

Санкт-Петербургский государственный университет

Мищенко Станислав Алексеевич

Выпускная квалификационная работа

Оптимизация работы сцены в приложении SOLVE

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование
информационных систем»*

Основная образовательная программа *СВ.5162.2020 «Технологии программирования»*

Научный руководитель:
старший преподаватель кафедры системного программирования, к.т.н Ю. В. Литвинов

Консультант:
ведущий программист-разработчик, «ВКонтакте» Д. С. Ярош

Рецензент:
младший научный сотрудник, Институт искусственного интеллекта С. Галочкин

Санкт-Петербург
2024

Saint Petersburg State University

Stanislav Mishchenko

Bachelor's Thesis

Scene optimization in SOLVE application

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2020 "Programming Technologies"*

Scientific supervisor:
C.Sc., senior lecturer Y.V. Litvinov

Consultant:
Lead developer, VK D.S. Jarosh

Reviewer:
Research Scientist, AIRI S. Galochkin

Saint Petersburg
2024

Оглавление

Введение	4
1. Постановка задачи	6
2. Обзор	7
2.1. Обзор приложения	7
2.2. Компонент сцены	7
2.3. Решения для рендеринга	10
3. Реализация системы рендеринга	18
4. Сравнение производительности и апробация	23
4.1. Апробация	24
5. Заключение	25
Список литературы	26

Введение

В настоящее время всё большее применение находят автономные робототехнические системы, используемые для выполнения различных задач на местности. При разработке такого типа устройств значительное внимание уделяется задаче локализации (определения положения робота в пространстве). Популярным методом решения данной задачи является SLAM (simultaneous localization and mapping) [19] — одновременная локализация и построение карты. Обычно SLAM-системы состоят из двух компонентов: фронтенда и бэкенда. Первый включает в себя распознавание ориентиров на кадрах (детекция) и их сопоставление на разных снимках (ассоциация), после чего второй компонент обрабатывает полученные данные для построения карты местности и локализации на ней робототехнического устройства.

Качество работы алгоритмов детекции и ассоциации фронтенда во многом определяет конечный результат работы всего SLAM алгоритма и, в связи с этим, часто возникает потребность в их отладке. На данный момент основным методом для выявления ошибок в рассматриваемых системах является визуализация ориентиров и ассоциаций в реальном времени.

Существуют SLAM-фреймворки, которые предоставляют такую возможность, но, как правило, визуализация в них является неотделимой от остальной реализации частью [3, 16], что создаёт трудности при отладке отдельных компонентов фронтенда. С целью решения данной проблемы в Лаборатории мобильной робототехники [15] ведётся разработка приложения SOLVE, интерфейс которого изображён на рисунке 1, предназначенного для визуализации ориентиров в SLAM-алгоритмах. Основным компонентом инструмента является сцена, отвечающая за отображение кадров с нанесёнными на них ориентирами. В качестве технологического стека используется язык программирования Kotlin, фреймворк TornadoFX и библиотека JavaFX. Поскольку наборы данных, использующиеся в рассматриваемой области, могут состоять из сотен или даже тысяч кадров, каждый из которых может содержать

несколько слоёв с ориентирами, к производительности компонента сцены предъявляются повышенные требования.

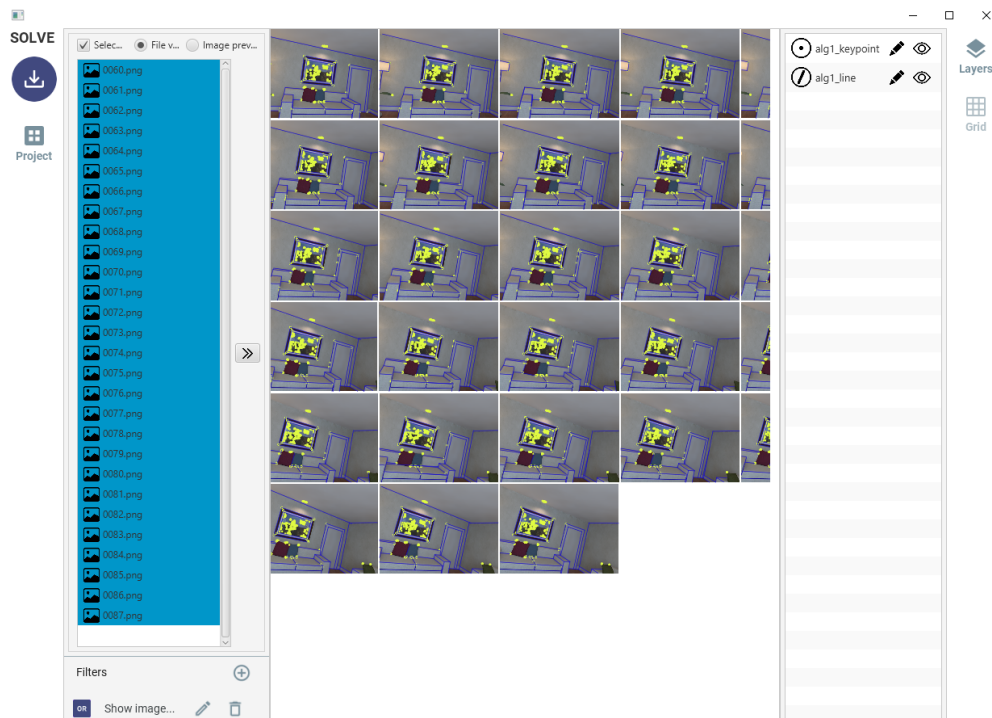


Рис. 1: Внешний вид приложения

Приложение SOLVE активно используется командой Лаборатории мобильной робототехники для оценки качества работы SLAM-алгоритмов, а также в качестве инструмента для визуализации ориентиров и ассоциаций. Текущее решение рассчитано на работу с наборами данных, содержащими небольшое количество кадров, в то время как на реальных наборах данных наблюдаются проблемы с производительностью компонента сцены и, как следствие, всего приложения в целом. Помимо этого, даже на небольших проектах критически возрастает объём потребляемой оперативной памяти, необходимой для корректной работы инструмента. Это является существенной проблемой, поскольку инструмент может быть использован одновременно с другими приложениями, необходимыми для работы со SLAM-алгоритмами. Проведённая апробация показала, что описанные проблемы являются существенными и негативно сказываются на пользовательском опыте.

