Санкт-Петербургский государственный университет

Группа

Проектирование и реализация UI фреймворка на основе идеологии immediate mode

Левков Данил Андреевич

Отчёт по преддипломной практике
в форме «Эксперимент»

Научный руководитель:
доц. каф. СП, к.ф.-м.н. Д. В. Луцив

Санкт-Петербург
2024

Оглавление

Введение **3**

1. Постановка задачи **4**

2. Обзор **5**

2.1. Имеющиеся решения 5
2.2. Лицензирование Qt 5
2.3. Описание Qt Widgets 6
2.4. Архитектура Qt Widgets 7
2.5. Использование GPU в Qt Widgets 8
2.6. Описание Qt Quick 8
2.7. Архитектура Qt Quick 9
2.8. Описание проблемы 11
2.9. Парадигма Immediate Mode GUI 12
2.10. MVVM 12
2.11. Dear ImGui 14

3. Решение **16**

3.1. Адаптивный layout 16
3.2. Обработка событий 17
3.3. Композиция элементов 18
3.4. Поддержка автотестов 19
3.5. Применение 20

4. Апробация **21**

4.1. Условия эксперимента 21
4.2. Исследовательские вопросы 21
4.3. Метрики 22
4.4. Обсуждение результатов 24

Заключение **27**

Список литературы **28**

Введение

В современном мире выбор инструментов для разработки нативных кроссплатформенных приложений является сложной задачей не смотря на небольшой спектр решений, представленных в этой области, по сравнению с многообразием систем, существующих в мире Web разработки. В первую очередь это касается проектов, для которых главным критерием является производительность, что накладывает ограничения как на стек используемых технологий, так и на количество связывающих их API-слоев. Таким образом возникает неизбежный компромисс между эффективностью кода и скоростью разработки пользовательского интерфейса. Ограничимся рассмотрением монолитных приложений, использующих C++ в качестве основного языка.

В описанных ограничениях самым популярным среди имеющихся решений является многопрофильный, кроссплатформенный фреймворк Qt. Однако ввиду постепенного ужесточения лицензии на продукты Qt Group во всем мире и частичного запрета на их использование в нашей стране, применение данного фреймворка становится всё менее оправданным. Также Qt изначально создавался как универсальный фреймворк широкого применения и зачастую это сказывается на его производительности в случае высоконагруженных систем, работающих с тяжелыми графическими вычислениями, такими как потоковое видео высокого разрешения, анимации, 2D и 3D сцены, которые тесно интегрированы в пользовательский интерфейс и не могут быть вынесены в независимые окна. Данная работа посвящена анализу подобных систем и разработке альтернативного узкоспециализированного фреймворка, позволяющего создавать современные интерфейсы.

1. Постановка задачи

Цель данного исследования заключается в выявлении слабых сторон производительности подсистемы рендеринга Qt и создании собственного фреймворка ImReady, лишённого обнаруженных недостатков. Для достижения цели были поставлены следующие задачи:

1. Выявить и структурировать архитектурные особенности фреймворка Qt разных версий.
2. На основе идеологии Immediate Mode разработать UI фреймворк, учитывающий ограничения данного подхода.
3. Разработать адаптивную layout-систему и модель обработки событий.
4. Добавить возможность тестирования графического интерфейса.
5. Сравнить производительность полученного фреймворка с Qt.

2. Обзор

2.1. Имеющиеся решения

Одним из основных игроков на рынке программного обеспечения для разработки пользовательских интерфейсов, применяемого для десктопных приложений, является The Qt Company. Их графический фреймворк обладает большим количеством преимуществ: кроссплатформенность, использование C++, активное сообщество разработчиков. Однако ввиду универсальности и тяжеловесности фреймворка, его использование не всегда оправдано в определенном множестве случаев. Именно поэтому сравнение производительности нашего фреймворка с аналогичными решениями от Qt представляет большой интерес. Перед тем, как перейти к собственному решению, предлагается более подробно рассмотреть две принципиально различные парадигмы формирования интерфейса с помощью Qt фреймворка: Qt Widgets и Qt Quick 2.

