканирование (НЛС) [4], мобильное лазерное сканирование (МЛС) [5], воздушное лазерное сканирование (ВЛС) [6]. Различаются они преимущественно методами привязки к глобальной системе координат, а также точностью и плотностью финального облака точек.

НЛС незаменимо при обследованиях закрытых помещений (in-door), тоннелей пещер и т.д. Технология НЛС используется для получения обмерных чертежей фасадов, топографических планов местности масштаба 1:500 и т.д. НЛС позволяет обследовать объекты размером до 0,5–2 см с точностью до 0,5–5 мм. Производительность НЛС около 1000–4000 кв. м при съемке фасадов в масштабе 1:50 до 4–20 га при съемке топографических планов масштаба 1:500 в день и несколько уступает другим видам ЛС [7].

МЛС является одним из наиболее технологичных и производительных методов съемки местности на текущий момент. Для производства измерений используется мобильный лазерный сканер, состоящий из нескольких лазерных сенсоров и нескольких фотокамер, который устанавливается, как правило, на автомобиль или на любое другое транспортное средство (например, водное судно или один из видов железнодорожного транспорта). Для привязки результатов сканирования к заданной системе координат используется навигационный блок, установленный на движущейся платформе и состоящий из одного или нескольких ГНСС (Глобальные Навигационные Спутниковые Системы) приемников и инерциальной навигационной системы. МЛС на скорости до 100 км/ч обеспечивает точность, необходимую для съемки масштаба 1:100-1:200 и плотность до 4000 точек/м^2. Средняя производительность – до 500 погонных километров в день при ширине полосы сканирования съемки до 250 метров [7].

Воздушное лазерное сканирование (ВЛС) - это современный метод сбора геопространственной информации о местности. ВЛС применяется для высокоточного картографирования линейных и площадных объектов в масштабах 1:500–1:5000 с воздушных носителей (самолет, вертолет, автожир). Суть метода лазерного сканирования заключается в измерении множества точек, принадлежащих земной поверхности и объектам, расположенным на ней, с помощью лазерного сканера (лидара, от англ. LIDaR — laser indetification, detection and ranging), установленного на борту движущегося воздушного судна. Среди основных характеристик отметим следующие: плотность точечной модели местности — до 15-20 точек на квадратный метр; точность точечной модели местности — до 5-7 см по высоте и 7-8 см в плане; размер пикселя на местности (детализация аэрофотосъемки и составляемых по ней ортофотопланов) — до 4-5 см [7].

Данные лазерного сканирования представлены облаком точек. Облако точек - это неупорядоченное множество точек в трехмерном евклидовом пространстве, полученное в результате трехмерного сканирования объекта, и представляющее поверхность этого объекта:

$$P = \{p_1, \dots, p_s\}. \quad (1.1)$$

Точка из облака точек представляет собой вектор (кортеж) координат (a_{coord}), нормалей (a_{normal}), цвета (a_{color}) и прочей информации, использующейся при построении облака точек:

$$p_i = \left(a^i_{coord}, a^i_{color}, a^i_{normal}, \dots\right), p_i \in P, \quad (1.2)$$

где $a_{coord} = (x, y, z)$, где x, y, z – трехмерные координаты точки;

$a_{color} = (r, g, b)$, где r, g, b – красная, зеленая и синяя компоненты цвета;

$a_{normal} = (nx, ny, nz)$, где nx, ny, nz – компоненты нормали.

Современные системы сканирования могут производить измерения со скоростью более миллиона точек в секунду [8]. Таким образом, количество точек в облаках точек может достигать сотен миллионов или нескольких миллиардов. Обработка таких объемов данных предоставляет высокие требования к вычислительным ресурсам и не позволяет в полной мере использовать возможности ЛС.

В настоящее время дискурс о развитии исследований в области обработки данных ЛС является открытым и чрезвычайно актуальным, так как именно это звено является лимитирующим в случае использования ЛС. С одной стороны, появились новые средства измерения, направленные на значительное расширение областей применения информационных технологий. С другой стороны, недостаточная проработка теоретических и методологических основ обработки результатов лазерного сканирования не позволяет использовать все возможности данных съемочных систем [7].

1.2. Организация процесса обработки данных лазерного сканирования. Этапы обработки

1.2.1. Распространенные форматы представления данных лазерного сканирования

Рассматривая задачу обработки облаков точек, необходимо провести анализ существующих форматов представления облаков точек лазерного сканирования. Целью такого анализа будет выявление наиболее распространенных форматов хранения облаков точек, на обработке которых стоит сосредоточиться в первую очередь, а также оценка их эффективности. Существующие форматы хранения облаков точек делятся на открытые и проприетарные. Проприетарные используются в закрытом программном обеспечении (TerraSolid, Bentley и пр.), в силу чего не будут рассматриваться в данной работе. Основные открытые форматы хранения облака точек, в свою очередь, используются повсеместно и поддерживаются многими программами, работающими с облаком точек.

Форматы хранения данных будут рассматриваться с точки зрения их распространенности, размера в файловой системе и поддержки основных атрибутов точки лазерного сканирования. Оценка размера облака точек будет производиться путем конвертации одного облака точек в разные форматы хранения. В качестве исходного будет использоваться облако точек, содержащее 1 193 056 точек (Рисунок 1.1) и представляющее результаты сканирования промышленного объекта. Атрибуты точки облака включают в себя координату и интенсивность.

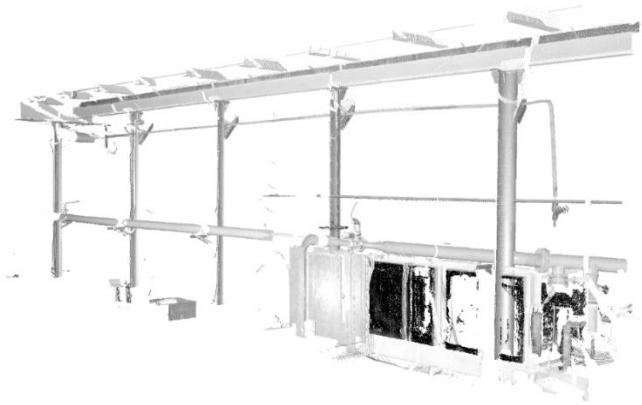

Рисунок 1.1 - Облако точек для сравнения размеров файлов

Рассмотрим основные форматы хранения облака точек.

ASCII (American Standard Code for Information Interchange) point cloud (xyz). Общий формат файлов точечных облаков, где данные пространственно представлены с использованием трехмерных координат (Таблица 1.1).

Таблица 1.1 – Характеристики формата ASCII point cloud

Категория	**Описание**
Распространенность	Средняя
Формат	Текстовый
Размер файла	65 116 Кбайт
Поддержка атрибутов точки	Общая спецификация отсутствует, поэтому содержимое файла может варьироваться в различном программном обеспечении

LASer (LAS) [9]**.** Наиболее распространенный формат хранения облака точек. Поддерживается большинством программных средств, ориентированных на работу с облаком точек (Таблица 1.2).

Таблица 1.2 – Характеристики формата LAS

Категория	**Описание**
Распространенность	Высокая
Формат	Бинарный
Размер файла	30 293 Кбайт (46% от размера ASCII point cloud)
Поддержка атрибутов точки	Поддерживает как множество встроенных типов атрибутов, так и возможность добавлять пользовательские. Среди поддерживаемых атрибутов: 10 общих классов точек и атрибуты, включающие в себя координату точки, интенсивность, количество отражений, класс и пр.

Compressed Las (LAZ) [10]**.** Вариант формата LAS, получаемый путем сжатия без потерь оригинального LAS файла (Таблица 1.3).

Таблица 1.3 – Характеристики формата LAZ

Категория	**Описание**
Распространенность	Высокая
Формат	Бинарный
Размер файла	13 226 Кбайт (20% от размера ASCII point cloud)
Поддержка атрибутов точки	Поддерживает те же возможности, что и оригинальный формат LAS

Point Cloud Data (PCD) [11]**.** Формат хранения данных используемый в библиотеке Point Cloud Library (PCL) [12]. Может быть как текстовым, так и бинарным (Таблица 1.4). В бинарной форме представляют собой полную копию облака точек, хранимого в оперативной памяти (за счет чего может быть достаточно быстро загружен).

Таблица 1.4 – Характеристики формата PCD

Категория	**Описание**
Распространенность	Низкая
Формат	Текстовый / Бинарный
Размер файла	23 303 Кбайт (35% от размера ASCII point cloud)
Поддержка атрибутов точки	Поддерживает стандартные типы данных (char, short, int, float, double) и возможность декларировать произвольный набор атрибутов в заголовке файла

E57 [13]**.** Относительно новый формат хранения облаков точек, позволяющий выполнять хранение облака точек, изображений и метаданных (Таблица 1.5). Позволяет хранить набор бинарных секций с данными и интерпретировать их при помощи иерархического описания на Extensible Markup Language (XML).

Таблица 1.5 – Характеристики формата E57

Категория	**Описание**
Распространенность	Средняя
Формат	Бинарный с текстовой (XML) частью
Размер файла	17 555 Кбайт (26% от размера ASCII point cloud)
Поддержка атрибутов точки	Поддерживает стандартные типы данных (integer, float, string), расширенные типы данных (scaled integer), бинарные данные (blob) и возможность декларировать произвольный набор атрибутов в заголовке файла

В данной работе для хранения и загрузки облаков точек используется формат LAS, как наиболее распространенный и поддерживаемый. Другим положительным качеством данного формата является то, что при необходимости обеспечения более компактного представления облаков точек возможно использование родственного формата LAZ с минимальными изменениями в исходном коде.

1.2.2. Этапы обработки данных лазерного сканирования

Процесс обработки данных лазерного сканирования может включать в себя ряд этапов, некоторые из которых выполняются полностью в автоматическом режиме, другие требуют вмешательства оператора. Состав и количество этапов также может различаться для наземного, мобильного и воздушного лазерного сканирования, а также может варьироваться в зависимости от поставленной задачи, качества исходных данных, необходимой точности. На каждом из этапов может использоваться отдельный набор алгоритмов для преобразования облака точек.

Рассмотрим обобщенное представление процесса обработки данных лазерного сканирования с целью определения этапов, которые могут потребовать применения механизмов обработки больших облаков точек.

С точки зрения процесса получения данных лазерного сканирования выделим этапы подготовки к съемке, съемки и первичной обработки данных. Сначала выполняется подготовка оборудования и планирование съемки. На этапе съемки производится калибровка, съем данных инерциальных систем и ГНСС, контрольных точек и непосредственно получение данных лазерного сканирования. На этапе первичной обработки производится совместная обработка инерциальных и ГНСС данных и контрольных точек, уравнивание облаков, устранение флуктуаций [14]. Данные этапы выполняются при помощи проприетарного программно-технического обеспечения систем лазерного сканирования, в силу чего затруднительно внесение в них каких-либо изменений.

С точки зрения процесса обработки данных лазерного сканирования, выделим этап предобработки и этап обработки облака точек. На этапе предобработки происходит загрузка облака точек во внутренний формат представления и подготовка структур данных, например, структур разбиения пространства, ускоряющих операции пространственного поиска. На этапе обработки происходит применение к загруженному облаку точек и построенным структурам данных алгоритмов обработки облака точек, включающих алгоритмы фильтрации шумов, прореживания, классификации, визуализации и прочих.

В данной работе рассматривается преимущественно этап предобработки облака точек, на котором выполняется построение структуры разбиения пространства, а также этап обработки, в котором предложенная структура разбиения применяется для выполнения обработки облаков точек в условиях ограниченного потребления оперативной памяти.

Методы, применяемые для структурирования данных ЛС и ограничения потребляемой памяти, можно представить в виде концептуальной модели (Рисунок 1.2). Как видно из рисунка, организация выборки и хранения данных может быть выполнена как с помощью структур разбиения (при необходимости ускорения операций пространственного поиска), так и путем последовательной выборки данных из неорганизованного облака точек. Далее в работе показано, что последовательная выборка имеет существенные ограничения, поэтому основное внимание в дальнейших исследованиях уделяется учету возможности хранения данных в виде октодерева и, с использованием этой особенности, поиску возможностей сокращения затрат времени на обмен данными с внешней памятью.

Рисунок 1.2 - Концептуальная модель организации обработки облака точек

1.3. Применение структур разбиения пространства для ускорения операций пространственного поиска

1.3.1. Применение операций пространственного поиска

В силу специфики лазерного сканирования, точки в облаке изначально расположены рядом, в соответствии с траекторией луча сканирования. Однако, предсказать их распределение не представляется возможным: расположение точек в облаке может быть изменено произвольным образом в результате операций совмещения облаков точек или в результате обработки. Таким образом, облако точек является неорганизованной структурой данных.

При обработке облака точек (прореживании, фильтрации, визуализации, вычислению нормалей и т.п.) обычно требуется информация о пространственном отношении между точками. Такая информация является результатом операций пространственного поиска, например: поиск ближайших соседей – поиск N ближайших соседей или соседей в определенном радиусе [15]; поиск точек в заданном объеме – объем может быть задан произвольной фигурой, но обычно это сфера, куб или параллелепипед.

В облаке, представленном последовательным массивом точек, операции поиска в общем случае требуют полного перебора облака точек, то есть вычислительная сложность составляет $O(n)$ [16]. Такой подход может быть оправдан, когда количество операций поиска невелико, но в случае применения операций поиска для каждой точки в облаке сложность подобной операции становится $O(n^2)$, что потребует применения более эффективных способов организации пространственного поиска.

Для ускорения подобных вычислений обычно используются специальные структуры данных, описывающие объем, занимаемый облаком точек, в виде набора непересекающихся областей и позволяющие идентифицировать каждую точку в облаке как принадлежащую к определенной области. При использовании подобной информации появляется возможность производить поиск с учетом данных о смежности областей, а также применять более быструю проверку на соответствие критериям поиска для групп точек, что позволяет снизить сложность поиска до $O(\log n)$ [16]. Будем называть подобную структуру данных структурой разбиения пространства [17].

1.3.2. Структуры разбиения пространства

Структуры разбиения пространства часто являются иерархическими. Это означает, что пространство (или область пространства) разделено на несколько областей, а затем одна и та же схема пространственного разбиения рекурсивно применяется к каждой из созданных таким образом областей. Области могут быть организованы в виде дерева, называемого деревом разбиения пространства. В зависимости от реализации структура разбиения может выполнять как индексацию, так и хранение облака точек. Некоторые структуры разбиения могут быть специально оптимизированы для определенного вида выборок, а некоторые не поддерживают специфичные виды поиска.

Существующие структуры разбиения пространства можно разделить на два класса [18]:

- Управляемые пространством (space-driven);
- Управляемые данными (data-driven).

Отличие между приведенными классами заключается в следующем: в первом случае производится разбиение пространства и определение принадлежности данных, во втором – разбиение данных и определение занимаемых регионов пространства.

Структуры разбиения, управляемые данными, обычно требуют меньше памяти и обеспечивают более быстрый доступ к данным, однако очень чувствительны к модификации облака точек, так как требуют перестройки при изменении данных.

Структуры разбиения, управляемые пространством, менее чувствительны к изменению облака точек, лучше подходят для использования совместно с вторичной системой хранения, так как разбиение на каждом уровне идентично, что позволяет выполнять сохранение меньшего количества информации.

Далее будут рассмотрены распространенные структуры разбиения пространства, применяемые при обработке облаков точек в условиях ограниченного потребления оперативной памяти.

1.3.3. Тайловый массив

Тайловый массив относится к структурам разбиения, управляемым пространством, в силу чего менее чувствителен к модификации данных. Данный подход может быть расширен для пространства любой размерности: в одномерном случае пространство разбивается на набор интервалов, в двумерном – прямоугольников, в трехмерном – кубов. Тайловый массив может использоваться в задачах визуализации [19].

Рассмотрим двумерный случай. Пусть дано исходное облако точек в двумерном пространстве. Построим описывающий прямоугольник этого облака точек (Рисунок 1.3).

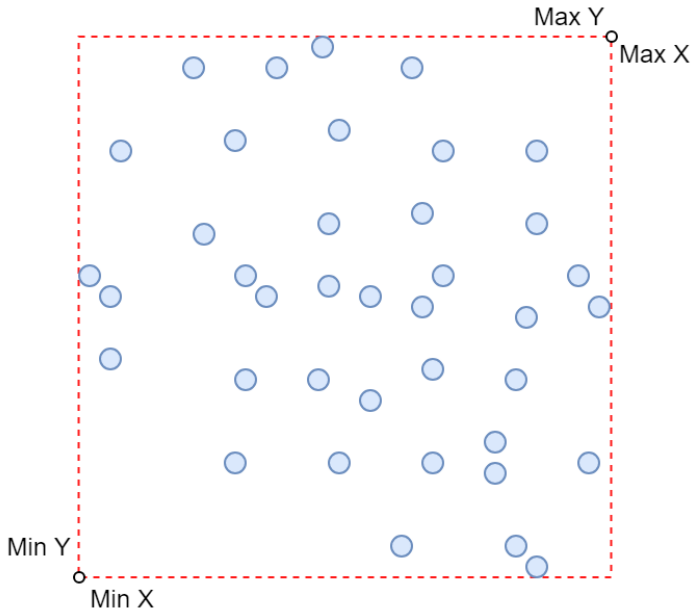

Рисунок 1.3 - Описывающий прямоугольник

Для структурирования облака точек создадим двумерный массив ячеек (Рисунок 1.4).

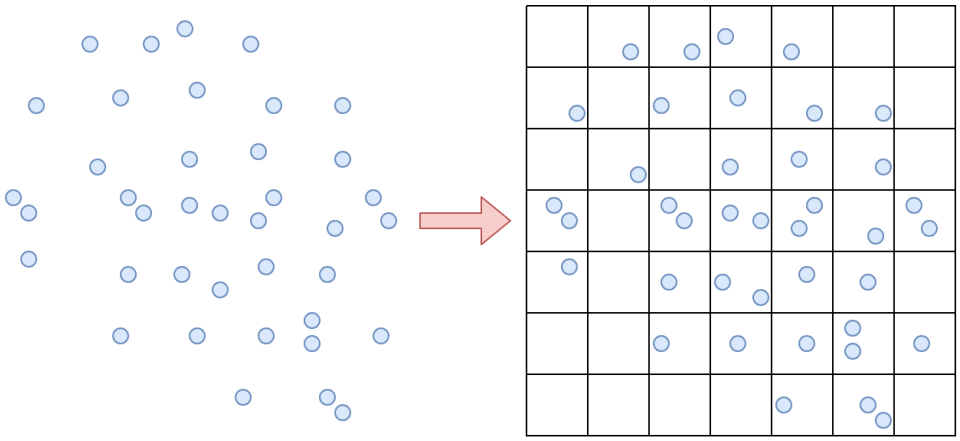

Рисунок 1.4 - Двумерный массив ячеек

В качестве примера рассмотрим операцию поиска ближайших соседей для такой структуры разбиения. Для поиска ближайших соседей точки *p* необходимо сначала найти ячейку, в которой эта точка находится. Если в этой ячейке находятся другие точки, необходимо применить алгоритм поиска для неструктурированных данных, описанный выше. Иначе необходимо повторить процедуру для соседей данной ячейки (Рисунок 1.5).

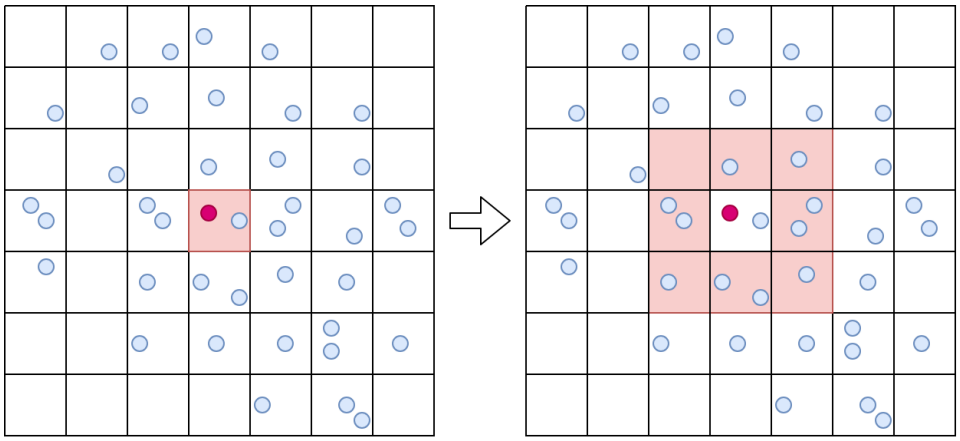

Рисунок 1.5 - Поиск соседних точек

Стоит отметить, что данный алгоритм плохо себя показывает в ряде случаев (Рисунок 1.6): когда выбрана слишком низкая плотность сетки и приходится производить сравнение с большим количеством неорганизованных точек в ячейке (а); когда искомые точки находятся слишком далеко и приходится перебирать слишком много ячеек (б) и когда плотность сетки избыточна (в).

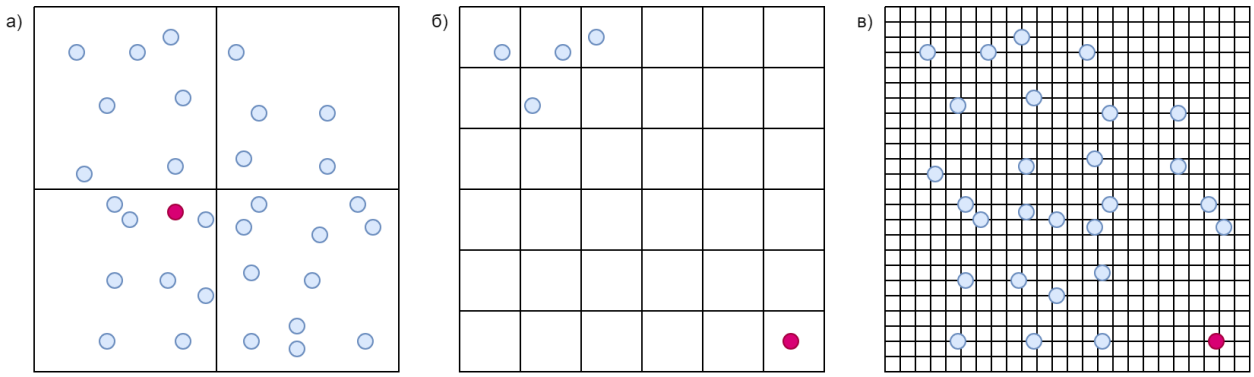

Рисунок 1.6 - Недостатки алгоритма: а - недостаточная плотность сетки, б - соседи слишком далеко, в - избыточная плотность сетки

Помимо вышеперечисленных недостатков стоит отметить большой расход памяти: зачастую для хранения всех ячеек требуется памяти больше, чем потребляет исходное облако точек.

1.3.4. К-мерное дерево

К-мерное дерево **(k-dimension tree, kd-tree)** [20] - это двоичное дерево, в котором разбиение пространства выполняется за счет рекурсивного подразбиения облака точек плоскостями на две части. Каждый не листовой узел можно рассматривать как плоскость, делящую пространство на два подпространства. Точки слева от этой плоскости представляют левое поддерево этого узла, а точки справа от плоскости представлены правым поддеревом. К-мерное дерево относится к структурам разбиения, управляемым данными (data-driven), в силу чего очень чувствительно к модификации облака точек, однако требует меньше памяти и обеспечивает более быстрый доступ к данным по сравнению с октодеревом.

Рассмотрим процесс построения k-мерного дерева для трехмерного облака точек (k=3). Пусть существует облако точек $P, |P| = n$. К-мерное дерево для P представляет собой бинарное дерево, которое рекурсивно разбивает пространство, занимаемое P. Корень дерева соответствует ограничивающему параллелепипеду (Axis-Aligned Bounding Box, AABB) облака точек P. Внутренние узлы дерева представляют собой плоскости $s_{k,x}$, которые рекурсивно разбивают пространство перпендикулярно к координатным осям. Листья содержат индексы всех точек, попадающих в определяемый ими объем. Примерный алгоритм построения k-мерного дерева можно увидеть в листинге 1.1.

Листинг 1.1 - Рекурсивное построение к-мерного дерева

```
// Принимает на вход массив точек (P) и их AABB (V)
// Возвращает внутренний узел дерева (Node) или лист (LeafNode)
Node* BuildTreeNode(Points P[], AABB V)
{
  if(ShouldFinish(P, V)) // Проверка условия остановки разбиения
     return new LeafNode(P);
  s = FindPlane(P, V); // Поиск подходящей плоскости s для разбиения P
  (V_l,V_r) = Split V with s;
  P_l = {p ∈ P | (p ∩ V_l) ≠ ∅}
  P_r = {p ∈ P | (p ∩ V_r) ≠ ∅}
  return new Node(s, BuildTreeNode(P_l,V_l), BuildTreeNode(P_r,V_r));
}
// Возвращает корневой узел дерева
Node* BuildTree(Points P[])
{
```

```
  V  = GetAABB(P); // Вычисление / Получение AABB
  return BuildTreeNode(P, V);
}
```

Обычно главная сложность при построении k-мерного дерева заключается в правильном выборе плоскости разбиения s и определении верного критерия остановки рекурсии. Наиболее часто используемым методом является разбиение по медиане, при котором ось разбиения a выбирается циклически, а плоскость расположена на пространственной медиане V по оси a:

$$a = D(V)\ mod\ 3, s = \frac{V_{max}^{a} - V_{min}^{a}}{2} + V_{min}^{a}, \quad (1.3)$$

где $D(V)$ – текущая глубина разбиения.

Разбиение обычно выполняется до тех пор, пока количество точек не упадет ниже определенного порога $K_{minPoints}$ или пока глубина разбиения не достигнет максимальной глубины $K_{maxDepth}$:

$$ShouldFinish(P, V) = |P| \leq K_{minPoints} \vee D(V) \geq K_{maxDepth}. \quad (1.4)$$

1.3.5. Октодерево

Октарные деревья широко используются при обработке облаков точек лазерного сканирования и находят применение в таких областях, как распознавание объектов [21–23], сегментация [24; 25], выделение примитивов [26], реконструкция поверхности [27], расчет нормалей [28] и пр.

Структура октодерева приведена на рисунке 1.7. Узел октодерева хранит в себе информацию о восьми потомках, а также вспомогательную информацию, которая может использоваться в процессе построения или обработки облака точек (например, координаты узла, его размер, уровень и т.п.). Листья октодерева хранят в себе информацию о точках, которые попадают в ячейку конкретного листа на данном уровне разбиения. Листья также могут содержать вспомогательную информацию [29].

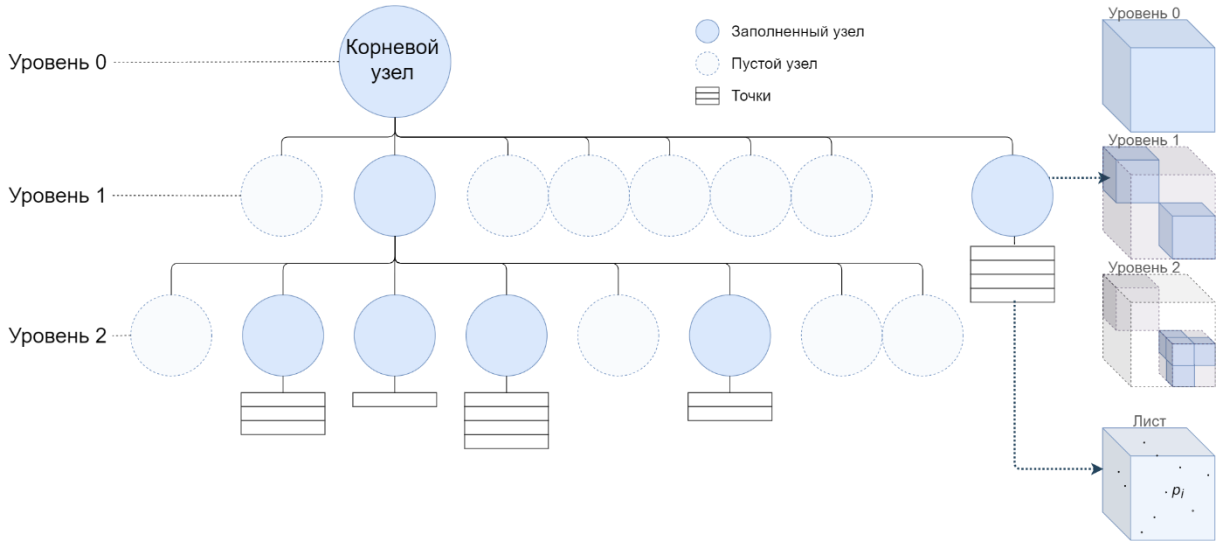

Рисунок 1.7 - Структура октодерева

Формирование октодерева производят при помощи рекурсивного добавления точек, начиная с корневого узла. На этапе построения при добавлении точек в любой узел производится проверка на удовлетворение критерия разбиения для данного узла [29]. Критерием разбиения как правило выбирается превышение заданного максимального количества точек в узле. При выполнении такого условия производится разбиение узла на восемь меньших октантов, точки родительского узла распределяются между ними по пространственному признаку. Критерием остановки служит превышение максимальной глубины дерева [29].

Часто построение октодерева выполняется в оперативной памяти. Это означает, что узлы, листья и принадлежащие им точки также будут храниться в оперативной памяти. Такое решение существенно ускоряет построение дерева, однако делает невозможным обработку облаков точек, не помещающихся в оперативную память [29].

1.3.6. Применение в алгоритмах обработки облаков точек

Рассмотрим применение структур разбиения пространства на примере алгоритма визуализации облака точек. Проблемой при визуализации облака точек является большой объем данных. Облако точек не всегда помещается в оперативную или видео память. Чтобы получить точки, непосредственно попадающие на экран, необходимо просмотреть все облако точек целиком (так как искомые точки могут быть случайным образом распределены по всему облаку). Пример визуализации облака точек можно увидеть на рисунке 1.8. На рисунке 1.9 красным цветом изображены точки, попадающие в пирамиду видимости.

Рисунок 1.8 - Точки, видимые на экране

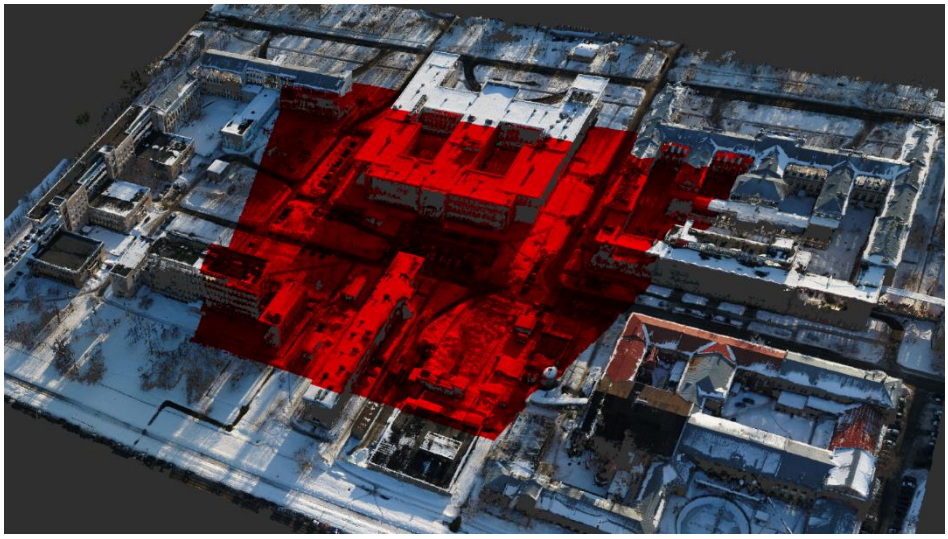

Рисунок 1.9 – Видимые точки в облаке точек

В ситуации, когда облако точек занимает несколько гигабайт, заново просматривать все облако точек каждый кадр не представляется возможным. Для решения подобной задачи обычно используются структуры разбиения пространства. Простейшим примером такой структуры является двумерная или трехмерная сетка. При построении такой структуры происходит определение попадания каждой точки из облака в ячейку сетки согласно координатам точки (Рисунок 1.10). Далее для каждой ячейки производится выборка соответствующих ей точек и сохранение в виде отдельной структуры данных. Таким образом, все точки, находящиеся в пространстве в пределах ячейки, лежат рядом в памяти.

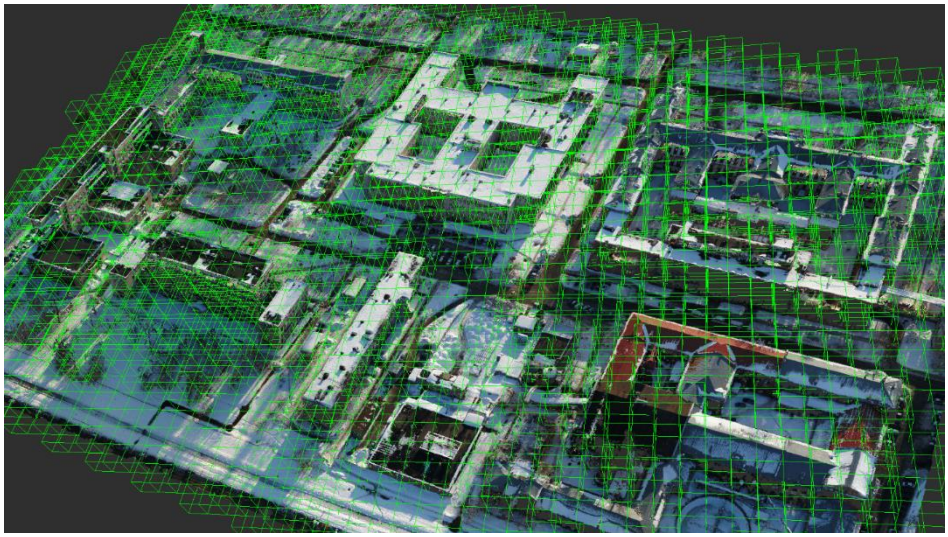

Рисунок 1.10 - Облако точек, разбитое на ячейки

Для получения видимых точек остается лишь найти видимые ячейки [30] и загрузить содержащиеся в них точки. Проверка пересечений пирамиды видимости виртуальной камеры с ячейкой структуры разбиения позволяет выполнять отсечение целых групп точек, что значительно сокращает объем вычислений. Пример такого подхода можно увидеть на рисунке 1.11. Красным цветом выделены реальные видимые точки, зеленым – ячейки, попадающие в пирамиду видимости.

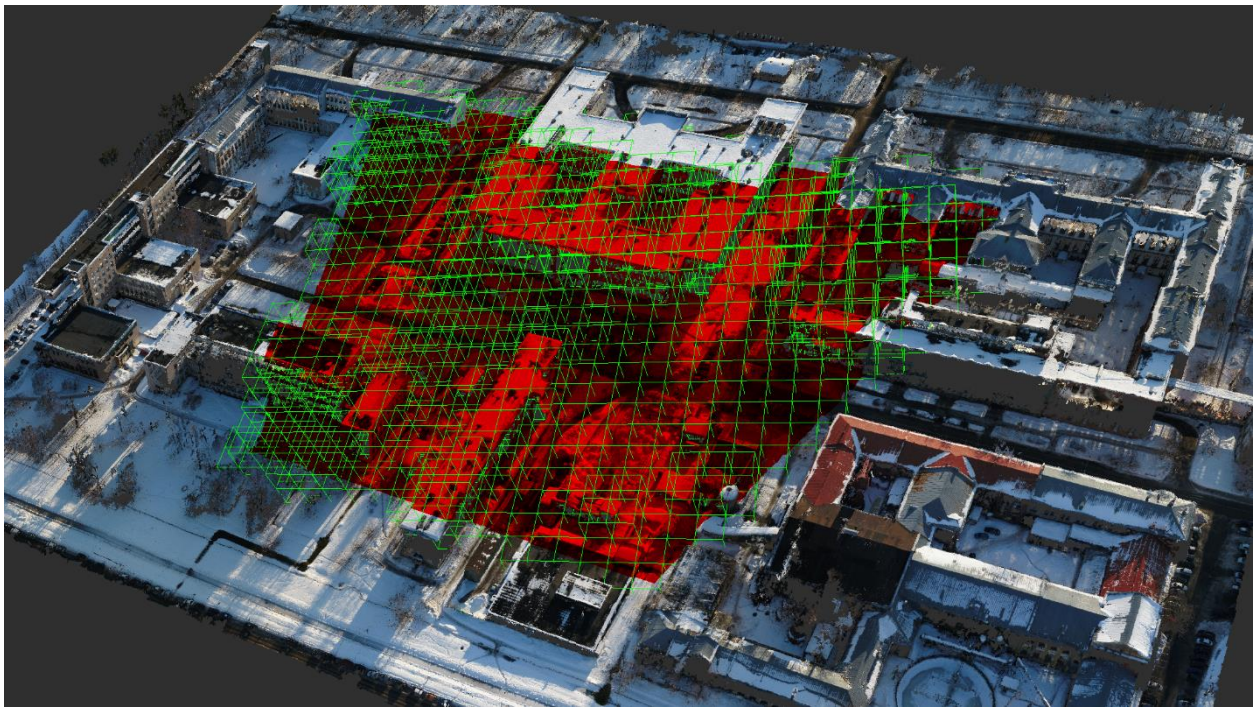

Рисунок 1.11 - Пример выборки точек для визуализации

В зависимости от требуемого результата могут применяться различные структуры разбиения пространства. Они могут различаться по времени выборки, принципу разбиения, сложности построения, дополнительно потребляемой памяти. Обычно для визуализации оправдано применение октодеревьев, которые за счет иерархической структуры позволяют сократить потребление памяти по сравнению с регулярными массивами ячеек, выполняя хранение только тех областей пространства, где содержатся какие-либо данные.

1.4. Обзор исследований в области организации обработки больших облаков точек

Для обоснования использования подходов к обработке больших облаков точек, применяемых в диссертационной работе, проведем анализ существующих исследований, рассматривающих организацию обработки больших облаков точек в ограниченном объеме оперативной памяти.

Можно выделить ряд основных подходов к решению данной задачи, включающих использование механизмов сжатия данных, потоковой обработки, структур разбиения пространства, а также ряд узкоспециализированных подходов.

Рассмотрим основные характеристики, позволяющие оценить рассмотренные подходы с точки зрения их использования для организации обработки больших облаков точек. Для этого лучше всего подходят количественные показатели, например, объем потребляемой оперативной памяти или производительность обработки. Но используемые алгоритмы подчинены различным целям, а данные, приведенные в статьях достаточно разнородны и получены на различающихся компьютерных системах, что затрудняет совокупную оценку.

Для оценки преимуществ и недостатков механизмов обработки больших облаков точек, используемых в рассмотренных подходах, можно использовать косвенные характеристики. С учетом того, что все рассмотренные подходы в той или иной мере позволяют выполнять обработку облаков точек больших, чем доступный объем оперативной памяти, можно выполнять их оценку с точки зрения накладных расходов на такую обработку и ограничений на использование для определенных видов обработки. Выделим эти характеристики.

Если обработка больших облаков осуществляется путем выгрузки неиспользуемых участков на вторичную систему хранения, то важной характеристикой будет количество создаваемых файлов, т.к. большое количество файловых записей может замедлить работу файловой системы, а также привести к исчерпанию доступного количества файловых записей. Так как в диссертационной работе для обеспечения возможности обработки больших облаков точек используется октодерево, необходимо рассмотреть другие подходы с целью обоснования использования структур разбиения пространства в целом и октодерева в частности. Также предпочтение будет отдано подходам, позволяющим выполнять модификацию облаков точек без перестроения используемых структур разбиения, что позволит расширить область применения.

Другой целью обзора является исследование методов, которые могут быть использованы в совокупности с предложенными в данной работе октодеревьями для увеличения их производительности.

1.4.1. Компактное представление данных

В статье Efficient processing of large 3d point clouds [31] предлагается реализация октодерева, применяемого для хранения и сжатия трехмерных данных без потери точности.

Для уменьшения структуры данных узла октодерева в статье предлагается сократить количество прямых указателей на данные потомков и полезную нагрузку узла, уменьшить размер указателей, используемых для адресации данных и потомков узла, а также отказаться от хранения размеров и позиции узла (так как они могут быть вычислены в процессе обхода).

Для уменьшения размеров полезной нагрузки в октодереве предлагается выполнить сжатие координат точек в облаке за счет использования для хранения чисел с плавающей точкой в облаке только двух байт. Для снижения потерь точности каждый бит координаты используется как приращение к координате нижнего левого переднего угла узла октодерева, в который входит данная координата. Для сокращения размеров цветовой компоненты точки используется тот же подход.

1.4.2. Использование потоковой обработки данных

В статье Stream-processing points [32] представлен подход, позволяющий выполнять потоковою обработку облака точек с использованием вторичных систем хранения данных. Предложенный подход позволяет выполнять последовательную обработку облака точек за счет их сортировки и смещения плоскости выборки в пространстве. Для обработки среза точек в оперативной памяти применяется набор конкатенируемых потоковых операторов. Для выполнения пространственного поиска используется динамическое полу-балансируемое k-мерное дерево, в котором содержатся данные текущей плоскости выборки. Результаты обработки последовательно записываются в файл.

В статье Flexible configurable stream processing of point data [33] рассматривается увеличение эффективности потоковой обработки из статьи [32] за счет введения концепции цепного оператора, действующего как глобальный оператор для цепочки обычных операторов. Подобный подход позволяет избавиться от старого механизма поиска ближайших соседей в пользу использования пространственной структуры из цепного оператора.

Недостатком подобного подхода является необходимость динамической балансировки структуры разбиения пространства для индексации точек текущего среза, что замедляет общий процесс обработки и ограничивает область пространства для поиска текущим срезом. С учетом того, что первичная выборка точек обеспечивается для всего файла облака точек, данный подход для любой обработки потребует считывания всего облака точек, тогда как в подходах с использованием структуры разбиения пространства на общем множестве точек имеется возможность отсечения не интересующих точек. Также данный подход ограничивает использование алгоритма обработки локальным множеством точек среза, что может усложнить его реализацию.

Преимуществом такого подхода можно назвать малое потребление оперативной памяти и возможность применения нескольких алгоритмов обработки за один проход считывания облака точек.

1.4.3. Использование тайлового массива

В работе Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures [34] представлена техника визуализации облака точек через интернет, обладающая возможностями визуализации облака точек с поддержкой уровней детализации на основе мульти-разрешения (multi-resolution) и использующая вторичные устройства хранения. Реализация такой техники производится при помощи неизбыточного метода организации точек, названного авторами Hierarchically Layered Tiles (HLT), и древовидной структурой, называемой авторами Tile Grid Partitioning Tree (TGPT).

Построение тайловой структуры HLT организовано следующим образом: ограничивающий прямоугольник облака точек делится на *T* тайлов одинакового размера, образующих регулярную сетку. Точки распределяются в тайлах с использованием их географического положения. Для каждого тайла в сетке точки разбиты на *L* слоев, при этом создается набор слоев с различной плотностью точек. Входной параметр, называемый коэффициентом понижающей дискретизации (*df*), определяет процент точек, которые разбросаны в каждом слое тайла. Точки в каждом слое равномерно распределены по поверхности.

Управление и обработка фрагментов, на которые разделено облако точек, осуществляется с помощью древовидной структуры TGPT. В отличие от других структур данных, таких как квадродерево или октодерево, которые могут быть полностью предварительно вычислены и сохранены на диске (на стороне сервера или на стороне клиента), TGPT является не статической предварительно вычисленной структурой, а структурой, сгенерированной на стороне клиента по мере необходимости и всегда соответствующей заданному региону интереса. TGPT хранится в оперативной памяти клиента. После определения региона интереса в TGPT создается корневой узел, который представляет все плитки, перекрывающие регион интереса.

Работа в условиях ограниченного потребления оперативной памяти достигается за счет сохранения точек каждого слоя каждого тайла в виде отдельного файла на вторичной системе хранения в сжатом формате LASzip (LAZ).

1.4.4. Использование K-мерного дерева

В работе Out-of-core visualization of classified 3d point clouds [35] авторами выполняется рендеринг больших облаков точек при помощи многослойного k-мерного дерева с поддержкой динамической детализации. Отличительной особенностью является возможность использования информации о классификации точек в облаке, т.е. можно применять различные способы рендеринга для точек различных классов. Построение kd-дерева выполняется при помощи вставки точек в гистограмму, как и нахождение медианного фрагмента и разбиения на левое-правое поддерево. Каждый узел дерева отвечает за определенный уровень детализации объекта. Обеспечение работы в условиях ограничений по оперативной памяти осуществляется путем выгрузки неиспользуемых узлов на вторичную систему хранения. Для каждого узла создается индивидуальный файл. Модификация дерева не поддерживается.

1.4.5. Использование октодерева

В работе An out-of-core octree for massive point cloud processing [18] рассматривается реализация октодерева, в которой для обеспечения выгрузки неактуальных узлов используется история взаимодействия с узлами. Для отслеживания узлов в памяти используется счетчик обращений, индивидуальный для каждого узла, а для повышения эффективности файловых операций происходит объединение нескольких узлов в один файл. Также обеспечивается возможность обновления данных в узле. Для записи данных на жестком диске создается иерархическая структура из файлов и директорий.

В работе Processing and interactive editing of huge point clouds [36] для эффективного хранения выбираются размеры узлов, которые соответствуют компромиссу между задержкой и пропускной способностью устройства ввода-вывода, узлы в оперативной памяти выгружаются согласно политике Least Recently Used (LRU, Вытеснение давно неиспользуемых) [37]. Хранение узлов на жестком диске обеспечивается при помощи набора файловых буферов, каждый из которых содержит блоки фиксированного размера. Между собой файлы различаются размером блока, который принимает значения степени двойки. Для снижения влияния задержек системы хранения все файловые операции выполняются асинхронно, в отдельном потоке. Поддерживается возможность модификации узлов октодерева.

В работе External memory management and simplification of huge meshes [38] рассмотрено решение задачи управления памятью при работе с полигональными сетками большого размера. Для решения задачи предложено использование октодерева, каждый узел которого выполняет хранение адреса участка данных полигональной сетки на вторичной системе хранения. Иерархическая структура октодерева при этом располагается в оперативной памяти, так как затраты памяти на ее хранение невелики. Поддерживаются возможности загрузки, модификации и сохранения участков полигональной сетки. Количество создаваемых файлов в файловой системе не указано.

В работе Interactions with gigantic point clouds [39] предложен способ взаимодействия с облаками точек, размер которых может превышать один миллиард точек. В работе для хранения облака точек используется октодерево, узлы которого загружаются и освобождаются при помощи системы кеширования, поддерживающей политику LRU. Поддерживаются возможности хранения уровней детализации для узлов октодерева, которые вычисляются при помощи просеивания точек на этапе построения октодерева. Поддерживается возможность модификации узлов. На каждый узел в процессе построения создается по одному файлу.

В работе Towards Efficient Implementation of an Octree for a Large 3D Point Cloud [40] рассматривается реализация октодерева для обработки больших облаков точек, в которой в каждом узле вместо указателя на данные в оперативной памяти хранится указатель на файл на жестком диске. Для получения данных требуется произвести их загрузку в оперативную память, какой-либо системы кеширования не предусмотрено. В работе также изучается влияние масштабирования октодерева для более точного соответствия описываемой области пространства к данным облака точек.

В статье Xsplat: External memory multiresolution point visualization [41] рассматривается реализация системы интерактивного рендеринга облака точек с построением структуры представления уровней детализации облака точек в файле в виде линейного списка блоков и последующего отображения их в оперативную память. Для хранения всех блоков используется один файл. Модификация блоков после построения не предусмотрена.

1.4.6. Прочие подходы

В статье Extracting roads from dense point clouds in large scale urban environment [42] описывается решение задачи выделения дорог на большом облаке точек (до миллиарда точек) для последующей обработки. Для решения задачи выполняется поиск дорог в картографической системе (Open Street Map), проекция дорог на облако точек, экспорт точек дорог и их окружения с разбиением на небольшие куски для последующего более точного выделения дорожного покрытия.