Данная работа посвящена оптимизации работы сцены для решения вышеупомянутых проблем.

1 Постановка задачи

Целью данной работы является оптимизация работы сцены в приложении SOLVE. Для её достижения были поставлены следующие задачи.

1. Выполнить обзор разрабатываемого приложения и используемых в нем технологий.
2. Выполнить обзор решений для рендеринга в рамках JavaFX-приложения.
3. Разработать систему для отрисовки кадров, ориентиров и ассоциаций с применением выбранного решения.
4. Разработать систему для управления сценой в рамках новой реализации.
5. Выполнить обзор методов измерения производительности сцены и сравнить реализации.
6. Провести апробацию приложения с оптимизированной сценой.

2 Обзор

В рамках текущего раздела был проведён обзор приложения, его компонентов и используемых технологий.

2.1 Обзор приложения

Приложение состоит из нескольких основных компонентов: модуля импорта проектов, сцены визуализации, каталога для формирования выборки кадров и панели настроек визуализации.

Модуль импорта проектов позволяет загружать в инструмент кадры и файлы, содержащие информацию об ориентирах. После того как проект был импортирован, все кадры становятся видны в модуле каталога в виде списка. В нём пользователь может формировать выборки из кадров, которые затем будут переданы для визуализации компоненту сцены. Последний ответственен за отрисовку кадров с размеченными на них ориентирами трёх типов: точками, линиями и плоскостями. Импортированный проект может содержать несколько наложенных друг на друга слоёв с ориентирами одного типа. Компонент настроек визуализации сцены позволяет изменять параметры отображения сцены, такие как размер сетки кадров или цвет и размер ориентиров.

В разработке приложения используется язык программирования Kotlin [12] и фреймворк TornadoFX [21], созданный как расширение для библиотеки JavaFX [6], которая позволяет создавать кроссплатформенные приложения для desktop-платформ. В процессе реализации инструмента применялся подход MVC (model-view-controller) [4].

2.2 Компонент сцены

При разработке приложения к модулю сцены был предъявлен ряд требований, затрагивавших не только функциональность компонента, но и эффективность его работы.

2.2.1 Требования к компоненту

Компонент должен уметь отображать сетку из кадров с размеченными на них ориентирами точек, линий и плоскостей. Пользователю также должен предоставляться интерфейс для перемещения по сцене в разных направлениях и изменения масштаба отображаемых кадров.

Поскольку импортируемый проект может содержать большое количество кадров, важно, чтобы компонент обладал высокой производительностью. Помимо этого, необходимо обеспечить виртуализацию на уровне данных для отображаемых кадров, поскольку размер некоторых проектов может превышать объём оперативной памяти, доступной на устройстве.

Подразумевается, что приложение будет использоваться параллельно с другими программами в качестве вспомогательного инструмента для работы со SLAM-алгоритмами, а значит в процессе работы с ним ресурсы системы будут использоваться другими процессами. По этой причине разрабатываемое приложение не должно требовать больших вычислительных мощностей устройства и не должно занимать большого количества оперативной памяти. Помимо этого, пользователи могут обладать разными операционными системами, поэтому разрабатываемое приложение должно быть кроссплатформенным.

2.2.2 Архитектура предыдущей реализации

Диаграмма классов для предыдущей реализации системы рендеринга сцены была взята из работы [22], посвящённой реализации первой версии компонента сцены, и представлена на рисунке 2.

Компонент **FrameView** отвечает за визуализацию одной клетки в сетке кадров сцены. Экземпляры данного класса поддерживают виртуализацию и могут получать новые данные в процессе навигации по сцене, меняя своё отображение.

Для отрисовки кадров и ориентиров плоскостей используется класс **FrameDrawer**, предоставляющий интерфейс для рисования на уровне отдельных пикселей. Для этих целей используются стандартные ком-