2.2. Лицензирование Qt

Фреймворк Qt поставляется по двойной лицензии: коммерческой и GPL/LGPL [11]. Первая предоставляет возможность использования последних версий Qt в проектах с закрытым исходным кодом, и при этом разрешает их настройку и изменение. Она включает как поддержку и исправление ошибок фреймворка, так и консультации, которые могут быть предоставлены разработчикам. В случае, если в коммерческом проекте предполагается использование Qt в качестве динамических библиотек в их неизменном виде, можно ограничится версиями, распространяемыми по лицензии LGPLv2. Однако, начиная с Qt 5.15 выпуски фреймворка с долгосрочной поддержкой (LTS) и offline-установщик стали доступны только по коммерческой подписке, а скачивание бинарных файлов возможно только с использованием учетной записи Qt Account. С переходом в 2016 году на LGPLv3 [3] для Qt 5.7 и более поздних версий, были введены дополнительные ограничения. Возникли сложности с использованием фреймворка для разработки мобильных приложений,

так как в этом случае необходимо предоставить конечному потребителю возможность заменить используемую версию Qt на другую, что с технической точки зрения довольно сложно, и зачастую неприемлемо с точки зрения безопасности ПО. Стоит также сказать, что операционные системы Windows 7 и 8 не будут более поддерживаться в Qt6 [1].

2.3. Описание Qt Widgets

В данной системе базовым элементом построения интерфейса является виджет, представляющий из себя полноценный C++ класс, для которого можно переопределить методы обработки событий, отрисовки и поведения в определенных ситуациях. Верстка, то есть композиция элементов, осуществляется с помощью явного указания отношений ребенок-родитель между виджетами и их связыванием посредством специальных классов, унаследованных от QLayout [14]. У такого подхода есть свои преимущества и недостатки. К плюсам можно отнести явное и однозначное описание структуры объектов и их свойств, так как оно задается посредством строго типизированного языка. Таким образом, практически исключаются ситуации, когда код, описывающий интерфейс, ведет себя не так как ожидал разработчик. С другой стороны, необходимость явно задавать иерархию с помощью вызова методов негативно сказывается на читаемости кода и его общей структуре, так как «скелет» интерфейса и его визуальная составляющая становятся сильно разнесены. Описанный эффект также является недостатком связки HTML и CSS. Ещё одним минусом написания UI с помощью виджетов является их низкая расширяемость. Это вытекает из синтаксиса классов в C++, который сам по себе не может обеспечить достаточной уровень гибкости и переиспользуемости объектов классов с множеством часто меняющихся параметров, при этом сохраняя возможность комбинировать их в различных сочетаниях. В той или иной степени эту проблему можно решить, применяя всевозможные паттерны проектирования [13]. Однако удобство их разработки и использования также не соответствует требованиям для простого, быстрого и интуитивно понятного описания

интерфейса.

Существует несколько встроенных способов стилизации QWidget. Во-первых, с помощью переопределения базового класса QStyle можно влиять на внешний вид стандартных GUI компонентов, таких как QPushButton, QComboBox и тд. Для этого необходимо знать enum идентификатор того компонента, или части компонента, которую необходимо поменять, что создает дополнительные сложности. Во-вторых, некоторые визуальные атрибуты доступны для редактирования в упрощенном виде, используя Qt Style Sheets, представляющие из себя текстовое описание с синтаксисом, похожим на CSS. Однако данное решение плохо себя зарекомендовало на практике, так как количество атрибутов стилей для конкретного виджета сильно ограничено, а сами они плохо задокументированы.

2.4. Архитектура Qt Widgets

В базовом сценарии фреймворк использует софтверную растеризацию на CPU, которая заключается в последовательной обработке сигналов QPaintEvent от родительских элементов к дочерним. Таким образом, формируется линейный набор команд отрисовки. Исполнителем этих команд является класс QPainter, который определяет базовый интерфейс работы с графическими примитивами и имеет различные реализации в зависимости от целевого устройства отрисовки (paint device). В качестве последнего могут выступать: принтер (QPrinter), текстура (QPixmap), изображение (QImage) и в самом распространенном варианте QWidget, который тоже является наследником QPaintDevice. Таким образом, чтобы поддержать описанный функционал, команды QPainter должны формироваться в линейном порядке и вызываться по очереди. Нельзя одновременно иметь два активных QPainter, привязанных к одному QPaintDevice. Данное ограничение является незначительным в случае софтверной растеризации, так как редкие перерисовки достаточно сложных сцен для CPU не являются сложной задачей даже при вычислении в одном потоке. Однако в случае использования графическо-

го процессора возможность объединять команды в группы и отправлять их на отрисовку параллельно является принципиальной. Описанная проблема будет более подробно освещена в работе далее.

2.5. Использование GPU в Qt Widgets

Для того, чтобы включить графический процессор в процесс растеризации интерфейса, достаточно переопределить QPainter таким образом, чтобы он исполнял команды графического API: Vulkan, Metal, DirectX или OpenGL. Добиться этого можно с помощью добавления в иерархию виджетов класса QOpenGLWidget, который автоматически выставит QOpenGLPaintDevice в качестве устройства отрисовки, переведет текущее окно в режим смешенной композиции с использованием 3D API, создаст и настроит OpenGL контекст, а также предоставит пользователю интерфейс для инициализации необходимых ресурсов (initializeGL), обновления размеров видимой области (resizeGL) и непосредственного рендеринга OpenGL сцены (paintGL). Внутри этих методов можно использовать привычные OpenGL команды либо напрямую, либо, что более предпочтительно, посредством кроссплатформенной обертки в виде QOpenGLFunctions.