1.4.7. Анализ рассмотренных подходов к обработке больших облаков точек

На основании проведенного обзора исследований в данном разделе произведен анализ алгоритмов обработки больших облаков точек, существующих структур представления информации из облака точек (структур разбиения пространства) и обоснование выбора типа структуры.

Рассмотрим сводную таблицу по интересующим нас характеристикам, описанным в разделе 1.4 (Таблица 1.6).

Таблица 1.6 – Основные характеристики рассмотренных подходов

№	Название работы/метода	Метод	Структура разбиения	Создаваемые файлы (N – кол-во узлов)	Возможность модификации
1	Efficient processing of large 3d point clouds	Компактное представление и сжатие	Октодерево	Нет	Нет
2	Stream-Processing Points	Потоковая обработка	Нет/Локальное k-мерное дерево	1	Нет (требует перезапуска)
3	Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures	Разбиение данных на тайлы, разбиение тайлов на слои по плотности	Тайловый массив	N	Нет
4	Out-of-core visualization of classified 3d point clouds	Хранение данных в k-мерном дереве с возможностью выгрузки	К-мерное дерево	N	Нет (требует перестроения)
5	An out-of-core octree for massive point cloud processing	Хранение данных в октодереве с возможностью выгрузки	Октодерево	N/8	Да
6	External memory management and simplification of huge meshes	Хранение файлового указателя в узле октодерева	Октодерево	Не указано	Да
7	Interactions with gigantic point clouds	Хранение данных в октодереве с возможностью кеширования и выгрузки	Октодерево	N	Да
8	Towards Efficient Implementation of an Octree for a Large 3D Point Cloud	Хранение файлового указателя в узле октодерева	Октодерево	N	Нет
9	Xsplat: External memory multiresolution point visualization	Использование механизма отображения памяти для хранения данных узлов	Октодерево	1	Нет
10	Extracting roads from dense point clouds in large scale urban environment	Сокращение потребляемой памяти за счет выделения точек интереса в отдельное облако	Нет	Нет	Нет

Детальное изучение существующих способов обработки облаков точек в условиях нехватки оперативной памяти позволяет выделить следующие подходы:

1. Сокращение объема загружаемых облаков точек за счет более компактного представления или сжатия. Может быть применен совместно с прочими подходами. Не позволяет полностью избежать проблемы нехватки памяти;
2. Применение потоковой обработки облака точек. Применяется для предварительно отсортированного облака точек. Обработка производится при помощи цепочки алгоритмов (фильтров), при этом единовременно в оперативной памяти находится локальный срез облака, все точки которого благодаря сортировке располагаются поблизости в пространстве. Для ускорения операций пространственного поиска для точек в срезе происходит динамическое обновление структуры разбиения пространства;
3. Выгрузка на вторичную систему хранения неиспользуемых на текущий момент данных. Позволяет решить проблему за счет использования устройств большей емкости, однако вызывает усложнение архитектуры системы и может привести к замедлению ее работы.

Как можно увидеть из приведенного обзора, в большинстве рассмотренных исследований предлагается использование вторичной системы хранения данных. Методы сжатия или компактного представления данных в той или иной степени находят свое применение во всех рассмотренных работах. Использование потоковой обработки применяется реже, ввиду более узкой специализации.

Рассмотрим более подробно способ обработки больших облаков точек при ограничении потребляемой памяти за счет выгрузки неактуальных данных на вторичную систему хранения. Исходя из проведенного обзора можно увидеть, что современные подходы к решению такой проблемы используют методы разбиения пространства совместно с системой кеширования данных. Такой подход позволяет добиться большей локальности участков, чем кеширование неупорядоченного облака точек. Среди рассмотренных реализаций используются следующие структуры разбиения пространства: k-мерное дерево, тайловый массив, октодерево.

Как было рассмотрено ранее, k-мерное дерево относится к структурам разбиения, управляемым данными, в силу чего очень чувствительно к модификации облака точек, однако требует меньше памяти и обеспечивает более быстрый доступ к данным по сравнению с октодеревом. Тайловые или воксельные массивы, а также октодерево относятся к структурам разбиения, управляемым пространством, что позволяет не перестраивать структуру при изменениях в облаке точек. Кроме того, они лучше подходят для использования совместно с вторичной системой хранения, так как разбиение на каждом уровне идентично, что позволяет выполнять сохранение меньшего количества информации.

Также стоит отметить различия в механизмах регуляции потребления оперативной памяти. В системах, реализующих выгрузку данных на вторичную систему хранения, преимущественно применяется подход с организацией системы кеширования, так как она позволяет использовать динамическую подгрузку и выгрузку данных (при этом в оперативной памяти содержится одновременно ограниченное количество данных). Отдельного внимания заслуживает подход с использованием механизма отображения данных, позволяющий переложить эту работу на операционную систему и предоставляющий доступ к данным по прямому указателю.

В большинстве подходов, в процессе построения создается большое количество файлов, что приводит к снижению производительности файловых операций. В некоторых файловых системах, таких как Ext4 [43], большое количество файлов может привести к исчерпанию доступного количества файловых записей в файловой системе.

1.5. Анализ проблемы обработки большого облака точек лазерного сканирования и выбор подхода к ее решению

Из анализа работ по обработке облака точек лазерного сканирования известны следующие факты:

- С развитием технологии лазерного сканирования и областей его применения скорость роста объема информации в облаке превышает скорость роста объема основной памяти в компьютерах;
- Программные библиотеки обработки облака точек ориентированы на размещение данных в основной памяти;
- Использование внешней памяти существенно увеличивает время обработки и требует разработки дополнительных средств при использовании стандартных программных библиотек.

В процессе анализа существующих реализаций обработки данных облака точек с использованием внешней памяти было сделано наблюдение, что механизмы файловой системы (в частности, способы идентификации файлов, получения к ним доступа и изменения их размеров), использующиеся при взаимодействии с внешней памятью, показывают падение производительности при работе с большим количеством файлов и могут вызывать ошибки при превышении максимального количества файлов в файловой системе или файловых дескрипторов процесса. Экспериментальное подтверждение данному наблюдению приведено в разделе 4.6.

Таким образом, существует возможность прироста производительности операций взаимодействия с внешней памятью для октодерева, использующего отдельные файлы для хранения блоков данных (с целью обеспечения возможности наращивания их размеров в результате заполнения октодерева), с использованием более низкоуровневого управления внешней памятью в пределах одного или нескольких файлов вместо более широко направленных механизмов файловой системы.

На основе данного наблюдения была выдвинута гипотеза об уменьшении затрат времени на обращения к внешней памяти при обработке облака точек за счет изменения способов доступа, размещения и идентификации блоков данных октодерева во внешней памяти, позволяющих сократить количество файловых операций и создаваемых файлов, а также снизить количество задержек, обусловленных файловой системой.

Эти вопросы рассматриваются во 2-й главе. С использованием результатов этих исследований в 3-й главе приведена разработка алгоритмов и структур данных для построения октодерева в условиях ограничения по оперативной памяти.

1.6. Постановка задачи снижения затрат времени на обмен с внешней памятью и определение показателей для оценивания эффективности реализации вычислительного процесса обработки

В данном разделе, на основании выдвинутой гипотезы об уменьшении затрат времени на обращения к внешней памяти, выполняется построение моделей формирования октодерева, компонентов вычислительного процесса обработки облака точек во внешней памяти, а также модели вычислительного процесса обработки облака точек на основе сетей Петри.

Модель формирования октодерева иллюстрирует взаимосвязи между данными облака точек и элементами структуры октодерева. Модель компонентов вычислительного процесса обработки облака точек демонстрирует многообразие вариантов организации вычислительного процесса и позволяет выделить приоритетные направления для дальнейшего исследования. Модель вычислительного процесса обработки облака точек на основе сетей Петри демонстрирует динамическую составляющую вычислительного процесса обработки. Вместе представленные модели позволяют выделить основные этапы и компоненты вычислительного процесса обработки облака точек, модификация которых приведет к повышению эффективности обработки.

Выдвигается предположение о модификации вычислительного процесса обработки двумя различными методами, для проверки которых выделяются показатели эффективности реализации вычислительного процесса обработки и выполняется декомпозиция общей задачи повышения эффективности реализации процесса обработки с учетом двух выделенных приоритетных направлений для исследования: на базе системы кеширования и механизма отображения памяти.

1.6.1. Модель формирования октодерева

Рассмотрим обобщенную модель формирования октодерева, основанную на онтологическом принципе и описывающую узлы, листья и связи между ними, а также их представление в оперативной и внешней памяти.

Модель можно представить в виде следующей совокупности объектов (Рисунок 1.12):

$$O = \langle P, N, R^n, L, R^{nl}, R^{lp}, V, V^{RAM}, V^{HDD} \rangle, \quad (1.5)$$

где

Исходные данные:

$P = \{p_1, \dots, p_s\}$ – множество точек в облаке точек;

Элементы структуры разбиения (октодерева):

$N = \{n_1, \dots, n_m\}$ – множество узлов в структуре разбиения;

$R^n = \{R_1^n, \dots, R_m^n\}$ – множество отношений между узлами;

$L = \{l_1, \dots, l_k\}$ – множество листьев структуры разбиения;

$R^{nl} = \{R_1^{nl}, \dots, R_k^{nl}\}$ – множество отношений между листьями и узлами структуры разбиения;

$R^{lp} = \{R_1^{lp}, \dots, R_s^{lp}\}$ – множество отношений между точками из облака точек и листьями структуры разбиения;

Структурные элементы представления данных в памяти компьютера:

$V = \{V_1, \dots, V_k\}$ – множество блоков данных, принадлежащих листьям в структуре разбиения;

$V^{RAM} = \{V_i^{RAM}\}, V^{RAM} \subset V, 0 \le i \le k$ – множество блоков данных в оперативной памяти, является подмножеством общего множества блоков данных;

$V^{HDD} = \{V_j^{HDD}\}, V^{HDD} \subset V, 0 \le j \le k$ – множество блоков данных во внешней памяти, является подмножеством общего множества блоков данных.

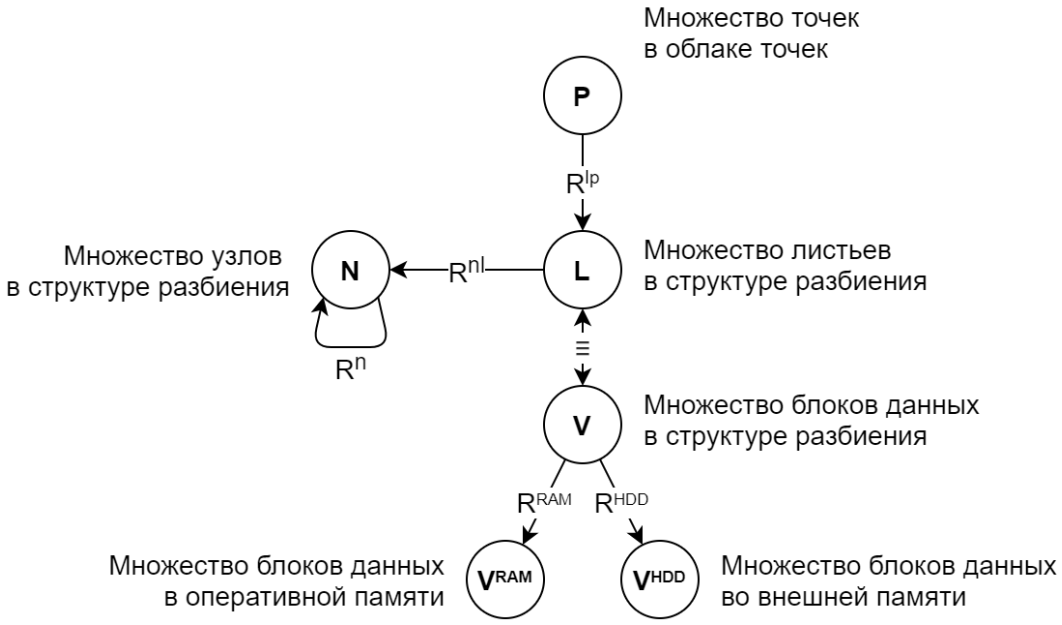

Рисунок 1.12 - Модель формирования октодерева

Приведенная модель демонстрирует взаимосвязи между исходным облаком точек, элементами иерархии октодерева и непосредственно блоками данных в оперативной или внешней памяти.

1.6.2. Анализ компонентов вычислительного процесса обработки облака точек

Рассмотрим проблемы организации вычислительного процесса обработки с учетом особенностей стандартных прикладных программ, ресурсов вычислительной системы, операционной системы (ОС) и дополнительных предложенных функций. Для этого разработаем концептуальную модель (Рисунок 1.13), учитывающую компоненты вычислительного процесса, применение которых направлено на сокращение потребления оперативной памяти либо на повышение производительности процесса обработки.

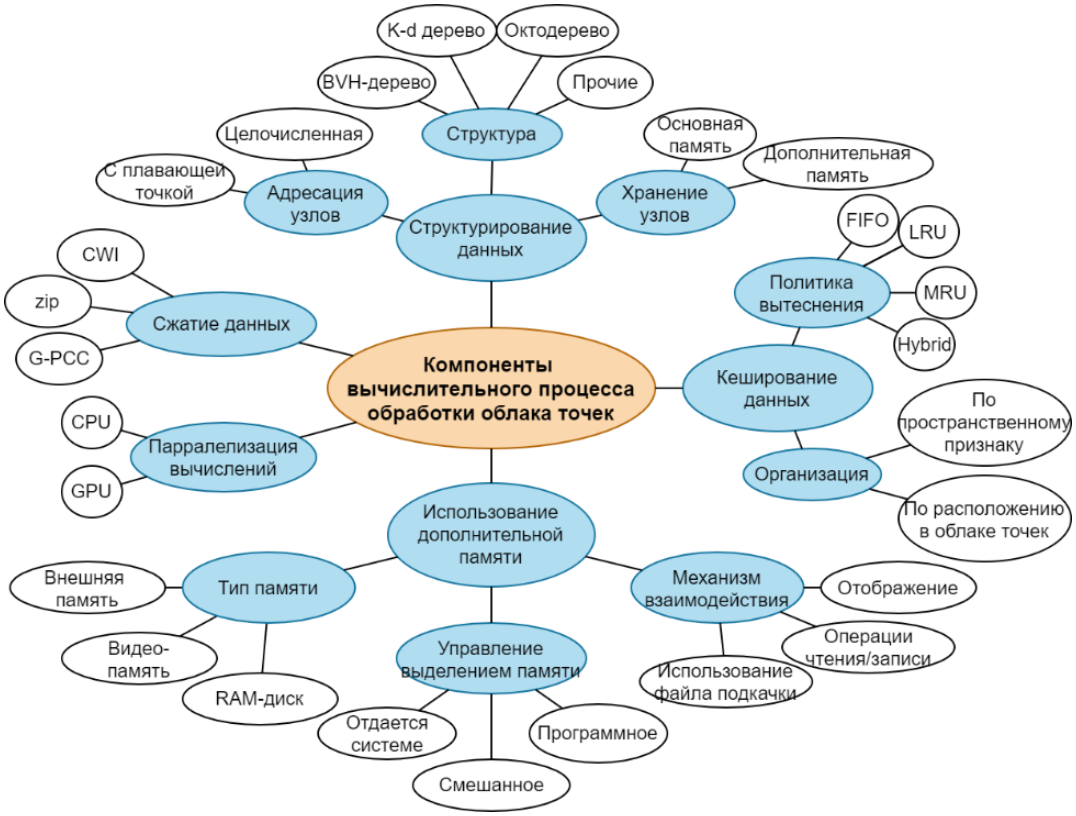

Рисунок 1.13 - Концептуальная модель компонентов вычислительного процесса обработки облака точек

Анализ существующих решений, производительности и трудоемкости реализации отдельных компонентов позволяет сократить список компонентов, подтвердив обоснованность использования:

– Октодерева в качестве структуры разбиения;
– Внешней памяти в качестве дополнительной памяти;

– Механизма отображения или операций чтения/записи для взаимодействия с внешней памятью;
– Механизмов низкоуровневого управления памятью для сокращения расходов на операции с внешней памятью.

Подтверждение обоснованности использования прочих компонентов требует дополнительного исследования. Для уточнения приоритетных направлений рассмотрим организацию вычислительного процесса обработки облака точек с использованием октодерева. Для этого исследуем структуру данных в октодереве и механизмы взаимодействия с такими данными для типовых примеров организации обработки в оперативной и внешней памяти.

На рисунке 1.14 приведена типовая структура данных в октодереве при обработке в оперативной памяти (слева), и во внешней памяти с использованием системы кеширования (справа). Адресация узлов, листьев и блоков данных в оперативной памяти производится при помощи указателя ($R^{pointer}$). Дополнительно для идентификации узлов и листьев в структуре разбиения (независимо от их расположения в оперативной памяти) используются идентификаторы (R^{id}). Стоит отметить, что в случае с файлом идентификатор преобразуется в имя файла при каждом сопоставлении.

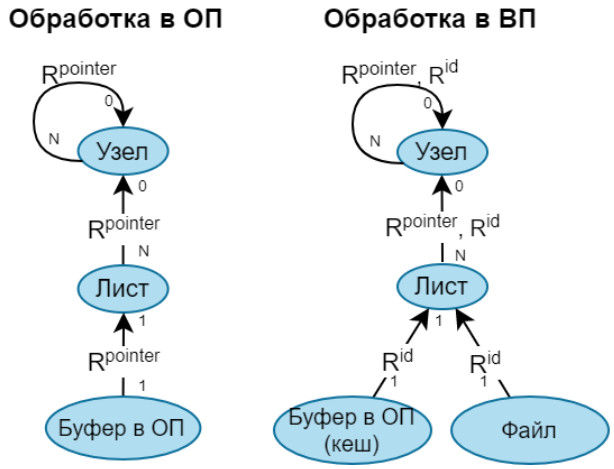

Рисунок 1.14 - Типовая структурная организация данных в октодереве

Для иллюстрации вычислительного процесса обработки рассмотрим процедуру вставки точки в октодерево, приведенную на рисунке 1.15. Как видно из рисунка, в случае обработки в оперативной памяти при добавлении точки достаточно произвести запрос буфера данных по указателю, выполнить изменение его размеров (для непрерывных массивов это может привести к реаллокации данных) и затем произвести запись точки. При обработке во внешней памяти процедура усложняется: производится сопоставление идентификатора файлу во внешней памяти, загрузка файла, запись данных и сохранение файла в момент вытеснения из кеша.

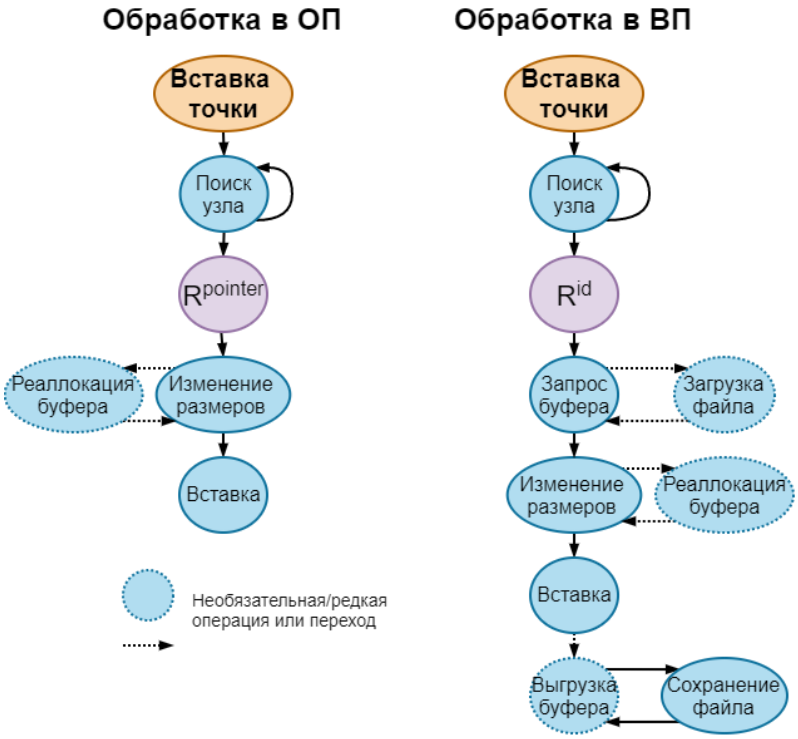

Рисунок 1.15 - Пример процедуры вставки точки в октодерево

1.6.3. Модель динамики вычислительного процесса обработки на основе сетей Петри

Процесс, как объект изучения, имеет структуру (статический характер) и поведение во времени (динамическая модель). Приведенные ранее в работе модели относятся к статике. В данном разделе выполняется построение и анализ динамической составляющей процесса обработки, что позволяет уточнить постановку задач для нового подхода к управлению процессом на определенных этапах, связанных с обращениями к ВП.

Удобным и наглядным языком для описания динамической составляющей вычислительного процесса является язык сетей Петри. Сеть Петри – это двудольный ориентированный граф с двумя типами вершин (позициями и переходами, изображаемыми соответственно кружками и полочками), дугами, соединяющими позиции с переходами и переходы с позициями, и начальной маркировкой, которая представляется вектором [44]. Сеть Петри представляет собой наглядную и хорошо формализованную модель поведения параллельных систем с асинхронными взаимодействиями. Она в компактной форме отражает структуру взаимоотношений элементов системы и динамику изменения ее состояний при заданных начальных условиях. Уровень абстракции модели очень высокий — он соответствует описанию взаимодействий в системе в терминах всего лишь двух понятий: событий и условий [45].

Представление сети Петри в виде двудольного графа позволяет задать структуру сети Петри статически. Динамика в модель вносится механизмом смены маркировки (разметки) позиций и соглашением о правиле срабатывания (реализации) переходов [45]. В рассмотренных далее моделях обобщенного вычислительного процесса предполагается, что логика срабатывания переходов зависит от используемых конечных алгоритмов обработки (напр. фильтрации, визуализации, сегментации и пр.), реализованных на базе рассматриваемых структур данных или вариантов организации вычислительного процесса. Из этого следует, что данная модель не может предсказать количество итераций в цикле обработки, или наиболее нагруженные (за счет частоты использования) переходы, но позволяет перечислить основные ветвления и переходы, а также выделить основные операции взаимодействия с ОП или ВП, не влияющие на конечные алгоритмы обработки.

Рассмотрим обобщенный процесс обработки облака точек в оперативной памяти с использованием октодерева (Рисунок 1.16). В таком процессе можно выделить последовательность операций обхода октодерева *TraverseLoop,* внутренний цикл обработки данных *InnerLoop*, и внешний цикл обхода и обработки узлов октодерева *OuterLoop*. Условия входа в данные циклы, равно как и количество итераций, зависит от конкретного алгоритма обработки. Непосредственно сам процесс обработки данных в общем виде можно представить в виде последовательности операций *чтения-и-обработки* (*Read* и $Process_2$) и/или *обработки-и-записи* ($Process_1$ и *Write*), выбор которых также осуществляется конкретным алгоритмом обработки.

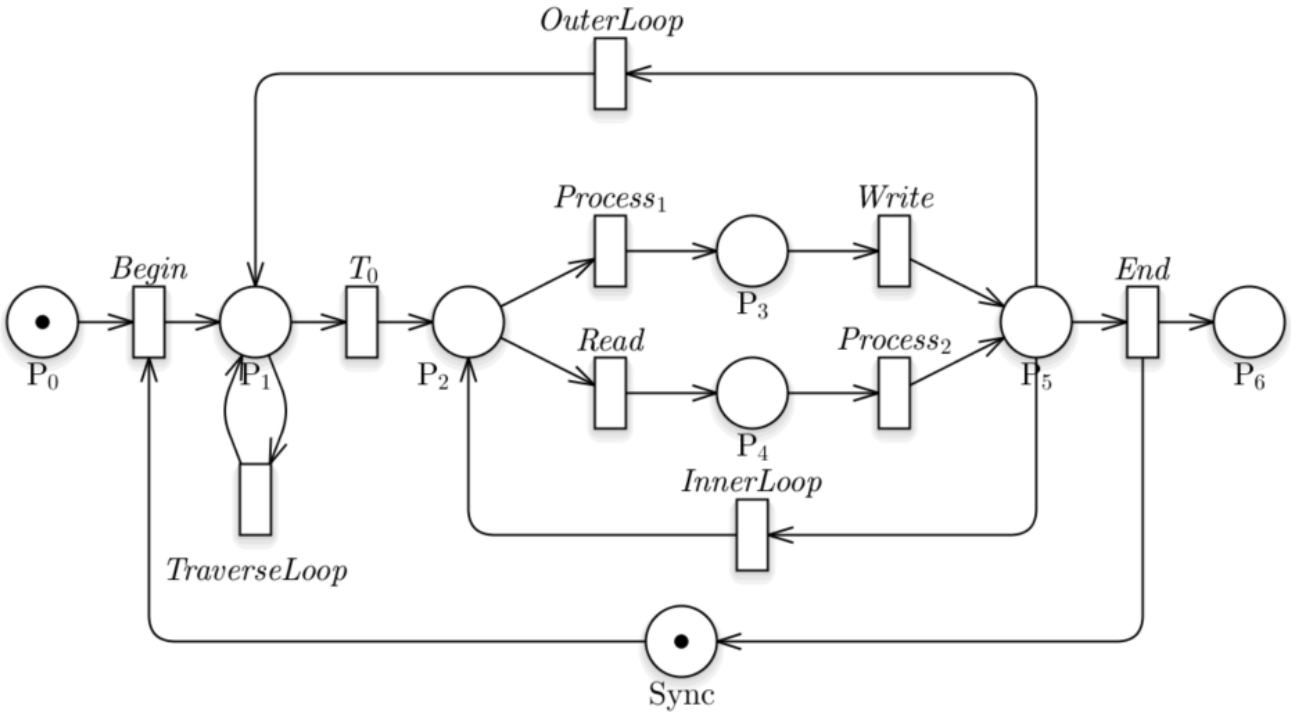

Рисунок 1.16 – Процесс обработки облака точек

В контексте данной работы нас интересуют операции взаимодействия с данными, которые условно разделены на операции чтения и записи (Рисунок 1.17). При обработке в оперативной памяти со стороны операций записи можно выделить операции выделения блока в оперативной памяти (*Allocate*), удаления блока (*Delete*), изменения размеров блока (*Reallocate*) и записи данных в блок (*Update* и *Append*). Со стороны операций чтения имеется непосредственно операция считывания данных из ОП (*Read*). Со стороны взаимодействия с октодеревом также присутствует операция сопоставления идентификатора узла в октодереве соответствующему блоку данных (*GetAddress*).

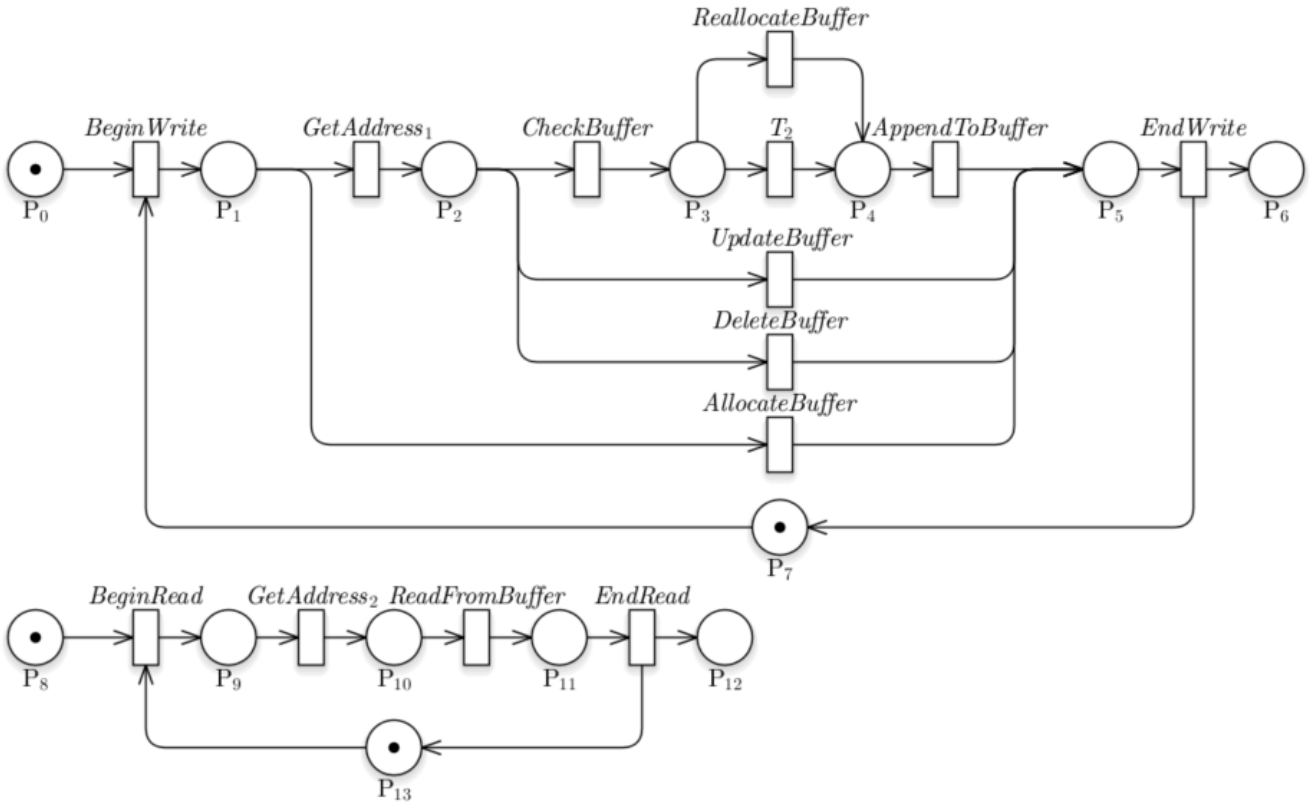

Рисунок 1.17 – Детализация операций чтения и записи при обработке в оперативной памяти

При обработке во внешней памяти (Рисунок 1.18) со стороны операций записи можно выделить операции создания файлов (*CreateFile*), удаления файла (*DeleteFile*), добавления данных в конец файла, что приводит к изменениям его размера (*Append*), и записи данных в файл (*Update*). Со стороны операций чтения имеется непосредственно операция считывания данных из файла (*Read*). Со стороны взаимодействия с файловой системой (ФС) присутствуют такие операции, как получение файлового дескриптора (*OpenFile*) и освобождение файлового дескриптора (*CloseFile*). Со стороны взаимодействия с октодеревом также присутствует операция сопоставления идентификатора узла в октодереве соответствующему адресу в ФС (*GetFilePath*).

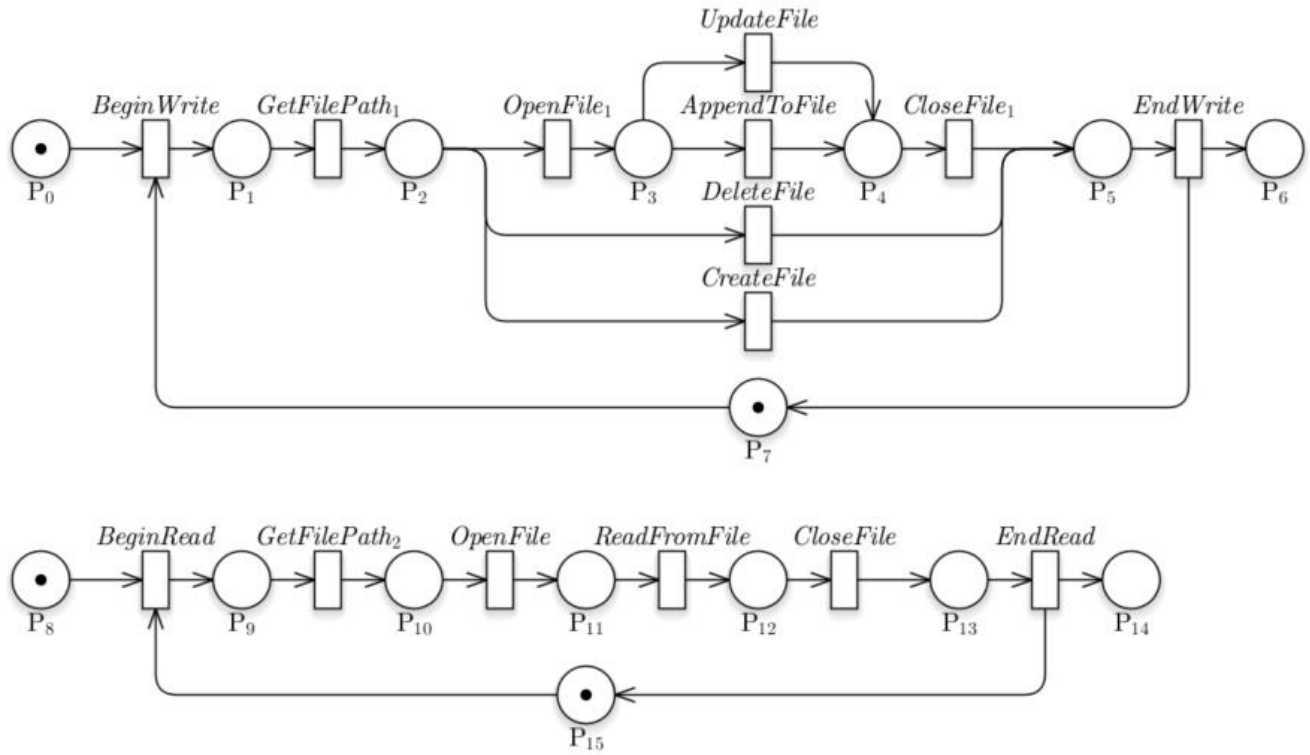

Рисунок 1.18 - Детализация операций чтения и записи при обработке во внешней памяти

При использовании внешней памяти процесс обработки может задействовать для хранения данных как исключительно внешнюю память, так и комбинацию подходов с использованием внешней и оперативной памяти.

Рассмотрим способы повышения эффективности процесса обработки облака точек при ограничении потребляемой оперативной памяти. Помимо повышения эффективности общих операций обработки облака точек (например, скорости обхода октодерева), был выделен ряд мер, связанных с взаимодействием с внешней памятью:

- Сокращение числа операций во внешней памяти за счет использования оперативной памяти;
- Сокращение числа операций с ФС за счет введения более низкоуровневых процедур доступа к данным;
- Сокращение количества создаваемых файлов для снижения нагрузки на ФС;

– Асинхронное выполнение части операций для снижения задержек взаимодействия с внешней памятью;
– Сокращение количества обменов с внешней памятью за счет модификации алгоритма обработки данных.

В дополнение к повышению эффективности взаимодействия с внешней памятью также были рассмотрены возможности повышения эффективности операций обхода и формирования октодерева. Они включают в себя меры по сокращению числа операций с плавающей точкой и модификацию алгоритма формирования, позволяющую сократить объем пересылаемых данных. На предложенной модели такие операции могут выполняться внутри *TraverseLoop, InnerLoop, OuterLoop* и *Process.*

1.6.4. Задача снижения затрат времени на обмен с внешней памятью и определение показателей для оценивания эффективности реализации вычислительного процесса обработки

В данной работе выдвигается предположение, что модификация структурной организации данных и вычислительного процесса обработки одним из предложенных далее методов позволит улучшить производительность обработки в ВП. Предложено два метода, отличающихся способом взаимодействия с внешней памятью и позволяющих произвести экспериментальное исследование предложенных мер по повышению эффективности вычислительного процесса. Первый метод позволяет исследовать процедуру обработки, не отказываясь полностью от операций файловой системы. Во втором методе функции файловой системы (в основном – выделение блоков данных (файлов) и наращивание их размеров) возьмет на себя операционная система и механизм динамической аллокации.

В первом методе предлагается модифицировать рассмотренную систему обработки во внешней памяти:

- Сократить количество операций с плавающей точкой, что позволит ускорить операции поиска и обхода узлов;
- Сократить количество создаваемых файлов, что позволит сократить накладные расходы на взаимодействие с ФС;
- Модифицировать процедуру формирования октодерева, что позволит сократить количество чтений из ВП;
- Выполнять часть операций асинхронно, что позволит снизить задержки, связанные с взаимодействием с ВП.

Во втором методе предлагается модифицировать рассмотренную процедуру обработки в оперативной памяти:

- Подменить буферы данных, выделенные в оперативной памяти, на буферы, выделенные в отображаемой памяти;
- Для управления блоками данных в отображаемой памяти использовать механизм динамической аллокации;
- Отказаться от операций с плавающей точкой в процессе обхода и поиска узлов;
- Модифицировать иерархическую структуру октодерева для применения целочисленных операций и упрощения доступа и модификации.

Пример модифицированной структуры данных приведен на рисунке 1.19. В случае использования системы кеширования добавляется возможность адресации участка файла, которая может быть применена для объединения всех отдельных файлов в один. При использовании механизма отображения исходная структура данных обработки в оперативной памяти практически не претерпевает изменений (что является одним из преимуществ данного метода), добавляется лишь скрытая взаимосвязь (R^{map}) между адресами в виртуальной памяти и смещением внутри файла.

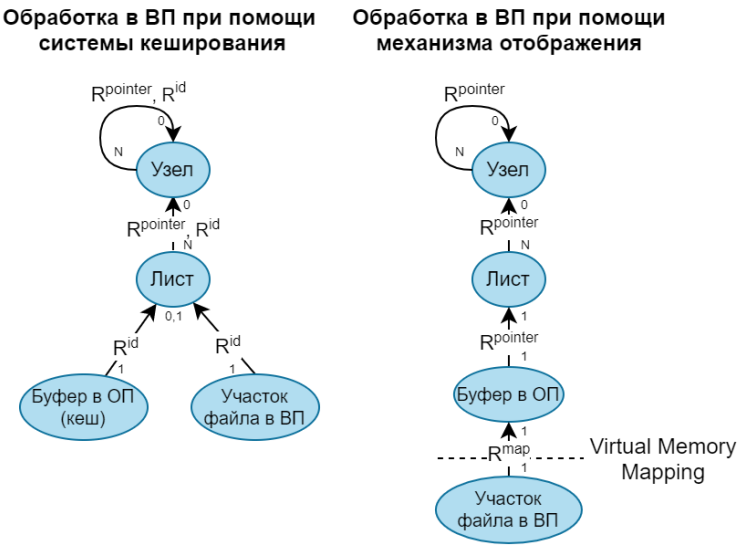

Рисунок 1.19 - Предложенные модификации структурной организации данных в октодереве

Выполним оценку эффективности реализации вычислительного процесса обработки. Так как количество вхождений операций чтения и записи, а также объем передаваемых данных зависит от применяемого алгоритма обработки и обрабатываемого облака точек, оценка количества операций в процессе обработки или на критическом пути исполнения не даст стабильных результатов.

В данной работе для получения оценки эффективности реализации вычислительного процесса обработки используется измерение общего времени обработки и сравнение его с временем обработки реализации в оперативной памяти на одинаковом наборе данных.

Рассмотрим основные операции, сокращение времени исполнения и количества которых приведет к повышению эффективности процесса обработки. Введем три основных категории таких операций:

$$U^{octree} = \{TraverseLoop, InnerLoop, OuterLoop, GetAddress, GetFilePath\}, \quad (1.6)$$

$$U^{ram} = \{AllocateBuffer, DeleteBuffer, UpdateBuffer, AppendToBuffer, ReadFromBuffer\},$$

$$U^{hdd} = \{OpenFile, CloseFile, CreateFile, DeleteFile, UpdateFile, AppendToFile, ReadFromFile\},$$

где U^{octree} – операции обхода октодерева и циклов алгоритма обработки;

U^{ram} – операции взаимодействия с оперативной памятью;

U^{hdd} – операции взаимодействия с файловой системой и внешней памятью.

Обозначения операций соответствуют приведенным на рисунках 1.16, 1.17 и 1.18.

Повышение эффективности процесса обработки может быть выполнено путем уменьшения общего времени, затрачиваемого на перечисленные выше операции, путем сокращения их количества или непосредственного времени исполнения. Допустимо также сокращение части более медленных операций (например, U^{hdd}) за счет применения более быстрых (U^{ram}), при условии соблюдения ограничений на объем потребляемой оперативной памяти.

Введем $X = \{x_i, i = 1 \dots n\}$ – конечное множество возможных вариантов реализации вычислительного процесса обработки облака точек, где x_i – номер варианта обработки. Тогда, согласно выдвинутой гипотезе, существует собственное подмножество вариантов реализации на основе системы кеширования $X^{cache} \subset X$ и собственное подмножество вариантов реализации на основе механизма отображения памяти $X^{mmap} \subset X$. Также в целях сравнения рассматриваются варианты реализации с обработкой в оперативной памяти x^{ram} и обработкой при помощи системы кеширования и механизмов ФС x^{std}.

Формализация вычислительного процесса обработки для различных вариантов обработки выполняется при помощи сетей Петри и представлена двудольным ориентированным мультиграфом:

$$G_\chi = \langle P_\chi, T_\chi, E_\chi \rangle, \tag{1.7}$$

где $\chi \in \{1,2,3,4\}$ – индекс, характеризующий вариант реализации вычислительного процесса: $x_1^{cache} \in X^{cache}, x_2^{mmap} \in X^{mmap}, x_3^{ram}, x_4^{std}$, причем x^{cache} и x^{mmap} соответствуют конкретным вариантам обработки на базе соответствующих компонентов вычислительного процесса, получение которых производится во второй главе;

P_χ – конечное непустое множество позиций;

T_χ – конечное непустое множество переходов (или событий);

E_χ – конечное множество дуг, $E_\chi \subseteq P_\chi \times T_\chi \cup T_\chi \times P_\chi$.

В качестве показателя эффективности предложенного варианта обработки предлагается выбрать показатель общих затрат времени обработки K_χ^{time}, которое требуется минимизировать:

$$K_\chi^{time} \to min, \tag{1.8}$$

и сравнить его с временем обработки реализации в оперативной памяти K_3^{time} на одинаковом наборе данных.

Рассмотрим ограничения, которым должны удовлетворять выбранные варианты обработки. Основным ограничением является пиковая потребляемая оперативная память в процессе обработки:

$$M_\chi^{ram}(t) \le M^{limit}, t \in \mathrm{T}, \tag{1.9}$$

где $M_\chi^{ram}(t)$ – объем потребления оперативной памяти в момент времени t;

M^{limit} – лимит потребления оперативной памяти;

$t \in \mathrm{T}$ – множество моментов времени обработки облака точек.

На основании приведенного обзора, выполним декомпозицию общей задачи повышения эффективности реализации процесса обработки на следующие подзадачи:

1. Анализ структурной организации октодерева и механизмов взаимодействия оперативной и внешней памяти. Исследование способов сокращения размеров узлов в ОП, их идентификации, а также повышения эффективности операций обхода и поиска в октодереве;
2. Исследование подмножества вариантов реализации вычислительного процесса обработки во внешней памяти на основе системы кеширования (X^{cache}) . Выбор политики вытеснения в системе кеширования. Организация локальности данных по пространственному признаку;

3. Исследование подмножества вариантов реализации вычислительного процесса обработки во внешней памяти на основе механизма отображения памяти (X^{mmap}). Исследование механизмов низкоуровневого управления памятью на внешней системе хранения.

1.7. Выводы

На основании вышеизложенного обзора можно сделать следующие выводы:

1. Задача сокращения потребляемой оперативной памяти при обработке облаков точек продиктована развитием технологии лазерного сканирования (и областей его применения), в результате которого скорость роста объема информации в облаке превышает скорость роста объема основной памяти в вычислительных системах.
2. Использование внешней памяти для сокращения потребления оперативной памяти в процессе обработки существенно увеличивает время обработки и требует разработки дополнительных средств при использовании стандартных программных библиотек. Программные библиотеки обработки облака точек ориентированы на размещение данных в основной памяти, что также усложняет применение внешней памяти для обработки облака точек.
3. Проведенный анализ методов и алгоритмов обработки больших облаков точек, существующих структур представления информации из облака точек (структур разбиения пространства) позволил выделить основные компоненты вычислительного процесса обработки облака точек с использованием внешней памяти, а также обосновать выбор октодерева в качестве базовой структуры данных.

4. Была выдвинута гипотеза об уменьшении затрат времени на обращения к внешней памяти при обработке облака точек, сформулированная на основе анализа проблемы с использованием формализованных моделей вычислительного процесса обработки облака точек, формирования октодерева, компонентов вычислительного процесса обработки облака точек во внешней памяти. Гипотеза дала возможность выделить этапы и компоненты процесса обработки, которые стали предметом детального исследования, что позволило предложить новые методы обработки.
5. Была проанализирована задача организации вычислительного процесса обработки данных облака точек при использовании внешней памяти, а также выделены показатели эффективности реализации вычислительного процесса обработки, что позволило выполнить декомпозицию общей задачи повышения эффективности реализации процесса обработки, а также выделить два приоритетных направления для исследований: на базе системы кеширования и механизма отображения памяти.

В данной главе были рассмотрены следующие положения диссертации:

1. Концептуальные модели формирования октодерева и вычислительного процесса обработки облака точек во внешней памяти, предназначенные для выделения компонентов и этапов, модификацией которых с учетом особенностей структуры облака точек можно сократить затраты времени при обработке.

2. Гипотеза об уменьшении затрат времени на обращения к внешней памяти при обработке облака точек за счет изменения способов доступа, размещения и идентификации блоков данных октодерева во внешней памяти, позволяющих сократить количество файловых операций и создаваемых файлов, а также снизить количество задержек, обусловленных файловой системой. Сформированные на базе данной гипотезы предположения об организации вычислительного процесса позволили выделить этапы и компоненты обработки, которые стали предметом детального исследования, что позволило предложить новые методы обработки, являющиеся основными результатами.

В данной главе были решены следующие задачи диссертации:

1. Системный анализ процессов формирования и использования октодерева по облаку точек ЛС при использовании внешней памяти, включающий анализ методов, алгоритмов и структур данных, применяемых для обработки в оперативной и внешней памяти, их декомпозицию на составные компоненты, исследование их взаимодействия между собой, центральным процессором, внешней и оперативной памятью с целью выделения компонентов, за счет новой организации которых можно уменьшить затраты времени.
2. Постановка задачи снижения затрат времени на обмен с внешней памятью, определение критерия и показателей для оценивания эффективности реализации вычислительного процесса обработки. Формулирование гипотезы об организации вычислительного процесса обработки и структур данных, позволяющей сократить временные затраты на использование внешней памяти, и формирование подзадач на последующее исследование и уточнение предположений, основанных на данной гипотезе.

Глава 2. Иерархическая модель октодерева и методы обработки больших облаков точек

В главе производится анализ и уточнение различных вариантов реализации вычислительного процесса обработки и механизмов взаимодействия оперативной и внешней памяти, предложенных в первой главе. Рассматривается организация данных в октодереве и механизмы взаимодействия оперативной и внешней памяти. Предложены структуры данных и алгоритмы, предназначенные для построения октодерева, позволяющего выполнять обработку больших облаков точек с использованием вторичных систем хранения. Рассматриваются модели октодерева на основе арифметики с плавающей точкой и целочисленной арифметики. Предложена иерархическая модель с использованием целочисленной арифметики.

В данной главе выполняются следующие задачи диссертации:

1. Разработка новых методов организации вычислительного процесса обработки облака точек, основанных на выдвинутой гипотезе. Разработка компонентов, алгоритмов и структур данных таких систем, исследование взаимодействий между ними, системным ПО, оперативной и внешней памятью.

Результаты, изложенные в данной главе были опубликованы соискателем в статьях [46].

2.1. Анализ задач обработки облака точек

В данном разделе рассматривается влияние группировки точек по пространственному признаку на производительность алгоритмов обработки облака точек, а также приводятся примеры сценариев обработки, учитывающих наличие пространственных отношений между точками в облаке.