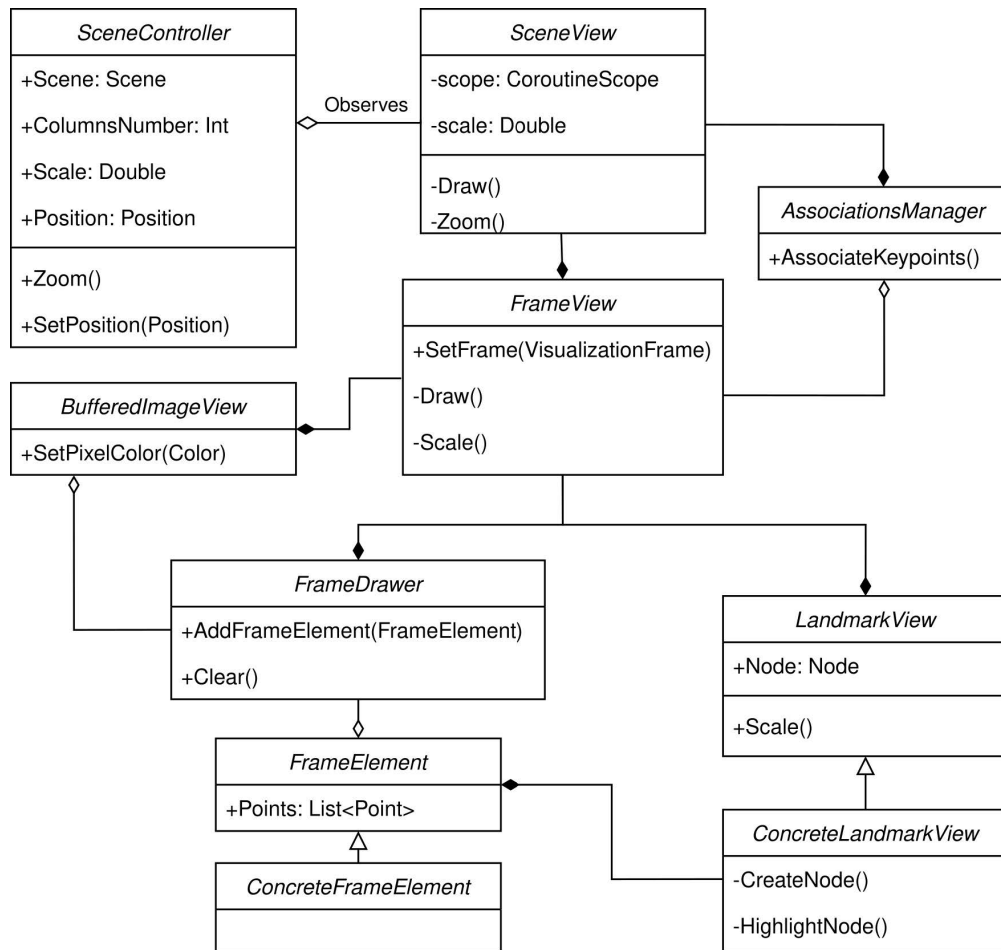


Рис. 2: Архитектура предыдущей реализации системы рендеринга сцены

поненты `PixelBuffer` и `WritableImage` библиотеки JavaFX.

Ориентиры точек и линий представлены в виде элементов визуального дерева приложения. Класс `LandmarkView` отвечает за визуализацию конкретного ориентира, сопоставляя ему геометрический примитив `Shape` и располагая его в иерархии компонентов приложения.

В предыдущей реализации для каждой клетки в сетке кадров могло выделяться несколько буферов. Это было обусловлено особенностями реализации логики, связанной с выделением ориентиров и изменением порядка их отрисовки, а также тем, что ориентиры плоскостей представлены в виде битовых масок. Помимо этого, имелаась необходимость в наложении слоёв ориентиров друг на друга с учётом прозрачности и цвета, что также создавало дополнительные вычислительные нагрузки. Таким образом, при отрисовке сцены, вне зависимости от используемо-

го масштаба, приходилось хранить в памяти и обрабатывать несколько слоёв полноразмерных изображений. Такой способ рендеринга требовал довольно большой вычислительной мощности для стабильной работы и потреблял много оперативной памяти. Даже на средних проектах, содержащих менее чем 100 кадров, наблюдались проблемы с производительностью. Проведённая апробация показала, что данные проблемы являются существенными и негативно сказываются на пользовательском опыте.

2.3 Решения для рендеринга

Для оптимизации работы компонента сцены необходимо было провести обзор решений для рендеринга, совместимых с текущим стеком приложения.

2.3.1 Нативные решения для рендеринга

JavaFX и стандартная библиотека Java предоставляют некоторые нативные решения для рендеринга. Из них можно выделить несколько основных:

- Canvas,
- PixelBuffer (WritableImage),
- PixelBuffer (BufferedImage).

В [1] можно найти сравнение их производительности при отображении множества движущихся частиц, по итогам которого наиболее эффективным решением для рендеринга с большим отрывом оказалась реализация PixelBuffer (Writable Image), а наименее эффективным – Canvas. Последнее решение является стандартным для рендеринга в JavaFX и обладает простым интерфейсом для рисования изображений и графических примитивов. PixelBuffer предоставляет лишь низкоуровневый интерфейс для работы с пикселями в буфере. В решении PixelBuffer с

использованием `WritableImage` мы изменяем значения пикселей напрямую, в отличие от решения с применением `BufferedImage`, где мы можем использовать готовые методы для работы с графикой, предоставляемые стандартной библиотекой AWT языка Java.

Решение `PixelBuffer (WritableImage)` не было взято в рассмотрение, поскольку при его использовании пришлось бы с нуля реализовывать алгоритмы растеризации и рисования графических примитивов.

Для измерения производительности двух оставшихся решений был организован тестовый стенд, представленный в ветке репозитория проекта [2]. В сравнении производилась отрисовка сетки изображений размером 15x15 кадров с разрешением 1024x768 пикселей. В качестве результата бралось среднее время рендеринга одного кадра для 1000 измерений. Первые 100 измерений были пропущены ради чистоты эксперимента.

Результаты для решения `Canvas` составили 0.590 ± 0.034 секунд на кадр, что не отвечает техническим требованиям к компоненту.

По результатам замеров производительности было решено также не брать в рассмотрение решение `PixelBuffer (BufferedImage)`, поскольку частота обновления экрана при отображении движущейся картинки с применением данного способа отрисовки была ниже значения в 24 кадра в секунду, что не соответствовало предъявленным требованиям к производительности.

2.3.2 Сторонние решения для рендеринга

Поскольку никакое из нативных решений для рендеринга не удовлетворило имеющимся требованиям, было решено рассмотреть сторонние библиотеки.

К рассматриваемым решениям был предъявлен ряд требований:

- рендеринг, использующий мощности GPU;
- кроссплатформенность;
- наличие разрешительной лицензии;
- поддержка решения разработчиками.

Для рассмотрения были выбраны четыре наиболее популярных библиотеки, представленных в таблице 1.

Как видно, все представленные решения обладают разрешительными лицензиями.

OpenGLFX и DriftFX являются библиотеками, использующими для рендеринга API OpenGL — платформонезависимый интерфейс к графической аппаратуре. Данные решения позволяют интегрировать OpenGL-контекст для рендеринга в приложение JavaFX. Оба решения могут использовать для вызовов API OpenGL [17] одну из двух библиотек: LWJGL [13] или JOGL [5]. Заметим, что OpenGLFX и DriftFX имеют некоторые проблемы с совместимостью. OpenGLFX при использовании с JOGL не работает на устройствах с macOS (с LWJGL такой проблемы нет). DriftFX, по словам разработчиков, не работает на устройствах с ОС Linux, обладающих интегрированными графическими процессорами Intel Graphics, а также на устройствах, использующих видеокарты Intel HD Graphics 4000.

FXGL является игровым фреймворком, реализованным на основе библиотеки JavaFX. Внедрение данного решения подразумевает исполь-

№	Библиотека	Лицензия	Последний релиз
1	OpenGLFX	Apache-2.0	3.0.5 (12.06.2023)
2	DriftFX	EPL-2.0	1.0.0 (8.09.2022)
3	FXGL	MIT	17.3 (30.03.2023)
4	NativeFX	Apache-2.0	0.2.0 (29.11.2019)

Таблица 1: Сравнение сторонних решений для рендеринга

зование навязанной фреймворком архитектуры для построения приложения, а также несёт большое количество лишних в контексте данной задачи зависимостей.

NativeFX является библиотекой, позволяющей выполнять рендеринг с использованием нативного API платформы, независимо от JVM. На момент проведения обзора, данное решение находилось в незаконченной стадии разработки, а последний релиз в репозитории датировался 29.11.2019, что говорит об отсутствующей поддержке библиотеки.

На основании этого было решено не брать на рассмотрение библиотеки FXGL и NativeFX, а рассмотреть подробнее решения OpenGLFX и DriftFX, использующие API OpenGL для рендеринга, сравнив их производительность.

В ветке репозитория проекта [18] был организован тестовый стенд для отрисовки набора изображений.

Необходимо было также реализовать тестовую систему, использующую некоторые оптимизации при отрисовке кадров. Для этого, с целью ускорить процесс тестирования производительности сравниваемых решений, был взят и частично адаптирован код из GitHub-репозитория MinimalLWJGLEngine [14].

Тестовая сцена представляет собой сетку из 500x500 текстур, взятых из тестового проекта, содержащего 39 разных изображений размера 640x480. Размер всех текстур на сцене уменьшен в 100 раз. При запуске программы «камера» двигается, захватывая разные участки сцены.

Вся логика, связанная с отрисовкой и вызовом функций OpenGL, написана с использованием библиотеки LWJGL и является общей для сравниваемых решений. Поэтому результаты данного тестирования показывают, насколько быстро та или иная реализация связывает полученное от GPU изображение с текстурой, отображаемой в окне приложения JavaFX.

Тестирование проводилось на компьютере следующей конфигурации:

- CPU: AMD Ryzen 5 3550H (2.1GHz);

- RAM: 16GB (2x8GB) 2666MHz;
- Storage: 256GB SSD, 1TB HDD;
- OS: Windows 10 64-bit;
- GPU: NVIDIA GeForce GTX1650 и интегрированная графика AMD Radeon Vega 8.

Отдельно были рассмотрены случаи для дискретной и интегрированной видеокарт.

Было проведено 1000 замеров промежутков времени, проходивших между очередными вызовами отрисовки окна рендеринга. Первые 200 вызовов были пропущены и не учитывались в расчётах. В процессе эксперимента были использованы два способа работы с памятью GPU в DriftFX: MainMemory и NVDXInterop. Первый метод является кроссплатформенным решением, а второй доступен только на устройствах с ОС Windows.

Результаты представлены в таблице 2.

Как видно из сравнения, DriftFX в среднем показал себя эффективнее на 10-11% OpenGLFX при использовании метода NVDXInterop. В случае, если рассматривать результаты DriftFX с универсальным методом MainMemory, то разница в производительности в незначительна и составляет менее 4%.

	Среднее время отрисовки (NVIDIA GTX1650m)	Среднее время отрисовки (AMD Radeon Vega 8)
OpenGLFX	31.60 ± 3.64 мс	64.31 ± 4.50 мс
DriftFX (MainMemory)	32.95 ± 3.54 мс	62.53 ± 4.89 мс
DriftFX (NVDXInterop)	28.69 ± 3.38 мс	58.56 ± 5.24 мс

Таблица 2: Сравнение сторонних решений для OpenGL-рендеринга

По итогам сравнения, производительность решений находится примерно на одном уровне. Если сравнивать библиотеки по другим критериям, то более подходящим решением является OpenGLFX, поскольку DriftFX имеет нерешённые проблемы, связанные с запуском на некоторых устройствах.

В результате проведённого сравнения было выбрано решение, использующее для рендеринга API OpenGL.

OpenGL использует для графических вычислений шейдеры [7] — небольшие программы, исполняемые на GPU. Последовательность шагов, исполняемых OpenGL для отрисовки изображения, называется пайплайном отрисовки [8], принцип работы которого изображён на рисунке 3¹.

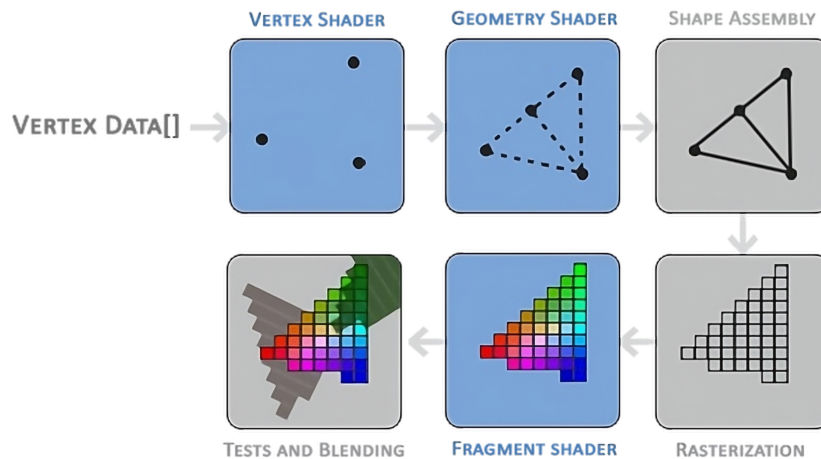


Рис. 3: Процесс выполнения пайплайна отрисовки

Обычно пайплайн отрисовки состоит как минимум из двух шейдеров, выполняемых последовательно: вершинного, отвечающего за работу с вершинами многогранников, и фрагментного, формирующего итоговое растровое изображение на уровне отдельных пикселей. В некоторых случаях также может использоваться геометрический шейдер, который находится между двумя предыдущими в пайплайне отрисовки. Он работает на уровне отдельных примитивов, представленных полигонами, отрезками и точками. На каждом этапе работы пайплайна

¹<https://learnopengl.com/Getting-started/Hello-Triangle>

операции распараллеливаются графическим ускорителем, за счёт чего достигается высокая скорость рендеринга изображения. Наиболее затратным процессом во всей последовательности шагов отрисовки является передача данных с CPU на GPU. Можно выделить следующие основные типы данных, передаваемых с процессора на видеокарту:

- вершинные данные,
- текстуры,
- uniform-данные.