2.1.1. Распределение точек в облаке

Рассмотрим подробнее топологию облака точек. Облако точек является неорганизованной структурой данных. Точки в облаке, полученном непосредственно после лазерного сканирования, могут иметь настолько предсказуемое расположение, насколько предсказуемо поведение луча лазерного сканера. Подобную информацию даже используют некоторые алгоритмы [47; 48] для ускорения процесса обработки облака точек. В качестве примера можно привести облако точек после лазерного сканирования (Рисунок 2.1), после обработки (Рисунок 2.2), и результат совмещения двух облаков точек (Рисунок 2.3). Нарастание индекса точек на изображениях изображено цветовым градиентом.

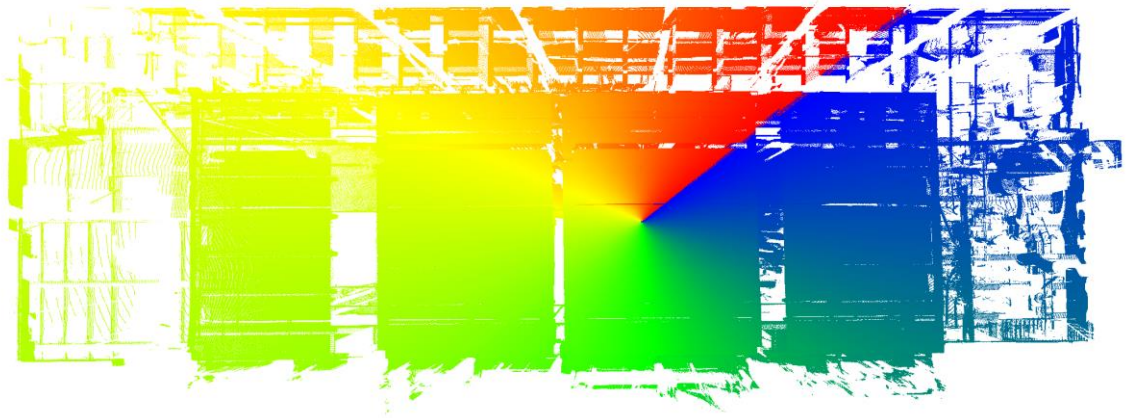

Рисунок 2.1 – Нарастание индекса точки в облаке точек лазерного сканирования

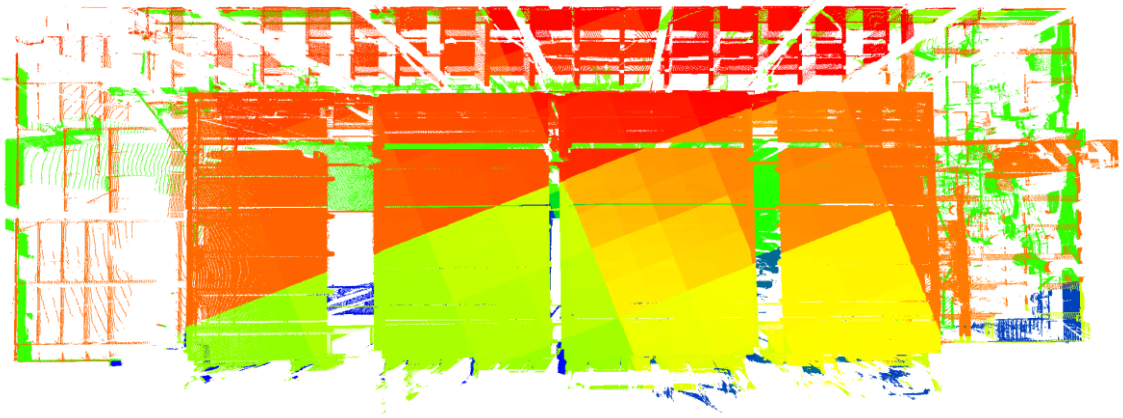

Рисунок 2.2 – Нарастание индекса точки после обработки

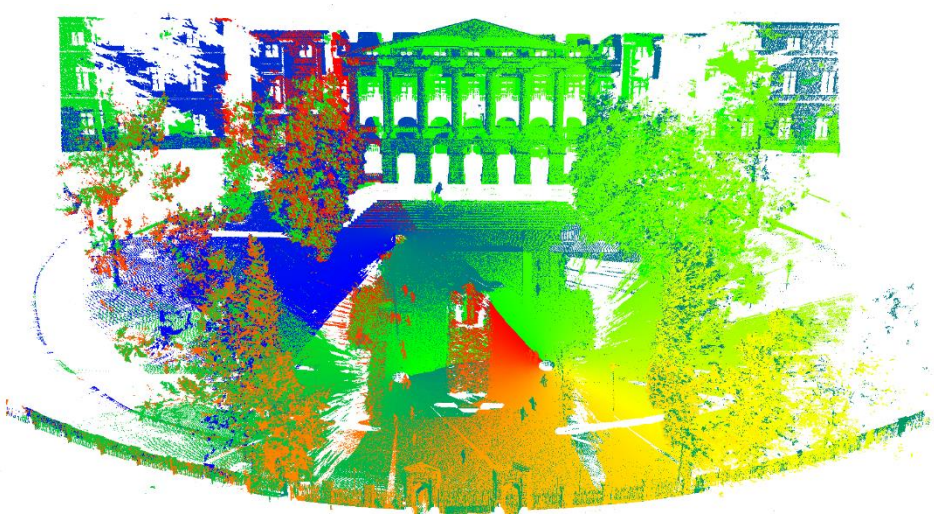

Рисунок 2.3 – Нарастание индекса точек при объединении облаков с двух точек сканирования

Для понимания цветового представления индекса точки, рассмотрим подробнее рисунок 2.1, на котором приведено облако точек, полученное непосредственно после лазерного сканирования. Лазерный сканер располагался в центре облака, а сканирование производилось путем поворота головки сканера в горизонтальной плоскости, с выполнением на каждой итерации сканирования вертикального сегмента. Если представить номер по порядку точки в облаке в виде цветового градиента, то соседние точки в облаке (но не в пространстве) будут иметь схожий цвет. Плавное нарастание градиента на рисунке 2.1 говорит о том, что точки, находящиеся рядом в облаке, также находятся рядом в пространстве. Разрыв говорит о том, что точки, находящиеся рядом в пространстве, находятся далеко друг от друга в облаке точек (например, разрыв, возникающий на стыке первых и последних линий сканирования).

Таким образом, приведенные рисунки показывают, что для некоей точки из облака ее соседи (точки, находящиеся рядом в пространстве) могут располагаться как поблизости в облаке точек (на той же или соседних линиях сканирования, рисунок 2.1), так и в другом конце облака точек (рисунки 2.2-2.3).

Для алгоритмов, не требующих информации о пространственном взаимоотношении между точками, расположение отдельных точек в облаке не имеет значения. Однако для алгоритмов, использующих такую информацию, требуется обеспечить возможность выборки точек в определенном пространственном регионе, что затруднительно сделать без структуры разбиения пространства.

В качестве примера алгоритма, требующего информацию о пространственном отношении между точками, рассмотрим алгоритм вычисления нормалей к поверхности, описываемой облаком точек, приведенный в статье [49]. Полученные нормали могут быть использованы для расчета освещения [50], сегментации [51] и в прочих алгоритмах. Для вычисления нормали определенной точки в облаке требуется выполнить поиск соседних с ней точек в определенном радиусе. В простейшем случае для полученных точек вычисляется такая плоскость, расстояние до которой у каждой из точек будет минимальным. Перпендикуляр к этой плоскости и будет искомой нормалью. Знак нормали можно вычислить, зная точку сканирования/взгляда. Пример такого алгоритма приведен на рисунке 2.4 (плоскость на рисунке сдвинута для наглядности).

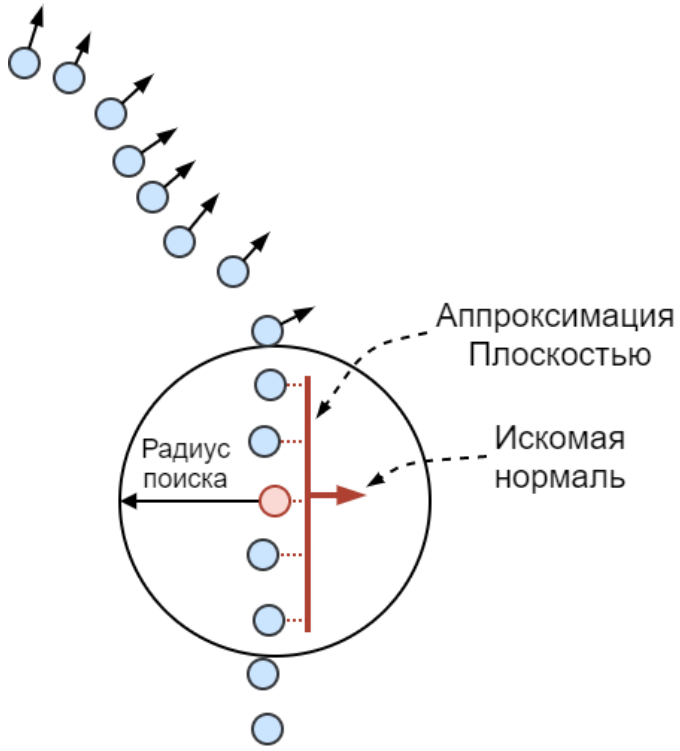

Рисунок 2.4 – Пример алгоритма вычисления нормалей в облаке точек

Для ускорения поиска в таких алгоритмах используются структуры разбиения пространства. В качестве полезной нагрузки в них могут храниться исходные точки из облака точек или их индексы в облаке точек. С точки зрения производительности первый вариант предпочтительнее, так как позволяет сгруппировать рядом в оперативной памяти близкие в пространстве точки, что существенно снижает вероятность кеш-промахов при обработке таких данных.

2.1.2. Обзор сценариев обработки облака точек

Выделим основные этапы обработки облака точек. Прежде всего, это загрузка исходных данных. Загрузка облака точек нужна для преобразования данных, полученных из некоего внешнего источника, во внутренний формат хранения данных, понятный используемому алгоритму. Под получением данных из внешнего источника подразумевается считывание из файла определенного формата, получение по сети и т.п. Так как в большинстве случаев исходное облако точек представлено в виде файла, далее мы будем рассматривать именно этот вариант.

Этап загрузки может быть совмещен с этапом предобработки. Именно на этом этапе выполняется построение структуры разбиения пространства. Далее выполняется этап обработки облака точек. В нашем случае под алгоритмом обработки подразумевается черный ящик, принимающий на вход облако точек и набор параметров. Выходными данными алгоритма тоже является облако точек и набор значений. Алгоритм может как изменять входное облако точек, так и создавать новое. Последовательно может применяться цепочка алгоритмов. Заключительным этапом является вывод результатов обработки (облака точек, изображения, и т.п.) в устройство вывода (экран, файл, и т.п.).

Рассмотрим возможные сценарии обработки облака точек алгоритмами, не требующими построения структуры разбиения пространства (Рисунок 2.5):

1. Алгоритм выполняет обработку облака точек путем последовательного считывания участков облака точек из файла, их изменением, и записью обратно в файл;
2. Перед вызовом алгоритма выполняется загрузка облака точек в оперативную память. В процессе обработки алгоритм выполняет изменение участка облака точек напрямую в оперативной памяти. Результат отправляется на визуализацию/записывается в файл;
3. Отличается от варианта №2 тем, что выполняет создание отдельного облака точек для каждого этапа обработки, что приводит к повышенному потреблению оперативной памяти;
4. Представляет собой вариант потоковой обработки облака точек. Считанный участок облака точек последовательно обрабатывается всеми алгоритмами, после чего процесс повторяется для следующего считанного участка. Подобный подход позволяет снизить количество файловых операций, а также отличается низким потреблением оперативной памяти. Недостатками такого подхода являются повышенные требования к реализации алгоритмов обработки.

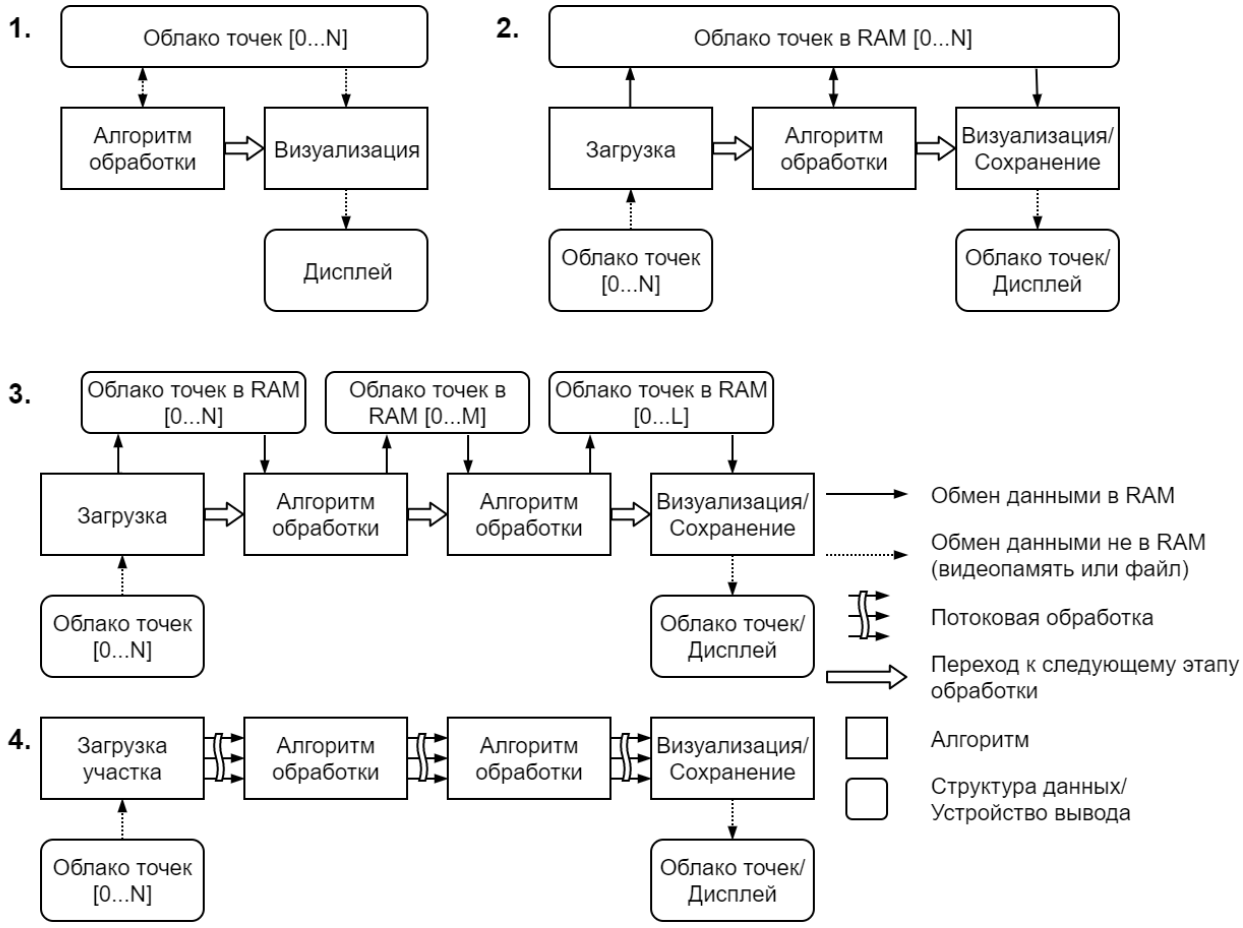

Рисунок 2.5 – Пример обработки облака точек без выполнения разбиения пространства

Рассмотрим возможные сценарии обработки облака точек с построением структуры разбиения пространства (Рисунок 2.6):

1. На этапе загрузки выполняется построение структуры разбиения пространства, представленной набором узлов, имеющих древовидную или линейную иерархическую модель и содержащих данные облака точек. Построение октодерева выполняется в оперативной памяти. Алгоритм обработки выполняет считывание узлов из структуры разбиения в произвольном порядке, изменяет данные и сохраняет результаты в структуре разбиения;
2. Отличается от варианта №1 тем, что позволяет выполнять сохранение содержимого узлов структуры разбиения на вторичные системы хранения, тем самым сокращает объем потребляемой оперативной памяти.

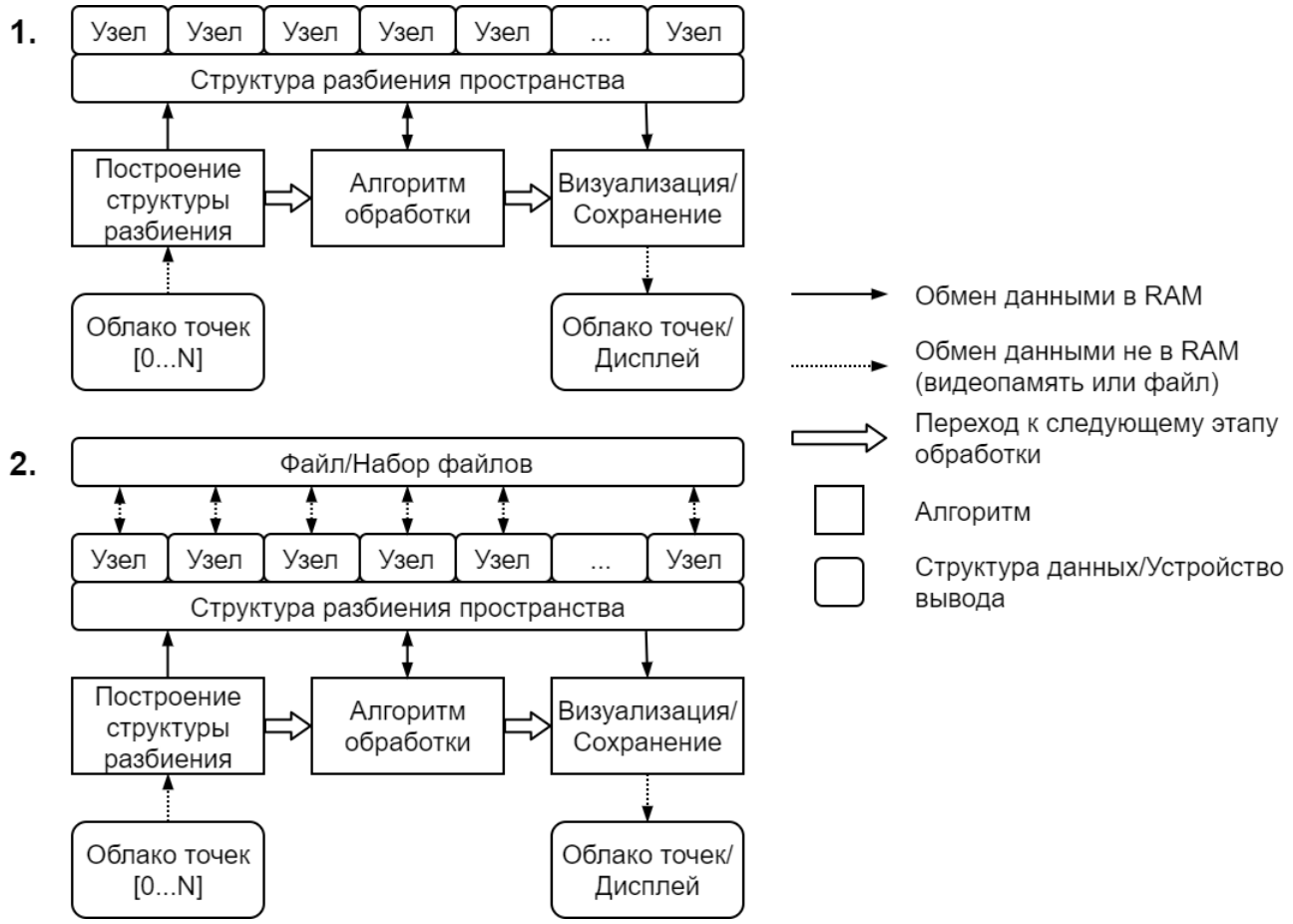

Рисунок 2.6 - Пример обработки облака точек с построением структуры разбиения пространства

Пример обработки облака точек без разбиения пространства используется, когда возможна последовательная обработка точек в облаке, т.е. когда из очередной точки исходного облака можно получить очередную точку результирующего облака. Примером такой операции может быть применение аффинного преобразования к облаку точек.

В случае, когда для получения очередной точки необходим произвольный набор точек из исходного облака (отвечающий определенному критерию), последовательная обработка невозможна. В таком случае обычно применяются вспомогательные структуры, позволяющие выполнить структурирование исходного облака точек. В результате такой обработки, все точки, отвечающие определенному критерию, будут расположены рядом в памяти. Как правило, критерием выступает пространственное расположение точки, а вспомогательной структурой является структура разбиения пространства.

Используя такое разбиение, процесс обработки также можно сделать последовательным: без использования структуры разбиения будет производиться считывание исходного облака точек, иначе – очередного блока в структуре разбиения (данные которой тоже могут храниться на жестком диске). Недостатком такого способа будут дополнительные временные затраты на построение структуры разбиения пространства, а также необходимость хранения дополнительных данных на жестком диске.

Рассмотрим пример обработки облака точек с построением структуры разбиения пространства с точки зрения потребления памяти. В качестве исходных данных примем, что:

- Облако точек содержит N точек;
- Результат алгоритма обработки содержит столько же точек, сколько исходное облако (для наглядности отображения);
- Размер точки в процессе обработки не меняется.

Без сохранения данных на жестком диске при размере исходного облака точек N в процессе работы потребуется N * (кол-во алгоритмов + 1) памяти (Рисунок 2.7). Рассмотрим худший случай, когда каждый алгоритм обработки создает собственный набор данных, а не изменяет данные предыдущего этапа обработки.

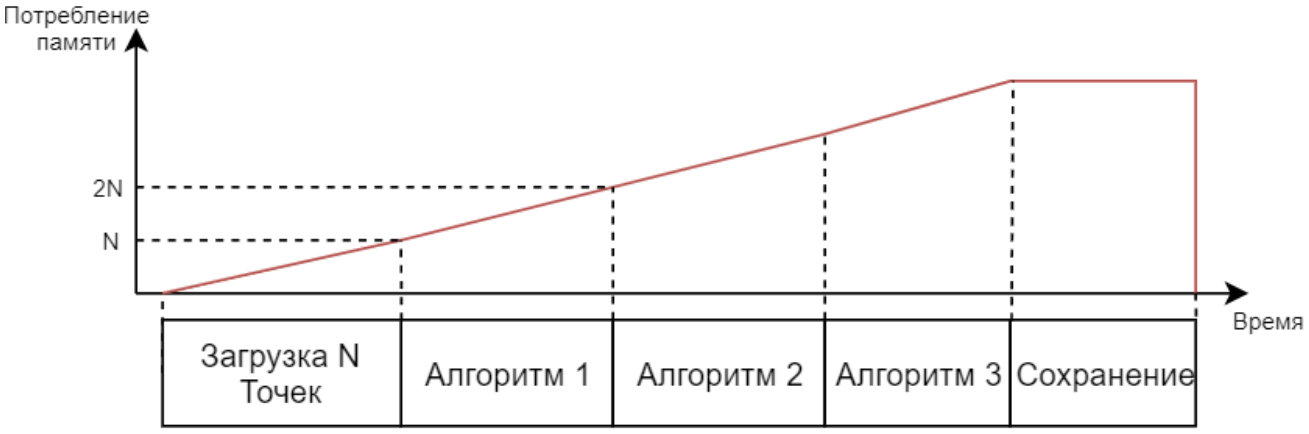

Рисунок 2.7 – Потребление памяти без сохранения данных на жестком диске

Однако в задачи работы так же входит реализация возможности ограничения потребляемой памяти. Такой контроль необходим как для результатов обработки, так и для данных, получаемых при разбиении пространства. Для реализации механизма ограничения потребляемой памяти дополним систему кешем в оперативной памяти с выгрузкой на жесткий диск (Рисунок 2.8).

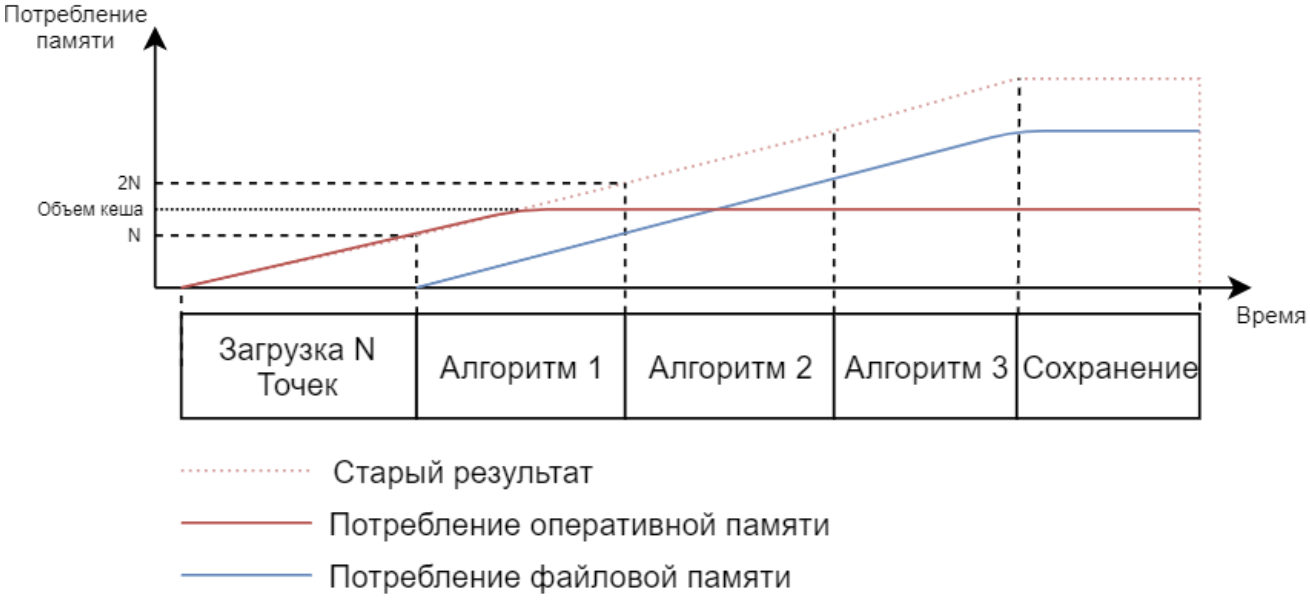

Рисунок 2.8 - Потребление памяти с кешированием на жесткий диск

Как видно из рисунка 2.8, при превышении заданного лимита потребления все не использующиеся в данный момент участки облака точек будут выгружаться на жесткий диск. Это особенно востребовано в случае с иерархическими структурами разбиения: наиболее часто запрашиваемые участки (более близкие к корню) будут всегда оставаться в кеше.

2.2. Получение оценок сложности доступа к информации из облака точек

2.2.1. Оценка сложности доступа к информации в облаке точек

Произведем оценку сложности доступа к данным в облаке точек для операций вставки, удаления и получения элемента (точки). Пусть облако точек представлено в виде динамического массива точек, последовательно расположенных в памяти.

Вставка элементов. При вставке элементов в динамический массив различают две ситуации: вставка в середину массива и вставка в конец. Учитывая, что отличительной особенностью такого массива является непрерывность в занимаемой памяти, для вставки в середину потребуется обеспечить перемещение всех последующих элементов на одну позицию вперед. Худшим случаем будет вставка в начало массива, что потребует перемещения всех элементов, что дает вычислительную сложность $O(n)$. При вставке элементов в конец массива нет необходимости производить перемещение элементов, но учитывая особенности механизма выделения памяти в таких массивах, при превышении его емкости потребуется выделение нового участка памяти и перемещение в него всех данных облака точек. В таком случае сложность добавления элемента в конец составит $O(n)$, однако сложность добавления n элементов составит $O(2n-1)$ (при увеличении размеров согласно степени двойки). Таким образом, амортизационная (усредненная) сложность добавления элемента составляет $O(1)$.

Удаление элементов. Аналогично с операцией вставки, удаление элементов также требует перемещения всех последующих, что дает вычислительную сложность $O(n)$. Удаление последнего элемента имеет вычислительную сложность $O(1)$, так как не требует перемещения элементов массива.

Получение элемента. Учитывая непрерывное расположение в памяти, местоположение элемента определяется его номером по порядку, что дает сложность доступа к элементу $O(1)$ и сложность обхода всего массива $O(n)$.

Более подробно оценки сложности приведены для динамического массива (и прочих) в [52].

2.2.2. Оценка сложности пространственного поиска в облаке точек

Пространственный поиск применяется при необходимости найти данные (точки), соответствующие определенному пространственному признаку. Таким признаком может быть, например, принадлежность некому пространственному объему или соседство с другой точкой. Такой поиск широко [53] применяется в алгоритмах обработки облака точек: удаления шумов [54], сегментации [55], визуализации [41], построения геометрических поверхностей [56] и пр.

На неподготовленном облаке точек такой поиск обычно занимает продолжительное количество времени. В общем случае, для поиска по определенному критерию (например, принадлежности пространственному объему) необходимо проверить на соответствие все точки в облаке. Поэтому для ускорения пространственного поиска обычно используются структуры разбиения пространства, которые позволяют сократить количество необходимых проверок.

Однако, прежде чем мы перейдем к рассмотрению подобных структур, стоит оценить производительность выполнения пространственного поиска на неструктурированных данных. Такой подход может быть оправдан, если сложность поиска будет меньшей, чем сложность построения структуры разбиения – т.е. точек будет очень мало.

Рассмотрим операцию поиска соседей произвольной точки p_i, принадлежащей облаку точек P.

Пусть существует вектор N_i, содержащий индексы k соседей точки $p_i \in P$ в порядке возрастания дистанции от точки p_i:

$$N_i = (n_1, \dots, n_k) : d(p_n, p_i) \leq d(p_{n+1}, p_i), n \neq p_i, |N_i| = k, \tag{2.1}$$

где d – евклидово расстояние между точками, вычисляемое по первому атрибуту точки (a_{coord}):

$$\begin{gathered} d(p_n, p_i) = d\left(a^n_{coord}, a^i_{coord}\right), \\ d\left(a^n_{coord}, a^i_{coord}\right) = (x_n - x_i)^2 + (y_n - y_i)^2 + (z_n - z_i)^2. \end{gathered} \tag{2.2}$$

Простейший подход к построению такого вектора заключается в последовательном расчете дистанций от точки p_i до каждой точки $p_n \neq p_i,\ p_n \in P$. При этом отслеживается k ближайших точек на текущий момент. Отслеживание может быть выполнено путем вставки в сортированный массив, содержащий максимум k точек (неактуальные точки удаляются).

Определим временную сложность построения такого вектора:

$$G = \langle P, N_i \rangle, \tag{2.3}$$

областью определения N_i является множество P:

$$P \to N_i, |P| = n. \tag{2.4}$$

Таким образом, нужно рассмотреть все элементы $p_j \in P$ и для каждого вычислить дистанцию до p_i, после чего, согласно полученной дистанции, выполнить вставку элемента в сортированный массив. Сложность такой вставки - $O(n)$, так как приходится выполнять последовательный поиск места для вставки.

Пространственный поиск в облаке точек без построения вспомогательных структур данных осуществляется при помощи полного перебора всех элементов. При поиске k ближайших элементов к определенной точке в пространстве вычислительная сложность поиска равна $O(kn)$.

Стоит отметить, что существует возможность оптимизации данного алгоритма, например, путем совершенствования алгоритма вставки или при помощи алгоритма быстрой сортировки. Согласно области определения N_i сложность такого алгоритма поиска будет не меньше, чем $O(n)$.

2.2.3. Оценка сложности доступа к информации в октодереве

Произведем оценку сложности доступа к информации в октодереве для операций разбиения узла, вставки, удаления и получения элемента. Октодерево является древовидной структурой, каждый узел которого содержит восемь потомков, а также содержит список точек, которые в него попадают. Максимальное количество точек в узле ограничено.

Разбиение узла. Если при добавлении новой точки в узел он содержит максимально допустимое количество элементов, происходит его разбиение на восемь подузлов. Согласно [57] в идеально сбалансированном октодереве, содержащем n листовых узлов и имеющим глубину разбиения $\log n - 1$, будет $\frac{8n-1}{7}$ узлов, а временная сложность полного разбиения октодерева, содержащего n точек составит $T(n) = Kn + 8 * T\left(\frac{n}{8}\right) = O(n)$, где K – стоимость обработки одной точки. Таким образом, амортизационная сложность разбиения одного узла составит $O(1)$.

Вставка элементов. Добавление нового элемента в октодерево требует поиска подходящего узла для вставки. Поиск выполняется рекурсивно, начинается с корневого узла и продолжается для его потомков до тех пор, пока не будет найден подходящий листовой узел. Учитывая то, что на каждом уровне разбиения размеры узла делятся пополам, процесс поиска подходящего узла будет иметь сложность $O(\log n)$ ($O(n)$ в случае сильно несбалансированного дерева), а процесс добавления новой точки в узел будет иметь сложность $O(1)$.

Удаление элементов. Процесс удаления точек из октодерева выполняется за $O(1)$, однако процесс перестроения узлов октодерева может занимать $O(n)$. С учетом того, что процесс перестроения не выполняется после удаления каждой точки, можно сказать, что удаление имеет временную сложность $O(1)$.

Получение элемента. Получение элемента также требует поиска содержащего элемент узла ($O(\log n)$). Выборка элемента из массива точек узла имеет сложность $O(1)$, так как точки в таком массиве расположены в памяти непрерывно.

2.2.4. Оценка сложности пространственного поиска в октодереве

Иерархическая организация октодерева позволяет на этапе поиска выполнять отсечение узлов, не пересекающихся в пространстве с искомым объемом. В общем случае поиска точек, находящихся в определенном радиусе, выполняется рекурсивный обход октодерева с отсечением узлов, не соприкасающихся со сферой поиска [53]. Сложность подобного поиска для облака точек размером n составляет $O(\log n)$ [58].

Поиск ближайших соседей точки является более сложным алгоритмом, и существует множество техник его реализации [59]. При использовании древовидной структуры разбиения обычно требуется не менее двух шагов, чтобы найти точного ближайшего соседа точки. Сначала выполняется поиск листового узла, соответствующего точке. Поскольку точка запроса может быть ближе к границе объема узла, чем к точкам данных, содержащимся в листовом узле, обратный поиск по дереву требуется в качестве второго шага для продолжения поиска в соседних узлах, что делает сложность поиска равной $O(L)$, где L – глубина дерева [60].

2.3. Анализ организации хранения октодерева облака точек в памяти компьютера

Пусть структура разбиения пространства представлена набором узлов $N = \{n_1, \dots, n_m\}$ и листьев $L = \{l_1, \dots, l_k\}$, каждый из которых имеет уникальный идентификатор $R^{id} = \left\{r^{id} \middle| r^{id} = id(x), id(x): x \to r^{id}, x \in N \cup L\right\}$, а также множество дуг, представляющих отношения между узлами и листьями $R^{nl} = \{R_1^{nl}, \dots, R_k^{nl}\}$.

Идентификатор в октодереве обычно привязан к расположению узла или листа в иерархии октодерева и позволяет осуществлять доступ к узлу вне зависимости от его расположения в оперативной (или внешней) памяти. Выбор способа идентификации оказывает существенное влияние на скорость операций обхода октодерева, которые используются в том числе при формировании октодерева и выполнении операций пространственного поиска.

Октодерево, сформированное по облаку точек, содержит листья (в которых выполняется хранение точек из облака, распределенных по пространственному признаку), узлы и дуги (которые представлены указателями, являющимися частью структуры данных узла). Иерархия узлов октодерева, листья, а также данные облака точек, распределенные между листьями, можно представить в виде блоков данных, которые могут быть расположены как в оперативной, так и во внешней памяти. Рассмотрим потребляемую память с такой позиции.

Учитывая необходимость быстрого доступа к узлам в иерархии октодерева, а также их совокупный размер, значительно меньший, чем размер облака точек, было принято решение выполнять их хранение в оперативной памяти. Листья октодерева, как часть иерархии, также хранятся в оперативной памяти, в отличии от данных облака точек, на которые они ссылаются. Таким образом, под блоками данных, хранение которых возможно как в оперативной, так и во внешней памяти, в дальнейшем подразумеваются блоки точек, распределенные между листьями октодерева. Для подсчета занимаемой ими оперативной и внешней памяти, введем:
V – множество блоков данных, принадлежащих листьям

$$V = \{v_1, \dots, v_k\}, \tag{2.5}$$

где v_i – блок данных, принадлежащий листу октодерева l_i;

R^{LeafId} – множество идентификаторов листьев, по которым также ведется обращение к блокам данным листьев

$$R^{LeafId} \subset R^{id} | \forall a \in R^{LeafId} : id^{-1}(a) \in L, \tag{2.6}$$

где $id^{-1}(a)$ – функция получения узла или листа по идентификатору;

U – упорядоченная последовательность запросов к блокам данных в процессе обработки

$$U = (u_1, \dots, u_n), u \in R^{LeafId}, \tag{2.7}$$

где u_i – это обращение к блоку данных листа октодерева по его идентификатору в момент времени t_i при условии, что время дискретно и в один момент времени может быть только одно обращение.

Введем $V^{RAM}(t)$ – множество блоков данных, находящихся в оперативной памяти в момент времени t, и $V^{HDD}(t)$ – множество блоков данных, находящихся во внешней памяти в момент времени t. Также можно выразить множество создаваемых файлов во внешней памяти как $F(t) = \{F_i\}, 0 \leq i \leq k$, где k – общее количество блоков данных во внешней памяти в момент времени t.

Для подсчета потребляемой памяти введем функцию получения размера блока данных в байтах $M(v): v \to b, v \in V, b \in \mathbb{R}$. Тогда объем потребляемой оперативной памяти можно выразить как $M^{RAM}(t) = \sum_{v \in V^{RAM}(t)} M(v)$, а объем потребляемой внешней памяти как $M^{HDD}(t) = \sum_{v_i \in V^{HDD}(t)} M(v)$.

Преобразование расположения данных при операциях обработки u_i соответствует определению:

$$f(t_i, U): V^{RAM}(t_i), V^{HDD}(t_i), F(t_i) \xrightarrow{U} V^{RAM}(t_{i+1}), V^{HDD}(t_{i+1}), F(t_{i+1}). \quad (2.8)$$

Пусть T^{HDD} представляет время, дополнительно затрачиваемое на работу с внешней памятью:

$$T^{HDD} = T^{read} + T^{write} + T^{service}, \quad (2.9)$$

где T^{read} – время, затраченное на считывание данных;

T^{write} – время, затраченное на запись данных;

$T^{service}$ – время, затраченное на операции обслуживания (создание файлов, получение файловых дескрипторов, увеличение размеров файлов и прочее).

2.4. Анализ иерархической модели октодерева и способов кодирования узлов

Одной из ключевых задач при проектировании структуры октодерева является определение минимально необходимого набора данных, требуемого для представления узла и обеспечения навигации по октодереву. Способ идентификации узлов напрямую влияет на занимаемую иерархией октодерева память, скорость обхода октодерева, гибкость иерархической структуры октодерева к различным видам входных данных. При условии обработки больших объемов данных особенно важным является выбор эффективного решения. Определим список важных характеристик рассматриваемых иерархических моделей октодерева:

- Занимаемое место в памяти – определяет размер структурных данных октодерева;
- Количество операций, необходимое для определения принадлежности точки потомку узла – определяет скорость обхода и скорость поиска по октодереву;
- Возможность масштабирования количества потомков узла и размерности пространства октодерева – не является необходимым, однако может помочь адаптировать структуру октодерева к различным вариациям входных данных.

Рассмотрим различные способы представления иерархической структуры октодерева.

2.4.1. Общая структура узла октодерева

Данный раздел является стартовой точкой в задаче проектирования узла октодерева. Рассмотрим избыточное представление такого узла, предложенного в работе [31]. Данный узел занимает в 64-разрядной архитектуре не меньше 100 байт и содержит координаты центра узла (center), размеры узла (size), восемь указателей на потомков (child), количество точек (nr_points) и указатель на массив точек (points).

Листинг 2.1 – Избыточное представление узла октодерева

```
struct OcTree {
   float center[3];
   float size[3];
   OcTree *child[8];
   int nr_points;
   float **points;
};
```

В той же работе предложена методика сокращения размеров узла за счет:

1. Отказа от хранения центра и размеров узла, так как они могут быть вычислены в момент обхода дерева;
2. Хранения одного указателя на массив потомков вместо восьми указателей на узлы, а также введения 8 бит для идентификации существующих потомков;
3. Использование урезанных 6-байтовых указателей;
4. Сокращение размера точки, путем использования только двух байт на координату.

Подобные меры позволяют сократить размер узла как минимум до 21 байта. В данной работе по умолчанию будут применяться подходы №1 и №2 из вышеприведенного списка, так как они позволяют сократить расходы памяти без большого ущерба производительности. Однако, в виду возможных проблем при кроссплатформенной реализации в данной работе не будет применен подход №3. По причине необходимости декодировки точек при расчетах и сложности интеграции со сторонними алгоритмами обработки облака точек не будет применен подход №4.

2.4.2. Представление узла октодерева на арифметике с плавающей точкой

Рассмотрим структуру узла октодерева, построенную с использованием методики, описанной в 2.4.1. Учитывая, что позиция и размеры узлов в октодереве определяются корневым узлом, нет необходимости в их хранении для каждого узла, так как они могут быть вычислены в процессе обхода. Стоит так же отметить, что для определения области пространства, занимаемой октодеревом, обычно используется арифметика с плавающей точкой, что позволяет адаптировать его к широкому диапазону входных данных. В таком случае узел будет задан тремя интервалами по каждой из координатных осей (пример для одномерного случая приведен на рисунке 2.9). При этом координата и размеры корневого узла октодерева могут быть представлены любым значением с плавающей точкой.

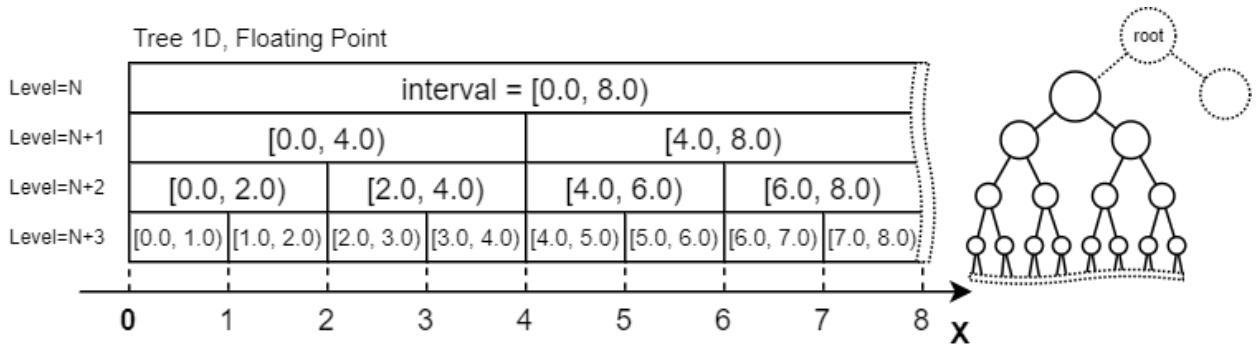

Рисунок 2.9 - Фрагмент октодерева на арифметике с плавающей точкой

Узел такого октодерева содержит в себе как минимум указатель на восемь потомков, 8 бит для идентификации непустых потомков, а также полезные данные в виде массива точек.

При добавлении новой точки в дерево происходит рекурсивная проверка на попадание точки в подузел, начиная с корневого узла. В общем случае, для получения подузла, в который попадает точка, можно использовать три операции сравнения, что изображено в виде алгоритма в листинге 2.2.

Листинг 2.2 – Алгоритм определения подузла, в который попадает точка

```
int GetInsertionIndex(vec3 pointCoord, vec3 nodeCenter)
{
  if(pointCoord.x < nodeCenter.x)
  {
    if(pointCoord.y < nodeCenter.y)
    {
      if(pointCoord.z < nodeCenter.z)
        return 0;
      else
        return 1;
    }
    else
    {
      if(pointCoord.z < nodeCenter.z)
        return 2;
      else
        return 3;
    }
  }
  else
  {
                // Same principle…
  }
}
```

Однако эффективность такого алгоритма может быть существенно улучшена за счет отказа от условных операторов. Расположение подузла внутри узла можно закодировать при помощи трех бит, по биту на ось разбиения (Таблица 2.1). Код подузла принимает значения от 0 до 7 и может использоваться в качестве индекса в массиве потомков узла, позволяя однозначно сопоставить пространственное местоположение и местоположение в памяти.

Таблица 2.1 – Идентификация потомков узла

Позиция относительно центра узла	Код	Наименование
$-x; -y; -z;$	*0b000*	Left Back Bottom
$+x; -y; -z;$	*0b001*	Right Back Bottom
$-x; +y; -z;$	*0b010*	Left Front Bottom
$+x; +y; -z;$	*0b011*	Right Front Bottom
$-x; -y; +z;$	*0b100*	Left Back Top
$+x; -y; +z;$	*0b101*	Right Back Top
$-x; +y; +z;$	*0b110*	Left Front Top
$+x; +y; +z;$	*0b111*	Right Front Top

Использование подобного индекса позволяет ускорить процесс вычисления подузла, которому принадлежит точка в пространстве за счет использования бинарной арифметики:

$$InsertionIndex = sgn(x - cx) * 2^1 + sgn(y - cy) * 2^2 + sgn(z - cz) * 2^3, \quad (2.10)$$

где sgn – операция определения знака числа (0 – для отрицательного, 1 для положительного), операции умножения на степень двойки эквивалентны операциям побитового сдвига;

x, y, z – координаты точки;

cx, cy, cz – координаты центра узла.

Рассмотрим прочие операции, необходимые для формирования октодерева. Так как узел теперь не выполняет хранение своих размеров и координат, необходимо определить способ их вычисления. Пусть октодерево задано следующими параметрами:

- *RootCoord* – координаты корневого узла октодерева;
- *RootSize* - размеры корневого узла октодерева;
- $Depth$ - максимальная глубина дерева.

Количество разбиений по стороне октодерева можно вычислить по формуле:

$$\mathrm{SideSize(level)} = 2^{\mathrm{level}}, \tag{2.11}$$

где *level* – уровень разбиения дерева.

Минимальным уровнем разбиения будет нулевой, что соответствует ***корневому*** узлу. Размер узла на определенном уровне разбиения можно вычислить по формуле:

$$NodeSize(level) = \frac{RootSize}{SideSize(level)}, \tag{2.12}$$

где *NodeSize* – размер узла на координатной прямой.

Координаты подузла можно вычислить, зная его ориентацию и координаты родительского узла:

$$OrientationBits(NodeIndex) \tag{2.13}$$
$$= (NodeIndex \,\&\, 0b001, NodeIndex \,\&\, 0b010, NodeIndex \,\&\, 0b100),$$

$$NodeCoord(level, orientationBits, parentCoord) \tag{2.14}$$
$$= parentCoord + NodeSize(level) * orientationBits,$$

где $OrientationBits$ – биты индекса подузла;

$NodeIndex$ – индекс подузла;

& – операция побитового “И”;

$0bxxx$ – двоичная форма записи числа;

$parentCoord$ – координаты родительского узла.

Максимальное количество узлов на определенном уровне разбиения можно вычислить по формуле:

$$LevelVolume(level) = 8^{level}. \tag{2.15}$$

В подходе с плавающей точкой координаты и размеры узлов представлены floating-point значениями. Такое решение позволяет поддерживать более широкий диапазон входных значений. Недостатком подобного подхода является снижение точности на слишком высоких и слишком малых величинах [61], а также повышенная сложность вычислений по сравнению с целочисленным подходом.

2.4.3. Представление узла октодерева на базе целочисленного идентификатора

Данное решение является переходным шагом между целочисленным и floating-point октодеревом. Его отличительной особенностью является использование целочисленного идентификатора для кодирования местоположения узла в октодереве, который при наличии информации о корневом узле позволяет получить размеры и местоположение узла. При этом иерархия октодерева остается аналогичной изображенной на рисунке 2.9 и сохраняется возможность позиционирования октодерева в пространстве при помощи floating-point координат.

Рассмотрим формат идентификатора узла (Рисунок 2.10). Идентификатор представлен 64-разрядным беззнаковым целым и сформирован способом, похожим на описанный в таблице 2.1: для идентификации расположения узла выше или ниже плоскости разбиения по каждой из осей (x,y,z) выделено по одному биту. Вместе эти три бита кодируют 8 возможных вариантов расположения в узле на определенном уровне разбиения. Каждые последующие три бита кодируют местоположение узла на очередном уровне разбиения (level 1 и level 2 на рисунке 2.10). Ведущий бит идентифицирует окончание разбиения (leading one на рисунке 2.10). Подобное решение примечательно тем, что позволяет получать индекс потомка или родителя узла путем выполнения простой комбинации операций смещения и маскирования бит идентификатора.

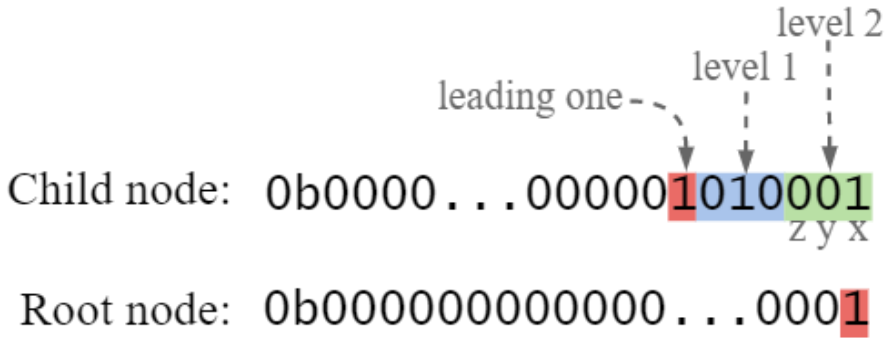

Рисунок 2.10 – Идентификатор узла октодерева в бинарной форме

Выполним расчеты основных параметров узла с использованием подобного идентификатора, который в дальнейшем будем называть *OctreeIndex*.

Уровень разбиения узла можно рассчитать при помощи взятия восьмеричного логарифма от идентификатора:

$$level(OctreeIndex) = \log_8 OctreeIndex, \quad (2.16)$$

где *level* – уровень разбиения.

Индекс потомка можно получить путем сдвига бит идентификатора влево и добавления бит, отвечающих за идентификацию потомка и описанных в таблице 2.1. Индекс родителя можно получить при помощи сдвига бит идентификатора вправо. Ориентацию узла можно получить в результате операции побитового “И” с числом 0b111. Определение размеров узла происходит аналогично описанному в п. 2.4.2.

Таким образом, появляется возможность идентификации узла в октодереве, которую можно использовать для получения идентификаторов потомков, родителей, а также уровня разбиения и размеров узла. Подобный идентификатор, в отличие от указателя, можно использовать для адресации узлов, выгруженных из оперативной памяти.

2.4.4. Представление узла целочисленного октодерева

В качестве альтернативы подходу с плавающей точкой можно предложить целочисленные вычисления. Согласно модели целочисленного октодерева, предложенной соискателем в статье [46] и основанной на реализации Volumetric Dynamic Grid для B+дерева из работы [62], используя иерархическую природу октодерева, можно обеспечить адресацию узлов в нем при помощи целочисленных координат, обеспечивающих однозначное сопоставление позиции узла в октодереве и его местоположение в пространстве. Размер узла в таком октодереве ограничен не сверху, как в подходе с плавающей точкой, а снизу, что позволяет наращивать размеры октодерева, указывая количество бит, выделенных на адресацию. Преимуществом такого подхода является возможность применения гибридного подхода в описании иерархии октодерева. Младшие биты ключа используются для адресации в октодереве, старшие - для адресации участков пространства в иной системе хранения данных. Таким образом, в отличие от классической реализации октодерева, описанной ранее, появляется возможность расширения участка пространства, занимаемого октодеревом, по мере добавления новых данных.

В отличие от октодерева с плавающей точкой, где минимальный размер узла стремится к нулю, в целочисленном октодереве минимальный узел будет иметь единичный размер.

Рассмотрим октодерево, заданное следующими параметрами:

- *D* – количество измерений в дереве (одномерное, двумерное, трехмерное);
- *Depth* – максимальная глубина дерева;
- *Dim* – логарифм от максимального количества разбиений узла вдоль оси координат. Отражает также количество бит, необходимое чтобы идентифицировать потомка. Для октодерева *Dim = 1*;
- *Resolution* – разрешение дерева, используется для представлений дробных интервалов.

Рассмотрим параметры узлов, которые будут использоваться при построении дерева:

$$NodeDim(level) = Dim * level, \quad (2.17)$$

где $NodeDim$ - логарифм от максимального количества разбиений узла вдоль оси координат для конкретного уровня разбиения, а также количество бит, необходимое? чтобы идентифицировать вложенных потомков;

$level$ – уровень разбиения;

$$NodeSize(level) = 2^{NodeDim(level)}, \quad (2.18)$$

где $NodeSize$ – размеры стороны узла в единичных узлах или максимальное количество разбиений вдоль оси координат;

$$NodeWorldSize(level) = Resolution * NodeSize(level), \quad (2.19)$$

где *NodeWorldSize* – размер узла в мировых координатах;

$$NodeCapacity(level) = 2^{D*NodeDim(level)}, \quad (2.20)$$

где $NodeCapacity$ – общее количество единичных узлов, которое содержит узел на данном уровне разбиения.

$$KeySize = NodeDim(Depth), \quad Depth = KeySize/Dim, \quad (2.21)$$

где $KeySize$ – размер координаты дерева в битах.

На рисунке 2.11 приведен пример разбиения узла на нулевом уровне при различном количестве измерений (*D*) и при различном количестве разбиений (*Dim*). Узел с параметрами *D=3* и *Dim=1* соответствует узлу октодерева.

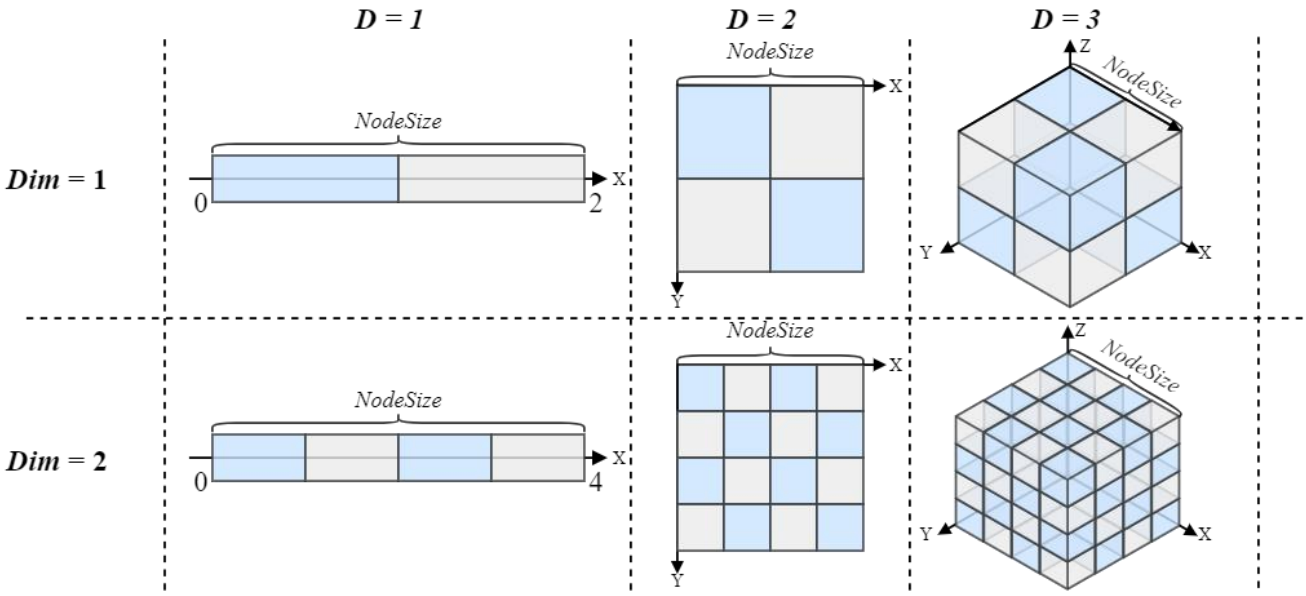

Рисунок 2.11 - Пример расположения потомков узла в пространстве при различном количестве измерений

Рассмотрим целочисленное одномерное ($D = 1$) дерево. В нем каждый узел будет иметь по два потомка ($Dim = 1$), в которых координата будет кодироваться при помощи трех бит ($Depth = 3$) (Рисунок 2.12).

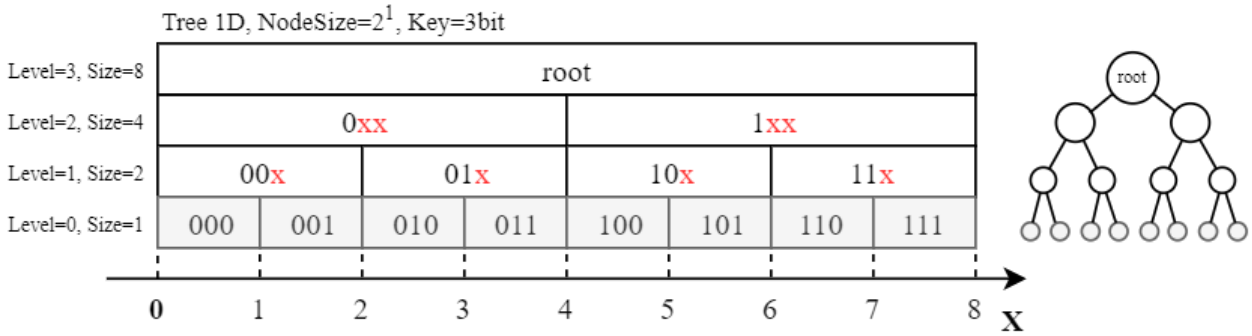

Рисунок 2.12 – Пример одномерного дерева, кодируемого при помощи одного бита на узел

Из рисунка 2.12 видно, что для идентификации узла такого дерева достаточно одного бита (0 – левый, 1 – правый). Таким образом конкретные биты координаты дерева позволяют идентифицировать узлы дерева. Иначе говоря, из координаты можно получить соответствующий узел на любом уровне разбиения.

Подобный подход может быть расширен до любого количества потомков на узел. Рассмотрим целочисленное одномерное ($D = 1$) дерево. Каждый узел в нем будет иметь по четыре потомка ($Dim = 2$), в которых координата будет кодироваться при помощи четырех бит ($Depth = 2$) (Рисунок 2.13).

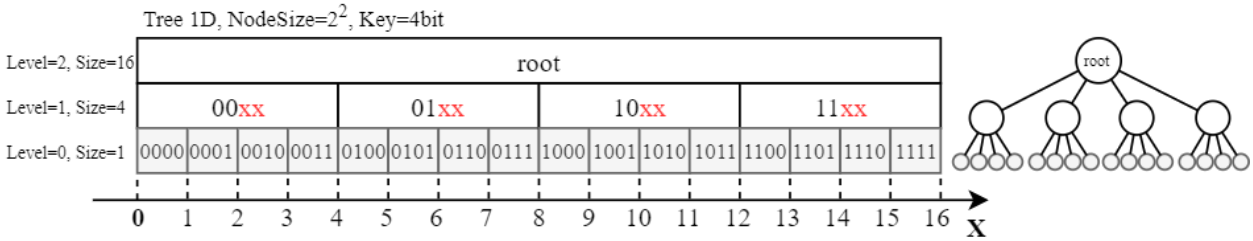

Рисунок 2.13 – Пример одномерного дерева, кодируемого при помощи двух бит на узел

Таким образом, в зависимости от количества возможных потомков узла, меняется количество бит, необходимых для идентификации определенного потомка. Вместе с этим соответственно изменяется количество уровней разбиения дерева, которое можно закодировать ключом определенного размера. Пример получения координаты узла на определенном уровне разбиения при различном количестве потомков узла приведен на рисунке 2.14.

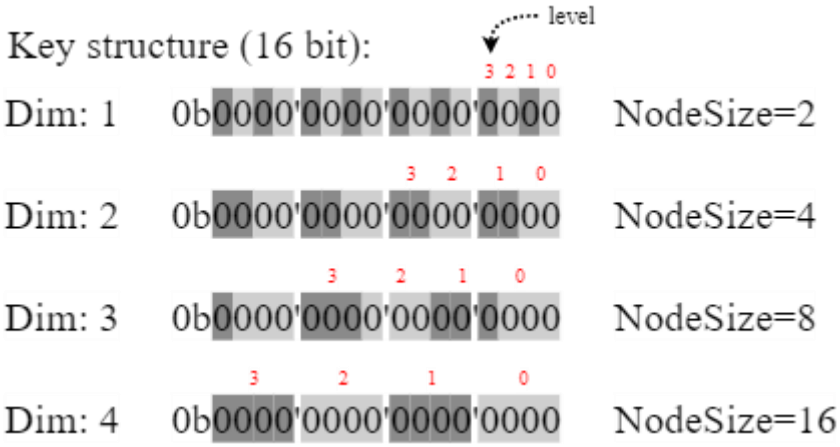

Рисунок 2.14 - Идентификация узлов целочисленной координатой при различном количестве потомков узла

Так как хранение потомков узла выполняется способом, аналогичным описанному в разделе 2.4.2, расположение подузла внутри узла также будет кодироваться при помощи трех бит, по биту на ось разбиения (Таблица 2.1). Однако формула определения попадания точки в узе теперь будет использовать только целочисленные операции:

$$InsertionIndex1D(x, level) = \frac{x \,\&\, (NodeSize(level) - 1)}{2^{NodeDim(level)}}$$
$$InsertionIndex(level) = InsertionIndex1D(x, level) * 2^1 + \quad (2.22)$$
$$InsertionIndex1D(y, level) * 2^2 \; + \; InsertionIndex1D(z, level) * 2^3 \,,$$

где x, y, z – целочисленные координаты в пространстве дерева;

& - операция побитового «И», умножение на степень двойки эквивалентно операции побитового сдвига.

2.4.5. Представление иерархии целочисленного октодерева

Рассмотрим иерархию узлов целочисленного октодерева. Для примера возьмем одномерное дерево ($D = 1$), каждый узел которого может содержать по четыре потомка ($Dim = 2$). Если использовать $KeySize = 16bit$, то подобное дерево сможет адресовать 65535 значений, а его максимальная глубина будет составлять 8 уровней.

При необходимости увеличения адресуемого диапазона можно увеличить размер ключа, что приведет к увеличению количества уровней дерева и замедлит скорость его обхода. Для решения подобной проблемы предлагается произвести интеграцию целочисленного дерева с ассоциативным массивом, что позволит использовать дерево с фиксированным размером ключа для любого диапазона значений.

Ассоциативные массивы [63] выполняют хранение элементов в виде пары {ключ:значение} и позволяют выполнять относительно быстрый поиск элементов по их ключам. Ассоциативные массивы плохо подходят для хранения точек, потому что не поддерживают операции пространственного поиска и не имеют древовидной структуры, полезной при раннем отсечении участков облака. Однако они могут быть использованы в совокупности с целочисленным октодеревом.

Пусть наше дерево имеет максимальную глубину в 3 уровня ($KeySize =$ 4 бита), но нам необходимо адресовать диапазон в 0-65535 значений. Для адресации такого диапазона нам необходимо еще 12 бит. Используем 16-битный ключ, первые 12 бит которого будут ключом в ассоциативном массиве, а последние 4 бита – ключом в целочисленном дереве. Полную схему такого решения можно увидеть на рисунке 2.15.

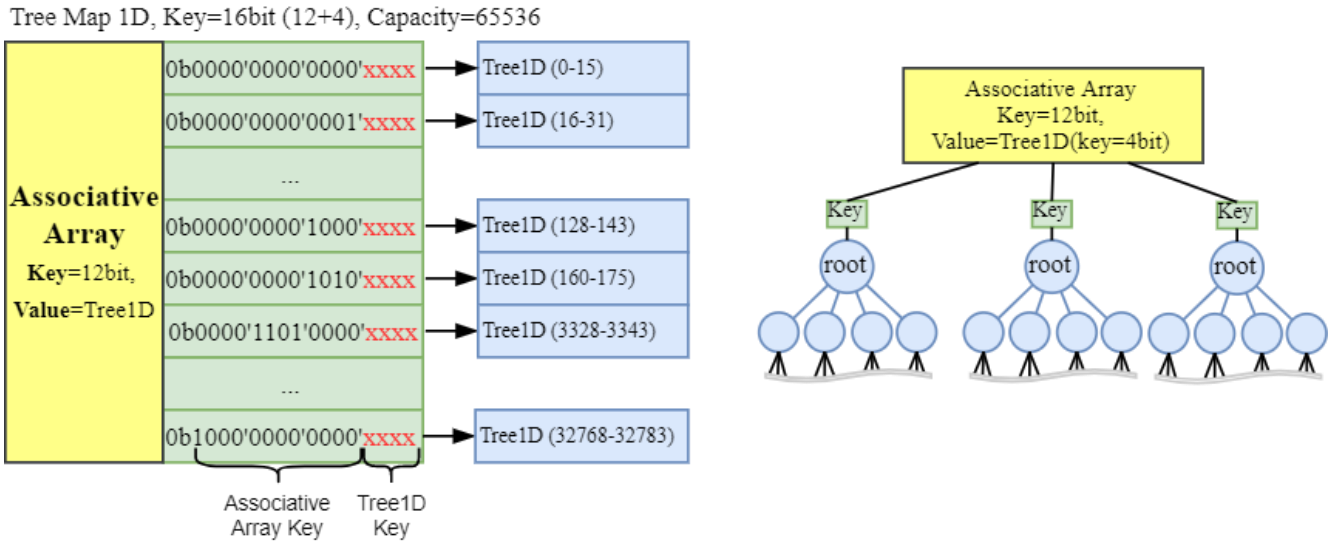

Рисунок 2.15 - Интеграция целочисленного дерева с ассоциативным массивом

2.4.6. Сравнение целочисленной и floating-point арифметики

Сравним операции определения узла, в который попадает точка, для рассматриваемых видов октодеревьев. Для этого исследуем приведенные ранее формулы и определим требуемое количество операций для вычисления. Воспользуемся тем фактом, что обе формулы имеют общую часть, которую можно не рассматривать.

Использование арифметики с плавающей точкой. Рассмотрим формулу, применяемую для вычисления позиции точки относительно центра узла вдоль оси x:

$$sgn(x - cx), \tag{2.23}$$

где sgn – операция получения знака выражения;

x – координата точки;

cx – центр узла.

В приведенной формуле стоит выделить следующие операции:

- Расчет cx – это floating-point операция, которая выполняется один раз при попадании точек в узел (так как узел не выполняет хранение данного параметра);
- $(x - cx)$ – это одна floating-point операция;
- sgn – это целочисленная операция определения бита знака у floating-point значения, выполняется за две операции (присутствует инверсия бита).

Таким образом, количество операций при выполнении вставки n точек в дерево глубиной $Depth$ составит:

$$Depth * n * (2\ floatOp + 2\ integerOp), \tag{2.24}$$

что при использовании $n = 100$, $Depth = 10$ составит 2000 floating point операций и 2000 целочисленных операций.

Алгоритм операции получения индекса вставки представлен в листинге 2.3.

Листинг 2.3 - Получение индекса вставки для координаты с плавающей точкой.

```
struct FloatCoord; // Floating point 3d vector {x,y,z}
struct FloatBBox;  // Floating point bounding box {min, max}
// Returns child index
uint8 GetInsertionIndex(FloatCoord crd, FloatBBox nodeBBox) {
        FloatCoord center = crd - nodeBBox.center(); // Coordinate relative to node center
   // Fast index calculation
   return !signbit(center[0]) |
          !signbit(center[1]) << 1u |
          !signbit(center[2]) << 2u;
}
```

Использование целочисленной арифметики. Рассмотрим формулу, применяемую для вычисления позиции точки относительно центра узла вдоль оси x:

$$\frac{x \,\&\, (NodeSize(level) - 1)}{2^{NodeDim(level)}}, \tag{2.25}$$

где x – целочисленная координата точки.

В приведенной формуле стоит выделить следующие операции:

– Расчет целочисленной координаты происходит путем умножения исходной на $Resolution$;

– Значение $(NodeSize(level) - 1)$ считается на этапе компиляции;

– Значение $2^{NodeDim(level)}$ считается на этапе компиляции.

Таким образом, количество операций вставки n точек в дерево глубиной $Depth$ составит:

$$n * (1\ floatOp) + Depth * n * (2\ integerOp), \tag{2.26}$$

что при использовании $n = 100$, $Depth = 10$ составит 100 floating point операций и 2000 целочисленных операций.

Алгоритм операции представлен в листинге 2.4.

Листинг 2.4 - Получение индекса вставки для целочисленной координаты

```
struct IntCoord; // Integer 3d vector {x,y,z}
struct IntBBox;  // Integer bounding box {min, max}
const int kLog2Dim = 1; // Dimension = 2^kLog2Dim

// Returns child index
uint8 GetInsertionIndex(IntCoord crd, int level) {
  // Calculated at compile time in lookup table (zero cost)
  const int kLog2Total = kLog2Dim + kLog2Dim * level;
  const int kLevelDim = (1 << kLog2Total) - 1;
  // Actual calculations
  return (((crd.x & kLevelDim) >> kLog2Total)) +
      (((crd.y & kLevelDim) >> kLog2Total) << 1 * kLog2Dim) +
      (((crd.z & kLevelDim) >> kLog2Total) << 2 * kLog2Dim);
}
```

2.5. Формирование структуры данных октодерева при различных методах использования памяти

В данном разделе производится анализ и уточнение компонент и деталей реализации для рассматриваемых в работе вариантов организации вычислительного процесса обработки. Дополнительно, в целях сравнения, рассматривается организация вычислительного процесса обработки в оперативной памяти.

Учитывая принципиально разные подходы к организации рассматриваемых моделей, целью данного анализа будет определение для каждого из методов:

- Наиболее подходящей иерархической модели октодерева;
- Организации механизма доступа к данным облака точек;
- Механизма ограничения потребляемой памяти;
- Определения характеристик, влияющих на производительность октодерева для различных конфигураций входных данных.

2.5.1. Метод формирования октодерева на базе двухуровневой асинхронной системы кеширования

Рассматриваемая система кеширования должна обеспечивать работу системы в условиях ограничений по оперативной памяти. На практике это означает, что данные облака точек в октодереве должны отслеживаться на протяжении всего цикла использования, обеспечивая подсчет занятой блоками данных оперативной памяти. При превышении заданного лимита часть блоков должна быть выгружена на жесткий диск. Таким образом, подобная система может быть представлена в виде двухуровневой системы кеширования, в которой, по аналогии с кешем процессора, первый уровень представлен более быстрой, но малочисленной оперативной памятью, а второй уровень – более медленной, но и более объемной внешней памятью.

Рассмотрим политику вытеснения, которая может использоваться для поддерживания заданного объема потребляемой памяти, чтобы определять, какие блоки следует выгружать в первую очередь. В ряде работ [15; 36; 39] хорошо себя зарекомендовала политика вытеснения LRU, в которой последний используемый блок помещается в начало очереди, вытесняя более старые в конец очереди. Описанный в работе [18] способ отслеживания на основе истории использования узлов тоже может быть применен, однако обладает следующим недостатком: количество отслеживаемых узлов ограничено, что лимитирует область применения.

Для реализации политики вытеснения LRU больше всего подходит двусвязный список, так как он позволяет производить вставку и удаление в начало и конец за O(1). Порядок использования в таком случае будет выглядеть следующим образом:

- При загрузке или использовании нового узла его идентификатор добавляется в начало списка;
- При необходимости освобождения памяти удаляется узел с идентификатором из конца списка;

– Последние используемые узлы остаются в начале списка и не удаляются при освобождении памяти.

Для идентификации узла была выбрана модель, рассмотренная в разделе 2.4.3, так как она позволяет однозначно определять местоположение узла как в пространстве, так и в октодереве. Также, в отличие от прямого указателя, позволяет идентифицировать узел вне зависимости от его местоположения в памяти.

Для отслеживания используемых узлов используем механизм подсчета ссылок на объекты, аналогичный рассмотренному применительно к узлам октодерева в работе [18], что позволит удалять из оперативной памяти неиспользуемые узлы.

Рассмотрим способ представления узлов на вторичной системе хранения, руководствуясь при этом следующими критериями:

– Обеспечение минимального количества создаваемых файлов, т.к. создание файловой записи может быть медленным, а большое количество файловых записей может превысить лимит, допустимый в используемой файловой системе;
– Обеспечение минимальных накладных расходов при хранении узлов октодерева, т.к. общее количество узлов в октодереве, построенном по большому облаку точек, может измеряться сотнями миллионов;
– Выполнение операций загрузки и сохранения узла за константное время, чтобы не допускать снижения производительности файловых операций октодерева при увеличении его размеров.

Исходя из приведенных выше критериев, был разработан способ представления, похожий на предложенный в работе [18]. Для сокращения количества создаваемых файлов в рассматриваемой работе потомки объединяются в один файл при сохранении, а иерархия октодерева представлена иерархией директорий (например, «Tree/0/3/4/2» для узла-ветви на уровне 4). Способ, используемый в данной диссертационной работе, тоже выполняет объединение узлов октодерева, однако позволяет сократить количество создаваемых файлов еще сильнее. Он основан на наблюдении, что при построении октодерева, в результате нерегулярной природы облака точек, цикл заполнения узла, его переполнения и последующего его разбиения на восемь потомков повторяется для одних и тех же точек множество раз, что приводит к необходимости удаления существующих узлов октодерева после их разбиения. В случае, если осуществлять хранение данных облака точек в ветвях октодерева, появится возможность при построении октодерева не выполнять разбиение узла при его переполнении, а продолжать добавлять точки в его подузлы. Когда узел заполнен, больше точек в него добавить нельзя, его можно перевести в состояние *завершенного* и присоединить к общему файлу. Таким образом, во вторичной системе хранения в виде отдельных файлов будет храниться только сводный файл *завершенных* узлов, а также вытесненные кешем узлы нижних уровней разбиения. По завершении формирования все оставшиеся файлы тоже перейдут в состояние *завершенных* и будут присоединены к общему файлу.

Рассмотрим возможность ускорения производимых файловых операций за счет использования для таких операций отдельного потока обработки. В работе [36] предложена реализация, предполагающая разделение операций обработки облака точек и операций подгрузки/выгрузки данных на два различных потока. При этом вытесненные из кеша узлы передаются в отдельный поток для последующего сохранения, а узлы, доступ к которым потребуются в ближайшем будущем, асинхронно подгружаются из файла. Метод, применяемый для обмена информацией между потоками не указан, поэтому в данной работе будет использоваться модель взаимодействия производитель-потребитель, реализованная при помощи асинхронной неблокирующей очереди [64], так как она позволит добиться высокой производительности при выполнении межпоточных операций. Протокол взаимодействия между потоками будет расширен операцией финализации узлов октодерева, позволяющей перевести узел в состояние *завершенного*.

Рассмотрим предложенный в данной работе процесс обработки облака точек при помощи системы кеширования. Для повышения эффективности процесса обработки в нем используется комбинация подходов с использованием внешней и оперативной памяти, сокращение количества обменов с внешней памятью за счет модификации алгоритма обработки данных и асинхронное выполнение части операций для снижения задержек взаимодействия с внешней памятью.

На базе модели процесса обработки на основе сетей Петри из раздела 1.6.3 сформирована модель вычислительного процесса обработки с использованием системы кеширования (Рисунок 2.16). В ней для наглядности опущен *TraverseLoop, InnerLoop* и *OuterLoop*. Непосредственно сам процесс обработки, как и в общем случае, сводится к последовательности операций чтения и записи (*ReadOp, WriteOp*), которые помещаются в очередь обработки (*Queue*) и исполняются в отдельном потоке (*Process*). Отдельно стоит отметить процедуры синхронизации: в случае операций записи поток обработки не дожидается завершения, однако в случае запроса на чтение данных поток приостанавливает исполнение, ожидая завершение операции.

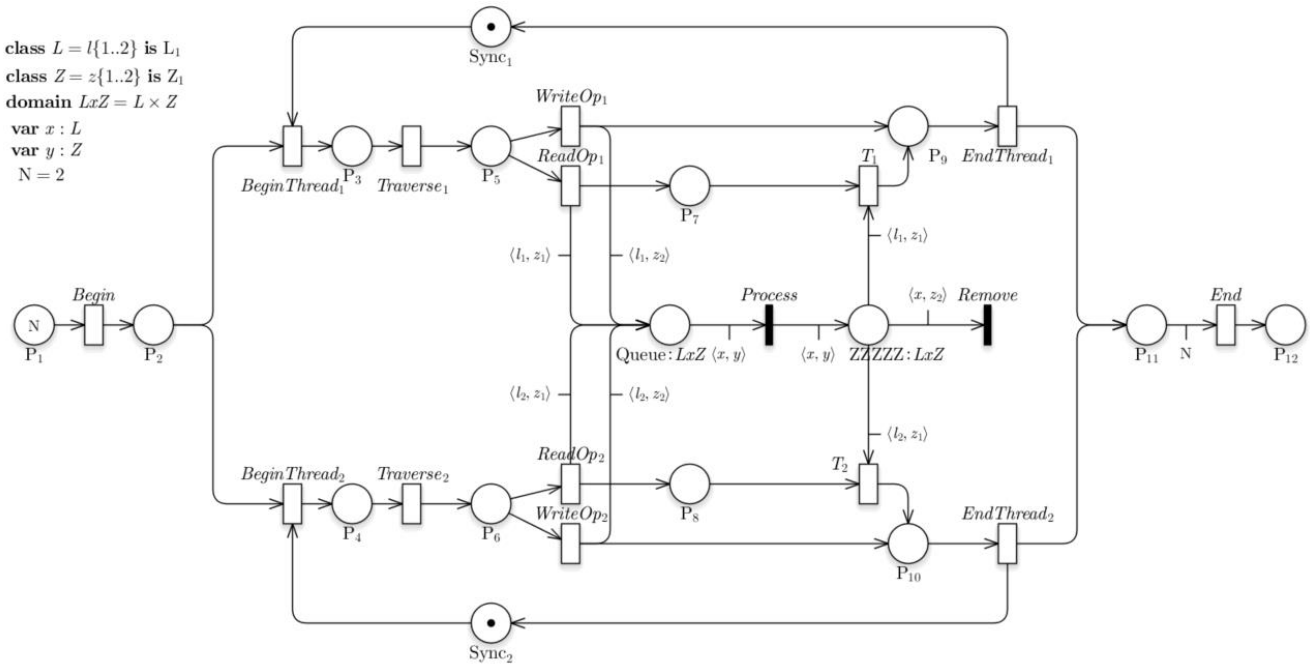

Рисунок 2.16 – Процесс обработки во внешней памяти при помощи системы кеширования

Рассмотрим более детально процедуры записи и считывания данных (Рисунок 2.17). Помимо стандартных операций взаимодействия с ФС (*GetFilePath, OpenFile, WriteToFile, ReadFile, CloseFile, DeleteFile*) присутствуют операции взаимодействия с системой кеширования (*ReadFromCache, WriteToCache, ClearCache*) и операция финализации данных. Последняя позволяет присоединить к общему файлу участок данных, модификация которого завершена, что позволяет сократить количество создаваемых файлов и в дальнейшем упростить доступ к таким данным.

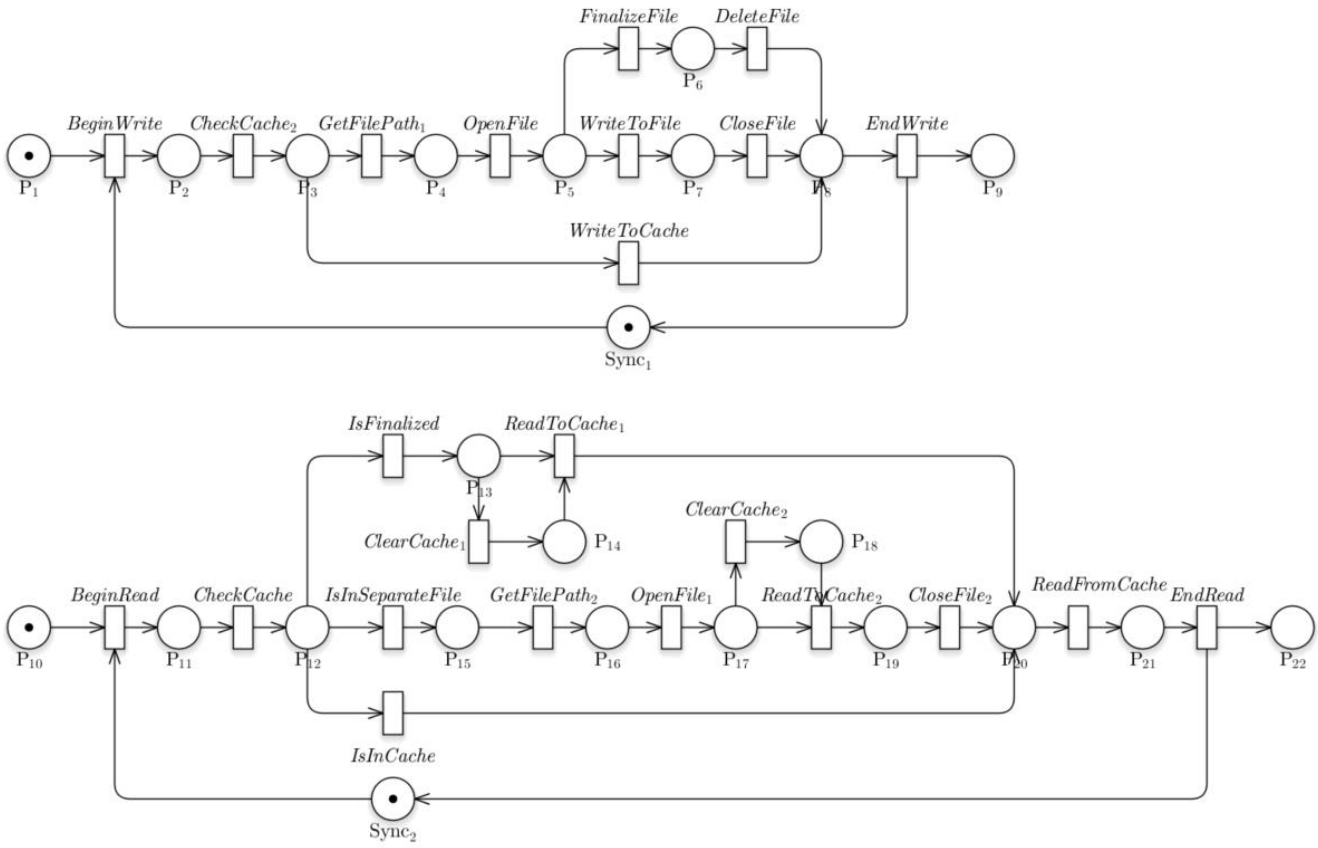

Рисунок 2.17 – Детализация операций чтения и записи при обработке во внешней памяти с помощью системы кеширования

Приведенная модель позволяет осуществить переход от высказанных ранее предположений об организации вычислительного процесса с использованием системы кеширования к алгоритму обработки и программному представлению.

2.5.2. Метод формирования октодерева с использованием механизма отображения памяти

Альтернативным подходом к управлению памятью при помощи системы кеширования, будет использование системного механизма отображения памяти, поддерживаемого ОС Linux / UNIX. Такой механизм позволяет выполнить отображение содержимого файла на диапазон виртуальных адресов в адресном пространстве приложения. Доступ к отображенной в памяти области адресного пространства будет запускать проверку в кеше страниц Linux и при необходимости считывать страницу из отображаемого файла в кеш страниц операционной системы. Таким образом, задача кеширования будет выполняться ядром операционной системы, что приведет к упрощению архитектуры приложения, а также позволит ускорить операции случайного доступа к данным.

Рассмотрим реализацию подобного механизма применительно к данным октодерева. Используя преимущества механизма отображения памяти, выделим следующие требования для разрабатываемой модели работы в условиях ограничений по оперативной памяти:

- Обеспечение минимального количества создаваемых файлов, т.к. создание файловой записи может быть медленным, а большое количество файловых записей может превысить лимит, допустимый в используемой файловой системе;
- Обеспечение прямого доступа к данным облака точек – использование прямого указателя для считывания/изменения данных позволит значительно упростить взаимодействие с данными;
- Возможность создания блоков данных произвольного размера – позволит выполнять хранение данных узлов октодерева в процессе построения октодерева, когда конечный размер блока неизвестен;
- Возможность редактирования данных октодерева после его построения, включая механизм изменения размеров узлов – позволит расширить спектр задач, в которых может быть применено октодерево.

Рассмотрим реализацию подобного механизма в существующих работах. В работе [41] рассматривается реализация системы интерактивного рендеринга облака точек с построением структуры представления уровней детализации облака точек в файле в виде линейного списка блоков и последующего отображения их в оперативную память. Но данная работа не рассматривает процесс построения октодерева в отображаемой памяти и, соответственно, не учитывает проблемы, возникающие при изменении размера данных узла при его заполнении.

В работе [65] рассматривается применение системы отображения памяти для получения доступа к участкам карты высот для визуализации ландшафта, однако она вновь не содержит решений задачи хранения узлов произвольного размера. В работе [66] рассматривается применение механизма отображения памяти для выполнения интерактивной визуализации больших полигональных моделей. Этап построения структуры разбиения пространства, более подробно описанный в работе [67], предполагает создание иерархии файлов и директорий.

Попробуем рассмотреть более общий подход к решаемой задаче. Пусть механизм отображения памяти позволяет выделить диапазон виртуальных адресов, который будет ссылаться на файл на вторичной системе хранения. Используем механизмы динамического выделения памяти (аллокации), которые используются для выделения памяти в куче, только вместо оперативной памяти используем выделенный диапазон виртуальных адресов.

Такое решение позволит выделять в отображаемой памяти участки данных произвольного размера, однако потребует решения проблемы фрагментации памяти, которая возникнет в результате неоднородного выделения и освобождения участков в процессе построения октодерева. Данная задача хорошо исследована. Например, в работе [68] приводится обзор существующих решений задачи динамического выделения памяти, а в работе [69] рассмотрена реализация системы динамического выделения памяти с использованием механизма отображения памяти. Таким образом, использование системы динамического выделения данных на диапазоне виртуальных адресов удовлетворяет поставленным требованиям и позволяет решить задачу хранения узлов октодерева в условиях ограничений на потребление оперативной памяти. Для представления иерархической модели используем модель октодерева на целочисленной арифметике. Это позволит добиться ускорения вычислений за счет меньшего количества операций с плавающей точкой.

На базе модели процесса обработки на основе сетей Петри из раздела 1.6.3 сформирована модель вычислительного процесса обработки с использованием механизма отображения памяти (Рисунок 2.18). Для повышения эффективности процесса обработки в нем используется комбинация подходов с использованием внешней и оперативной памяти, сокращение числа операций с ФС за счет введения более низкоуровневых процедур доступа к данным и сокращение количества создаваемых файлов для снижения нагрузки на ФС.

Одной из его отличительных особенностей является возможность практически бесшовной интеграции в любой из процессов обработки в оперативной памяти: так же, как и при работе с оперативной памятью, взаимодействие с данными сводится к операциям выделения и освобождения блоков данных и прямого доступа по указателю на данные. Из-за этого общая модель процесса обработки идентична процессу обработки в оперативной памяти.

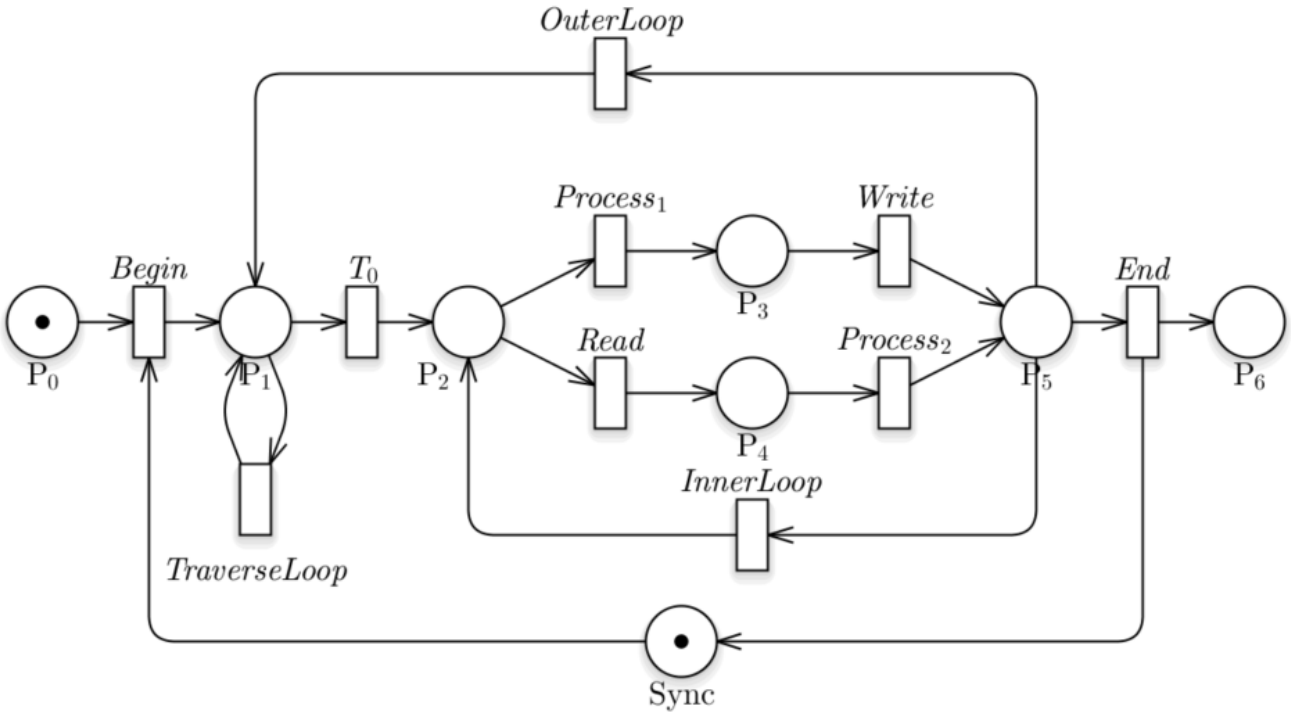

Рисунок 2.18 - Процесс обработки во внешней памяти при помощи механизма отображения памяти

Рассмотрим более детально процедуры записи и считывания данных (Рисунок 2.19). Отказ от хранения блоков данных в индивидуальных файлах повлек за собой потерю возможности наращивания размеров блоков средствами файловой системы. Теперь все блоки данных хранятся в одном файле, и при необходимости изменения его размеров выполняется операция реаллокации, что влечет за собой фрагментацию данных. Однако, за счет более узкой специализации и отказа от части операций ФС, такой подход должен показывать более высокую производительность. Как и в случае с использованием оперативной памяти, со стороны операций записи можно выделить операции выделения блока в ВП (*Allocate*), удаления блока (*Delete*), изменения размеров блока (*Reallocate*) и записи данных в блок (*Update* и *Append*). Для управления размещением данных вводятся операции поиска свободного блока данных заданного размера (*FindFreeBlock*), выделения блока (*UseFreeBlock*), освобождения блока (*FreeBlock*), проведения дефрагментации блока и его соседей (*DefragmentBlock*).

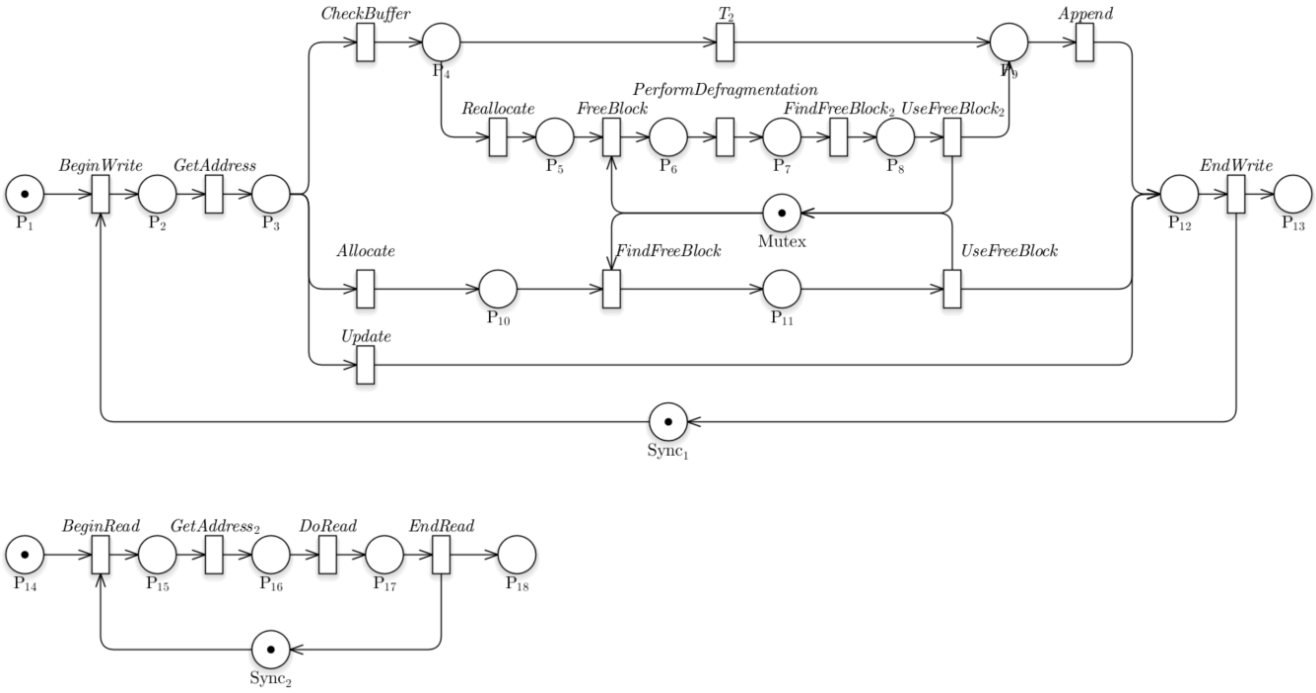

Рисунок 2.19 – Детализация операций чтения и записи при обработке во внешней памяти с помощью механизма отображения памяти

Приведенная модель позволяет осуществить переход от высказанных ранее предположений об организации вычислительного процесса с использованием механизма отображения к алгоритму обработки и программному представлению.

2.5.3. Обзор предложенных методов

Формирование октодерева в оперативной памяти. Объем используемой оперативной памяти в таком октодереве не снижается за счет использования внешней памяти и может превысить лимит потребления. Не удовлетворенное требование к ограничению объема потребления оперативной памяти делает данный метод непригодным для решения поставленной задачи.

$$M^{RAM}(P, x_3^{ram}, t_i) > M^{limit}, \quad (14)$$

$$K_{time}(P, x_3^{ram}) = 0,$$

$$K_{files}(P, x_3^{ram}, t_i) = 0,$$

где x_3^{ram} - вариант реализации с обработкой в оперативной памяти;

P – произвольное облако точек, чей размер превышает доступные объемы оперативной памяти;

K_{files} – количество созданных файлов в момент времени t_i.