Вершинные данные [11] используются для отрисовки многоугольников и любых других графических примитивов и передаются с процессора на видеокарту для каждого кадра. Вершинные данные содержат любые специфичные для конкретных вершин данные. Текстуры [9] используются для отрисовки изображений на видеокарте и требуют предварительной загрузки в память GPU. Uniform-данные [10] доступны в любой момент исполнения пайплайна отрисовки, и используются для хранения данных, являющихся общими для всех отображаемых элементов. Наибольший объём данных, передаваемых с CPU на GPU, как правило, приходится на вершинные данные. По этой причине для достижения максимальной производительности пайплайна отрисовки необходимо уделять внимание сокращению объёма передачи именно этого типа данных.

2.3.3 Требования к новому компоненту сцены

Одной из основных проблем предыдущей реализации было использование отдельных контекстов рендеринга для клеток в сетке кадров. С учётом этого, в новой реализации было решено использовать один контекст рендеринга для отрисовки всей сцены. Таким образом, рендеринг сцены не зависел бы от используемого масштаба или разрешения кадров, не требовал бы больших вычислительных ресурсов и не потреблял бы большое количество оперативной памяти.

Помимо этого, необходимо было поддерживать существующую функциональность приложения, связанную с отображением кадров и виртуализацией, настройками визуализации и управлением сценой.

Таким образом, к компоненту сцены, использующему API OpenGL для рендеринга, был предъявлен ряд требований:

- отображение выборки кадров с размеченными на них ориентирами и ассоциациями;
- отрисовка сцены в рамках одного контекста рендеринга;
- виртуализация на уровне отображения и на уровне данных;
- наличие интерфейса для перемещения и масштабирования сцены;
- поддержка существующих настроек визуализации.

Содержимое сцены можно условно разделить на составляющие, представленные кадрами, слоями с ориентирами трёх типов и ассоциациями. По такому же принципу было решено организовать новую систему рендеринга.

3 Реализация системы рендеринга

Система рендринга сетки кадров, помимо вершинного и фрагментного шейдеров, использует геометрический шейдер для оптимизации рендринга. Это осуществляется за счёт уменьшения объёма данных, которые приходится передавать с CPU на GPU: для отрисовки кадра передаётся лишь его индекс, а все остальные вычисления производятся на видеокарте. При импорте проекта выделяется фиксированный набор буферов под текстуры, которые переиспользуются в процессе работы, из-за чего не приходится каждый раз заново выделять память под кадры на стороне GPU.

Для отрисовки ориентиров используются вершинный и фрагментный шейдеры. В случае ориентиров точек и линий на GPU передаётся информация об индексе кадра и координатах вершин. Для отображения плоскостей используются текстуры битовых масок. При таком способе визуализации видеокарта выполняет отрисовку пикселей параллельно, используя фрагментный шейдер. В этом состоит отличие от предыдущей реализации, где из битовых масок предварительно извлекались координаты пикселей плоскостей, которые затем последовательно переносились в компонент `PixelBuffer`, что требовало больших вычислительных ресурсов.

Все системы рендеринга помимо собственных оптимизаций используют механизм батчинга (пакетной отрисовки). Его суть заключается в объединении небольших наборов вершин отображаемых графических элементов в одну большую последовательность. За счёт этого значительно сокращается количество вызовов между CPU и GPU, что увеличивает производительность пайплайна отрисовки.

Навигация по сцене и масштабирование были сделаны при помощи использования матриц перемещения и масштабирования при расчёте координат вершин отображаемых графических элементов.

Диаграмма классов системы рендеринга представлена на рисунке 4.