Использование системы кеширования данных. Данный метод выполняет управление размещением блоков данных при помощи двухуровневой системы кеширования, в которой первым уровнем является размещение блока в оперативной памяти, а вторым уровнем – размещение блока на жестком диске. При вытеснении блоков используется политика Least Recently Used (LRU). Потребляемая память лимитируется заданным значением, однако в процессе создается большое количество файлов, в худшем случае соразмерное количеству узлов в октодереве. Затраты времени на выполнение сервисных операций растут с количеством создаваемых файлов.

$$M^{RAM}\left(P, x_1^{cache}, t_i\right) \le M^{limit}, \tag{15}$$

$$K_{time}(P, x_1^{cache}) = T^{read} + T^{write} + T^{service},$$

$$K_{files}\left(P, x_1^{cache}, t_i\right) = \begin{cases} k, 0 \le i < n, \\ 1, i = n \end{cases},$$

где P – произвольное облако точек, чей размер превышает доступные объемы оперативной памяти;

x_1^{cache} – метод обработки, принадлежащий вариантам обработки на основе системы кеширования X^{cache};

$T^{service} = T^{create} + T^{open} + T^{append}$, T^{create} – затраты времени на создание файла, T^{open} – затраты времени на операции получения файлового дескриптора, T^{append} – затраты времени на наращивание размеров файла;

k – количество узлов октодерева;

n – количество операций с октодеревом.

Использование механизма отображения памяти. Данный механизм позволяет выполнять отображение содержимого файла на диапазон виртуальных адресов процесса, что дает возможность напрямую работать с сохраненными блоками точек как с данными в оперативной памяти, а также сократить количество создаваемых файлов до одного.

$$M^{RAM}(P, x_2^{map}, t_i) \leq M^{limit}, \quad (16)$$

$$K_{time} = T^{cache} + T^{write} + T^{service},$$

$$K_{files}(P, x_2^{map}, t_i) = 1,$$

где x_2^{map} – метод обработки, принадлежащий вариантам обработки на основе системы отображения памяти X^{map};

$T^{service} = T^{allocate} + T^{free}, T^{allocate}$ – затраты времени на аллокацию блока данных на отображаемой памяти, T^{free} – затраты времени на освобождение блока данных в отображаемой памяти.

2.6. Выводы

В данной главе был проведен анализ организации октодерева и механизмов взаимодействия оперативной и внешней памяти, предложены структуры данных и алгоритмы, предназначенные для построения октодерева, позволяющего выполнять обработку больших облаков точек с использованием вторичных систем хранения. Произведен анализ и уточнение альтернатив организации вычислительного процесса обработки, предложенных в первой главе. Рассмотрены модели октодерева на основе арифметики с плавающей точкой и целочисленной арифметики. Предложена иерархическая модель с использованием целочисленной арифметики. Рассмотрены задачи организации иерархической модели октодерева, обеспечения работы с использованием внешних систем хранения, снижения нагрузки на файловую систему путем сокращения количества создаваемых файлов. Выявлены компоненты системы обработки больших облаков точек, определены связей между ними, их функции и цели, и механизмы управления ресурсами системы.

1. Проведенная оценка сложности доступа к данным облака точек в исходном виде и после построения октодерева показала, что структурирование данных по пространственному признаку, в частности, при помощи октодерева, позволяет сократить затраты времени на операции пространственного поиска и выборки данных при помощи группировки близко расположенных в пространстве точек.
2. Исследование способов сокращения размеров узлов в ОП, их идентификации, а также повышения эффективности операций обхода и поиска в октодереве позволило построить иерархическую модель октодерева с использованием арифметики с плавающей точкой, предложить способ идентификации узла при помощи целочисленного идентификатора и разработать иерархическую модель октодерева с использованием целочисленной арифметики.
3. Анализ организации данных в октодереве и механизмов взаимодействия оперативной и внешней памяти, позволил произвести уточнение альтернатив организации вычислительного процесса обработки, предложенных в первой главе. Предложен способ организации механизма асинхронного двухуровневого кеширования для октодерева. Предложен способ организации октодерева с использованием механизма отображения памяти и алгоритмов динамической аллокации.

В данной главе были рассмотрены следующие положения диссертации:

1. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе асинхронной двухуровневой системы кеширования, использующий внешнюю память при превышении обрабатываемой информацией объема оперативной памяти и позволяющий сократить количество создаваемых файлов для снижения влияния обменов с внешней памятью на производительность.

2. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе механизма отображения памяти, использующий внешнюю память для хранения информации с возможностью прямого доступа и интерактивной модификации, а также позволяющий ускорить выполнение операций доступа к данным октодерева за счет кодирования с использованием целочисленной арифметики и сократить количество создаваемых файлов до одного для снижения влияния обменов с внешней памятью на производительность.

В данной главе были решены следующие задачи диссертации:

1. Разработка новых методов организации вычислительного процесса обработки облака точек, основанных на выдвинутой гипотезе. Разработка компонентов, алгоритмов и структур данных таких систем, исследование взаимодействий между ними, системным ПО, оперативной и внешней памятью.

Глава 3. Разработка алгоритмов и структур данных для построения октодерева в условиях ограничений по оперативной памяти

В главе рассматривается реализация вычислительного процесса обработки на основе системы кеширования и механизма отображения памяти. Предложен алгоритм построения октодерева с использованием системы кеширования и возможностью объединения заполненных узлов в общий файл. Для обеспечения возможности использования механизма отображения памяти в октодереве предложена реализация алгоритма динамической аллокации на отображаемой памяти. Для обеспечения возможности обработки больших облаков точек сторонними библиотеками предложен способ обработки больших облаков точек путем внедрения системы аллокации отображаемой памяти в сторонние библиотеки для linux систем.

В данной главе выполняются следующие задачи диссертации:

1. Разработка новых методов организации вычислительного процесса обработки облака точек, основанных на выдвинутой гипотезе. Разработка компонентов, алгоритмов и структур данных таких систем, исследование взаимодействий между ними, системным ПО, оперативной и внешней памятью.

Результаты, изложенные в данной главе были опубликованы соискателем в статьях [29; 70; 71].

3.1. Загрузка облака точек

Как было отмечено в разделе 2.1.2, этап построения октодерева тесно связан с процессом загрузки облака точек – в момент загрузки происходит считывание данных облака точек, их конвертация во внутренний формат представления и заполнение октодерева, которое в дальнейшем будет использоваться для доступа к данным облака точек.

В качестве входного формата облака точек был выбран формат LAS, как наиболее широко используемый на текущий момент. Учитывая большой размер загружаемых облаков точек, рассмотрим два подхода при загрузке облаков точек: разбиение большого облака на секции меньшего размера; загрузка облака по частям. Для построения октодерева предпочтительнее второй подход. Он поддерживается библиотеками LASTools [72] и PDAL [73], широко использующимися для загрузки облаков точек формата LAS. Однако в данных библиотеках загрузка и предобработка участков облака точек производится последовательно, что вносит задержки файловой системы в процесс обработки загруженного участка облака точек. Под предобработкой подразумевается конвертация данных облака точек во внутренние структуры данных, включая получение координат точек и прочих их атрибутов, а также конвертация во внутреннюю систему координат.

В опубликованной соискателем (в соавторстве) статье [70] был рассмотрен используемый в данной работе алгоритм, позволяющий выполнить разделение задач загрузки участка облака точек из файловой системы и его предобработки на несколько потоков.

Алгоритм, применяемый для загрузки и предобработки участков облака точек, основан на использовании многопоточной обработки с асинхронной неблокирующей очередью для коммуникации между потоками. На вход он принимает путь к файлу облака точек в формате LAS, а на выходе по запросу выдает считанные и предобработанные участки облака точек. Алгоритм загрузки и предобработки участков облака точек применительно к построению октодерева приведен листинге 3.1.

Листинг 3.1 – Алгоритм загрузки участков облака точек

```
Reader r(filepath); // LAS Point Cloud reader
Octree octree; // Point Cloud Octree

// Queue for freed buffers (Lock-free)
Queue<Buffer> freeQ(queueLength);
// Queue for loaded buffers (Lock-free)
Queue<Buffer> loadedQ(queueLength);
```

```
// Loading thread
auto t1 = std::thread([&]
{
  while(/* not all points readed from file */) {
    // Pop free buffer from queue
    Buffer b = freeQ.pop();
    // Read file chunk to buffer
    r.readChunk(b, b.size());
    // Push points to queue
    loadedQ.push(b);
  }
});

// Octree construction thread
auto t2 = std::thread([&]
{
  while(/* not received stop signal */) {
    if(Buffer b = loadedQ.pop()) {
      // Insert loaded points to octree
      octree.insertPoints(b);
      freeQ.push(b);
    }
  }
});

// Wait for threads ends
t1.join();
t2.join();
```

3.2. Разработка алгоритма и структуры октодерева с применением асинхронной системы двухуровневого кеширования

В данном разделе приводится описание вычислительного процесса и структуры данных октодерева при обработке во внешней памяти с применением механизма отображения памяти, являющегося решением задачи диссертации «Разработка структуры данных и алгоритмов формирования октодерева для обработки больших облаков точек, а также системы управления процессами загрузки/выгрузки данных на внешнюю память при обработке облака точек прикладными программами стандартной библиотеки».

3.2.1. Использование целочисленного идентификатора для адресации узлов октодерева

Для адресации узлов октодерева в системе кеширования применим целочисленные идентификаторы, рассмотренные в разделе 2.4.3. Данный идентификатор представлен 64 битным беззнаковым целым (uint64_t), содержащим по три бита на уровень разбиения и один бит для обозначения последнего адресуемого уровня. Таким образом, данный идентификатор может адресовать 2^{63} узла или 21 уровень разбиения.

3.2.2. Хранение иерархии октодерева с использованием целочисленного идентификатора

Классически узлы октодерева хранятся в виде иерархии, в которой родительский узел выполняет хранение восьми узлов-потомков. Для хранения узлов октодерева используем хэш-таблицу из стандартной библиотеки C++, далее называемой STL (Standard Template Library) [74; 75]. Хэш-таблица представлена контейнером под названием unordered_map [76], который является реализацией ассоциативного массива и позволяет хранить пары ключ:значение. Ключом в нашем случае будет идентификатор узла, а значением – структура данных узла.

3.2.3. Организация системы кеширования и политики вытеснения

Для хранения списка узлов, загруженных в оперативную память, используется двусвязный список. Для снижения количества аллокаций реализуем собственный двусвязный список, позволяющий использовать повторно старые элементы вместо их повторной аллокации. Элемент такого списка (Листинг 3.2) является стандартным при реализации двусвязного списка и хранит указатель на следующий и предыдущий узлы. Отличием может быть хранение отдельно идентификатора узла (KeyT) и указателя на порожденный в оперативной памяти узел (ValueT), что обусловлено удобством при реализации некоторых вспомогательных функций.

Листинг 3.2 – Элемент двусвязного списка

```
struct Node
{
   Node* prev; // Указатель на предыдущий узел
   Node* next; // Указатель на следующий узел
   KeyT key; // Идентификатор узла
   ValueT value; // Указатель на узел в оперативной памяти
};
```

Структура самого двусвязного списка отличается от общепринятой реализации тем, что в ней также хранится указатель на функцию, вызываемую при удалении узла (m_destroyCb) и указатель на пул удаленных элементов (m_pool) (Листинг 3.3).

Листинг 3.3 – Структура двусвязного списка

```
struct LinkedList
{
   // ...
   // Указатель на функцию, вызываемую при удалении узла
   NodeDestroyCb m_destroyCb;
   // Указатель на первый элемент списка
   Node* m_front;
   // Указатель на последний элемент списка
   Node* m_back;
   // Указатель на пул удаленных узлов
   Node* m_pool;
   // Количество элементов в списке
   size_t m_size;
}
```

Хранение пула удаленных элементов позволяет вместо создания нового элемента в куче получать его из списка удаленных элементов (при наличии). Соответственно, при удалении элемента, его необходимо поместить в список удаленных (Листинг 3.4).

Листинг 3.4 – Операции добавления и удаления из списка

```
// Добавление элемента в список
inline Node* push(const KeyT& key, const ValueT& val)
{
   Node* node;
   if(m_pool)
   {
```

```
        node = pool_pop();
        node->key = key;
        node->value = val;
    }
    else
        node = new Node(key, val);
    .....
}

// Удаление элемента из списка
inline void pop()
{
    if(m_front == m_back) {
        pool_push(m_back);
        m_front = m_back = NULL;
    }
    else {
        pool_push(m_back);
        m_back = m_back->prev;
        m_back->next = NULL;
    }
    .....
}
```

Стоит отметить, что реализация двусвязного списка с использованием преаллоцированных блоков памяти для хранения элементов списка может оказаться еще более эффективной.

Рассмотрим реализацию механизма подсчета занимаемой оперативной памяти. Пусть каждый узел октодерева выполняет хранение массива данных произвольного размера, представляющего точки из облака точек. Тогда размер узла в оперативной памяти можно вычислить, сложив размер самого узла и размер массива точек. Следует так же учитывать накладные расходы, возникающие при хранении массива данных при помощи стандартных контейнеров STL. Учитывая все вышеприведенные факторы, можно обеспечить подсчет занимаемой данными облака точек оперативной памяти. Существует возможность также включить в расчеты любые другие данные, например, занимаемые иерархией октодерева, однако это влечет за собой необходимость введения соответствующих механизмов подсчета.

Чтобы отслеживать потребляемую память в реальном времени, достаточно прибавлять или вычитать из счетчика потребляемой памяти значение дельты в момент загрузки, создания или удаления узла. Для поддержания заданного уровня потребления оперативной памяти M необходимо в момент выделения новой памяти размера N (при создании узла или увеличении его данных) освободить узлы из конца двусвязного списка до обеспечения значения потребляемой памяти в $(M - N)$.

Рассмотрим реализацию политики LRU, выполняемую при помощи предложенного двусвязного списка. Для обеспечения ее работы достаточно отслеживать операции создания и загрузки узла, а также изменения его данных. При возникновении любой из таких ситуаций данный узел помещается в конец двусвязного списка. Учитывая, что освобождение элементов происходит с начала списка, часто используемые узлы будут оставаться в оперативной памяти дольше всего.

3.2.4. Асинхронная работа со вторичной системой хранения

Рассмотрим реализацию механизма асинхронного выполнения файловых операций, приведенного в разделе 2.5.1 и позволяющего частично скрыть задержки файловой системы.

Пусть для работы с файлами используется отдельный поток, принимающий управляющие команды при помощи асинхронной неблокирующей очереди сообщений. Различные варианты подобной очереди широко используются в многопоточном программировании. В данной работе используется реализация неблокирующей очереди с одним потребителем и одним производителем, предложенная в [77]. Выбор данной реализации обусловлен наличием кольцевого буфера, позволяющего лимитировать количество команд в очереди, а также хорошей общей производительностью.

Управляющая команда содержит тип выполняемой операции, указатель на данные облака точек и индекс узла, для которого производится заявленная операция. Сформируем список команд, необходимый для взаимодействия со вторичной системой хранения:

- Загрузка данных узла октодерева в кеш – выполняет считывание данных из отдельного или общего файла;
- Добавление точек к узлу октодерева – если блок не финализирован, но выгружен на вторичную систему хранения, то подобная команда позволит добавить новые точки к блоку без необходимости полной его загрузки в оперативную память;
- Обновление данных узла октодерева – при изменении данных узла в оперативной памяти может потребоваться их обновление на вторичной системе хранения. Команда также может выполняться для финализированных блоков при условии, что количество точек в узле не было изменено;
- Финализация блока данных – при заполнении блока в процессе построения октодерева данная команда производит его присоединение к общему файлу;
- Очистка блока точек – позволяет удалить файл, принадлежащий нефинализированному блоку точек.

Также рассмотрим общие управляющие команды:

- Выполнение синхронизации – применяется для ожидания завершения выполнения команд в очереди;
- Остановка потока – сообщает потоку о необходимости завершения цикла обработки сообщений.

Синхронизация выполняется при необходимости ожидания завершения определенной операции или ряда операций; реализована при помощи подачи в очередь команд примитива синхронизации, основанного на использовании условной переменной (conditional variable) [78], алгоритм работы с которой приведен в листинге 3.5. Для выполнения синхронизации отправитель помещает данный примитив в очередь команд, после чего вызывает метод wait, что блокирует его поток исполнения до момента вызова метода notify при исполнении данной команды получателем. Такой подход гарантирует, что все команды, предшествующие команде синхронизации, будут выполнены по завершении метода wait.

Листинг 3.5 – Примитив синхронизации на основе условной переменной

```
struct Sync // Примитив синхронизации с использованием условной переменной
{
   // Блокирует поток исполнения до вызова метода notify
   inline void wait() {
      std::unique_lock<std::mutex> locker(m_lock);
      while(!m_notified)
         m_cv.wait(locker);
   }
   // Снимает блокировку
   inline void notify() {
      std::unique_lock<std::mutex> locker(m_lock);
      m_notified = true;
      m_cv.notify_one();
   }
private:
   std::mutex  m_lock;
   std::condition_variable m_cv;
   bool m_notified = false;
};
```

Цикл обработки команд для потока работы с файлами приведен в листинге 3.6. Подобная организация позволяет производить выборку команд из очереди сообщений, а при отсутствии команд в очереди передавать управление планировщику потоков для экономии ресурсов процессора.

Листинг 3.6 – Цикл обработки команд

```
bool resumeFlag = true;
while(resumeFlag)
{
   while(CacheOP* op = m_asyncQueue.front())
   {
      m_asyncQueue.pop();
      switch(op->type)
      {
      case kFinish:
         resumeFlag = false;
         break;
      case kSync:
         // Синхронизация
         break;
      case kAppend:
         // Добавление новых точек к узлу
         break;
      case kLoad:
         // Загрузка узла
         break;
      case kDiscardBlock:
         // Очистка узла
         break;
      case kFinalizeBlock:
         // Финализация узла
         break;
      case kUpdateBlock:
         // Обновление данных узла
         break;
      default:
         assert(false);
      }
   }
   // Передача управления планировщику потоков,
   // если нет входящих команд
   std::this_thread::yield();
}
```

3.2.5. Алгоритм построения октодерева

Построение октодерева осуществляется путем последовательной вставки блоков точек, получаемых в результате процесса загрузки, описанного в разделе 3.1. Было реализовано два варианта алгоритма построения октодерева – рекурсивный и параллельный. Рекурсивный алгоритм (Листинг 3.7) выполняет для каждого узла октодерева процедуру добавления точек, при заполнении узла производит его финализацию, а затем повторяет ту же процедуру уже для его потомков. Заполненные узлы при этом игнорируются. Переполнения стека при этом не происходит ввиду ограничения на максимальное количество уровней в октодереве (21 уровень).

Листинг 3.7 – Алгоритм рекурсивного заполнения облака точек

```
void insertPoints(const Point* points, const Index* indices, size_t size, const OctreeIndex nodeId)
{
   // Обращение к кешу для выяснения текущего количества точек в узле.
   // Не требует выполнения файловых или асинхронных операций
   const size_t blockSize = cache()->pointCount(nodeId);
   // Определение количества точек, которые будут записаны в узел
   size_t sizeToWrite = 0;
   if(blockSize < MaximumBlockSize)
   {
      sizeToWrite = std::min(MaximumBlockSize - blockSize, size);
      size -= sizeToWrite;
   }
   if(size)
   {
      std::vector<Index> actualIndices[8];
      const Index* firstPointIndex = indices + sizeToWrite;
      const Index* const lastPointIndex = firstPointIndex + size;
      // Распределение индексов точек по потомкам узла
      while(firstPointIndex < lastPointIndex)
      {
         uint8_t childIndex = ComputeChildIndex(points[*firstPointIndex]);
         actualIndices[childIndex].push_back(*firstPointIndex);
         ++firstPointIndex;
      }
      for(int i = 0; i < 8; i++)
      {
         if(actualIndices[i].empty())
            continue;
         // Рекурсивный вызов функции для вставки точек в узел потомка
         insertPoints(points,
            actualIndices[i].data(),
```

```
            actualIndices[i].size(),
            nodeId.childIndex(i));
      }
   }
   if(sizeToWrite)
   {
      // Добавление новых точек к узлу
      cache()->append(extractPointBlock(indices, sizeToWrite), index.code);
      // Если узел заполнен - выполнить финализацию
      if(blockSize + sizeToWrite == MaximumBlockSize)
      {
         cache()->finalize(index.code);
         cache()->unload(index.code);
      }
   }
}
```

Параллельная реализация алгоритма выполняется так же, как рекурсивная, однако вместо рекурсивного вызова происходит добавление запроса в очередь выполнения, которая обрабатывается асинхронно при помощи пула потоков [79]. Такая реализация позволяет снизить негативный эффект, возникающий при частом создании новых потоков, т.к. порождается только по одному потоку на ядро. Параллельность достигается за счет того, что каждый поток обрабатывает свою ветвь октодерева, и потоки не пересекаются по используемым ими данным. Но при подобном подходе сложно устранить задержки, возникающие при пересечении регионов памяти, попадающих в кеш разных ядер процессора [80]. В таком случае принято добавлять пустую зону памяти, равную размеру линии кеширования ядра, между участками памяти, обрабатываемыми разними ядрами, что проблематично сделать для узлов октодерева, и особенно для блока загруженных точек, лежащего в массиве данных. Тем не менее, реализация параллельного алгоритма обработки облака точек все же позволяет ускорить процесс построения облака точек.

3.2.6. Общая схема программной реализации октодерева на базе системы кеширования

Рассмотрим общую схему программной реализации октодерева на базе системы кеширования (Рисунок 3.1). На рисунке изображен центральный компонент (Octree Interface), который предоставляет основные методы для выборки, обхода и построения октодерева, алгоритм которого приведен в разделе 3.2.5. Для хранения узлов октодерева используется ассоциативный массив (Hash Table, описан в разделе 3.2.2), применение которого описано в разделе 3.2.2. В качестве ключа в нем используется целочисленный идентификатор узла (описан в разделе 2.4.3). Для хранения данных облака точек, находящихся на текущий момент в оперативной памяти, а также для обеспечения политики вытеснения при превышении установленного лимита потребления, используется двусвязный список (LRU Cache), описанный в разделе 3.2.3. Для взаимодействия с файловой системой применяется асинхронный подход, при котором все операции выполняются в отдельном потоке (File Thread), взаимодействие с которым происходит при помощи неблокирующей очереди сообщений (Async Queue), что описано в разделе 3.2.4

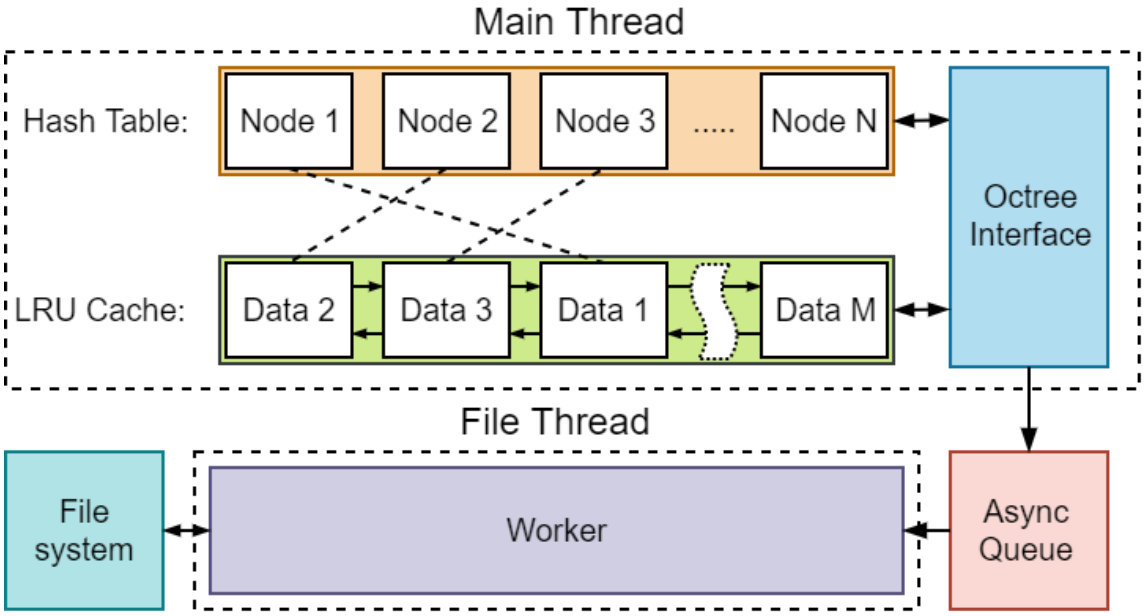

Рисунок 3.1 – Общая схема программной реализации октодерева на базе системы кеширования

3.3. Разработка алгоритма и структуры октодерева с применением механизма отображения памяти

В данном разделе приводится описание вычислительного процесса и структуры данных октодерева при обработке во внешней памяти с применением механизма отображения памяти, являющегося решением задачи диссертации «Разработка структуры данных и алгоритмов формирования октодерева для обработки больших облаков точек, а также системы управления процессами загрузки/выгрузки данных на внешнюю память при обработке облака точек прикладными программами стандартной библиотеки».

3.3.1. Разработка структуры узла октодерева на базе целочисленной арифметики

Структура узла в данном октодереве существенно отличается от предложенной в октодереве на базе системы кеширования. С использованием механизма отображения памяти нам не требуется введения громоздких механизмов кеширования, что существенно упрощает архитектуру октодерева. Мы также не будем использовать ассоциативный массив для хранения узлов октодерева, а будем придерживаться классической схемы [81], в которой родительский узел выполняет хранение восьми потомков. Это обусловлено отличиями схемы работы в условиях ограничений по оперативной памяти, в которой нам не требуется отдельной идентификации узлов для хранения в системе кеширования. Стоит также отметить, что в случае применения классической схемы для хранения узлов октодерева временные задержки для операций обхода станут более предсказуемыми, а также пропадет задержка, возникающая при перестройке таблицы хеширования при ее заполнении.

Примем, что иерархия октодерева хранится в оперативной памяти для повышения скорости доступа. Каждый узел октодерева выполняет хранение массива данных, который располагается в отображаемой памяти, механизмы работы с которым будут представлены в последующих разделах. Кроме хранения массива данных узлом выполняется хранение указателя на массив потомков, а также битовые маски, указывающие на то, какие потомки не пусты, что было рассмотрено в разделе 2.4.1.

Каждый узел отвечает за представление диапазона целочисленных значений, который сужается по мере уменьшения его уровня. На нулевом уровне ширина диапазона будет равна единице. Имея целочисленную координату, можно получить узел, ее содержащий, на любом уровне разбиения. Как было описано в разделе 2.4.4, иерархия октодерева организована так, что определенные биты целочисленной координаты в пространстве позволяют адресовать потомков узла на определенном уровне разбиения. Имея трехмерную координату, представленную тремя целыми числами со знаком (например, int32_t), младшие биты каждого из чисел будут определять положение узла на нулевом уровне разбиения, следующие по порядку – на первом и так далее до максимального уровня разбиения. Подобное решение позволяет значительно сократить временные затраты на обход октодерева.

3.3.2. Хранение иерархии октодерева

Рассмотрим механизм представления иерархии целочисленного октодерева. При использовании целочисленной координаты для адресации рассматриваемое дерево может содержать по одному уровню разбиения на каждый бит координаты (в качестве фактора разбиения - единица). Однако учитывая, что чем больше уровней разбиения, тем медленнее скорость обхода, а также то, что октодеревья, построенные по облакам точек, в основном содержат от восьми до четырнадцати уровней, мы бы хотели использовать ограниченное количество младших бит координаты для адресации в октодереве. Остальные биты могут быть использованы в качестве ключа в ассоциативном массиве, как было рассмотрено в разделе 2.4.5. Подобное решение снизит глубину дерева и позволит выполнять адресацию больших участков пространства без существенного замедления работы октодерева, так как количество запросов к ассоциативному массиву будет гораздо меньшим (в отличие от октодерева на базе системы кеширования).

Пусть младшие биты координаты отвечают за местоположение узла в целочисленном октодереве. Тогда обнулив младшие биты в координате, ее можно будет использовать в качестве ключа в ассоциативном массиве.

3.3.3. Алгоритм динамической аллокации на отображаемой памяти

Механизм отображения памяти позволяет выполнять отображение содержимого файла на диапазон виртуальных адресов процесса. При этом в 64-х разрядной системе на адрес выделяется 48 бит, что позволяет адресовать таким образом до 256 терабайт данных. На практике выполнение отображения содержимого файла на диапазон адресов в адресном пространстве процесса осуществляется при помощи вызова метода *mmap* [82] с указанием файлового дескриптора и ширины необходимого диапазона адресов (Листинг 3.8). Размер файла при этом не обязан соответствовать данному диапазону, что позволяет наращивать размер файла по мере необходимости. Стоит отметить, что попытка доступа по адресу, выходящему за границы файла, приведет к ошибке времени исполнения. Метод *mmap* возвращает указатель на адрес начала данных в отображенной памяти, доступ к которым приведет к считыванию соответствующей информации из файла.

Листинг 3.8 – Пример выполнения отображения памяти в ОС Linux

```
// Создание файла и получение файлового дескриптора
int fd = ::open(filePath,
    O_RDWR | O_CREAT,
    static_cast<mode_t>(0600));

void* data =
    mmap(
        // Начальный адрес
        nullptr,
        // Размер адресного пространства
        m_capacity,
        // Открыть на чтение и запись
        PROT_READ | PROT_WRITE,
        // Общий доступ,
        // Не резервировать данные в RAM и файле подкачки
        MAP_SHARED | MAP_NORESERVE,
        // Дескриптор файла
        fd,
        // Смещение
        0);
```

Рассмотрим применение механизма отображения памяти для хранения данных узлов октодерева. Пусть каждый узел выполняет хранение массива точек. Учитывая, что количество точек в узле может быть любым, вплоть до заданного максимума, для хранения точек используем реализацию динамического массива (std::vector [83]) из стандартной библиотеки C++. Такой выбор позволит хранить точки в виде последовательного массива в памяти, что положительно скажется на скорости их обработки и общем количестве операций выделения данных, а также позволит выполнять операции изменения размеров массива с выделением новой памяти при превышении точками текущей емкости массива. Недостатки данного подхода происходят из его достоинств: учитывая линейную организацию памяти, операции удаления элемента не из конца массива имеют сложность *O(n)*, а операции вставки в конец массива при превышении его емкости потребуют реаллокации всех его данных.

Для обеспечения работы с отображаемой памятью выполним подмену аллокатора в используемом динамическом массиве. Для реализации такой возможности выполним наследование от класса «std::allocator» [84], и переопределим методы, вызываемые при выделении и освобождении памяти (Листинг 3.9). Таким образом мы сможем обойти стандартный механизм выделения памяти и самостоятельно управлять выделением памяти для данного динамического массива, что позволит нам использовать отображаемую память для хранения данных массива.

Листинг 3.9 – Подмена аллокатора для динамического массива

```
template<typename T, size_t kAlignment = alignof(T)>
class MMapAllocator : public std::allocator<T>
{
   // ....
   // Выделение памяти
   T* allocate(size_type n, const void *hint = nullptr) {
      // Метод выделения памяти на отображенном диапазоне адресов
      return mmap_alloc(n * sizeof(T), kAlignment);
   }
   // Освобождение памяти
   void deallocate(pointer p, size_type n) {
      // Метод освобождения памяти на отображенном диапазоне адресов
```

```
        mmap_free(p);
    }
};
```

Так как стандартный системный аллокатор не поддерживает выделение памяти в отображаемом адресном пространстве, нам придется собственноручно реализовать подобную возможность. Рассмотрим требования к системе выделения памяти, необходимые для обеспечения работы совместно с динамическими массивами. Как минимум, требуется обеспечить возможность как выделения, так и освобождения участков памяти. Размер выделяемых участков не является константным, что добавляет необходимость обеспечения возможности выделения участков памяти произвольного размера. Таким образом, мы не сможем воспользоваться моделями линейного аллокатора [85], стекового аллокатора [86] и аллокатора на пуле [87], что приводит нас к необходимости решения проблемы фрагментации памяти. К счастью, подобная проблема хорошо исследована. Разрабатываемая система выделения памяти имеет довольно узкую область применения, это делает ее структурно более простой, чем системы общего назначения, и существует возможность снижения фрагментации памяти до приемлемого уровня. Причем производительность не уступает производительности при общесистемных методах. Пример распределения памяти можно увидеть на рисунке 3.2.

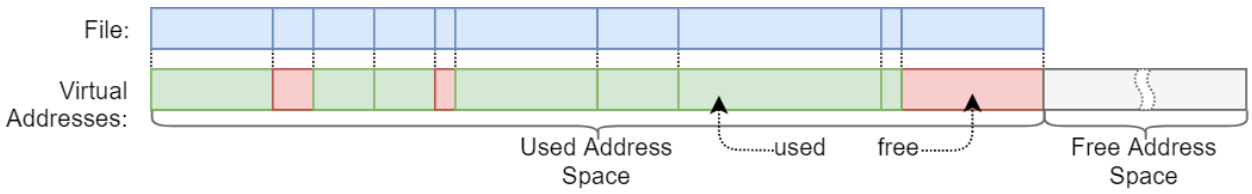

Рисунок 3.2 – Динамическая аллокация на отображаемой памяти

3.3.4. Пулы памяти

Для управления выделенной памятью используем так называемые пулы памяти, которые позволяют выделять блок памяти заданного размера для последующего использования. Разработано две реализации:

- RAM Pool – осуществляет выделения участка памяти в оперативной памяти при помощи системного аллокатора;
- MMap File Pool – осуществляет порождение файла в системе хранения, выделение диапазона виртуальных адресов и привязку файла.

Пул памяти позволяет произвести выделение линейного блока памяти для последующего использования в динамическом аллокаторе. Для возможности масштабирования поддерживается операция изменения размеров блока, т.е. аллокатор сможет запрашивать дополнительные объемы памяти по мере увеличения размера занимаемых данных, аналогично использованию команды *brk* [88] для увеличения сегмента данных процесса в классической имплементации *malloc* [89]. В таком случае при использовании пула на отображаемой памяти производится изменение размеров используемого файла (без изменения диапазона виртуальных адресов), а при использовании пула на оперативной памяти происходит только изменение счетчика текущего размера (в силу того, что размер такого пула не может быть изменен без перемещения его в другое место в памяти, он создается сразу на максимально допустимый объем).

3.3.5. Динамическое выделение памяти

Алгоритм динамического выделения памяти основан на описанном автором в работе [71]. Он использует пулы памяти в качестве базовых блоков памяти, в которых будет производиться выделение более мелких участков по запросу пользователя. В случае, если места в пуле недостаточно для выделения нового участка – у пула будет запрошено увеличение размеров на заданную величину.

Выделенные и свободные участки памяти хранятся в виде двусвязного списка блоков, содержащих данные, а также верхний (Header) и нижний (Footer) заголовки (Рисунок 3.3). Заголовки содержат информацию о размере блока, используется ли он, а также позволяют найти следующий и предыдущий. Изначально для всего свободного блока памяти, выделенного пулом, создается один пустой блок.

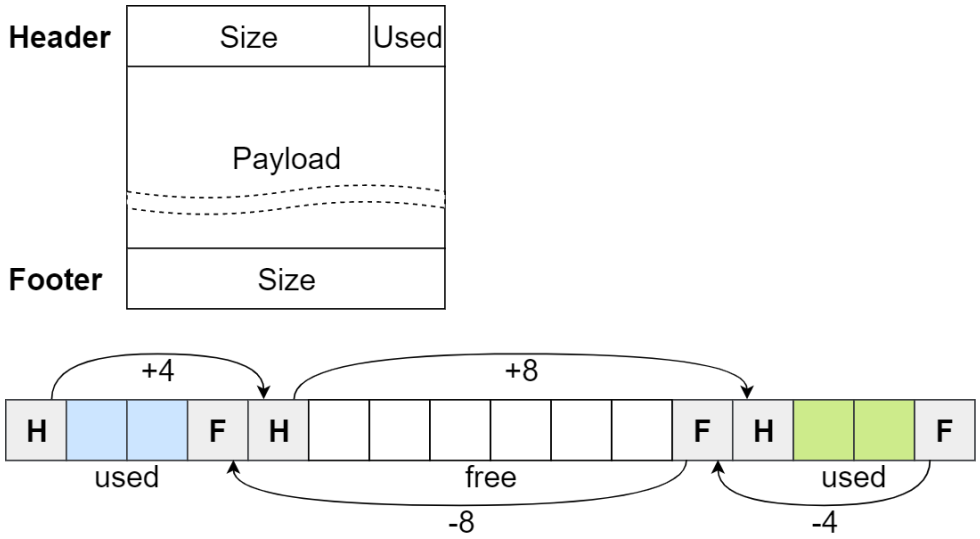

Рисунок 3.3 - Хранение участков данных

В силу необходимости выделения и освобождения участков произвольного размера появляются проблемы с фрагментацией данных (Рисунок 3.4). Возможность проведения дефрагментации, аналогичной производимой в файловых системах, отсутствует, так как перемещение участка памяти сделает все прямые указатели на нее неработоспособными. Фрагментацию можно снизить, введя возможность объединения свободных блоков. В случае, если при запросе выделения памяти не найдено свободного блока подходящего размера, необходимо запросить у пула увеличение его размеров.

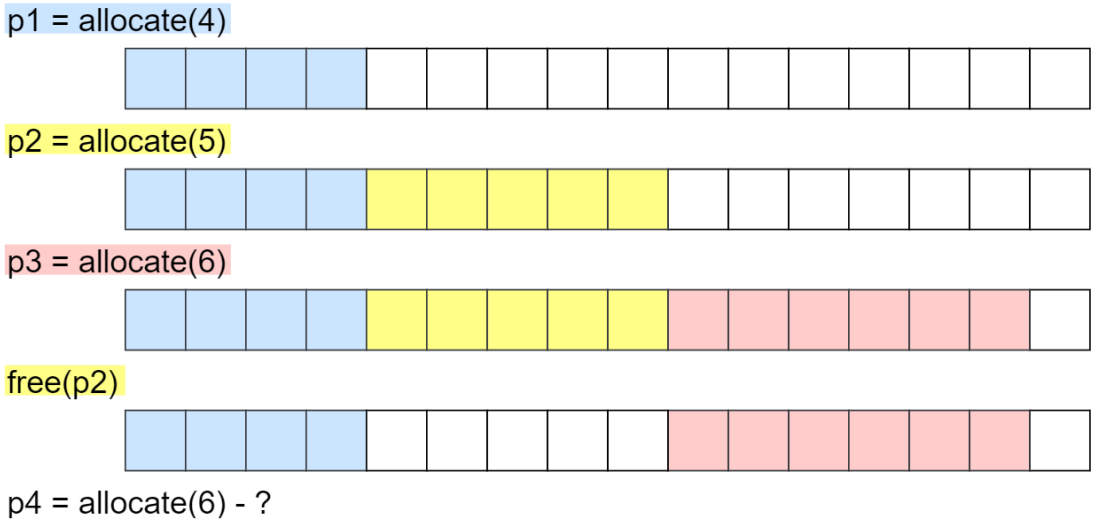

Рисунок 3.4 - Фрагментация данных при выделении памяти

При необходимости аллокации нового участка в памяти следует выполнить поиск наиболее подходящего свободного блока. Перебор всех блоков плохо подходит, так как имеет линейную сложность. Для быстрого поиска аллокатор хранит информацию о местоположении и размере всех блоков в виде отсортированного (по размеру блока) списка. Поиск по такому списку имеет логарифмическую сложность.

Для освобождения выделенных данных указанный блок просто помечается удаленным и помещается в список свободных блоков. Тогда после удаления всех блоков мы получим сильно фрагментированный массив пустых блоков. Поэтому при удалении блока его необходимо объединять с соседними пустыми блоками (Рисунок 3.5).

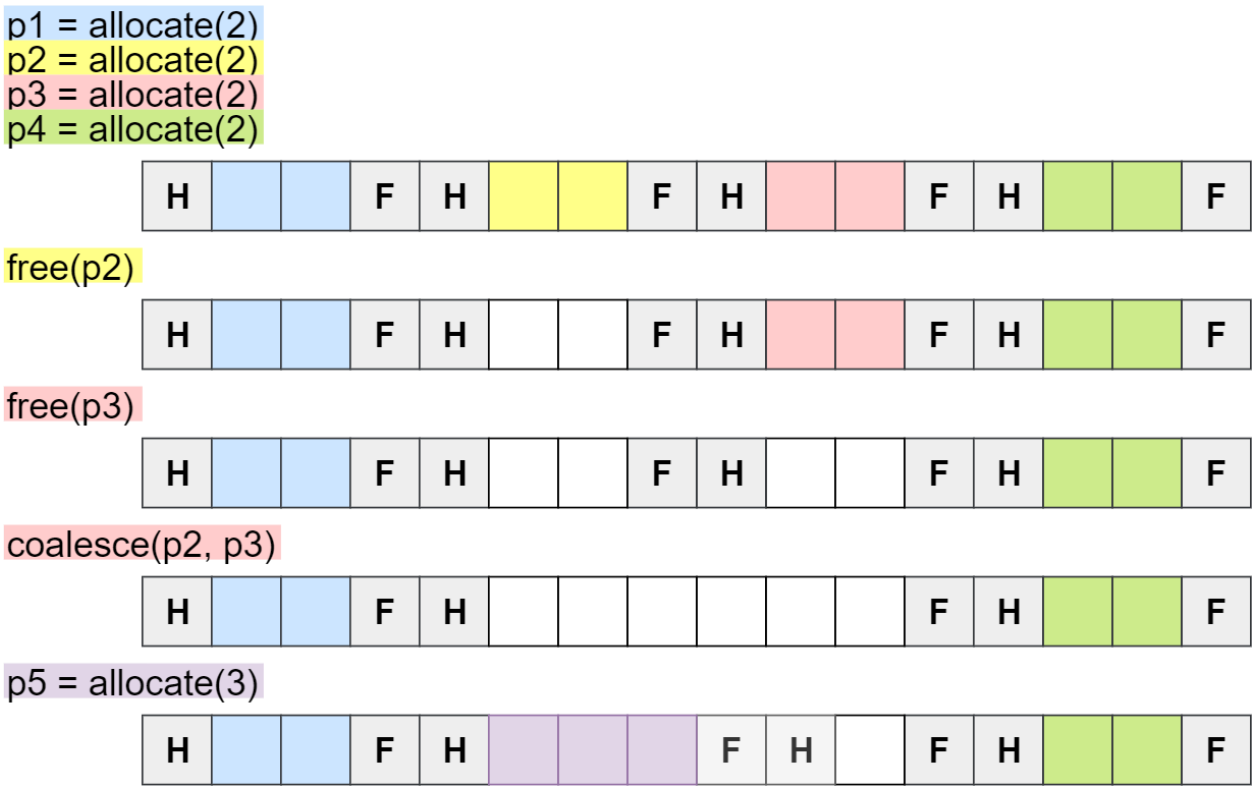

Рисунок 3.5 - Объединение блоков при освобождении памяти

Суммируя вышесказанное, система динамической аллокации имеет следующий набор функций:

– **allocate** – выделение нового блока памяти;
– **free(p)** – освобождение выделенного блока памяти по указателю **p**;

– **coalesce(p1, p2)** – объединение двух соседних блоков памяти **p1** и **p2**;
– **findBlock(size)** – поиск свободного блока памяти, размером больше или равного **size**;
– **requestMemory(size)** – запрос новой памяти размером **size** у пула.

3.3.6. Общая схема программной реализации октодерева на базе механизма отображения памяти

Рассмотрим общую схему программной реализации октодерева на базе механизма отображения памяти на рисунке 3.6. Иерархия октодерева представлена комбинацией классической организации узлов (Node) октодерева и ассоциативным массивом (Associative Array). Взаимодействие с октодеревом реализовано при помощи компонента (Octree Interface), содержащего основные методы доступа к данным. Данные узлов октодерева представлены динамическим массивом (Data), содержащим указатель на отображаемую память (Mapped Memory).

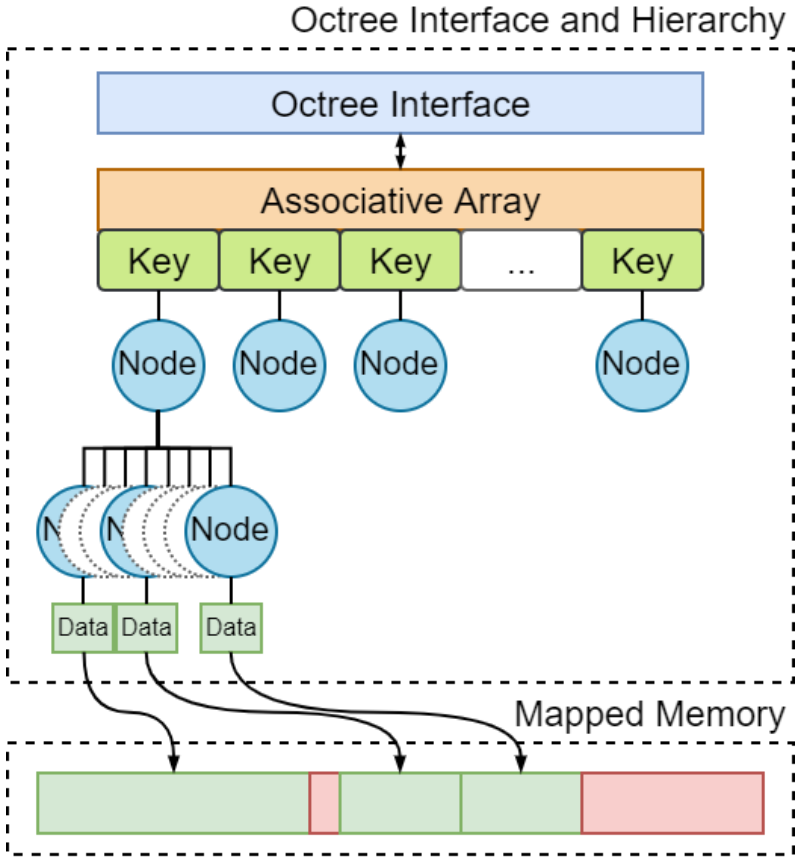

Рисунок 3.6 – Общая схема программной реализации октодерева на базе механизма отображения памяти

3.4. Построение цилиндрических проекций больших облаков точек при помощи октодерева

Обычно для решения задачи построения цилиндрической проекции больших облаков точек не требуется больших объемов памяти, так как построение проекции может быть выполнено путем последовательного считывания точек из облака точек, их проекции и записи в изображение. Однако в данной задаче требовалось выполнять хранение всех точек, попадающих в пиксель финального изображения, для обеспечения быстрой выборки и последующей обработки сторонними средствами.

Рассмотрим решение подобной задачи при помощи октодерева. Пусть дано облако точек, для которого требуется построение цилиндрической проекции. Ось цилиндра, для которого будет выполнено построение развертки, задана двумя трехмерными координатами S_1 и S_2 . Проекцию необходимо выполнить в изображение с шириной W и высотой H.

Для построения растровой проекции необходимо найти такое преобразование, которое позволит перевести координату точки из исходного пространства P_{src} в пространство P_{dst} , в котором соблюдаются следующие условия:

- Начало системы координат расположено в точке S_1;
- Ось X направлена вдоль оси цилиндра;
- Масштаб оси X таков, что для точки S_1 значение координаты равно 0, а для точки S_1 значение координаты равно W;
- Координата точки по оси Y представляет угол наклона точки вокруг оси цилиндра;
- Масштаб оси Y таков, что для угла -180° значение координаты равно 0, а для угла 180° значение координаты равно H;
- Координата точки по оси Z представляет кратчайшее расстояние между осью цилиндра и точкой.

После преобразования системы координат все точки добавляются в октодерево. При выборке из такого октодерева оси X и Y будут соответствовать осям X и Y изображения, а ось Z будет содержать глубину точки, т.е. ее расстояние от оси проекции.

3.5. Способ обработки больших облаков точек путем внедрения системы аллокации отображаемой памяти в сторонние библиотеки для linux систем

В рамках исследования алгоритмов обработки облака точек ЛС было разработано программное обеспечение для решения проблемы выделения трубопровода в природно-технических системах. В качестве основы для применения алгоритмов обработки облаков точек была использована библиотека Point Cloud Library (PCL). Однако данная библиотека не позволяет выполнять обработку больших облаков точек при помощи используемых алгоритмов. Для обеспечения возможности обработки больших облаков точек в библиотеку PCL было произведено внедрение системы динамической аллокации на отображаемой памяти, описанной автором в работе [90]. В данном разделе будут рассмотрены алгоритмы, применяемые для решения задачи выделения цилиндрических объектов, а также способ внедрения системы динамической аллокации на отображаемой памяти для обеспечения возможности обработки больших облаков точек.

3.5.1. Разработка алгоритмов выделения цилиндрических объектов из облака точек с применением системы динамической аллокации на отображаемой памяти

При лазерном сканировании остро стоит вопрос дешифрирования и векторизации исходных данных. Ввиду большого количества разнообразных природно-технических систем (ПТС) с уникальными характеристиками, не существует единого способа классификации и обработки данных, полученных путем лазерного сканирования. В рамках данной работы будут рассмотрены алгоритмы применимые к отдельному классу ПТС — трубопроводу.

Один из наиболее часто используемых примитивов в такой системе – это цилиндрическая поверхность. Среди прочих примеров цилиндрических поверхностей природно-технических систем – труба, дерево (конус), дымовая труба (цилиндр или конус), столб (в том числе многогранный), тоннель, в некоторых случаях, кабель/провод. Выделение таких элементов требуется для:

- Проведения измерений;
- Подсчета статистики;
- Снижения визуальной нагрузки при отображении облака точек;
- Экспорта в геоинформационные системы (ГИС).

Таким образом исследования в данном направлении могут быть полезны для автоматизации процессов оценки состояния ПТС.

Рассмотрим последовательность алгоритмов, применяемых для решения задачи выделения цилиндрических поверхностей. Можно выделить три основных этапа обработки:

- Фильтрация неинформативных точек;
- Кластеризация;
- Поиск цилиндрических поверхностей.

В программе данные этапы реализованы в виде последовательности алгоритмов, которые вместе образуют конвейер обработки облака точек. Входными данными для каждого алгоритма является набор параметров, а также облако точек в формате PCL, основанное на стандартной реализации динамического массива и расположенное в оперативной памяти. Ниже будет приведено описание каждого этапа конвейера, применяемого в процессе решения данной задачи, его входных и выходных параметров, а также используемых алгоритмов.

3.5.2. Удаление малоинформативных точек

Обычно облака точек сильно зашумлены, поэтому необходимо провести процесс фильтрации для того, чтобы избавиться от малоинформативных точек, которые снижают качество получаемого результата и увеличивают время обработки. Для решения этой задачи был разработан соответствующий алгоритм. В данном алгоритме используются следующие средства из библиотеки PCL:

- Алгоритм статистической фильтрации удаленных точек на основе алгоритма Statistical Outlier Removal;
- Алгоритм вокселизации VoxelGrid.

Алгоритм Statistical Outlier Removal является двухпроходным алгоритмом. На первом проходе для каждой точки вычисляется средняя дистанция до k ближайших соседей, k является настраиваемым значением. Далее вычисляется среднее и стандартное отклонение для средних дистанций всех точек, при помощи данных параметров рассчитываются границы дистанции по формуле *mean* + *stddev_mult* * *stddev*, где *stddev_mult* является настраиваемым значением. На втором проходе все точки классифицируются как внутренние или внешние в зависимости от значения их средней дистанции до соседей. Все внешние точки считаются малоинформативными.

Алгоритм VoxelGrid предназначен для уменьшения количества точек в облаке точек [91]. Поверх исходного облака строится трехмерная сетка вокселей, после чего для каждого вокселя все точки, находящиеся внутри, аппроксимируются их центроидом. В результате получается разряженное облако точек, которое не потеряло своих геометрических свойств. Данный алгоритм позволяет значительно ускорить дальнейшую обработку облака точек.

Описанные выше средства вместе формируют алгоритм «Noise Reduction». Данный алгоритм принимает на вход облако точек для обработки, количество ближайших соседей и порог удаления выбросов, требуемые для работы алгоритма Statistical Outlier Removal. Размер вокселя для алгоритма VoxelGrid считается автоматически, исходя из средней плотности облака точек. Результат работы алгоритма можно увидеть ниже на рисунке 3.7.

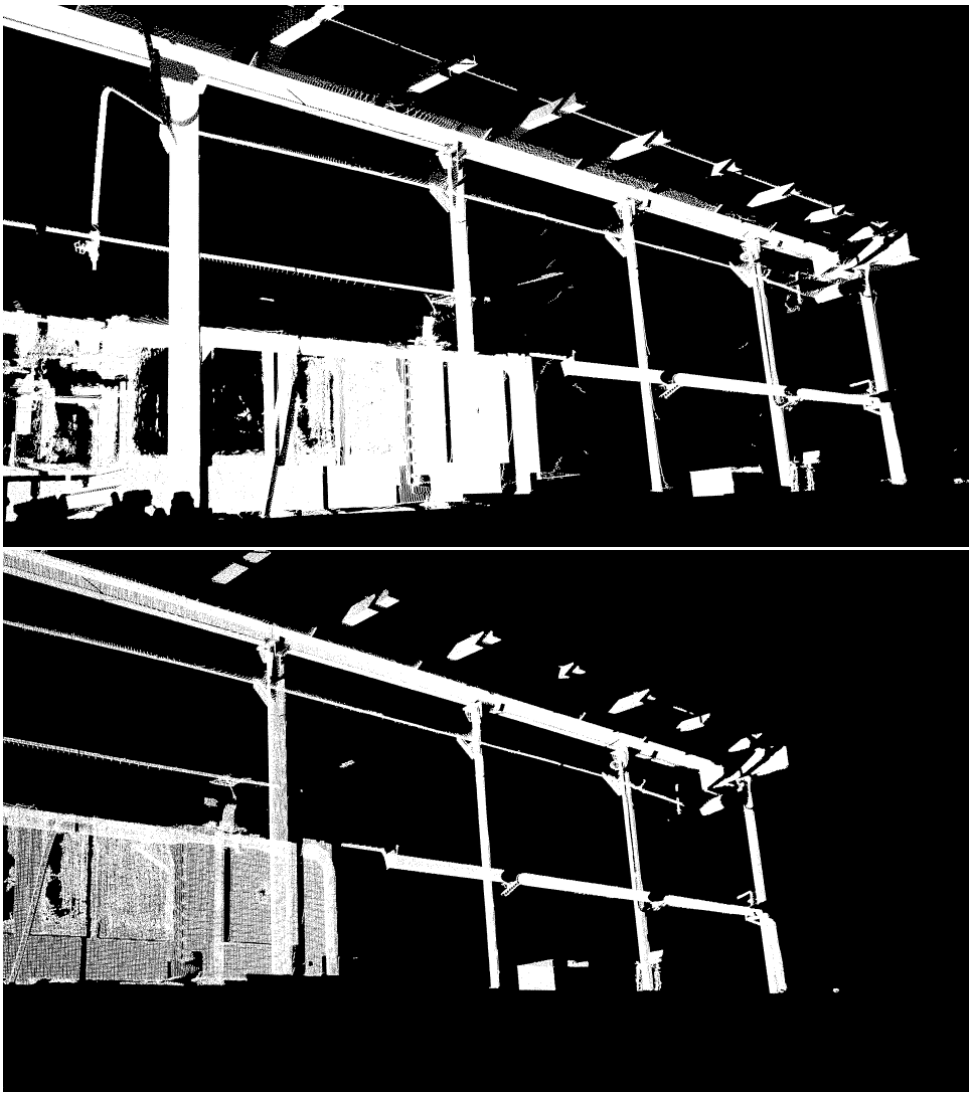

Рисунок 3.7 - Результат работы алгоритма «Noise Reduction» - до (сверху) и после (снизу)

3.5.3. Удаление неровных и плоских поверхностей

Алгоритм удаления слишком шумных или слишком плоских сегментов так же предназначен для удаления малоинформативных точек. Его применение основывается на том, что искомые цилиндрические поверхности имеют определенный радиус кривизны, а значит слишком плоские или наоборот слишком зашумленные сегменты облака точек скорее всего не будут принадлежать искомому цилиндру. Для каждой точки берется k соседей, где k — настраиваемый параметр, и по ним методом наименьших квадратов строится описывающая плоскость. После этого рассчитывается среднеквадратическое отклонение точки от плоскости. Точки с отклонением, не удовлетворяющим заданному критерию, исключаются из выборки.

Алгоритм удаления неровных и плоских поверхностей принимает на вход облако точек для обработки, а также параметры минимальной и максимальной кривизны поверхности для отсечения. Результат работы алгоритма можно наблюдать ниже на рисунке 3.8: красные точки представляют собой найденные плоские поверхности, зеленые – слишком шумные поверхности. Все подсвеченные точки будут удалены в результате работы алгоритма.

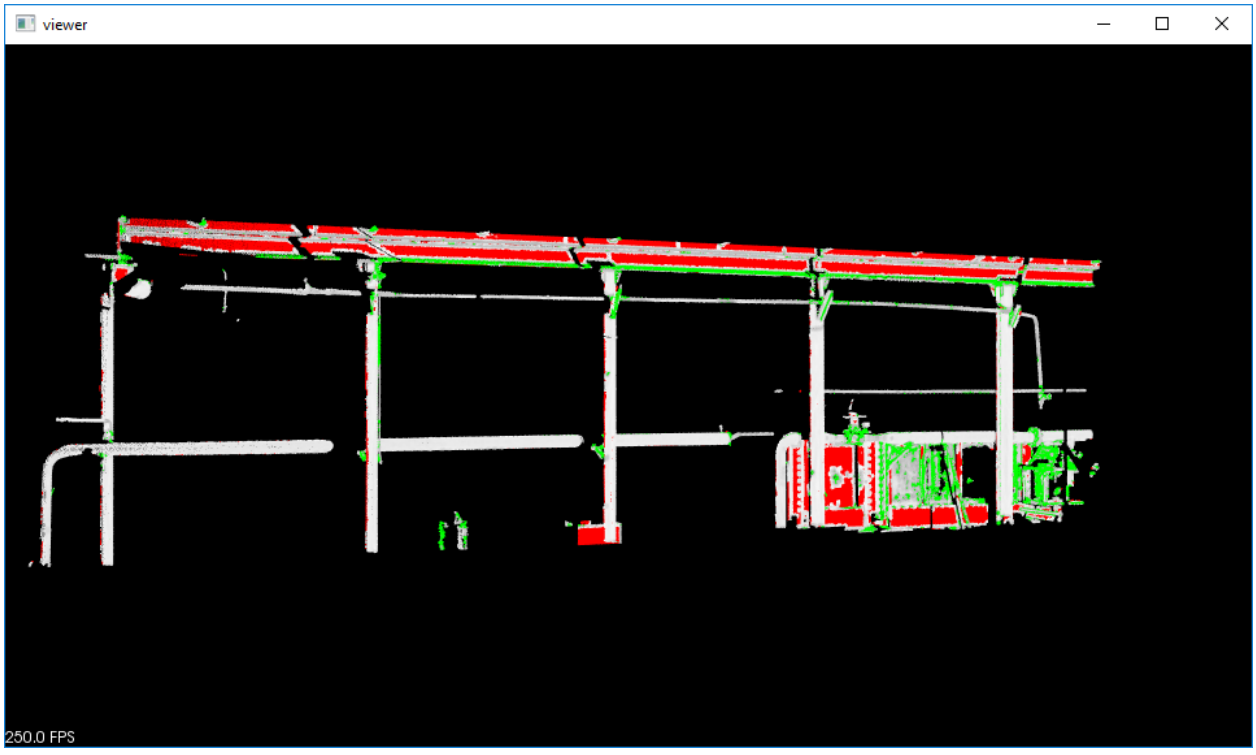

Рисунок 3.8 - Результат работы алгоритма «Plane Extraction»

3.5.4. Кластеризация

Для сокращения объема вычислений при поиске цилиндрических объектов выполняется разбиение отфильтрованного облака точек на отдельные кластеры при помощи алгоритма *Euclidean Cluster Extraction* представленного в библиотеке PCL (Рисунок 3.9).

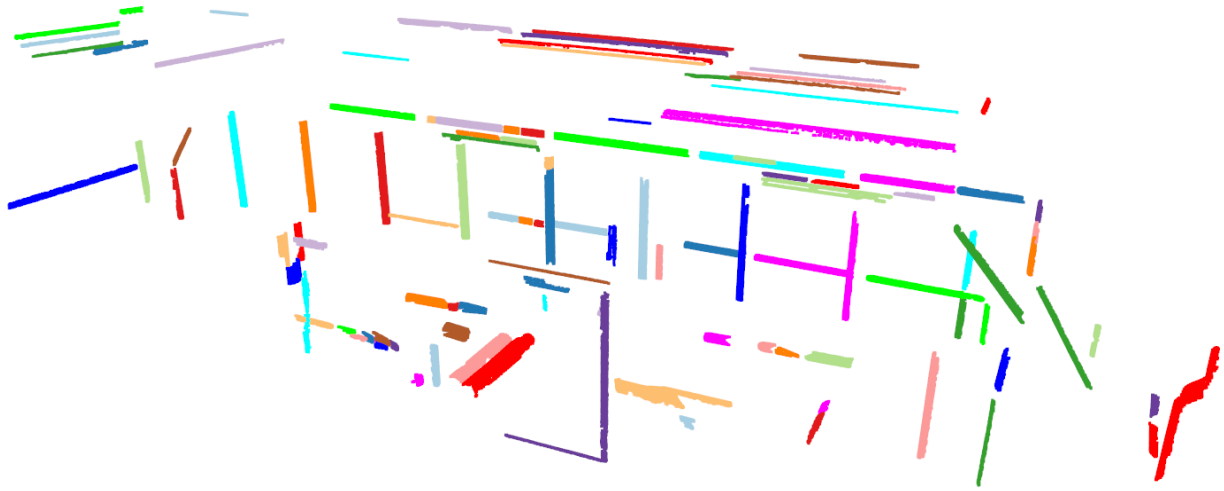

Рисунок 3.9 – Пример кластеризации отфильтрованного облака точек

3.5.5. Поиск цилиндрических поверхностей

Поиск цилиндрических поверхностей производится при помощи алгоритма Random sample consensus (RANSAC). Алгоритм основан на сборе статистики о входных данных. Из общего набора входных точек случайным образом выбирается некоторое подмножество фиксированного размера, которое аппроксимируется геометрическим примитивом. Общее количество точек входного набора, оказавшихся вблизи полученного примитива, запоминается. Этот процесс повторяется несколько раз. Геометрический примитив, вблизи которого оказалось наибольшее число точек, с высокой вероятностью является наилучшей аппроксимацией всего множества входных точек [92].

Алгоритм поиска цилиндрических поверхностей также поддерживает поиск примитивов с заданными характеристиками (радиус, направление и т.д.). В качестве входных параметров алгоритм принимает отфильтрованное и разбитое на кластеры облако точек, коэффициент вклада нормалей при поиске цилиндра, количество итераций алгоритма RANSAC, коэффициент вклада расстояний при поиске цилиндра, радиус искомых труб (минимальный и максимальный), количество итераций поиска, минимальное количество точек, которые могут принадлежать цилиндру. На выходе алгоритма можно получить список цилиндров и облако точек без цилиндрических поверхностей.

Результат работы алгоритма можно увидеть ниже на рисунке 3.10. Цветом подсвечены точки, принадлежащие найденным цилиндрам, линии отмечают оси цилиндров.

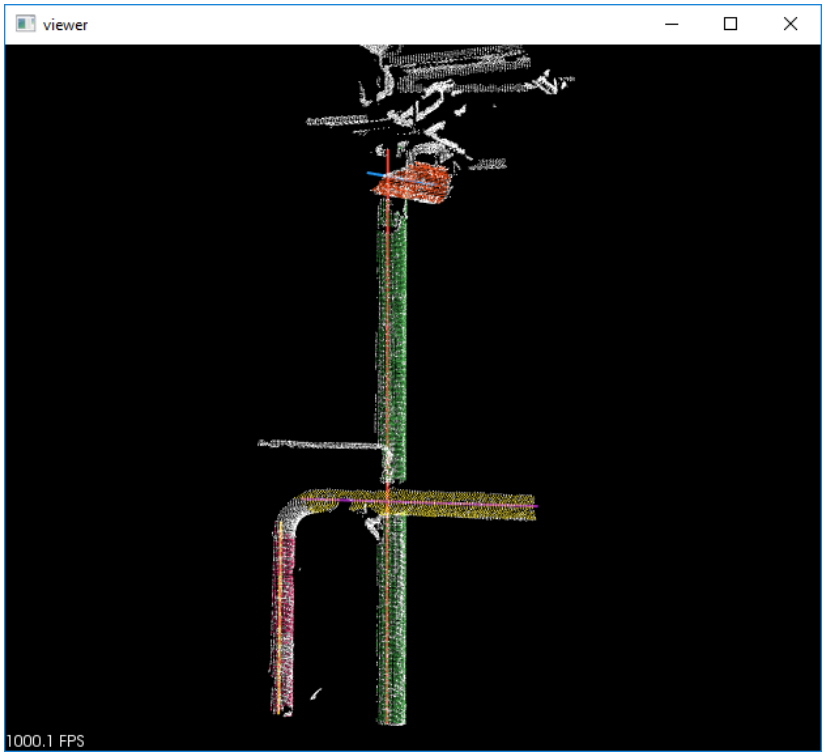

Рисунок 3.10 - Результат работы алгоритма поиска цилиндрических поверхностей

3.5.6. Внедрение системы динамической аллокации данных

Для обеспечения возможности обработки больших облаков точек необходимо сократить объемы данных, находящихся в оперативной памяти, с минимальными изменениями в исходном коде используемой библиотеки. Было рассмотрено два возможных способа:

1. Внедрение аллокатора в реализацию динамического массива PCL – позволяет с минимальными изменениями в исходном коде библиотеки обеспечить ее работу на вторичной системе хранения;
2. Полная подмена системного аллокатора – позволяет обеспечить перенаправление всех запросов на выделение и освобождение данных в предложенный аллокатор. Не требует никаких изменений в исходном коде библиотеки, однако может негативно повлиять на сложные программные компоненты.

В первом случае требуется изменить исходные коды библиотеки, чтобы используемая реализация динамического массива в библиотеке PCL была объявлена с применением аллокатора на отображаемой памяти. Само изменение невелико, но ввиду того, что данный динамический массив объявлен во многих местах в библиотеке, изменения, которые потребуется внести, будут довольно значительными. Кроме того, подобные изменения существенно усложняют переход на новые версии библиотеки.

Во втором случае для Linux-подобных операционных систем была рассмотрена возможность переопределения __malloc_hook и __free_hook [93], которая позволила использовать в функциях malloc и free свою реализацию аллокатора. Стоит отметить, что подобная глобальная подмена аллокатора привела к нестабильной работе пользовательского интерфейса и некоторых других компонентов программы, что было решено при помощи ограничения минимального размера выделяемого участка памяти, для которого будет использоваться предложенный аллокатор.

Для демонстрации данного подхода будет использоваться второй вариант реализации, по причине простоты его имплементации. Однако в общем случае стоит рекомендовать первый подход, так как он позволяет локализовать использование аллокатора для отдельных компонент системы, а также обладает большей надежностью.

3.6. Выводы

В третьей главе приведено описание алгоритмов и структур данных, применяемых при построении октодерева и обработки облака точек в условиях ограниченного потребления оперативной памяти. При проектировании алгоритмов и структур данных были применены системный и объектно-ориентированный подходы с целью выявления основных компонентов системы, определения связей между ними, их функций и целей, а также для создания необходимых абстракций и интерфейсов при реализации.

1. Рассмотрен процесс загрузки облака точек в формате LAS. Для ускорения процесса загрузки предложено использование асинхронного подхода к процессу загрузки и предобработки участков облака точек.
2. Рассмотрены алгоритмы и структуры данных, использующиеся в октодереве на базе системы кеширования. Рассмотрена реализация LRU системы кеширования с использованием асинхронной неблокирующей очереди. Предложен алгоритм построения октодерева, позволяющий сократить количество создаваемых файлов.
3. Рассмотрены алгоритмы и структуры данных, использующиеся в октодереве на базе механизма отображения. Предложена реализация иерархической структуры октодерева на базе целочисленной арифметики. Предложен способ организации хранения данных октодерева при помощи механизма отображения памяти. Предложен алгоритм динамической аллокации данных на отображаемой памяти.

4. Рассмотрен алгоритм построения цилиндрических поверхностей облаков точек, использующий октодерево для обеспечения обработки облака точек с использованием вторичной системы хранения, рассмотрен алгоритм выделения цилиндрических поверхностей, использующий алгоритм динамической аллокации для обеспечения обработки облака точек с использованием вторичной системы хранения.

В данной главе была рассмотрена программная реализация следующих положений диссертации:

1. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе асинхронной двухуровневой системы кеширования, использующий внешнюю память при превышении обрабатываемой информацией объема оперативной памяти и позволяющий сократить количество создаваемых файлов для снижения влияния обменов с внешней памятью на производительность.
2. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе механизма отображения памяти, использующий внешнюю память для хранения информации с возможностью прямого доступа и интерактивной модификации, а также позволяющий ускорить выполнение операций доступа к данным октодерева за счет кодирования с использованием целочисленной арифметики и сократить количество создаваемых файлов до одного для снижения влияния обменов с внешней памятью на производительность.

В данной главе были решены следующие задачи диссертации:

1. Разработка новых методов организации вычислительного процесса обработки облака точек, основанных на выдвинутой гипотезе. Разработка компонентов, алгоритмов и структур данных таких систем, исследование взаимодействий между ними, системным ПО, оперативной и внешней памятью.

Глава 4. Экспериментальная апробация и оценка эффективности

Данная глава посвящена экспериментальному исследованию предложенных октодеревьев, а также алгоритмов обработки облаков точек с использованием предложенных октодеревьев. Предложенные алгоритмы сравниваются с существующими решениями, рассматриваются их недостатки и ограничения.

Результаты, изложенные в данной главе были опубликованы соискателем в статье [29].

В данной главе выполняются следующие задачи диссертации:

1. Планирование и проведение экспериментальных исследований с целью оценки эффективности предложенных методов для различных задач обработки облака точек и подтверждения правильности выдвинутой гипотезы. Определение показателей, характеризующих вычислительный процесс обработки облака точек при использовании внешней памяти, и разработка вычислительных экспериментов для получения этих показателей и сравнения с существующими реализациями. Разработка методики выбора параметров октодерева в зависимости от целевой направленности обработки. Исследование возможности применения предложенных решений в сторонних программных библиотеках обработки облаков точек, ориентированных на работу в оперативной памяти, и разработка соответствующих практических рекомендаций.

4.1. Исходные данные для тестирования

Облака точек, при помощи которых будет производиться экспериментальная оценка, приведены в таблице 4.1. Выбор облаков точек производится так, чтобы обеспечить дифференциацию по размеру, плотности, топологии и методу съемки в целях демонстрации работы алгоритмов на широком спектре входных наборов данных. Представленные облака точек были получены в результате решения реальных промышленных и исследовательских задач.

Таблица 4.1 – Облака точек для тестирования

№	Название	Кол-во точек, млн	Размер, ГБ
1	five.las	1.2	0.03
2	polytech.las	51.6	1.8
3	smolny.las	126	4.3
4	molodezhnoe.las	1560	53.4
5	road.las	21	0.685

Рассмотрим подробнее предложенные облака точек. Облака точек различаются по количеству точек, размеру, плотности, методу съемки. На изображениях ниже будет приведен внешний вид облаков точек, а также их плотность. Плотность рассчитывалась путем подсчета количества соседей для каждой точки в радиусе 15 см. Чем ближе значение такого радиуса к средней плотности облака точек, тем более точны результаты, поэтому данный радиус был выбран, чтобы удовлетворять всем облакам точек, предоставляя единую шкалу для сравнения плотностей. На изображении также приведена логарифмическая шкала плотности, на которой представлено соотношение цвета раскраски и значения плотности.

five.las. Данное облако точек позволяет исследовать поведение системы при малом количестве точек. Источником облака точек является лазерное сканирование. Изображение облака приведено на рисунке 4.1, а плотность на рисунке 4.2.

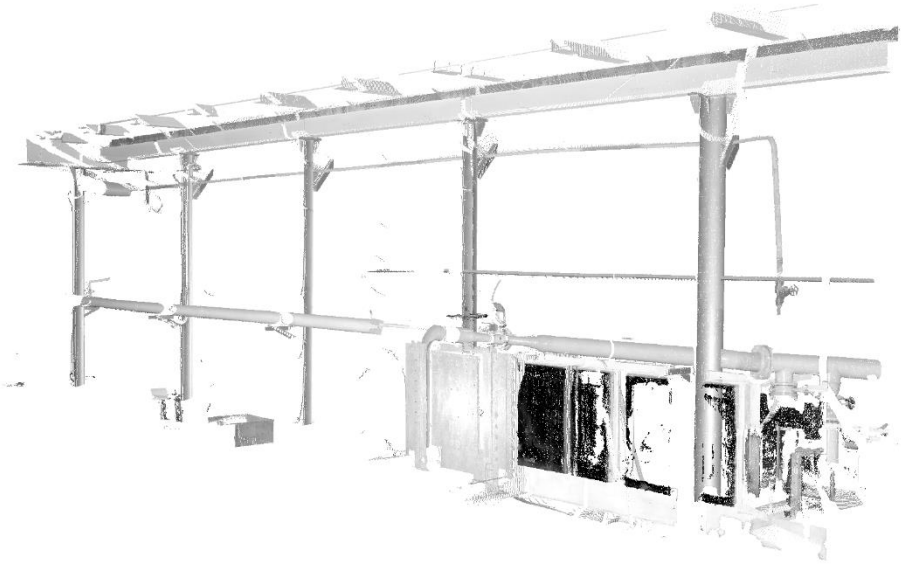

Рисунок 4.1 – Изображение облака точек five.las

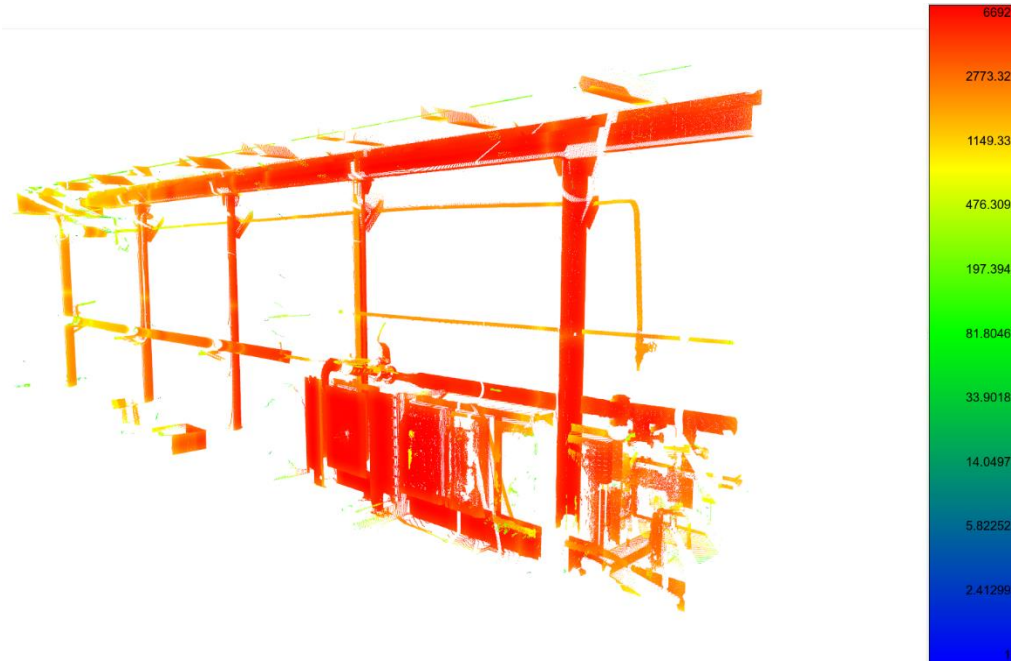

Рисунок 4.2 – Плотность облака точек five.las

polytech.las. Данное облако точек представляет результат сканирования Научно-Исследовательского Корпуса СПбПУ и его окрестностей при помощи дрона с видеокамерами, в силу чего обладает более равномерным распределением плотности, однако имеет пространственные искажения. Изображение облака приведено на рисунке 4.3, а плотность на рисунке 4.4.

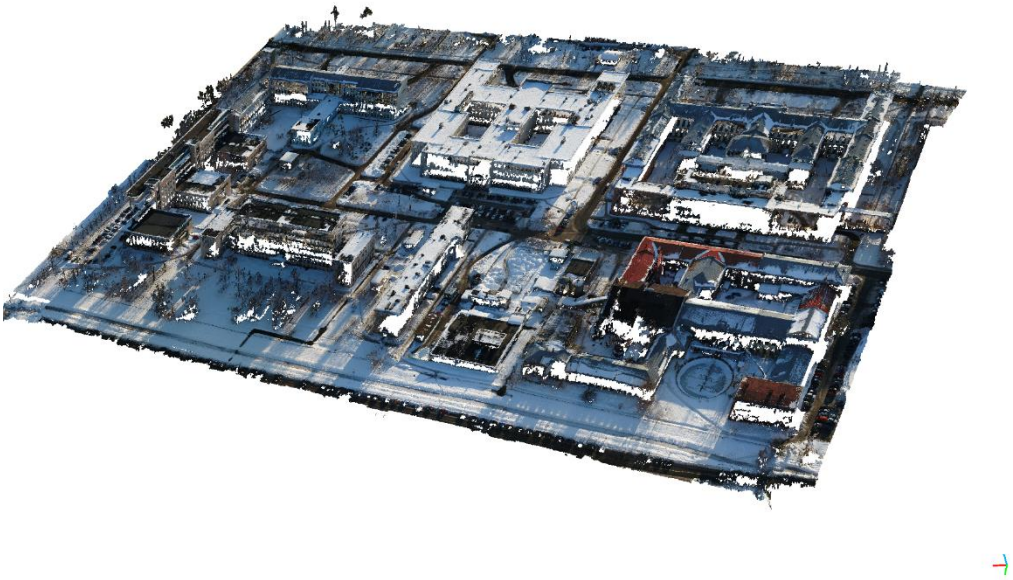

Рисунок 4.3 - Изображение облака точек polytech.las

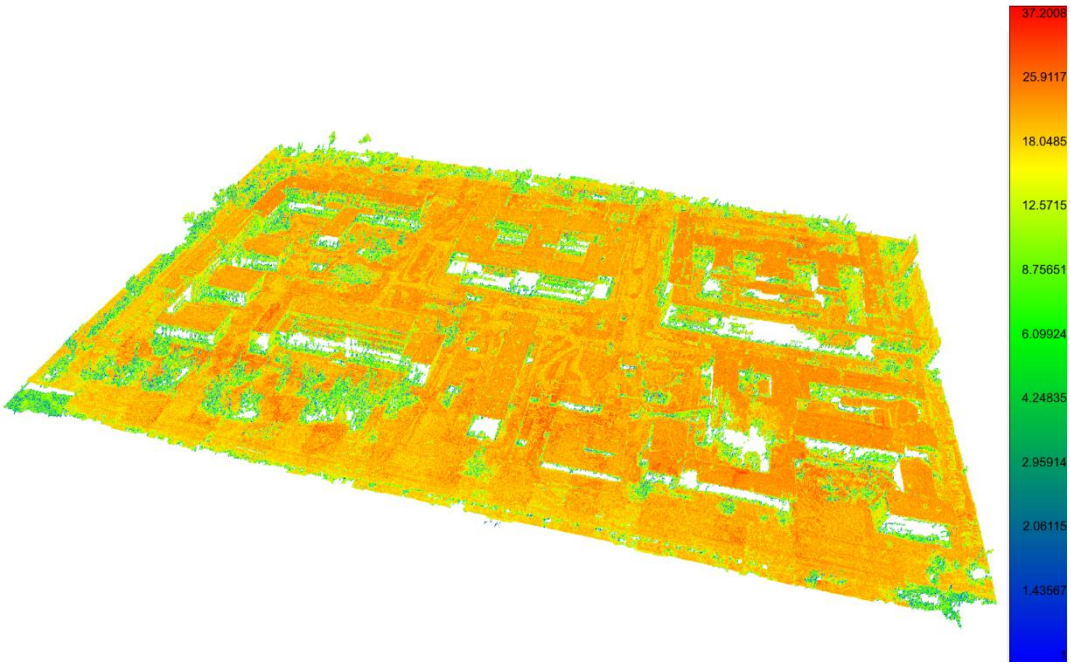

Рисунок 4.4 - Плотность облака точек polytech.las

smolny.las. Данное облако уже содержит значительное количество точек и является результатом совмещения данных с трех точек сканирования. Изображение облака приведено на рисунке 4.5, а плотность на рисунке 4.6.

Рисунок 4.5 - Изображение облака точек smolny.las

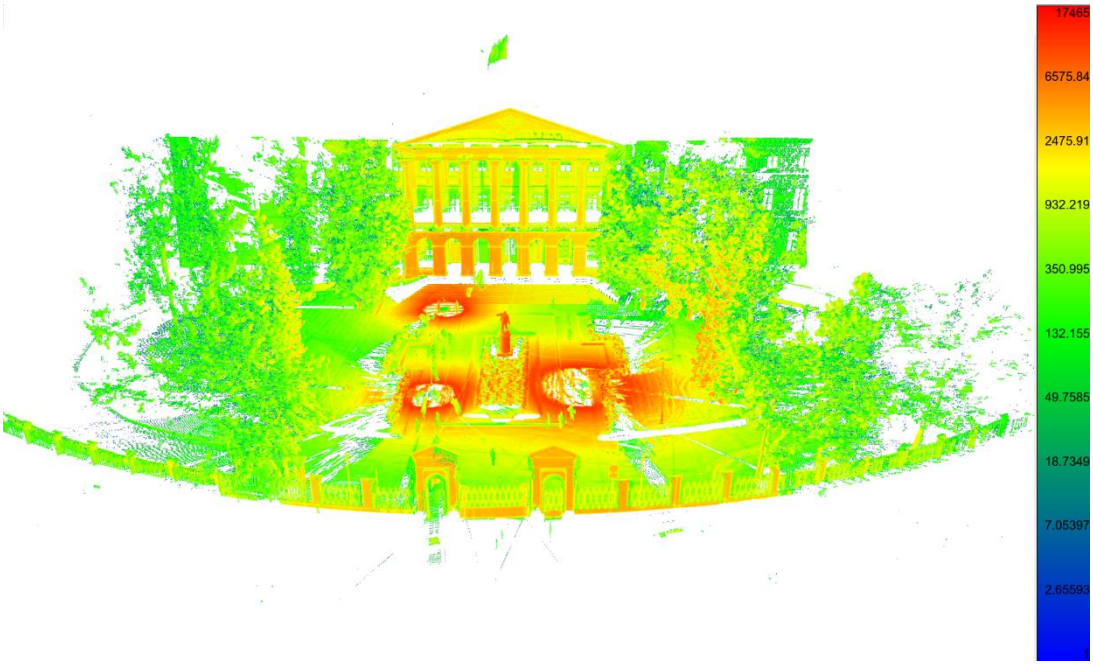

Рисунок 4.6 - Плотность облака точек smolny.las

molodezhnoe.las. Данное облако содержит полтора миллиарда точек и уже не может быть загружено в память целиком в большинстве вычислительных систем. Является результатом совмещения данных с многочисленных точек сканирования. Изображение облака приведено на рисунке 4.7, а плотность на рисунке 4.8.

Рисунок 4.7 - Изображение облака точек molodezhnoe.las

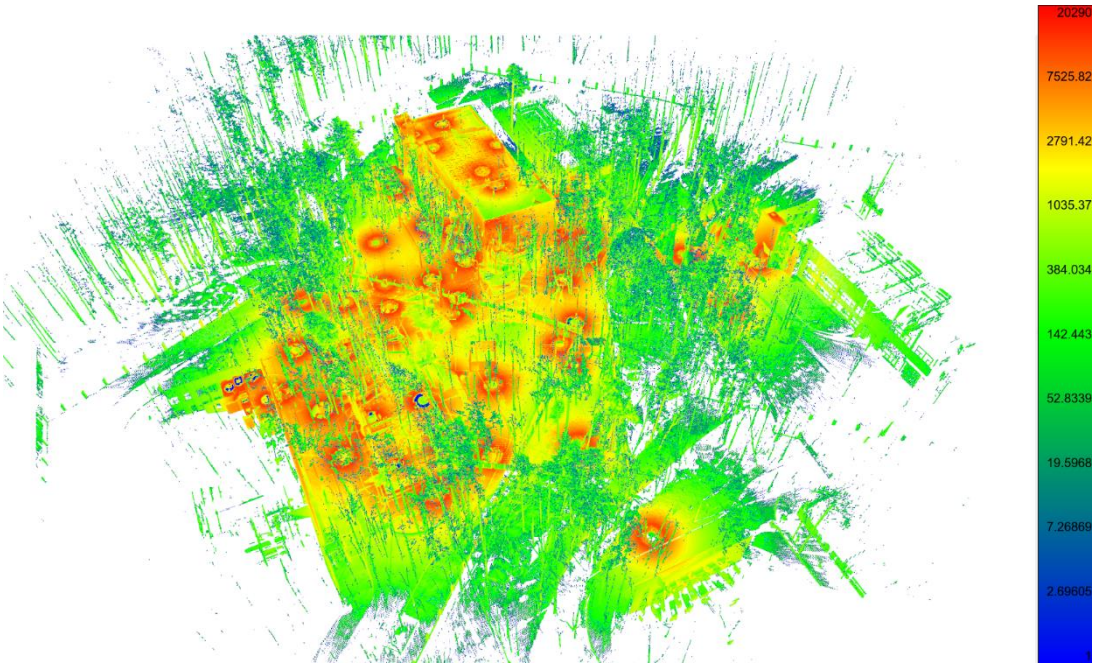

Рисунок 4.8 - Плотность облака точек molodezhnoe.las

road.las. Данное облако используется в алгоритме построения цилиндрической проекции и является результатом мобильного лазерного сканирования дороги. Изображение облака приведено на рисунке 4.9, а плотность на рисунке 4.10.

Рисунок 4.9 - Изображение облака точек road.las

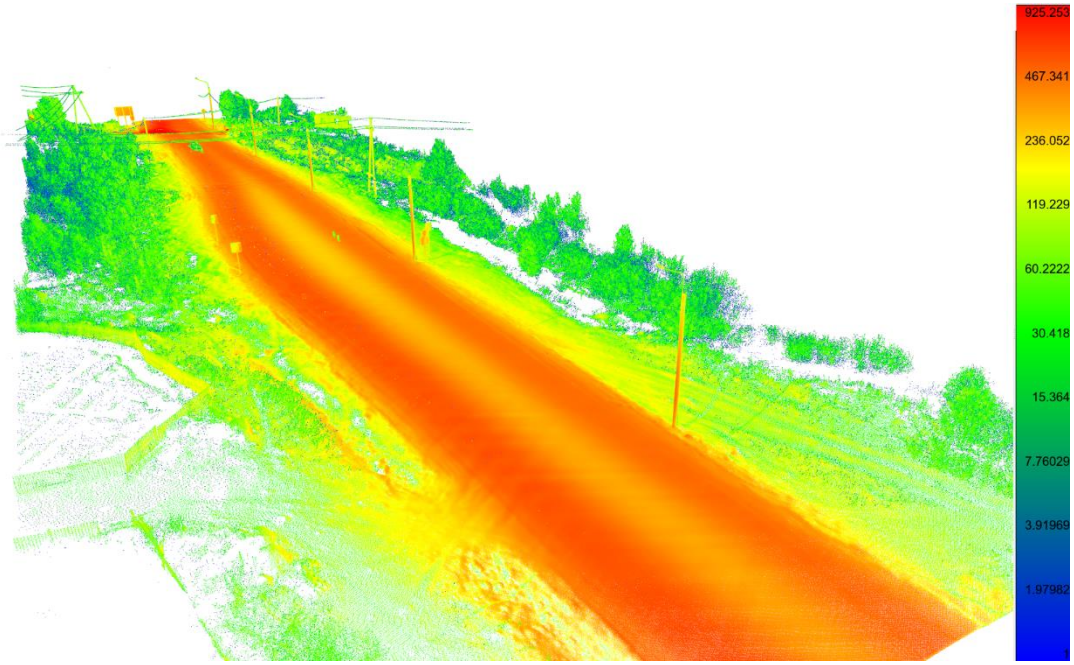

Рисунок 4.10 - Плотность облака точек road.las

4.2. Экспериментальная оценка алгоритмов формирования октодерева

В данном разделе будет рассмотрен процесс построения октодеревьев с использованием стандартных и предложенных алгоритмов. Результаты опубликованы автором в статье [29]. В качестве исходных облаков точек будут использованы *five.las*, *polytech.las*, *smolny.las*, *molodezhnoe.las*, что позволит продемонстрировать работу октодеревьев на широком диапазоне входных данных, различающихся по количеству точек, их плотности и методу съемки.

4.2.1. Разработка системы показателей, характеризующих процесс структурирования данных, превышающих объем основной памяти, и разработка вычислительных экспериментов для получения этих показателей

Выделим основные показатели, позволяющие охарактеризовать процесс структурирования данных, превышающих объем основной памяти. Прежде всего стоит обратить внимание на объем потребляемой оперативной памяти. Данный показатель позволяет доказать корректность работы системы в задаче уменьшения потребляемой оперативной памяти, а также наглядно демонстрирует зависимость потребляемой памяти от различных параметров октодерева. Для оценки производительности предложенных октодеревьев используем показатели времени построения и времени поиска в октодереве. Данные показатели позволяют оценить снижение производительности по сравнению с работой в оперативной памяти, а также продемонстрировать зависимость скорости построения и поиска от различных параметров октодерева.

Рассмотрим процесс экспериментальной оценки предложенных октодеревьев, использующий выделенные показатели. Целью экспериментальной оценки будет измерение:

- Объема потребления памяти в процессе построения октодерева;
- Скорости построения октодерева в зависимости от различных параметров узлов;

– Скорости поиска в построенном октодереве в зависимости от различных параметров узлов.

Выделим набор тестов, необходимых для анализа производительности и потребления памяти октодеревом:

– Тест №1 предназначен для оценки зависимости времени построения октодерева от используемого размера узла. Для октодерева на базе системы кеширования дополнительно измеряется скорость построения в зависимости от установленных размеров кеша. Тест производится путем измерения скорости построения октодерева различных облаков точек;
– Тест №2 предназначен для оценки потребляемой памяти в процессе построения октодерева. Тест производится путем измерения потребляемой памяти в процессе построения октодерева для определенного облака точек;
– Тест №3 предназначен для оценки зависимости скорости поиска в октодереве от используемого размера узла. Для октодерева на базе системы кеширования дополнительно измеряется скорость поиска в зависимости от установленных размеров кеша. Тест производится путем измерения скорости поиска в октодереве для различных облаков точек.

Для оценки снижения производительности при использовании вторичных систем хранения те же тесты будут проведены с использованием только оперативной памяти, за исключением облака точек *molodezhnoe.las*, так как оно слишком велико.

Для проведения вычислительных экспериментов будем использовать следующую конфигурацию тестового стенда:

Центральный процессор (ЦПУ): Intel® Core™ i7-7820HQ CPU @ 2.90 GHz

ОП: 2 x SODIMM DDR4 2400 MHz (2x8 GB)

ВП: Samsung SSD 850 PRO 1 TB

ОС: Ubuntu 18.04

ФС: Ext4

4.2.2. Экспериментальная оценка октодерева на базе системы кеширования

Тест №1: Время построения дерева. На рисунке 4.11 и рисунке 4.12 представлен результат анализа производительности построения октодерева при использовании системы кеширования. На графиках сплошная линия используется для отображения результатов обработки в файловой памяти, пунктирная – для отображения результатов обработки в оперативной памяти, цвет соответствует используемому облаку точек (зеленый – *five.las*, красный – *polytech.las*, бирюзовый – *smolny.las*, розовый – *molodezhnoe.las*). Для этого проведены измерения времени построения дерева для различных облаков точек в оперативной (RAM) и файловой (HDD) памяти путем изменения размера узла и размера кеша. Такой тест позволяет выяснить оптимальные параметры дерева, а также продемонстрировать разницу в скорости построения дерева в оперативной памяти и с использованием жесткого диска.

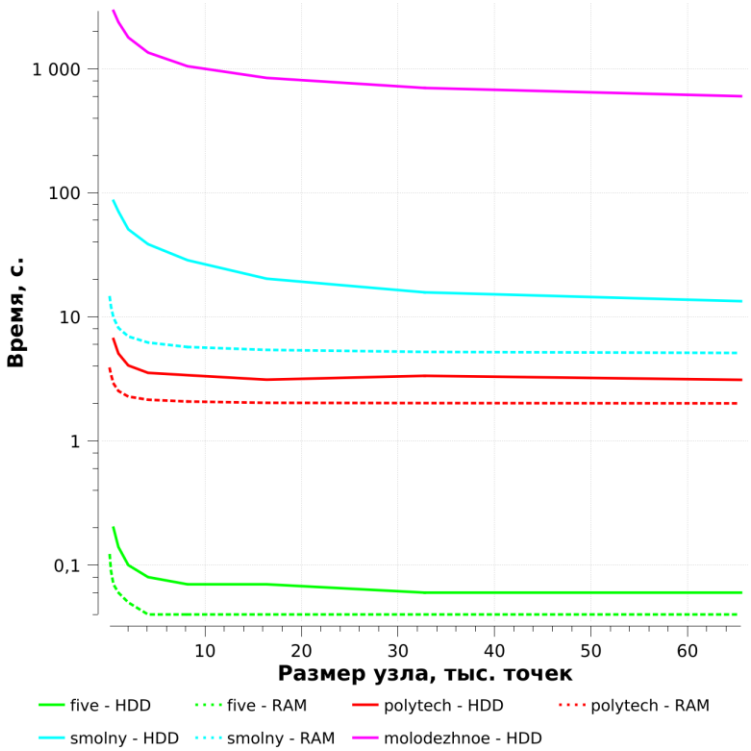

Рисунок 4.11 - Зависимость скорости построения дерева от максимального размера узла

Как видно на рисунке 4.11, построение дерева значительно замедляется при использовании узлов маленького размера. Это объясняется прямой зависимостью между размером узла и глубиной дерева: чем меньше узел, тем на большую глубину спускается очередная точка из облака при построении. В случае с файловыми операциями размер узла влияет особенно сильно, так как приводит к значительному увеличению количества создаваемых файлов. Увеличение размера узла также негативно влияет на скорость поиска по дереву, что будет продемонстрировано в дальнейших тестах.