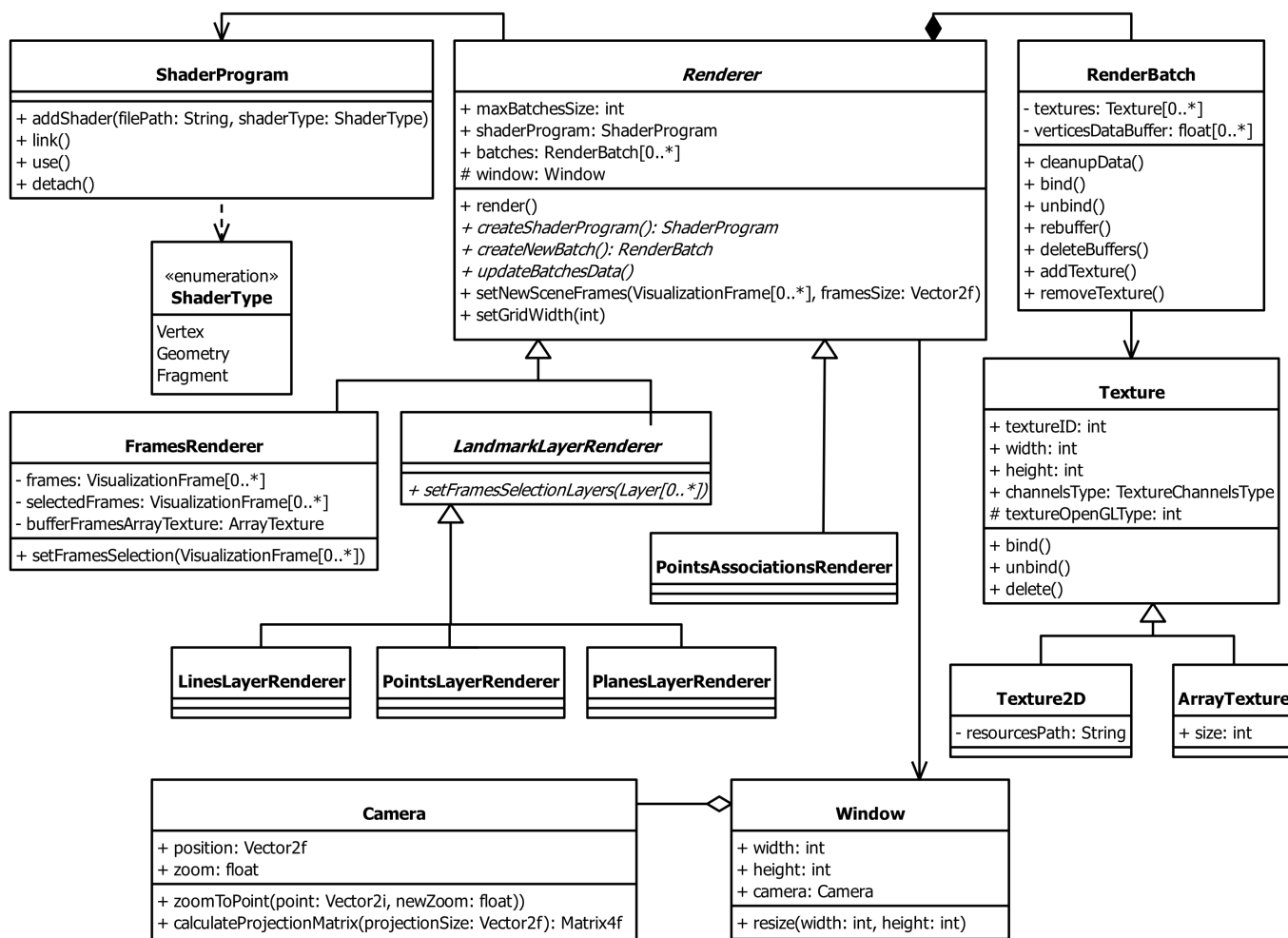


Рис. 4: Диаграмма классов системы рендеринга

Ключевым компонентом в данной диаграмме является абстрактный класс **Renderer**, отвечающий за процесс рендеринга. В зону его ответственности помимо отрисовки входит также управление шейдерной программой, представленной классом **ShaderProgram** и организация процесса батчинга при помощи использования объектов типа **RenderBatch**.

От класса **Renderer** наследуются классы **FramesRenderer** и **LandmarkLayerRenderer**, отвечающие соответственно за отображение кадров и ориентиров.

Класс **FramesRenderer** имеет несколько публичных методов, позволяющих изменять набор отображаемых кадров и размер сетки фреймов. При импорте проекта заранее выделяется фиксированный набор буферов для хранения текстур. Буферы инициализируются текстура-

ми, соответствующими начальным кадрам, и в процессе перемещения камеры, представленной классом `Camera`, происходит их переиспользование путём замещения старых текстур, пропавших из зоны видимости сцены, новыми. Таким образом обеспечивается виртуализация на уровне визуализации и на уровне данных при отрисовке кадров сцены.

Классы `PointsLayerRenderer`, `LinesLayerRenderer` и `PlanesLayerRenderer` отвечают за отрисовку ориентиров точек, линий и плоскостей соответственно. Экземпляр каждого из этих классов связан с некоторым слоем ориентиров на сцене. Таким образом, нет необходимости передавать информацию, связанную с настройками визуализации, для каждого из ориентиров. Параметры настроек визуализации, которые являются общими для всего слоя, передаются однократно в виде набора `uniform`-параметров.

Класс `PointAssociationsRenderer` отвечает за визуализацию ассоциаций точек и использует механизм отрисовки, схожий с тем, что используется для ориентиров линий.

На рисунках 5, 6 и 7 продемонстрирован внешний вид ориентиров и ассоциаций в новой реализации компонента сцены. Видео с демонстрацией его работы доступно по ссылке².