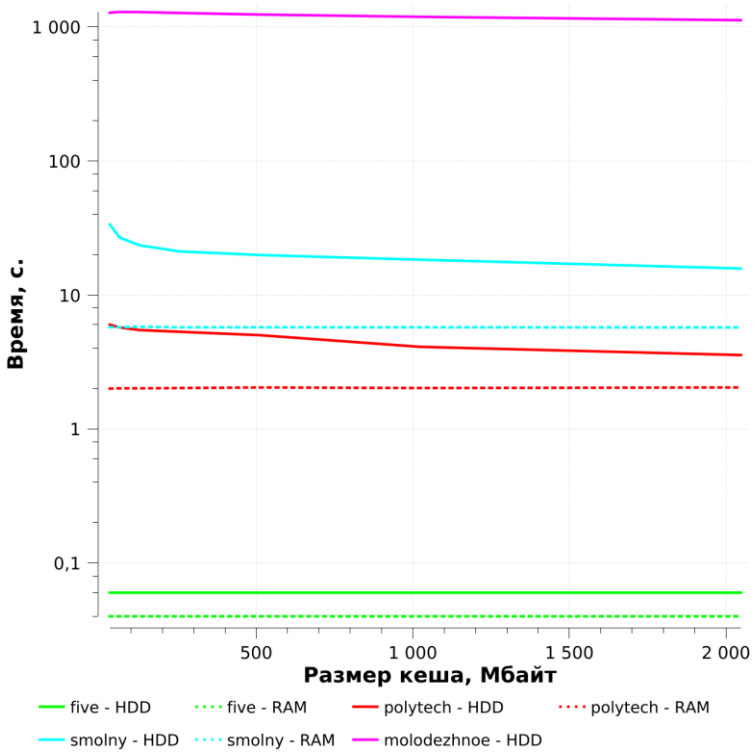

Рисунок 4.12 - Зависимость скорости построения дерева от используемого объема кеша

Из рисунка 4.12 видно, что увеличение объема кеша ускоряет процесс построения дерева. Тем не менее, значительного вклада он не вносит, так как добавляемые точки обычно разбросаны по облаку неравномерно.

По результатам проведенного теста можно утверждать, что увеличение размера узла или объема кеша приводит к ускорению построения октодерева. Также можно заметить, что характер изменений практически не зависит от исследуемого облака точек. Для выбора наилучшего значения размера узла необходимо также провести измерения скорости поиска по построенному дереву. Что касается выбора подходящего размера кеша, то рекомендуется использовать максимально допустимое значение в вашей системе, но не превышающее размер наибольшего обрабатываемого облака точек.

Тест №2: Потребление памяти и файловые операции. Проверим корректность работы системы кеширования в условиях ограничения потребляемой оперативной памяти. В данном тесте приведены графики потребления оперативной и файловой памяти. На рисунке 4.13 приведен график зависимости объема потребления L1 (RAM) и L2 (HDD) от количества загруженных точек в процессе построения октарного дерева. Потребление оперативной памяти (RAM) отображено зеленым цветом, размер данных на жестком диске (HDD) отображен синим цветом, ограничение потребляемой памяти (Capacity) отображено красным цветом. На рисунке 4.14 приведен график файловых операций в процессе построения октарного дерева. Тестирование производилось на облаке точек *smolny.las* с установленным максимальным объемом кеша в 1024 мегабайта, количество считанных данных приведено зеленым цветом, а количество записанных данных приведено красным цветом.

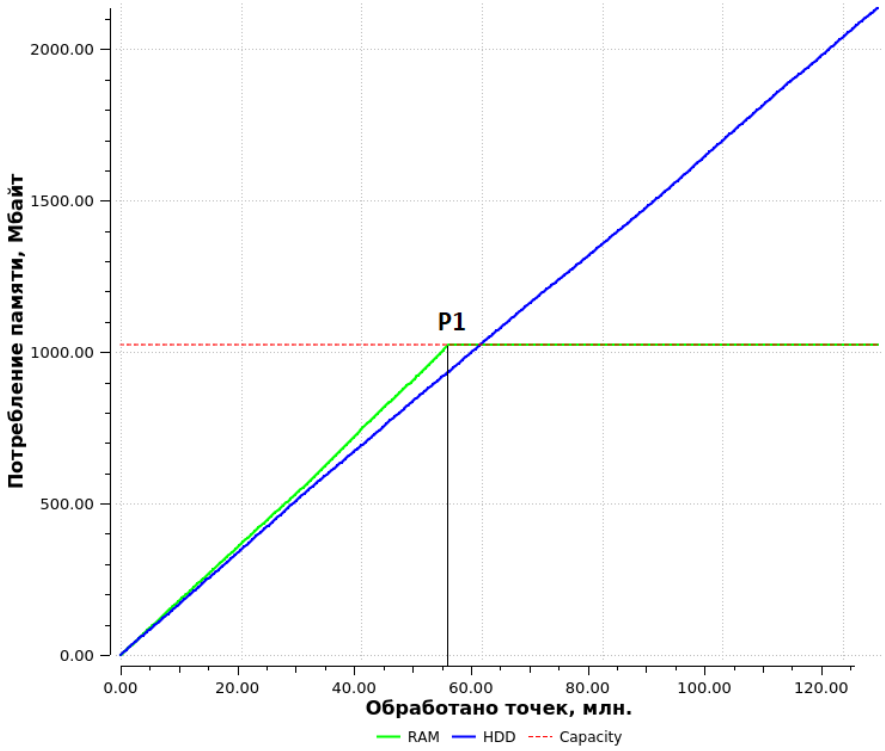

Рисунок 4.13 - Потребление памяти в процессе построения дерева

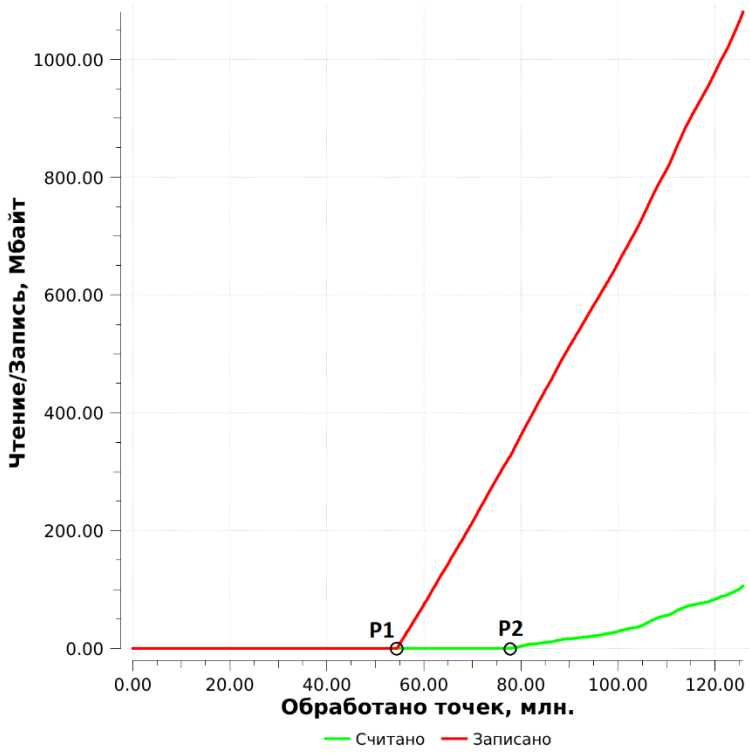

Рисунок 4.14 - Файловые операции в процессе построения дерева

Как видно из приведенных графиков, сначала происходит заполнение кеша до значения его емкости (точка P1), параллельно с выгрузкой заполненных узлов на жесткий диск. Расход оперативной памяти остается в пределах заданных значений за счет использования дискового пространства. Также на рисунке 4.14 можно увидеть, как растет интенсивность записи на жесткий диск после заполнения кеша (точка P1), а также как возрастает количество операций считывания по мере приближения к концу процесса построения (точка P2). Последнее вызвано загрузкой недозаполненных узлов, выгруженных из кеша ранее.

Результаты данного теста позволяют сделать вывод об успешности объединения октодерева и системы кеширования, так как потребление памяти оставалось в пределах заданных лимитов на протяжении всего процесса обработки облака точек.

Тест №3: Скорость поиска. Если в предыдущих тестах мы оценивали влияние размеров узла и объема кеша на скорость построения, то в данном тесте целью будет измерение скорости поиска в уже построенном дереве. Данный тест позволит оценить, насколько замедляется скорость поиска при больших размерах узлов, а также насколько увеличение объемов кеша ускоряет поиск. На рисунке 4.15 приведен график зависимости скорости поиска отдельных точек от размеров узлов, на рисунке 4.16 приведен график зависимости скорости поиска узлов октодерева от размеров узлов, на рисунке 4.17 приведен график зависимости скорости поиска от объемов кеша. На графиках сплошная линия используется для отображения результатов поиска в файловой памяти, пунктирная – для отображения результатов поиска в оперативной памяти, цвет соответствует используемому облаку точек (зеленый – *five.las*, красный – *polytech.las*, бирюзовый – *smolny.las*, розовый – *molodezhnoe.las*). В процессе тестирования для каждого исследуемого параметра производится большое количество операций поиска точек, находящихся в определенном радиусе от случайной точки из облака. В качестве результата используется среднее значение времени поиска.

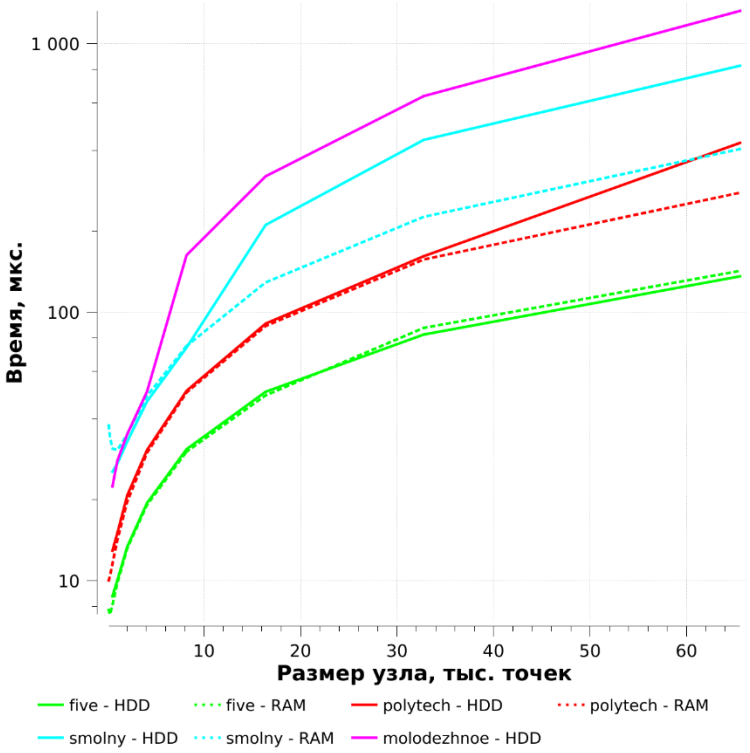

Рисунок 4.15 – Скорость поиска отдельных точек в октодереве в зависимости от размера узла

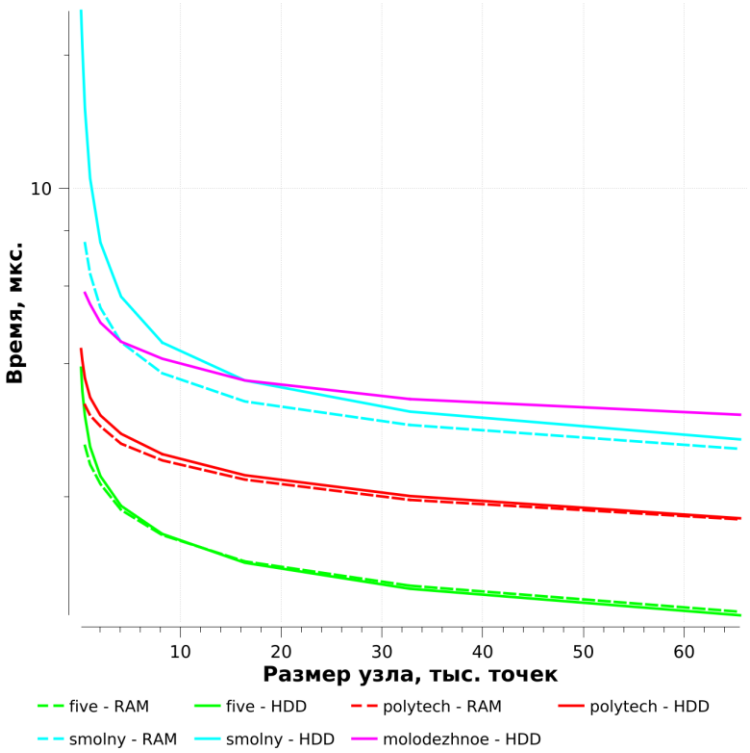

Рисунок 4.16 – Скорость поиска узлов октодерева в зависимости от размеров узла

Как видно из графика на рисунке 4.15, чем больше количество точек в узле, тем медленнее происходит поиск отдельных точек. Данный результат обусловлен механизмом поиска: после проверки попадания узла в радиус поиска, алгоритм выполняет проверку для всех точек, входящих в узел (т.к. попадание узла не гарантирует попадание всех точек узла). Соответственно, чем больше точек в узле, тем большее количество проверок отдельных точек предстоит провести. В случае, когда размер узла невелик, большая часть точек отсекается на этапе более дешевой проверки узлов. Однако, значения скорости поиска отдельных узлов, приведенные на рисунке 4.16, дают обратный результат по сравнению с поиском отдельных точек: чем больше количество точек в узле, тем быстрее происходит поиск. Это объясняется тем, что увеличение максимального количества точек в узле снижает общее количество узлов и глубину иерархии октодерева.

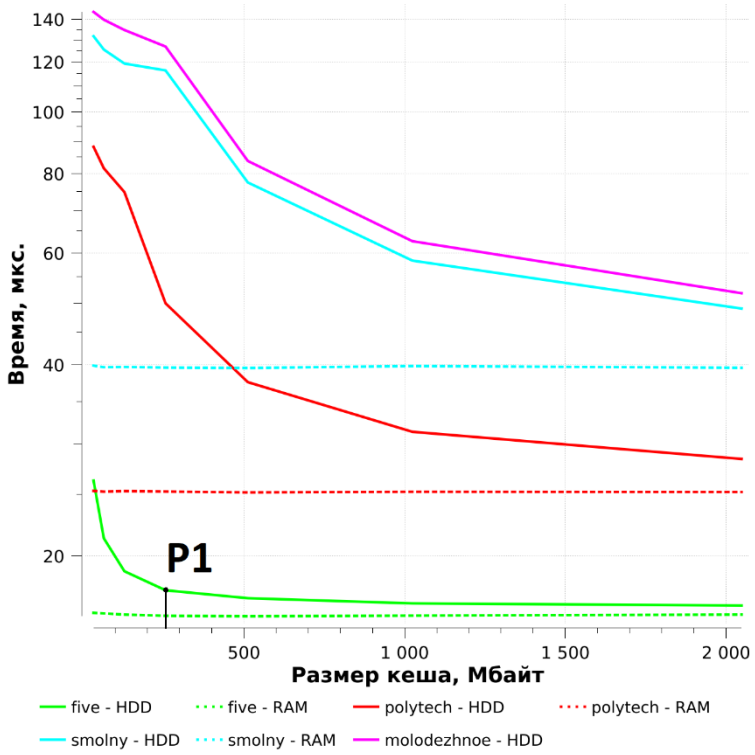

Рисунок 4.17 - Скорость поиска в октодереве в зависимости от объема кеша

На рисунке 4.17 можно наблюдать ускорение операций поиска при увеличении объема кеша. Стоит отметить, что оптимальный объем кеша возрастает с увеличением размера облака точек – это можно увидеть по тому, как насыщается график *five.las* в точке P1, а также по динамике изменения графиков *polytech.las* и *smolny.las*.

Результаты проведенных тестов подтверждают, что октодерево успешно справляется с работой в условиях ограничений по оперативной памяти, т.к. отслеживаемые объемы данных (а именно – полезная нагрузка узлов октодерева) не превышают заданного значения. Так же проведенное экспериментальное тестирование позволило установить, что увеличение объема кеша положительно влияет на скорость поиска, в то время как при увеличении объема узла скорость поиска замедляется. Таким образом, при выборе подходящего размера узла следует также руководствоваться необходимой скоростью поиска. Нами было выбрано значение в 2048 точек, как компромисс между скоростью построения и скоростью поиска. Способ выбора подходящего размера кеша остается неизменным.

4.2.3. Экспериментальная оценка октодерева с использованием механизмов отображения памяти

Тест №1: Время построения дерева. На рисунке 4.18 представлен результат анализа производительности построения октодерева при использовании системы кеширования. На графике сплошная линия используется для отображения результатов обработки в файловой памяти, пунктирная – для отображения результатов обработки в оперативной памяти, цвет соответствует используемому облаку точек (зеленый – *five.las*, красный – *polytech.las*, бирюзовый – *smolny.las*, розовый – *molodezhnoe.las*). Для этого проведены измерения времени построения дерева для различных облаков точек памяти путем изменения размера узла. Такой тест позволит выяснить оптимальные параметры дерева, а также оценить его скорость построения.

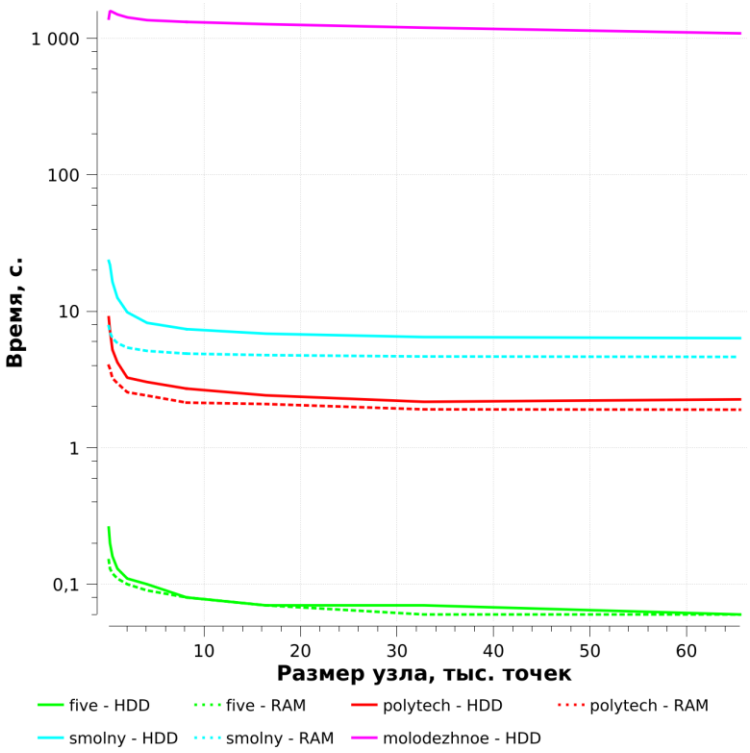

Рисунок 4.18 - Зависимость скорости построения дерева от максимального размера узла

Как видно на рисунке 4.18, построение дерева значительно замедляется при использовании узлов маленького размера. Это объясняется прямой зависимостью между размером узла и глубиной дерева: чем меньше узел, тем на большую глубину спускается очередная точка из облака при построении.

По результатам проведенного теста можно утверждать, что увеличение размера узла приводит к ускорению построения октодерева. Также можно заметить, что характер изменений практически не зависит от исследуемого облака точек. Для выбора наилучшего значения размера узла необходимо также провести измерения скорости поиска по построенному дереву.

Тест №2: Потребление оперативной памяти. Далее проверим корректность работы системы отображения памяти в условиях ограничения потребляемой оперативной памяти. В данном тесте приведены графики потребления оперативной памяти. На рисунке 4.19 приведен график зависимости потребления оперативной памяти от используемых размеров узла. На графике сплошная линия используется для отображения результатов обработки в файловой памяти, пунктирная – для отображения результатов обработки в оперативной памяти, цвет соответствует используемому облаку точек (зеленый – *five.las*, красный – *polytech.las*, бирюзовый – *smolny.las*, розовый – *molodezhnoe.las*). Как видно из графика, при уменьшении максимального числа точек в узле количество оперативной памяти, используемой октодеревом, повышается. Такой рост потребляемой памяти обусловлен увеличением количества узлов, что приводит к разрастанию иерархии октодерева, расположенной в оперативной памяти.

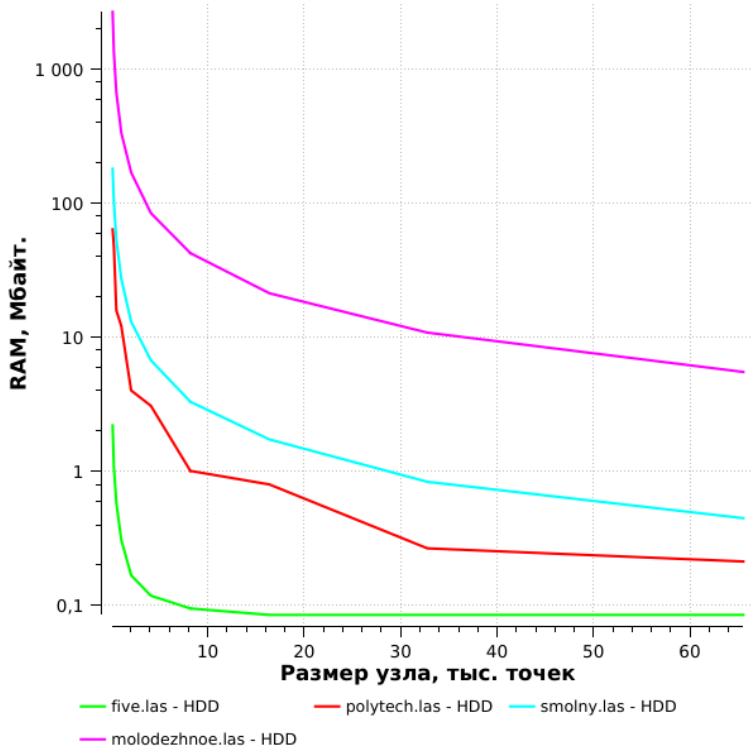

Рисунок 4.19 – Потребление оперативной памяти в зависимости от размера узла октодерева

Тест №3: Скорость поиска. Целью данного теста будет измерение скорости поиска в уже построенном дереве. Данный тест позволит оценить, насколько замедляется скорость поиска при больших размерах узлов. На рисунке 4.20 приведен график зависимости скорости поиска отдельных точек от размеров узлов, а на рисунке 4.21 приведен график зависимости скорости поиска узлов октодерева от размеров узла. На графике сплошная линия используется для отображения результатов поиска в файловой памяти, пунктирная – для отображения результатов поиска в оперативной памяти, цвет соответствует используемому облаку точек (зеленый – *five.las*, красный – *polytech.las*, бирюзовый – *smolny.las*, розовый – *molodezhnoe.las*). В процессе тестирования для каждого исследуемого параметра производится большое количество операций поиска точек, находящихся в определенном радиусе от случайной точки из облака. В качестве результата используется среднее значение времени поиска.

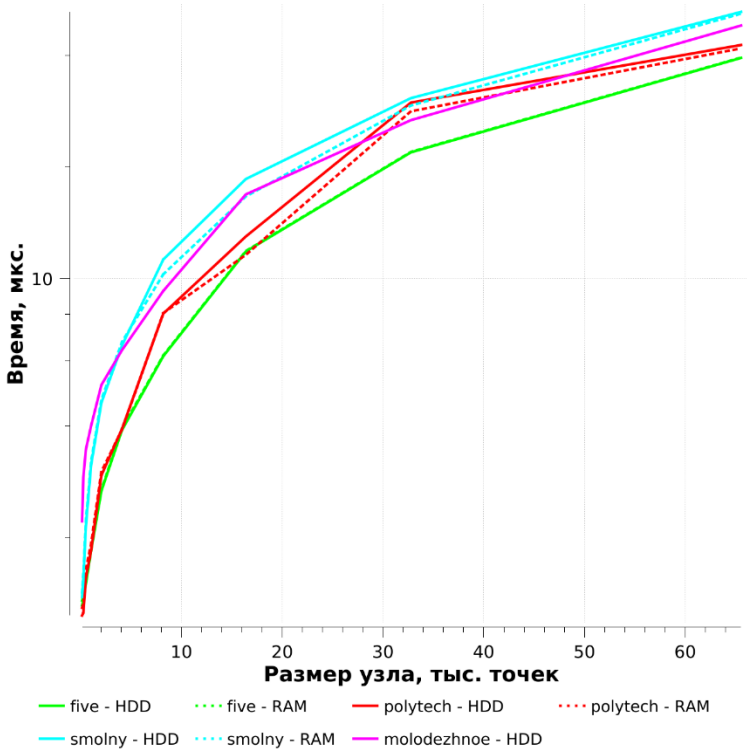

Рисунок 4.20 - Скорость поиска отдельных точек в зависимости от размера узла

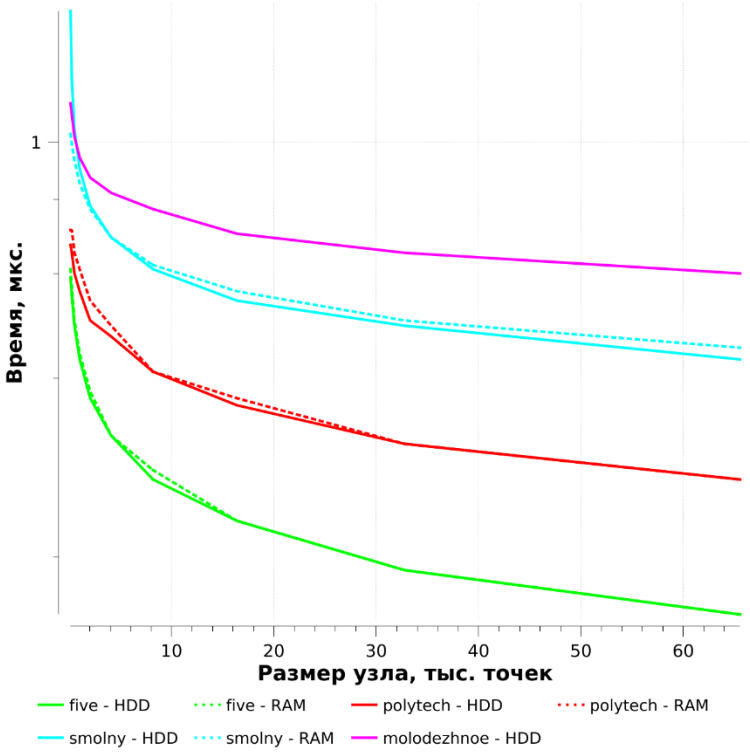

Рисунок 4.21 – Скорость поиска узлов октодерева в зависимости от размера узла

Как видно из графика на рисунке 4.20, чем больше количество точек в узле, тем медленнее происходит поиск. Данный результат обусловлен механизмом поиска: после проверки попадания узла в радиус поиска, алгоритм выполняет проверку для всех точек, входящих в узел (т.к. попадание узла не гарантирует попадание всех точек узла). Соответственно, чем больше точек в узле, тем большее количество проверок отдельных точек предстоит провести. В случае, когда размер узла невелик, большая часть точек отсекается на этапе более дешевой проверки узлов. Обратный результат можно увидеть на графике на рисунке 4.21: скорость поиска нарастает при увеличении размеров узла. Это объясняется тем, что увеличение максимального количества точек в узле снижает общее количество узлов и глубину иерархии октодерева.

4.2.4. Методика выбора параметров процесса формирования октодерева

Рассмотрим результаты экспериментального анализа предложенных октодеревьев, приведенного в разделах 4.2.2 и 4.2.3. В данном анализе производилась оценка влияния параметров октодеревьев на скорость выполнения операций построения и поиска. Совокупные результаты анализа приведены в таблице 4.2, из которой видно, что тенденция к ускорению или замедлению операций не зависит от типа предложенного октодерева (исключая влияние объемов кеша, который в явном виде отсутствует у октодерева на базе механизма отображения памяти).

Таблица 4.2 – Влияние параметров октодерева на операции построения и поиска

Исследуемое октодерево	**Операция**	**Увеличение максимального количества точек в узле**	**Увеличение объемов кеша**
Октодерево на базе системы кеширования	Построение	Ускоряет	Ускоряет
	Поиск точек	Замедляет	Замедляет
	Поиск узлов	Ускоряет	Ускоряет
Октодерево на базе механизма отображения	Построение	Ускоряет	Отсутствует
	Поиск точек	Замедляет	Отсутствует
	Поиск узлов	Ускоряет	Отсутствует

Суммируя информацию, приведенную в разделах 4.2.2 и 4.2.3 сделаем ряд утверждений:

- Увеличение объемов кеша положительно влияет на скорость выполнения операций построения и поиска;
- Увеличение максимального количества точек в узле положительно влияет на скорость построения октодерева и скорость поиска узлов, но отрицательно влияет на скорость поиска индивидуальных точек.

Таким образом, для выбора наиболее подходящих параметров октодерева можно использовать следующую методику:

1. Установить объем кеша в максимально допустимое значение для используемой компьютерной системы;
2. В случае, если разрешение выборки и пространственного поиска ограничивается размером узла (например, в задачах визуализации, когда проверка на видимость и отправка на отрисовку производится для узлов целиком, не спускаясь до проверки индивидуальных точек), использовать высокий размер узла, ограниченный сверху максимально допустимым объемом точек в используемом алгоритме обработки;
3. В случае, если требуется обеспечивать индивидуальную выборку или поиск отдельных точек рекомендуется использовать низкий размер узла, ограниченный сверху требуемым разрешением октодерева, вплоть до одной точки на узел.

В случае, если требуется соблюсти компромисс между скоростью построения октодерева и скоростью поиска, рекомендуется отдавать предпочтение увеличению скорости поиска, так как данная операция может оказать большее влияние на совокупную производительность вычислений.

4.3. Сравнение с существующими реализациями

Проведем сравнение исследуемой системы с распространенными библиотеками и программами, предназначенными для обработки и визуализации облаков точек. Целью данного сравнения будет доказательство того, что предложенная система позволяет производить обработку больших облаков точек в ограниченном объеме оперативной памяти без существенного ущерба в производительности по сравнению с системами, выполняющими работу полностью в оперативной или внешней памяти. Для сравнения были выбраны следующие программы и библиотеки:

– Point Cloud Library (PCL) - библиотека с открытым исходным кодом, широко использующаяся для обработки облаков точек;
– CloudCompare [94] - программное обеспечение с открытым исходным кодом, предназначенное для загрузки, обработки и визуализации облаков точек.

Сравнение будет производиться путем измерения времени, необходимого на загрузку облака точек и построения октодерева. В качестве параметров октодерева на базе системы кеширования будут использованы размер узла - 2048 точек, объем кеша - 1024 мегабайта. Данные параметры выбраны как наиболее подходящие с точки зрения компромисса между скоростью построения и скоростью поиска. Результаты сравнения приведены в таблице 4.3. В графе потребляемой памяти дополнительно указан процент занимаемой тестовым приложением оперативной памяти от размера исходного облака точек.

Таблица 4.3 – Сравнение скорости построения и потребляемой памяти для различных реализаций октодерева

Метод обработки	**Параметр**	**Облако точек**			
		five.las *0.03 ГБ*	*polytech.las* *1.8 ГБ*	*smolny.las* *4.3 ГБ*	*molodezhnoe.las* *53.4 ГБ*
Октодерево на базе системы кеширования	Время, сек	0.06	4.6	18.6	1075
	Память, ГБ	0.0334 *(111%)*	0.517 *(29%)*	0.586 *(13%)*	0.986 *(1.8%)*
Октодерево на базе отображения памяти	Время, сек	0.13	3.8	10	1043
	Память, ГБ	0.0334 *(5%)*	0.004 *(<1%)*	0.014 *(<1%)*	0.180 *(<1%)*
Point Cloud Library (ОП)	Время, сек	0.122	4.6	12.5	Нехватка памяти
	Память, ГБ	0.0066 *(22%)*	1.8 *(100%)*	4.3 *(100%)*	Нехватка памяти
Point Cloud Library (ВП)	Время, сек	1.5	32.8	152.3	Много файлов
	Память, ГБ	0.0121 *(40%)*	0.131 *(7%)*	0.420 *(10%)*	Много файлов
Cloud Compare	Время, сек	0.435	18.2	58.5	Нехватка памяти
	Память, ГБ	0.114 *(380%)*	1.6 *(88%)*	3.6 (83%)	Нехватка памяти

Из результатов сравнения видно, что потребление памяти у предложенных октодеревьев значительно меньше, чем у рассмотренных аналогов, работающих в оперативной памяти. Скорость построения тоже отличается в лучшую сторону, это может быть вызвано как оптимизациями алгоритма построения, так и модификациями в алгоритме загрузки участков облака точек. Однако при сравнении с аналогами стоит учитывать вероятность влияния сторонних параметров, не поддающихся установке и измерению, а именно различиями в политиках добавления точек, пространственном разрешении октодеревьев, наличии дополнительных процессов, увеличивающих время построения.

Стоит также отметить потребление оперативной памяти для октодерева на базе системы кеширования: при установленном ограничении потребления памяти в 1024 Мбайт, общее потребление памяти поднимается выше установленного лимита. Это обусловлено тем, что лимитируется потребление памяти полезной нагрузкой узлов октодерева. Прочие данные программы, расположенные в оперативной памяти, например, иерархия октодерева, не подлежат ограничению (что рассмотрено в разделе 3.2.3).

4.4. Апробация алгоритма построения растровых проекций при помощи октодерева на базе механизма отображения памяти

Рассмотрим применение октодерева для выполнения построения растровых проекций. Описание подобной системы было приведено в разделе 3.4.

Для произведения проекции выполняется считывание участков облака точек, которые затем преобразуются в цилиндрическую систему координат, в которой ось X соответствует расположению точки вдоль оси проекции, ось Y – углу наклона точки относительно оси цилиндра, а ось Z – расстоянию точки от оси цилиндра. Масштаб осей задается так, чтобы максимальные координаты по осям X и Y соответствовали заданным размерам изображения растровой проекции. Преобразованные точки затем добавляются в октодерево.

Результат построения такого октодерева для облака точек *road.las* приведен на рисунке 4.22.

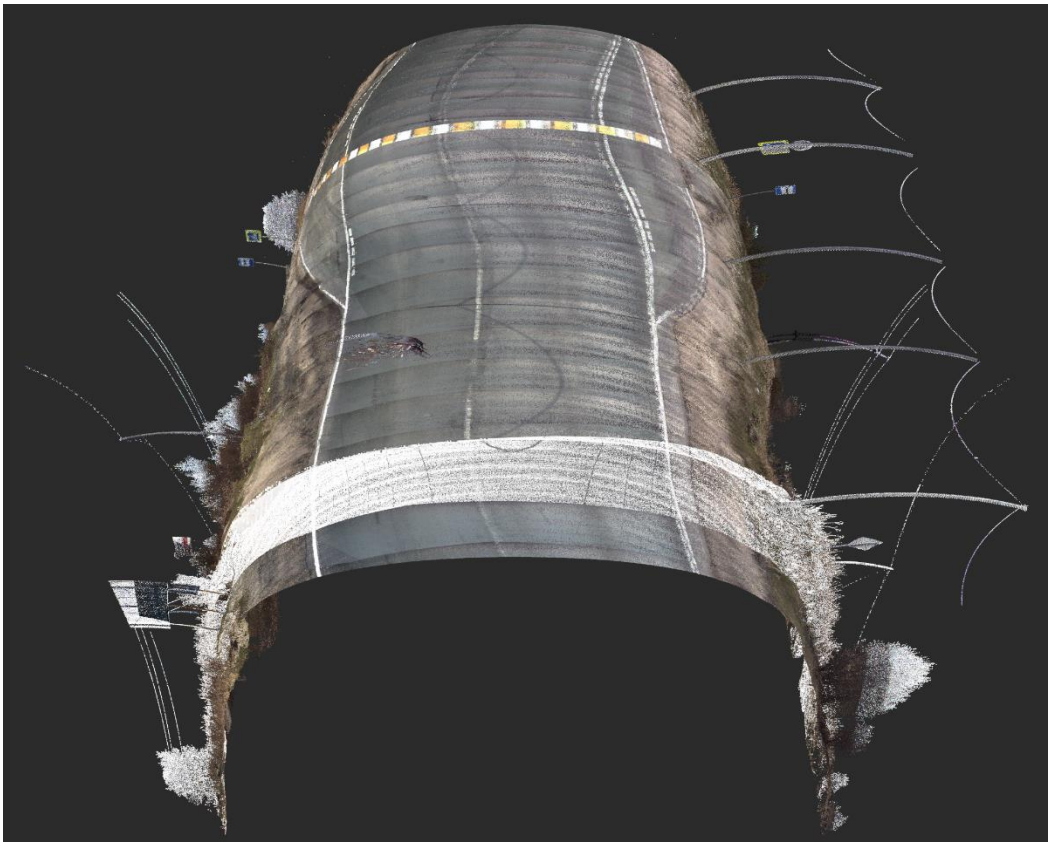

Рисунок 4.22 – Внешний вид октодерева, содержащего спроецированные точки

Далее по построенному октодереву выполняется построение изображения растровой проекции. Результат такой проекции содержит пустоты, которые можно увидеть на рисунке 4.23, возникающие из-за нерегулярной структуры облака точек и того, что точечное представление не содержит информации о топологии поверхности сканируемого объекта.

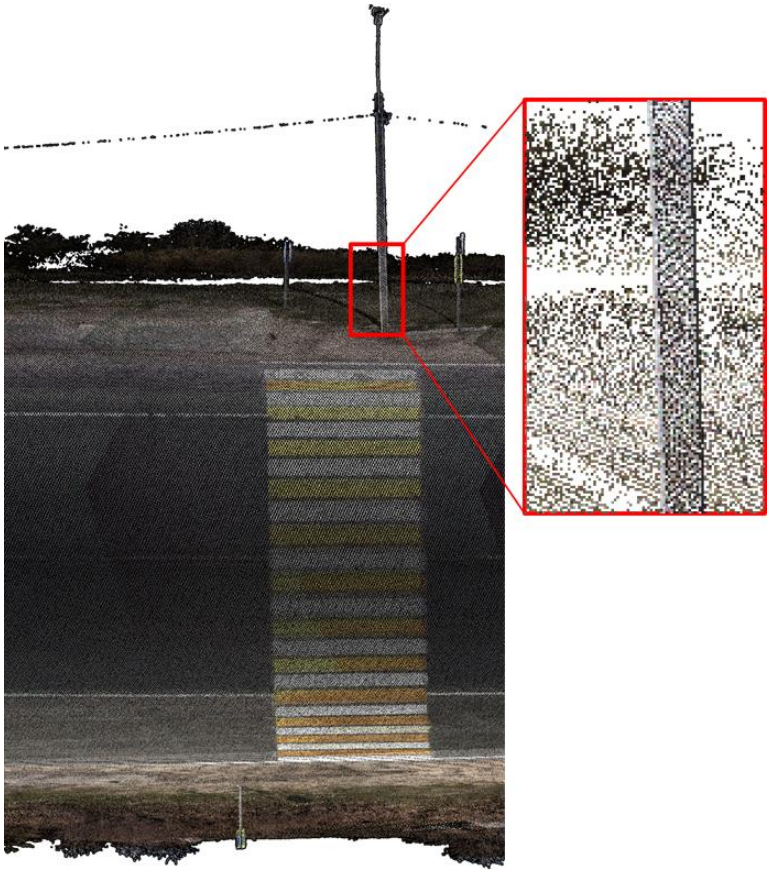

Рисунок 4.23 – Пустоты, возникающие при проекции облака точек

Для устранения пустот в данной задаче была применена триангуляция Делоне [95] с целью переноса вычислений на видеокарту в виде задачи растеризации треугольников. Такое решение было отчасти обусловлено большими размерами изображений (до 32768x32768), участки которых отрисовывались при помощи видеокарты. Также применение этого метода позволило добиться интересных результатов при интерполяции пикселей пустот. В общем случае классический подход с заполнением пикселей заднего плана [96; 97] будет гораздо эффективнее. Результаты применения алгоритма заполнения пустот можно увидеть на рисунке 4.24.

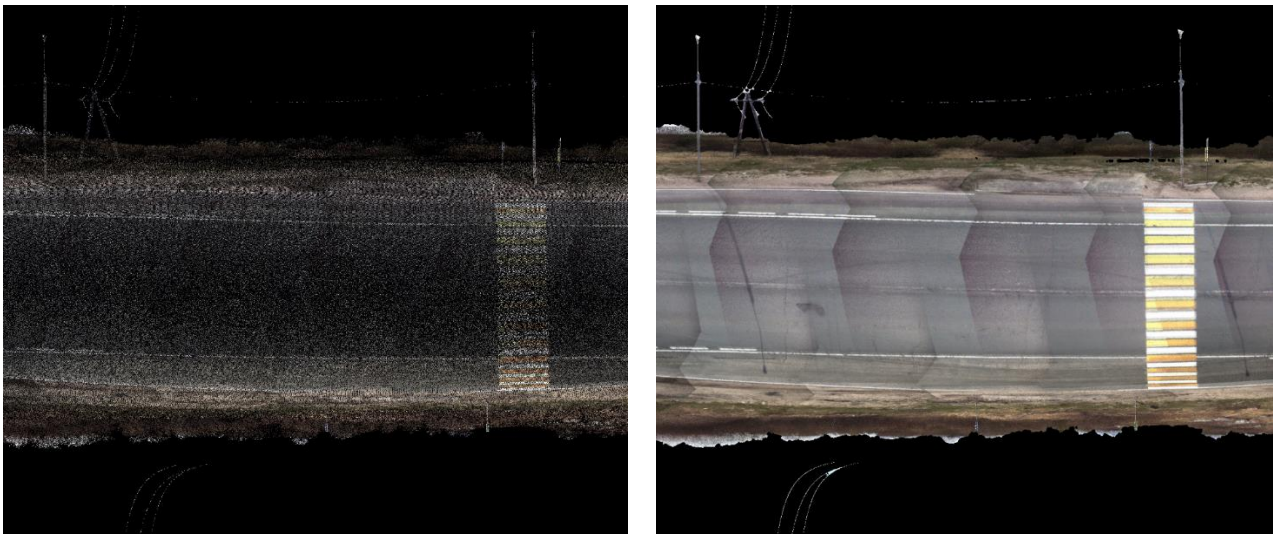

Рисунок 4.24 – Проекция облака точек без заполнения пустот (слева) и с заполнением пустот (справа)

4.5. Экспериментальная оценка системы динамического выделения памяти

Сравнение разработанного аллокатора со стандартным системным аллокатором приводится автором в статье [71]. Для воссоздания ситуации, возникающей при выделении памяти в узлах облака точек, выполним вставку большого количества точек в набор массивов, призванных симулировать узлы октодерева. При выборе массива для вставки будет использоваться нормальное распределение (Рисунок 4.25).

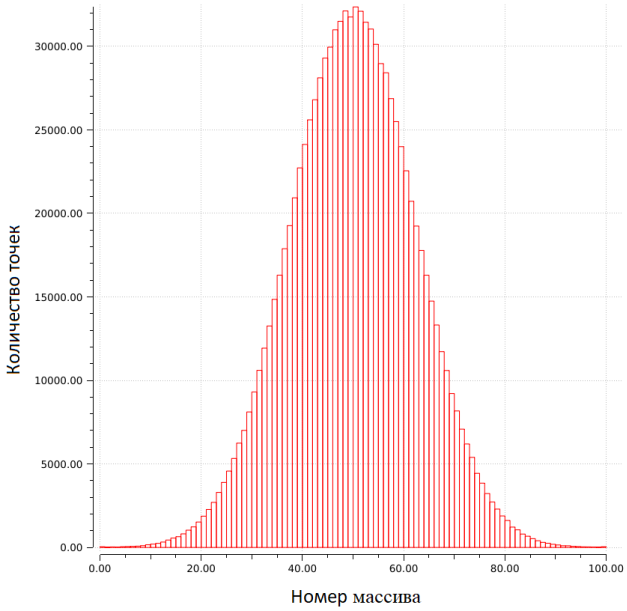

Рисунок 4.25 - Распределение одного миллиона точек по ста массивам

Несмотря на то, что в узле будут использоваться массивы (вектор), для сравнения проведем тот же эксперимент для ассоциативного массива и двусвязного списка. Результаты тестирования приведены в таблице 4.4.

Таблица 4.4 – Результаты тестирования массивов

Кол-во массивов на 1000000 точек	Время заполнения ассоциативного массива, мс		Время заполнения двусвязного списка, мс		Время заполнения векторного массива, мс	
	RAM	HDD	RAM	HDD	RAM	HDD
100	510.8	529.8	39.1	49.6	21.1	23.6
1000	468.2	473.0	42.4	49.9	25.3	29.8
10000	412.9	418.3	49.4	56.4	37.1	47.4

Как видно из результатов тестирования, использование аллокатора на отображаемой памяти работает несколько медленнее (на 10-20%), чем при использовании системного аллокатора. Подобное поведение вызвано работой с файловой памятью вместо оперативной и отсутствием ряда оптимизаций, имеющихся в системном аллокаторе.

Для оценки эффективности использования памяти построим карту занятых и свободных участков в пуле памяти при использовании аллокатора на отображаемой памяти (Рисунок 4.26). Как видно из рисунка, доля пустых участков после обработки не превышает 6.3% и в среднем остается постоянной. Это говорит о том, что механизм объединения пустых участков, а также способ поиска участков для вставки справляются задачей снижения фрагментации памяти.

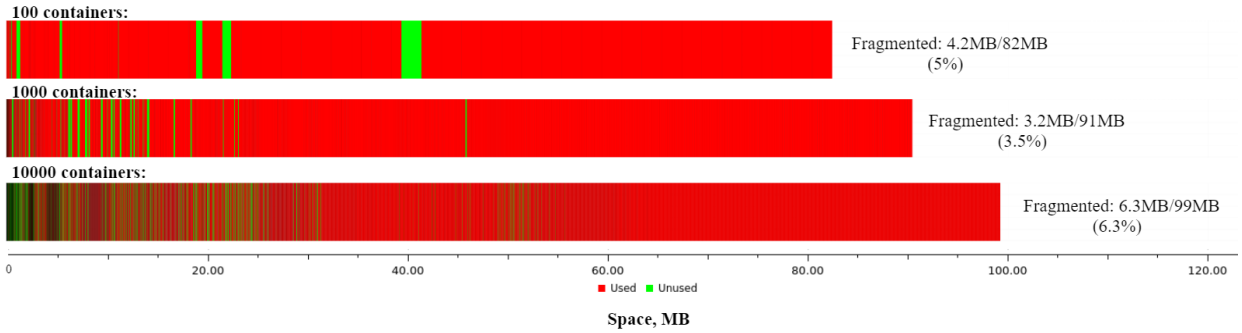

Рисунок 4.26 – Карта памяти после вставки одного миллиона элементов

4.6. Экспериментальная оценка прироста производительности при сокращении числа используемых файлов в процессе работы со вторичной системой хранения

Часто используемым подходом при формировании октодерева во внешней памяти является создание на каждый узел октодерева отдельного файла с данными. Такой подход обусловлен особенностями формирования октодерева, при которых количество точек в узлах нарастает по мере формирования и заранее неизвестно. При обработке больших облаков точек количество создаваемых файлов исчисляется миллионами, что приводит к повышенной нагрузке на файловую систему (ФС). Можно выделить следующие проблемы, возникающие при использовании большого количества файлов:

1. Максимальное количество файловых записей в некоторых ФС ограничено, что может приводить к отказам;
2. Максимальное количество файловых дескрипторов на процесс ограничено:
   - Использование одного файлового дескриптора на процесс потребует частого повторения операции открытия файла, с поиском по пути в ФС (критично для большого количества маленьких файлов);
   - Использование всех доступных файловых дескрипторов потребует их отслеживания;
3. Файловая система более универсальна, за счет чего проигрывает узкоспециализированным решениям с низкоуровневым управлением потребляемой внешней памятью.

При этом первая и вторая проблема могут быть решены путем настройки операционной системы (что предъявляет дополнительные требования к оператору).

Для сравнения производительности при использовании одного или множества файлов, рассмотрим следующую методику тестирования.

4.6.1. Методика оценки

Оценка производительности взаимодействия с вторичной системой хранения будет произведена путем измерения времени, необходимого на запись тестовых данных, выполненную по одному из сценариев. Все вычисления выполняются в однопоточном режиме.