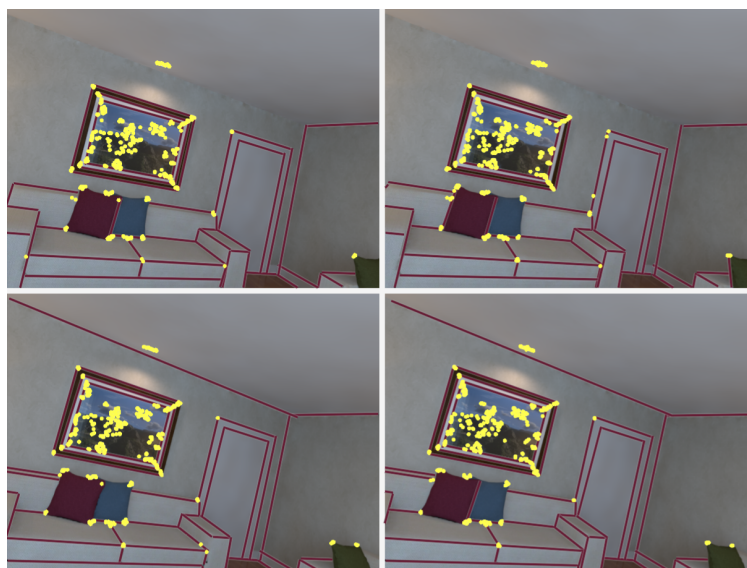


Рис. 5: Визуализация ориентиров точек и линий

²https://drive.google.com/file/d/1Hx9EUyPjsH4MCMXTc_FiVVavI3RQEosA

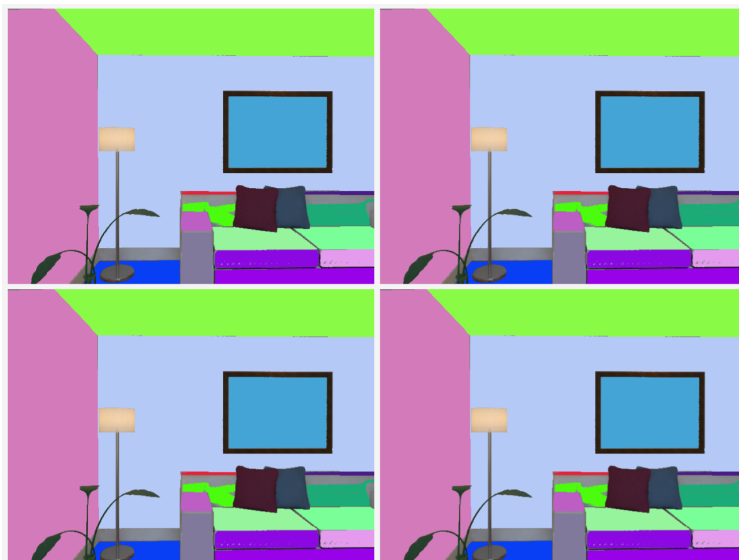


Рис. 6: Визуализация ориентиров плоскостей



Рис. 7: Визуализация ассоциаций точек

Описанная выше система рендеринга используется для отрисовки изображения в окне рендеринга, предоставляемом компонентом `OpenGLFXCanvas` библиотеки `OpenGLFX`. Для упрощения взаимодействия с функциональностью библиотеки были реализованы вспомогательные классы, диаграмма классов для которых представлена на рисунке 8.

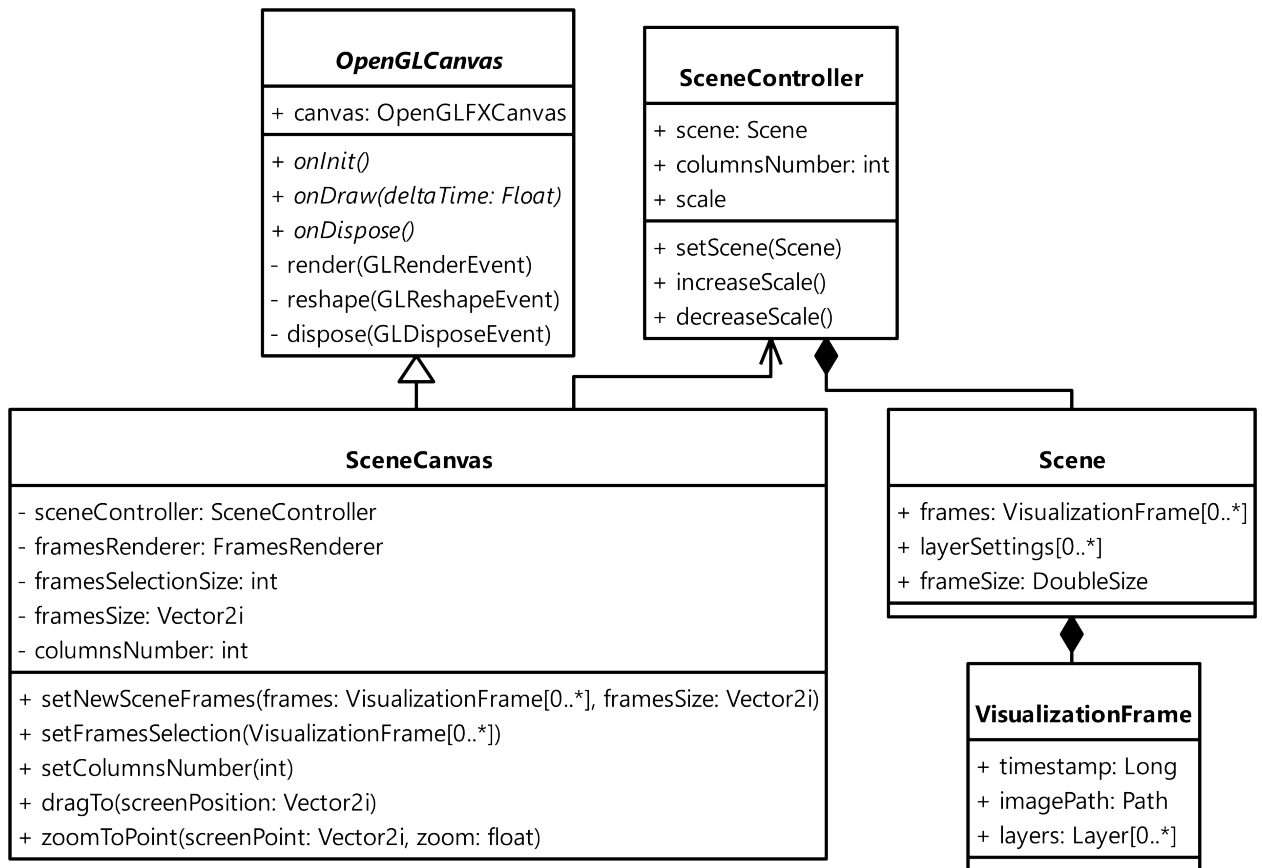


Рис. 8: Диаграмма классов компонента рендеринга

Абстрактный класс `OpenGLCanvas` агрегирует в себе логику, связанную с инициализацией компонента окна рендеринга, предоставляемого библиотекой `OpenGLFX`. Данный класс наследуется компонентом `SceneCanvas`, реагирующим на изменения параметров сцены, содержащихся в контроллере `SceneController` и вызывающим соответствующие методы системы рендеринга сетки кадров сцены `FramesRenderer`. Помимо этого, `SceneCanvas` отслеживает события мыши в системе ввода и на основе полученных данных реализует логику перемещения по сцене и приближения её содержимого.

4 Сравнение производительности и апробация

Для сравнения производительности двух реализаций был сформирован тестовый стенд, в котором производилось измерение времени отрисовки содержимого сцены на проекте, содержащем 100 кадров с ориентирами. Во время эксперимента производилась навигация по сцене и её масштабирование. Было произведено 2000 замеров времени для каждой из реализаций, первые 200 вызовов отрисовки были пропущены и не учитывались при расчётах. Сравнение производилось на компьютере следующей конфигурации:

- CPU: AMD Ryzen 5 3550H (2.1GHz);
- RAM: 16GB (2x8GB) 2666MHz;
- Storage: 256GB SSD, 1TB HDD;
- OS: Windows 10 64-bit;
- GPU: интегрированная графика AMD Radeon Vega 8.

В таблице 3 представлены результаты эксперимента.

Как видно из результатов, среднее время отрисовки для новой реализации компонента сцены в 4 раза меньше, чем у предыдущего решения, а частота кадров намного стабильнее. Большое среднеквадратическое

	Среднее время отрисовки кадра, мс
Предыдущая реализация компонента сцены	72.0 ± 45.0
Новая реализация компонента сцены	17.4 ± 2.5

Таблица 3: Сравнение производительности компонентов сцен

отклонение времени отрисовки предыдущей реализации обусловлено периодическими кратковременными зависаниями приложения. Полученное значение средней частоты кадров для новой реализации составляет 57 кадров в секунду, против 14 кадров в секунду у предыдущего решения. Стоит сказать, что максимально допустимая частота обновления сцены составляет 60 кадров в секунду.

4.1 Апробация

Для того чтобы оценить приложение с оптимизированной сценой на предмет удобства использования, была проведена апробация с использованием методики SUS (System Usability Scale) [20]. Данная система оценивания представляет собой опросник, состоящий из 10 вопросов, каждый из которых имеет пять вариантов ответов, отражающих разную степень удовлетворённости продуктом. На основе полученных ответов формируется оценка по стобалльной шкале. В рамках апробации участникам было предложено выполнить два небольших задания, связанных с использованием инструмента на различных наборах данных. В апробации приняли участие 7 сотрудников Лаборатории мобильной робототехники, и по результатам опроса итоговый балл составил 74.29 (оценка B), что говорит о том, что пользоваться приложением достаточно удобно.

5 Заключение

В ходе выполнения работы были выполнены следующие задачи:

1. Выполнен обзор разрабатываемого приложения и используемых в нем технологий.
2. Выполнен обзор решений для рендеринга в рамках JavaFX-приложения.
3. Разработана система для отрисовки кадров, ориентиров и ассоциаций с применением выбранного решения.
4. Разработана система для управления сценой в рамках новой реализации.
5. Выполнен обзор методов измерения производительности сцены, проведено сравнение реализаций.
6. Проведена апробация приложения с оптимизированной сценой.

Ссылка на репозиторий GitHub с исходным кодом разрабатываемого инструмента: github.com/prime-slam/SOLVE.

Ссылка на аккаунт, с которого велась разработка: github.com/mists.

Список литературы

- [1] Baimagambetov Almas. High Performance Rendering in JavaFX. — URL: <https://foojay.io/today/high-performance-rendering-in-javafx/> (дата обращения: 2024-04-28).
- [2] Built-in render solutions comparison repository. — URL: <https://github.com/prime-slam/SOLVE/tree/render-solution-tests> (дата обращения: 2024-04-28).
- [3] Cartographer repository. — URL: <https://github.com/cartographer-project/cartographer> (дата обращения: 2024-04-28).
- [4] Design Patterns: Model-View-Controller. A Compound Design Pattern. — URL: https://cs.smu.ca/~porter/csc/465/notes/design_patterns_mvc.html (дата обращения: 2024-04-28).
- [5] JOGL official site. — URL: <https://jogamp.org/jogl/www/> (дата обращения: 2024-04-28).
- [6] JavaFX library official site. — URL: <https://openjfx.io/> (дата обращения: 2024-04-28).
- [7] Khronos OpenGL Wiki. OpenGL shaders. — URL: <https://www.khronos.org/opengl/wiki/Shader> (дата обращения: 2024-04-28).
- [8] Khronos OpenGL Wiki. Rendering Pipeline Overview. — URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (дата обращения: 2024-04-28).
- [9] Khronos OpenGL Wiki. Texture. — URL: <https://www.khronos.org/opengl/wiki/Texture> (дата обращения: 2024-04-28).
- [10] Khronos OpenGL Wiki. Uniform (GLSL). — URL: [https://www.khronos.org/opengl/wiki/Uniform_\(GLSL\)](https://www.khronos.org/opengl/wiki/Uniform_(GLSL)) (дата обращения: 2024-04-28).

- [11] Khronos OpenGL Wiki. Vertex Specification. — URL: https://www.khronos.org/opengl/wiki/Vertex_Specification (дата обращения: 2024-04-28).
- [12] Kotlin programming language official site. — URL: <https://kotlinlang.org/> (дата обращения: 2024-04-28).
- [13] Lightweight Java Game Library official site. — URL: <https://www.lwjgl.org/> (дата обращения: 2024-04-28).
- [14] MinimalLWJGLEngine repository. — URL: <https://github.com/ambrosiogabe/MinimalLWJGLEngine> (дата обращения: 2024-04-28).
- [15] Mobile Robotics Lab official site. — URL: <https://sites.skoltech.ru/mobilerobotics/> (дата обращения: 2024-04-28).
- [16] ORB-SLAM2 repository. — URL: https://github.com/raulmur/ORB_SLAM2 (дата обращения: 2024-04-28).
- [17] OpenGL official site. — URL: <https://www.opengl.org/> (дата обращения: 2024-04-28).
- [18] OpenGL libraries comparison repository. — URL: <https://github.com/prime-slam/SOLVE/tree/gpu-rendering-libs-tests> (дата обращения: 2024-04-28).
- [19] Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age / Cesar Cadena, Luca Carlone, Henry Carrillo et al. // [IEEE Transactions on Robotics](#). — 2016. — Vol. 32, no. 6. — P. 1309–1332.
- [20] System Usability Scale. — URL: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html> (дата обращения: 2024-04-28).
- [21] TorandoFX framework official site. — URL: <https://tornadofx.io/> (дата обращения: 2024-04-28).

- [22] Плоскин А. Е. Разработка сцены инструмента для визуализации и редактирования распознавания ориентиров в SLAM алгоритмах // Сайт кафедры Системного Программирования Математико Механического факультета СПбГУ.— 2023.— URL: https://se.math.spbu.ru/thesis/texts/Ploskin_Aleksandr_Evgen'evich_Spring_practice_3rd_year_2023_text.pdf (дата обращения: 2024-04-28).