Тестовые данные получены при помощи захвата операций, выполняемых в процессе формирования октодерева для облаков точек *five.las, polytech.las, smolny.las*, и предварительно сохранены в оперативную память. Структурно тестовые данные представлены массивом, каждый элемент которого содержит идентификатор узла, код операции (добавление точек в узел или удаление узла) и данные для записи.

Для сравнения производительности были выбраны следующие сценарии:

1. Использование динамической аллокации на отображаемом файле — позволяет продемонстрировать скорость работы при использовании одного файла;
2. Использование стандартных файловых операций, по одному файлу на узел:
   - Использование одного файлового дескриптора — для каждой файловой операции будет произведено открытие соответствующего файла, выполнение записи и закрытие;
   - Использование всех доступных файловых дескрипторов — при нехватке закрываются все открытые файлы.

Для проведения вычислительных экспериментов будем использовать следующую конфигурацию тестового стенда:

ЦПУ: Intel® Core™ i7 CPU 950 @ 3.07 GHz

ОП: 2 x DIMM DDR4 2400 MHz (3x6 GB)

ВП: Samsung SSD 860 PRO 256 GB

ОС: Ubuntu 18.04

ФС: Ext4 (Fourth Extended Filesystem), NTFS (NT File System), Btrfs (B-tree file system).

4.6.2. Результаты сравнения

Тестовые данные для сравнения были получены при формировании октодеревьев с максимальным размером узла в 2048 точек. Приблизительное распределение точек в узлах октодерева можно увидеть на рисунке 4.27 (кривая упрощена, чтобы исключить выбросы). Из приведенного графика видно, что значительная часть узлов содержит минимум точек.

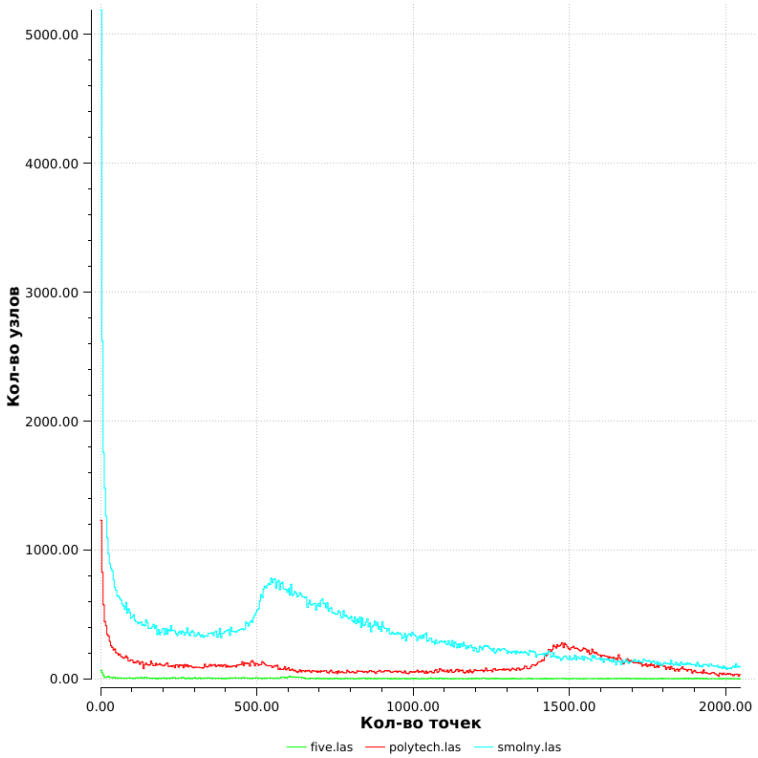

Рисунок 4.27 – Распределение точек в узлах октодерева

Рассмотрим полученные результаты сравнения, приведенные в таблице 4.5. Из них видно, что взаимодействие с внешней памятью при помощи динамической аллокации на отображаемом файле быстрее подходов, использующих множество файлов. Следующим по скорости работы идет подход с использованием всех доступных процессу файловых дескрипторов, затем подход с использованием одного дескриптора. Последний подход существенно медленнее за счет того, что затраты на формирование пути в ФС и открытие файла превышают затраты на запись данных (при большом количестве маленьких файлов и при наличии множества операций добавления данных в один файл).

Из рассмотренных файловых систем лучше всего себя зарекомендовала Ext4. Следующей за ней идет btrfs, затем NTFS. Последняя показала существенно более низкую производительность, чем две остальные, что может быть вызвано как самой организацией ФС, так и (что более вероятно) несовершенством драйвера файловой системы в операционной системе на основе Linux.

Таблица 4.5 – Результаты сравнения сценариев взаимодействия с файловой системой

Облако точек	**Файловая система**	**Время исполнения сценария, сек.**		
		Один файл на октодерево	Один файл на узел, много дескрипторов	Один файл на узел, один дескриптор
five.las	Ext4	**0.042**	0.160	**4.077**
	NTFS	0.073	0.651	53.47
	Btrfs	0.044	**0.154**	5.195
polytech.las	Ext4	**0.603**	**1.788**	**2.802**
	NTFS	8.816	15.43	19.01
	Btrfs	0.717	3.311	7.513
smolny.las	Ext4	**7.092**	**8.905**	**39.17**
	NTFS	36.47	65.10	495.7
	Btrfs	11.44	16.76	53.37

Полученные результаты позволяют сделать вывод, что сокращение количества маленьких файлов при формировании октодерева является оправданной операцией.

4.7. Экспериментальная оценка алгоритма выделения цилиндрических объектов на базе механизма отображения памяти

Для анализа производительности динамической аллокации на отображаемой памяти было произведено экспериментальное исследование алгоритма выделения цилиндрических поверхностей, использующего для расчетов библиотеку PCL. Эксперимент был поставлен путем сравнения скорости выполнения и потребления памяти на различных этапах алгоритма при работе с использованием стандартного системного аллокатора (т.е. в оперативной памяти) и с использованием аллокатора на отображаемой памяти (т.е. в файловой памяти). Более подробно проведенные эксперименты описаны в статье [98].

Измерения производились для следующих этапов алгоритма:

- Загрузка – на данном этапе происходит загрузка облака точек;
- Вокселизация – на данном этапе происходит построение воксельной сетки с целью прореживания исходного облака точек до определенной плотности. По завершении этапа исходное облако точек заменяется прореженным, что приводит к сокращению потребляемой памяти;
- Расчет нормалей, удаление шумов, фильтрация – данные алгоритмы для работы требуют построения дополнительных структур данных, что видно по потребляемой памяти;
- Кластеризация и RANSAC – на данном этапе производится итеративное применение алгоритмов кластеризации и поиска цилиндрических объектов, которые можно увидеть по пикам на графике;
- Фильтрация результатов – фильтрация найденных цилиндрических объектов, объединение сонаправленных сегментов, удаление выбросов.

На рисунке 4.28 приведены графики обработки облака точек *factory.las* в оперативной и во внешней памяти. Стоит отметить, что на графике потребления внешней памяти приведен размер занятых блоков памяти, а общий размер файла равен пиковому потреблению в процессе обработки. Как видно из графиков, скорость обработки при использовании внешней памяти незначительно уменьшилась (147 секунд в оперативной памяти и 156 секунд во внешней памяти).

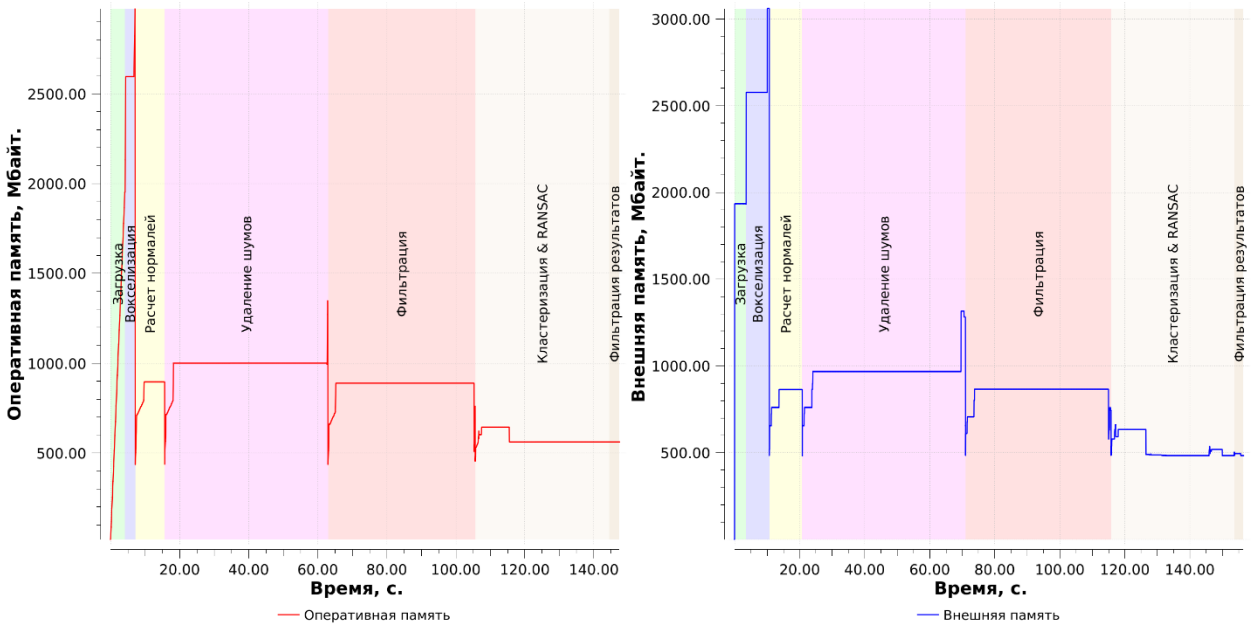

Рисунок 4.28 – Сравнение скорости обработки и потребления памяти при обработке облака точек *factory.las* в оперативной памяти (слева) и во внешней памяти (справа)

Рассмотрим потребление оперативной памяти при обработке во внешней памяти. Как видно из графика на рисунке 4.29, объем потребления оперативной памяти в таком случае существенно меньше, чем при обработке исключительно в оперативной памяти. Оперативная память при обработке выделяется благодаря ограничению на использование аллокации во внешней памяти при размере выделяемого блока больше 1 мегабайта. Пики потребления на графике соответствуют построению и использованию *k-деревьев*, которые при построении совершают множество выделений небольших участков данных.

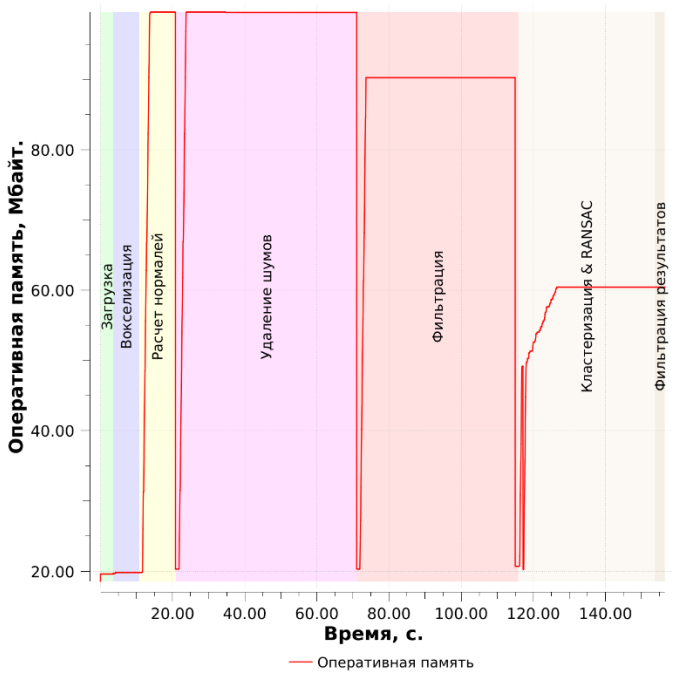

Рисунок 4.29 - Потребление оперативной памяти при обработке облака точек с использованием вторичной системы хранения

Рассмотрим потребление памяти при обработке облаков точек, чей размер превышает доступные объемы оперативной памяти. Для этого произведем обработку облака точек *molodezhnoe.las*, занимающего 53 ГБ (Рисунок 4.30).

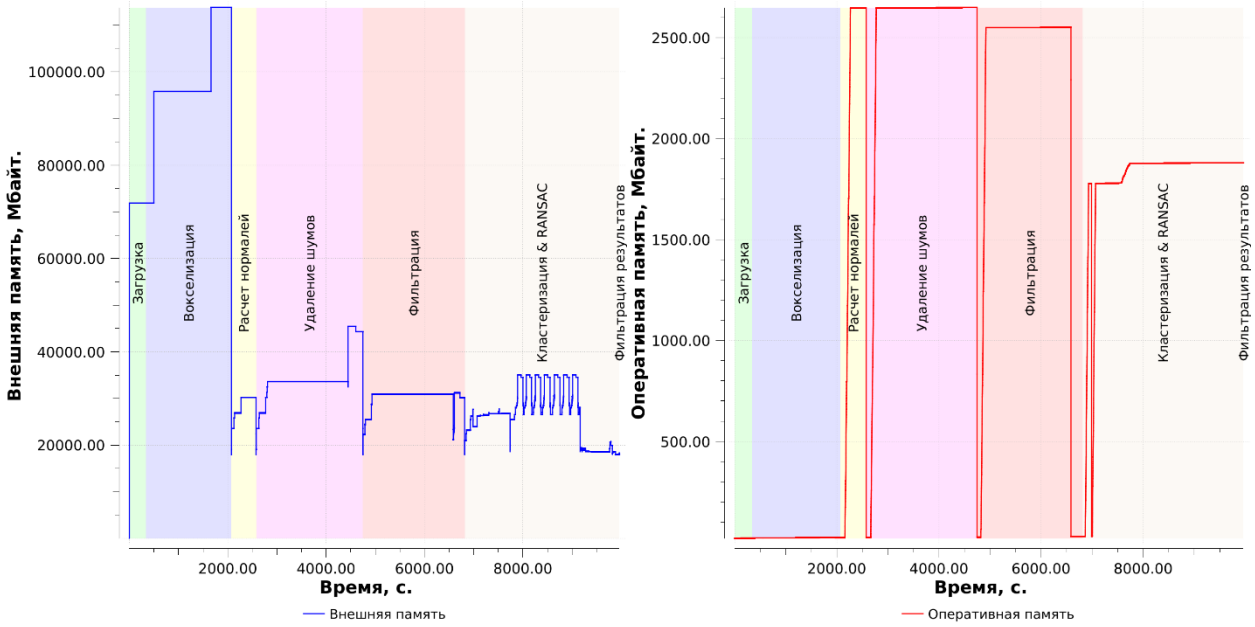

Рисунок 4.30 – Потребление внешней памяти (слева) и оперативной памяти (справа) при обработке облака точек *molodezhnoe.las*

Как видно на приведенных графиках, пиковое потребление файловой памяти (115 Гб) и среднее потребление файловой памяти (~30 Гб) значительно превышает доступные объемы оперативной памяти (16 Гб). Потребление оперативной памяти в процессе обработки (на графике справа), как и в предыдущих экспериментах, обусловлено выделением участков данных меньше мегабайта с использованием системного аллокатора.

В процессе тестирования также были выявлены недостатки подхода с полной подменой системного аллокатора: в процессе работы алгоритма были зафиксированы ошибки при работе с памятью в пользовательском интерфейсе тестового приложения. Для обеспечения стабильной работы потребовалась изоляция процесса вычислений в отдельном потоке (с использованием системного аллокатора в остальных потоках). Разнесение временных промежутков исполнения также помогло справиться с проблемой.

Полученные результаты позволяют сделать вывод, что предложенный метод позволяет выполнять обработку больших облаков точек с использованием алгоритмов, рассчитанных на применение исключительно в оперативной памяти, без существенного снижения производительности. Таким образом, подобный подход может быть использован в программном обеспечении, выполняющем обработку и визуализацию облаков точек как в качестве основной структуры хранения облаков точек, так и в качестве вспомогательного механизма для обеспечения работы в условиях ограничений по оперативной памяти. Однако, в связи с очевидными недостатками полной замены системного аллокатора, рекомендуется применять подход с частичной заменой, доступный при использовании контейнеров стандартной библиотеки C++.

4.8. Выводы

В четвертой главе проведено экспериментальное исследование предложенных методов формирования октодерева, а также основанных на них алгоритмов обработки облаков точек.

1. Проведенные эксперименты показали преимущества предложенных в работе методов перед рассматриваемыми альтернативами, выполняющими обработку в оперативной и внешней памяти, а именно: существенное сокращение потребляемой оперативной памяти (до 99% от размера исходного облака точек), низкое падение производительности в сравнении с альтернативами, выполняющими обработку в ОП (от 0% до 48% времени обработки), увеличение производительность в сравнении с альтернативами, выполняющими обработку в ВП (время обработки до 6.5% от альтернатив).
2. Анализ потребляемой памяти показал, что оба предложенных октодерева позволяют производить обработку облаков точек, чей размер превышает доступные объемы оперативной памяти. В случае использования системы кеширования объем измеряемых данных в оперативной памяти при использовании октодерева не превышает заданного лимита потребления. В случае использования механизма отображения памяти фактическое потребление памяти процессом минимально за счет выполнения процессов кеширования на уровне операционной системы.
3. Установлено, что увеличение объемов кеша положительно влияет на скорость выполнения операций построения и поиска, увеличение максимального количества точек в узле положительно влияет на скорость построения октодерева и скорость поиска узлов, однако отрицательно влияет на скорость поиска индивидуальных точек.
4. На основании анализа оценки влияния параметров октодеревьев на скорость выполнения операций построения и поиска была сформирована методика выбора параметров октодерева для разработки программного обеспечения.
5. Сравнение предложенных октодеревьев с аналогами, работающими в оперативной памяти, показало, что предложенные реализации не только позволяют снизить потребление оперативной памяти, но и не уступают в скорости построения.

6. Проведенное экспериментальное исследование октодерева на базе отображаемой памяти в задаче выполнения растровых проекций подтвердило его работоспособность.
7. Проведенное экспериментальное исследование внедрения алгоритма динамической аллокации в библиотеку обработки облаков точек PCL показало снижение потребляемой оперативной памяти без существенных потерь в производительности.
8. Экспериментальное тестирование алгоритма динамической аллокации позволило продемонстрировать возможность интеграции с контейнерами STL и показало снижение производительности в худшем случае на 21% по сравнению с использованием оперативной памяти. Построение карты занимаемой памяти позволило продемонстрировать максимальный уровень фрагментации в 6.3%.

В данной главе были решены следующие задачи диссертации:

1. Планирование и проведение экспериментальных исследований с целью оценки эффективности предложенных методов для различных задач обработки облака точек и подтверждения правильности выдвинутой гипотезы. Определение показателей, характеризующих вычислительный процесс обработки облака точек при использовании внешней памяти, и разработка вычислительных экспериментов для получения этих показателей и сравнения с существующими реализациями. Разработка методики выбора параметров октодерева в зависимости от целевой направленности обработки. Исследование возможности применения предложенных решений в сторонних программных библиотеках обработки облаков точек, ориентированных на работу в оперативной памяти, и разработка соответствующих практических рекомендаций.

Заключение

В работе решена задача снижения затрат времени на обмен с внешней памятью за счет организации хранения октодерева облака точек в оперативной и внешней памяти и организации управления обработкой информации с учетом особенностей структуры октодерева. При значительном сокращении потребляемой оперативной памяти (до 1% от размера исходного облака точек) обеспечивается незначительное увеличение времени обработки в сравнении с альтернативами, выполняющими обработку в ОП (от 0% до 48%), значительное снижение времени обработки в сравнении с альтернативами, выполняющими обработку в ВП (время обработки до 6.5% от альтернатив). Были выявлены компоненты системы обработки больших облаков точек, определены связи между ними, их функции и цели, механизмы управления ресурсами системы. Был применен объектно-ориентированный подход для создания необходимых абстракций и интерфейсов в процессе реализации. Предложенные алгоритмы, модели и методы, а также их практическая реализация, решают актуальную научно-техническую задачу обработки больших облаков точек при условии наличия ограничений на потребление оперативной памяти и позволяют снизить требования к вычислительным ресурсам системы. В ходе решения данной задачи были получены следующие результаты, составляющие итоги исследования:

1. Рассмотрены проблемы обработки и формирования октодерева, обоснована актуальность и проведен системный анализ проблемы формирования октодерева по облаку точек ЛС при использовании внешней памяти. Сформирована гипотеза об уменьшении затрат времени за счет предлагаемых способов размещения октодерева и организации взаимодействия между оперативной и внешней памятью.

2. Произведена постановка задачи снижения затрат времени на обмен с внешней памятью за счет организации хранения октодерева облака точек в оперативной и внешней памяти и организации управления обработкой информации с учетом особенностей структуры октодерева. Определены критерии эффективности реализации вычислительного процесса обработки, построены концептуальные модели организации обработки облака точек, формирования октодерева, компонентов вычислительного процесса обработки облака точек во внешней памяти. Разработаны модели вычислительного процесса обработки облака точек, анализ которых позволил предложить ряд модификаций, повышающих эффективность процесса обработки.
3. На основании выдвинутых гипотез об организации вычислительного процесса и взаимодействия между оперативной и внешней памятью разработаны методы и алгоритмы решения задачи организации управления вычислительным процессом обработки данных облака точек во внешней памяти, основанные на применении асинхронной системы кеширования и механизма отображения памяти. Разработаны иерархические структуры данных октодерева, включающие реализацию на целочисленной арифметике и на базе целочисленного идентификатора.
4. Разработано программное обеспечение для выполнения обработки облаков точек ЛС с применением предложенных алгоритмов. Предложен способ обработки больших облаков точек путем внедрения системы аллокации на отображаемой памяти в сторонние библиотеки для linux систем. Разработано программное обеспечение для обработки больших облаков точек с использованием сторонней библиотеки совместно с предложенными алгоритмами.

5. Произведенное сравнение потребления оперативной памяти и скорости обработки с альтернативами, использующими оперативную и внешнюю память, показало преимущества предложенных методов. Экспериментально доказано повышение производительности при сокращении количества файлов, сокращения потребляемой памяти при сохранении производительности при внедрении в существующие библиотеки. Разработана методика выбора параметров октодерева, необходимых для разработки программного обеспечения.

Рекомендации и перспективы дальнейшей разработки темы. Совершенствование иерархической структуры октодерева с целью уменьшения ее объема и увеличения производительности поиска, а также обеспечения адаптации для большего количества вариаций входных данных, совершенствование механизмов хранения данных с целью ускорения процедур выборки и сохранения данных. Обеспечение модульности для получения возможности тестирования и замены отдельных компонентов октодерева. Исследование различных политик кеширования. Использование алгоритмов сжатия данных для октодерева на основе системы кеширования.

Соответствие паспорту специальности. Диссертационное исследование соответствует следующим областям (номера соответствуют пунктам в паспорте специальности):

п. 4 – Разработка **методов** и **алгоритмов** решения задач системного анализа, оптимизации, управления, принятия решений и **обработки информации**;

п. 5 – Разработка **специального математического и алгоритмического обеспечения** систем анализа, оптимизации, управления, принятия решений и **обработки информации**;

п. 12 – **Визуализация**, **трансформация** и **анализ** информации на основе **компьютерных методов** обработки информации.

Список сокращений и условных обозначений

ВЛС	Воздушное Лазерное Сканирование
ВП	Внешняя память
ГИС	Геоинформационные системы
ГНСС	Глобальные Навигационные Спутниковые Системы
ЛС	Лазерное Сканирование
МЛС	Мобильное Лазерное Сканирование
НЛС	Наземное Лазерное Сканирование
ОП	Оперативная память
ОС	Операционная система
ПО	Программное Обеспечение
ПТС	Природно-Технические Системы
РИД	Результат интеллектуальной деятельности
ФС	Файловая Система
ЦПУ	Центральное Процессорное Устройство
AABB	Axis-Aligned Bounding Box
ASCII	American Standard Code for Information Interchange
Btrfs	B-tree file system
Ext4	Fourth Extended Filesystem
HDD	Hard Disk Drive
HLT	Hierarchically Layered Tiles
LAS	LASer - Формат хранения облака точек
LAZ	LAS Zip, Compressed Las - Формат хранения облака точек
LIDaR	Laser identification, detection and ranging
LRU	Least Recently Used, Вытеснение давно неиспользуемых
NTFS	NT File System
PCD	Point Cloud Data - Формат хранения облака точек
PCL	Point Cloud Library
RAM	Random Access Memory
RANSAC	Random sample consensus
STL	Standard Template Library - Стандартная библиотека шаблонов
TGPT	Tile Grid Partitioning Tree
XML	Extensible Markup Language

Список литературы

1. *Bohler, W.* Comparison of 3D laser scanning and other 3D measurement techniques / W. Bohler // Recording, Modeling and Visualization of Cultural Heritage. – 2006. – P. 89-99.

2. A review of the use of terrestrial laser scanning application for change detection and deformation monitoring of structures. Vol. 49 / W. Mukupa [et al.]. – 2017.

3. *Wang, K.* Car-based laser scanning system of ancient architecture visual modeling / K. Wang, J. Zhang // Communications in Computer and Information Science. – 2017. – Vol. 698. – P. 251-256.

4. *Кошанулы, К.Е.* Возможности, преимущества и недостатки наземного лазерного сканирования / К.Е. Кошанулы // Интерэкспо ГЕО-Сибирь. – 2017. – Т. 9. – № 1. – С. 27-30.

5. *Позняк, И.И.* Метод оценки колейности автомобильных дорог с использованием мобильного лазерного сканирования / И.И. Позняк, И.Г. Масурадзе, Ш.Г. Масурадзе // Славянский форум. – 2017. – № 2. – С. 127-133.

6. *Pirotti, F.* State of the art of ground and aerial laser scanning technologies for High-resolution topography of the Earth Surface / F. Pirotti, A. Guarnieri, A. Vettore // European Journal of Remote Sensing. – 2013. – Vol. 46. – № 1. – P. 66-78.

7. *Лыткин, С.А.* Методы информационного моделирования при реновации здания: выпускная квалификационная работа бакалавра: 20.03. 02-Природообустройство и водопользование; 20.03. 02_01-Природообустройство / С.А. Лыткин. – 2018.

8. Leica Geosystems AG. HDS7000 [Электронный ресурс]. – URL: http://w3.leica-geosystems.com/downloads123/hds/hds/HDS7000/brochures-datasheet/HDS7000_DAT_en.pdf (дата обращения: 11.08.2019).

9. *The American Society for Photogrammetry & Remote Sensing*. LAS SPECIFICATION VERSION 1.4 – R13 15 July 2013 / The American Society for Photogrammetry & Remote Sensing. – 2013.

10. *Isenburg, M.* LASzip: Lossless compression of lidar data / M. Isenburg // Photogrammetric Engineering and Remote Sensing. – 2013. – Vol. 79. – № 2. – P. 209-217.

11. Point Cloud Library. The PCD (Point Cloud Data) file format [Электронный ресурс]. – URL: http://pointclouds.org/documentation/tutorials/pcd_file_format.php (дата обращения: 21.10.2019).

12. *Rusu, R.B.* 3D is here: Point Cloud Library (PCL) / R.B. Rusu, S. Cousins // 2011 IEEE International Conference on Robotics and Automation. – IEEE, 2011. – P. 1-4.

13. *Huber, D.* The ASTM E57 file format for 3D imaging data exchange / D. Huber // Three-Dimensional Imaging, Interaction, and Measurement. – 2011. – Vol. 7864. – P. 78640A.

14. *Сарычев, Д.С.* Обработка данных лазерного сканирования / Д.С. Сарычев // САПР и ГИС автомобильных дорог. – 2014. – № 1 (2). – С. 16-19.

15. *Sankaranarayanan, J.* A fast all nearest neighbor algorithm for applications involving large point-clouds / J. Sankaranarayanan, H. Samet, A. Varshney // Computers and Graphics (Pergamon). – 2007. – Vol. 31. – № 2. – P. 157-174.

16. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration / J. Elseberg [et al.] // Journal of Software Engineering for Robotics (JOSER). – 2012. – Vol. 3. – № 1. – P. 2-12.

17. *Serpa, Y.R.* A comparative study on a novel drawcall-wise visibility culling and space-partitioning data structures / Y.R. Serpa, Y.R. Serpa, M.A.F. Rodrigues // Proceedings of the XV SBGames. – 2016. – P. 36-43.

18. An out-of-core octree for massive point cloud processing / K. Wenzel [et al.] // Rs.Tudelft.Nl. – 2011. – P. 2011-2014.

19. *Schnabel, R.* A Parallelly Decodeable Compression Scheme for Efficient Point-Cloud Rendering / R. Schnabel, S. Möser, R. Klein // Eurographics Symposium on Point-Based Graphics. – 2007. – P. 119-128.

20. *Bentley, J.L.* Multidimensional Binary Search Trees Used for Associative Searching / J.L. Bentley // Communications of the ACM. – 1975. – Vol. 18. – № 9. – P. 509-517.

21. *Wang, J.* SigVox – A 3D feature matching algorithm for automatic street object recognition in mobile laser scanning point clouds / J. Wang, R. Lindenbergh, M. Menenti // ISPRS Journal of Photogrammetry and Remote Sensing. – 2017. – Vol. 128. – P. 111-129.

22. Object Detection in Point Clouds Using Conformal Geometric Algebra / A. Sveier [et al.] // Advances in Applied Clifford Algebras. – 2017. – Vol. 27. – № 3. – P. 1961-1976.

23. *Wang, L.* A Voxel-Based 3D Building Detection Algorithm for Airborne LIDAR Point Clouds / L. Wang, Y. Xu, Y. Li // Journal of the Indian Society of Remote Sensing. – 2019. – Vol. 47. – № 2. – P. 349-358.

24. Segmentation of Individual Trees from TLS and MLS Data / L. Zhong [et al.] // IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing. – 2017. – Vol. 10. – № 2. – P. 774-787.

25. Voxel-based segmentation of 3D point clouds from construction sites using a probabilistic connectivity model / Y. Xu [et al.] // Pattern Recognition Letters. – 2018. – Vol. 102. – P. 67-74.

26. Geometric Primitive Extraction from Point Clouds of Construction Sites Using VGS / Y. Xu [et al.] // IEEE Geoscience and Remote Sensing Letters. – 2017. – Vol. 14. – № 3. – P. 424-428.

27. *Tang, Y.* Multi-scale surface reconstruction based on a curvature-adaptive signed distance field / Y. Tang, J. Feng // Computers and Graphics (Pergamon). – 2018. – Vol. 70. – P. 28-38.

28. Robust Normal Estimation for 3D LiDAR Point Clouds in Urban Environments / R. Zhao [et al.] // Sensors. – 2019. – Vol. 19. – № 5. – P. 1248.

29. *Беляевский, К.О.* Формирование октодерева по облаку точек при ограничении объема оперативной памяти / К.О. Беляевский // НТВ СПбГПУ. Информатика. Телекоммуникации. Управление. – 2019. – Т. 12. – № 4. – С. 97-110.

30. *Futterlieb, J.* [FTB16] Smooth Visualization of Large Point Clouds / J. Futterlieb, C. Teutsch, D. Berndt // IADIS International Journal on Computer Science and Information Systems. – 2016. – Vol. 11. – № 2. – P. 146-158.

31. *Elseberg, J.* Efficient processing of large 3D point clouds / J. Elseberg, D. Borrmann, A. Nuchter // 2011 XXIII International Symposium on Information, Communication and Automation Technologies. – IEEE, 2011. – P. 1-7.

32. *Pajarola, R.* Stream-processing points / R. Pajarola // Proceedings of the IEEE Visualization Conference. – 2005. – P. 31.

33. *Boesch, J.* Flexible configurable stream processing of point data / J. Boesch, R. Pajarola // 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, WSCG'2009 - In Co-operation with EUROGRAPHICS, Full Papers Proceedings. – 2009. – P. 49-56.

34. *Deibe, D.* Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures / D. Deibe, M. Amor, R. Doallo // International Journal of Geographical Information Science. – 2019. – Vol. 33. – № 3. – P. 593-617.

35. *Richter, R.* Out-of-core visualization of classified 3D point clouds / R. Richter, S. Discher, J. Döllner // Lecture Notes in Geoinformation and Cartography. – 2015. – Vol. PartF3. – P. 227-242.

36. Processing and interactive editing of huge point clouds from 3D scanners / M. Wand [et al.] // Computers and Graphics (Pergamon). – 2008. – Vol. 32. – № 2. – P. 204-220.

37. *Puzak, T.R.* Analysis of cache replacement-algorithms / T.R. Puzak // Doctoral Dissertations. – 1985.

38. External Memory Management and Simplification of Huge Meshes / P. Cignoni [et al.] // IEEE Transactions on Visualization and Computer Graphics. – 2003. – Vol. 9. – № 4. – P. 525-537.

39. *Scheiblauer, C.* Interactions with Gigantic Point Clouds / C. Scheiblauer. – 2014.

40. *Han, S.* Towards Efficient Implementation of an Octree for a Large 3D Point Cloud / S. Han // Sensors. – 2018. – Vol. 18. – № 12. – P. 4398.

41. *Pajarola, R.* XSplat: External memory multiresolution point visualization / R. Pajarola, M. Sainz, R. Lario // Proceedings of the 5th IASTED International Conference on Visualization, Imaging, and Image Processing, VIIP 2005. – 2005. – P. 628-633.

42. *Boyko, A.* Extracting roads from dense point clouds in large scale urban environment / A. Boyko, T. Funkhouser // ISPRS Journal of Photogrammetry and Remote Sensing. – 2011. – Vol. 66. – № 6. – P. S2-S12.

43. *Cao, M.* Ext4: The Next Generation of Ext2/3 Filesystem. / M. Cao, S. Bhattacharya, T. Ts'o // LSF. – 2007.

44. *Мараховский, В.Б.* Моделирование параллельных процессов. Сети Петри. Курс для системных архитекторов, программистов, системных аналитиков, проектировщиков сложных систем управления / В.Б. Мараховский, Л.Я. Розенблюм, А.В. Яковлев // Санкт-Петербург: Профессиональная литература, АйТи-Подготовка. – 2014.

45. *Простителенко, О.В.* Моделирование дискретных систем на основе сетей Петри: учебное пособие / О.В. Простителенко, В.И. Халимон, А.Ю. Рогов. – Санкт-Петербург: СПбГТИ(ТУ), 2017. – 69 с.

46. *Беляевский, К.О.* Использование целочисленной арифметики для формирования октодерева / К.О. Беляевский, М.В. Болсуновская // Неделя науки СПбПУ. – 2019. – Т. 1. – С. 123-125.

47. Scan Line Based Road Marking Extraction from Mobile LiDAR Point Clouds / L. Yan [et al.] // Sensors. – 2016. – Vol. 16. – № 6. – P. 903.

48. Laser-Based Online Sliding-Window Approach for UAV Loop-Closure Detection in Urban Environments / A. Wang [et al.] // International Journal of Advanced Robotic Systems. – 2016. – Vol. 13. – № 2. – P. 61.

49. *Boulch, A.* Fast and robust normal estimation for point clouds with sharp features / A. Boulch, R. Marlet // Eurographics Symposium on Geometry Processing. – 2012. – Vol. 31. – № 5. – P. 1765-1774.

50. *Remondino, F.* Da nuvem de pontos à superfície: o problema de modelagem e visualização - From point cloud to surface: the modeling and visualization problem / F. Remondino // International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences. – 2003. – Vol. XXXIV. – P. 24-28.

51. A new segmentation method for point cloud data / H. Woo [et al.] // International Journal of Machine Tools and Manufacture. – 2002. – Vol. 42. – № 2. – P. 167-178.

52. *Microsoft*. STL Containers [Электронный ресурс]. – URL: http://msdn.microsoft.com/en-us/library/vstudio/1fe2x6kt.aspx (дата обращения: 15.10.2019).

53. *Behley, J.* Efficient radius neighbor search in three-dimensional point clouds / J. Behley, V. Steinhage, A.B. Cremers // Proceedings - IEEE International Conference on Robotics and Automation. – 2015. – Vols. 2015-June. – P. 3625-3630.

54. Discrete point cloud filtering and searching based on VGSO algorithm / F. Hu [et al.] // Proceedings - 27th European Conference on Modelling and Simulation, ECMS 2013. – 2013. – P. 850-856.

55. Octree-based region growing for point cloud segmentation / A.V. Vo [et al.] // ISPRS Journal of Photogrammetry and Remote Sensing. – 2015. – Vol. 104. – P. 88-100.

56. *Kuhn, A.* Incremental Division of Very Large Point Clouds for Scalable 3D Surface Reconstruction / A. Kuhn, H. Mayer // Proceedings of the IEEE International Conference on Computer Vision. – 2015. – Vols. 2015-Febru. – P. 157-165.

57. *Shagam, J.* Dynamic irregular octrees / J. Shagam, J. Pfeiffer Jr // Most. – 2003.

58. *Bédorf, J.* A sparse octree gravitational N-body code that runs entirely on the GPU processor / J. Bédorf, E. Gaburov, S. Portegies Zwart // Journal of Computational Physics. – 2012. – Vol. 231. – № 7. – P. 2825-2839.

59. *RezaAbbasifard, M.* A Survey on Nearest Neighbor Search Methods / M. RezaAbbasifard, B. Ghahremani, H. Naderi // International Journal of Computer Applications. – 2014. – Vol. 95. – № 25. – P. 39-52.

60. *Drost, B.H.* Almost constant-time 3D nearest-neighbor lookup using implicit octrees / B.H. Drost, S. Ilic // Machine Vision and Applications. – 2018. – Vol. 29. – № 2. – P. 299-311.

61. *Goldberg, D.* What every computer scientist should know about floating-point arithmetic / D. Goldberg // ACM Computing Surveys (CSUR). – 1991. – Vol. 23. – № 1. – P. 5-48.

62. *Museth, K.* VDB / K. Museth // ACM Transactions on Graphics. – 2013. – Vol. 32. – № 3. – P. 1-22.

63. *Koenig, A.* Associative arrays in C + + / A. Koenig // Proc. USENIX Conference. – 2016. – P. 1-12.

64. *Valois, J.D.* Implementing Lock-Free Queues / J.D. Valois // Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems. – 1994. – P. 64-69.

65. *Lindstrom, P.* Terrain simplification simplified: A general framework for view-dependent out-of-core visualization / P. Lindstrom, V. Pascucci // IEEE Transactions on Visualization and Computer Graphics. – 2002. – Vol. 8. – P. 239-254.

66. *Wald, I.* An interactive out-of-core rendering framework for visualizing massively complex models / I. Wald, A. Dietrich, P. Slusallek // ACM SIGGRAPH 2005 Courses on - SIGGRAPH ’05. – New York, New York, USA: ACM Press, 2005. – P. 81-92.

67. *Wald, I.* Interactive Distributed Ray Tracing of Highly Complex Models / I. Wald, P. Slusallek, C. Benthin. – 2001. – P. 277-288.

68. Dynamic storage allocation: A survey and critical review / P.R. Wilson [et al.] // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). – 1995. – Vol. 986. – P. 1-116.

69. NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines / C. Wang [et al.] // Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012. – 2012. – P. 957-968.

70. Multithreading in Laser Scanning Data Processing / V. Badenko [et al.] // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). – 2019. – Vol. 11619 LNCS. – P. 289-305.

71. *Беляевский, К.О.* Использование механизма отображения памяти при формировании октодерева облака точек / К.О. Беляевский, М.В. Болсуновская // Неделя науки СПбПУ. – 2019. – Т. 1. – С. 126-128.

72. *Hug, C.* Advanced lidar data processing with LasTools / C. Hug, P. Krzystek, W. Fuchs // International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives. – 2004. – Vol. 35.

73. PDAL: Point cloud Data Abstraction Library Release 2.0.1 [Электронный ресурс]. – URL: https://pdal.io/PDAL.pdf (дата обращения: 01.10.2019).

74. *Musser, D.R.* STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library / D.R. Musser, G.J. Derge, A. Saini. – Addison-Wesley Professional, 2001.

75. *Josuttis, N.M.* The C++ STL A Tutorial and Reference. Vol. 53 / N.M. Josuttis. – Addison-Wesley, 2013. – 1689-1699 p.

76. unordered_map [Электронный ресурс]. – URL: https://en.cppreference.com/w/cpp/container/unordered_map (дата обращения: 01.10.2019).

77. rigtorp/SPSCQueue [Электронный ресурс]. – URL: https://github.com/rigtorp/SPSCQueue (дата обращения: 13.05.2019).

78. *Boehm, H.* Threads and Shared Variables in C++ / H. Boehm. – 2011. – P. 55-77.

79. *Majumdar, S.* Parallel Quick Sort using Thread Pool Pattern / S. Majumdar, I. Jain, A. Gawade // International Journal of Computer Applications. – 2016. – Vol. 136. – № 7. – P. 36-41.

80. *Liu, T.* Sheriff : Detecting and Eliminating False Sharing / T. Liu, E.D. Berger // Science. – 2010. – P. 1-11.

81. *Meagher, D.* Geometric modeling using octree encoding / D. Meagher // Computer Graphics and Image Processing. – 1982. – Vol. 19. – № 2. – P. 129-147.

82. mmap - Linux Programmer’s Manual [Электронный ресурс]. – URL: http://man7.org/linux/man-pages/man2/mmap.2.html (дата обращения: 15.10.2019).

83. Vector - Cppreference.Com [Электронный ресурс]. – URL: https://en.cppreference.com/w/cpp/container/vector (дата обращения: 15.10.2019).

84. Allocator - Cppreference.Com [Электронный ресурс]. – URL: https://en.cppreference.com/w/cpp/memory/allocator (дата обращения: 15.10.2019).

85. *Hill, S.* a Simple Fast Memory Allocator / S. Hill // Graphics Gems III (IBM Version). – 1992. – P. 49-50.

86. *Tofte, M.* A Theory of Stack Allocation in Polymorphically Typed Languages / M. Tofte, J. Talpin. – 1993. – P. 1-72.

87. *Zhao, Q.* Dynamic memory optimization using pool allocation and prefetching / Q. Zhao, R. Rabbah, W.-F. Wong // ACM SIGARCH Computer Architecture News. – 2005. – Vol. 33. – № 5. – P. 27.

88. brk (2) - Linux Programmer’s Manual [Электронный ресурс]. – URL: http://man7.org/linux/man-pages/man2/brk.2.html (дата обращения: 15.10.2019).

89. *Kamp, P.-H.* Malloc (3) revisited. / P.-H. Kamp // USENIX Annual Technical Conference. – 1998. – P. 45.

90. *Беляевский, К.О.* Применение динамической аллокации на отображаемой памяти для обработки больших облаков точек в библиотеке PCL / К.О. Беляевский // Известия Самарского научного центра Российской академии наук. – 2020. – Т. 22. – № 1. – С. 56-64.

91. Point cloud data filtering and downsampling using growing neural gas / S. Orts-Escolano [et al.] // The 2013 International Joint Conference on Neural Networks (IJCNN). – IEEE, 2013. – P. 1-8.

92. *Дегтярева, А.* Line fitting, или методы аппроксимации набора точек прямой / А. Дегтярева, В. Вежневец // Компьютерная графика и мультимедиа. Выпуск. – 2003. – № 1. – С. 3.

93. MALLOC_HOOK(3) - Linux Programmer’s Manual [Электронный ресурс]. – URL: http://man7.org/linux/man-pages/man3/malloc_hook.3.html (дата обращения: 15.10.2019).

94. *Girardeau-Montaut, D.* CloudCompare: 3D point cloud and mesh processing software / D. Girardeau-Montaut // Webpage: http://www.cloudcompare.org. – 2015.

95. *Paul Chew, L.* Constrained delaunay triangulations / L. Paul Chew // Algorithmica. – 1989. – Vol. 4. – № 1-4. – P. 97-108.

96. *Rosenthal, P.* Image-space point cloud rendering / P. Rosenthal, L. Linsen // Proceedings of Computer …. – 2008. – № May 2017. – P. 1-8.

97. *Botsch, M.* Efficient high quality rendering of point sampled geometry / M. Botsch, A. Wiratanaya, L. Kobbelt // Eurographics Workshop on Rendering. – 2002. – P. 53-64.

98. *Беляевский, К.О.* Применение динамической аллокации на отображаемой памяти для обработки больших облаков точек в библиотеке PCL / К.О. Беляевский // Известия Самарского научного центра Российской академии наук. – 2020. – Т. 22. – № 1.

Приложение А. Акты внедрения

Ниже приводятся копии актов о внедрении результатов диссертационного исследования.

МИНОБРНАУКИ РОССИИ
федеральное государственное автономное
образовательное учреждение высшего образования
«Санкт-Петербургский политехнический
университет Петра Великого»
(ФГАОУ ВО «СПбПУ»)

ИНН 7804040077, ОГРН 1027802505279,
ОКПО 02068574
Политехническая ул., 29, Санкт-Петербург, 195251
тел.: +7(812)297 2095, факс: +7(812)552 6080
office@spbstu.ru

УТВЕРЖДАЮ
Проректор по научной работе
В.В. Сергеев
«07» 09 2020 г.

Акт реализации результатов диссертационной работы Беляевского Кирилла Олеговича, выполненной на тему «Методы и алгоритмы формирования и использования октодерева для обработки облака точек лазерного сканирования в ограниченном объеме оперативной памяти» и подготовленной для соискания ученой степени кандидата технических наук по специальности 05.13.01 «Системный анализ, управление и обработка информации (технические системы)»

Настоящий акт составлен о том, что следующие результаты диссертационной работы Беляевского К.О.:

1. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе асинхронной двухуровневой системы кеширования.
2. Метод и алгоритм предобработки информации облака точек лазерного сканирования, заключающийся в ее структурировании путем формирования октодерева на базе механизма отображения памяти.
3. Алгоритм динамической аллокации на отображаемой памяти.
4. Способ обработки больших облаков точек путем интеграции системы аллокации отображаемой памяти в сторонние библиотеки для linux систем

внедрены в следующем научном проекте, выполняемом в СПбПУ:

- ФЦП, 14.584.21.0025, «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014 - 2020 годы" по теме: «Исследование и разработка алгоритмов и программных средств по обработке, хранению и визуализации данных лазерного сканирования и фотосъемки» (Уникальный идентификатор проекта RFMEFI58417X0025)

Полученные Беляевским К.О. результаты используются в качестве алгоритмического и методического обеспечения для разработки программного комплекса, предназначенного для обработки, хранения и визуализации данных лазерного сканирования. Разработанный с использованием результатов диссертационной работы Беляевского К.О. программный комплекс позволяет кратно снизить потребление оперативной памяти при выполнении типовых операций обработки облаков точек, сохраняя при этом приемлемую производительность.

Научный руководитель работ _____ Баденко В.Л.

Общество с ограниченной ответственностью
“ЭкоСкан”
190121, г. Санкт-Петербург, наб. реки Пряжки, д. 32
Тел. +7 (812) 702-6178, моб. +7 (921) 998-7157

ЭКОСКАН

18.02.2020 г. г. Санкт-Петербург

Акт внедрения

Настоящий акт подтверждает, что основной результат диссертационной работы Беляевского К.О. на тему «Методы и алгоритмы формирования и использования октодерева для обработки облака точек лазерного сканирования в ограниченном объеме оперативной памяти», заключается в новом подходе к организации обработки больших облаков точек. Этот подход основывается на использовании октодерева совместно с системой кеширования и механизмом отображения памяти.

Данное решение апробировано компанией ООО «ЭкоСкан» в программном комплексе для обработки, хранения и визуализации данных лазерного сканирования и фотосъемки. Практическим результатом диссертационной работы является использование предложенного подхода для снижения загрузки оперативной памяти в процессе работы с облаками точек. Это позволяет выполнять обработку облаков точек лазерных отражений, размер которых значительно превышает доступные объемы оперативной памяти.

Директор
ООО “Экоскан”, к.т.н. Науменко А.И.

Технический директор
ООО “Экоскан”, к.т.н. Виноградов К.П.

Общество с ограниченной ответственностью "ЭкоСкан" Для документов Санкт-Петербург