В задачах, требующих управления большим количеством интерактивных элементов, рекомендуется использовать связку QGraphicsScene/View, входящую в состав Graphics View Framework [5]. В нем применяется алгоритм Binary Space Partitioning (BSP) для быстрого поиска элементов на сцене. Это позволяет значительно ускорить нахождение и обновление объектов, однако не влияет на принцип работы системы отрисовки.

2.6. Описание Qt Quick

Перед тем как перейти к новому подходу в организации процесса подготовки кадра отметим, что Qt Quick первой версии использовал старый подход, описанный в предыдущих параграфах.

Версия	**Qt Quick 1**	**Qt Quick 2**
Основа	Qt Scripts, Qt Widgets	Qt Gui
Рендеринг	QGraphicsView, QPainter	QWindow, QOpenGL-like

Таблица 1: Сравнение модулей Qt Quick разных версий.

Главным образом переход на вторую версию обусловлен ограниченностью старой модели, использующей QPainter. С помощью неё нельзя организовать автоматическую группировку однотипных команд отрисовки для GPU, так называемый draw call batching. Это является принципиальным моментом для уменьшения времени, которое тратится на синхронизацию с CPU и на переключение состояний. Зачастую эти действия требуют больше вычислительных ресурсов, чем полезная нагрузка [10]. На рисунке 1 наглядно представлена разница между линейным исполнением команд QPainter и группировкой непересекающихся примитивов, за которую отвечает Scene Graph Renderer из Qt Quick 2.

Важно отметить, что в Qt Quick нет понятия FPS, то есть перерисовка кадров происходит по требованию, когда это необходимо, а не по таймеру с фиксированной частотой. Ещё одной отличительной особенностью Qt Quick является применение языка QML [4], использующего JavaScript JIT компилятор. В отличие от системы Style Sheets, которая лишь предоставляет интерфейс для редактирования существующих визуальных характеристик класса QStyle, QML является полноценным инструментом для описания не только визуальной составляющей, но и структуры, поведения и связанных характеристик графических компонентов.

2.7. Архитектура Qt Quick

В Qt Quick 2 существует два основных потока: GUI thread и Scene Graph thread. GUI-поток отвечает за обработку пользовательских событий, таких как нажатия клавиш и клики мыши. В нем также происходит обновление элементов интерфейса под воздействием бизнес-логики, на-

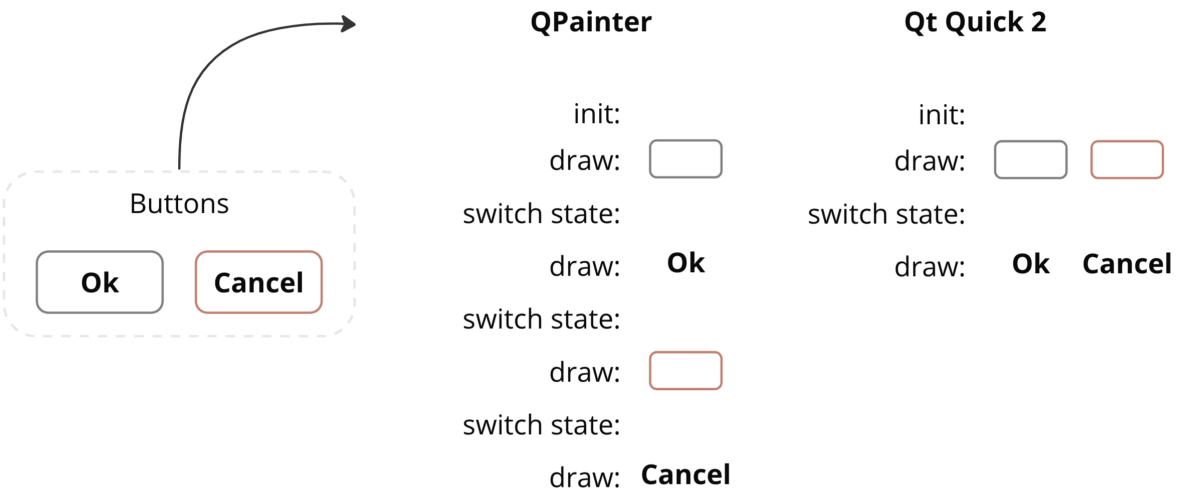

Рис. 1: Draw call batching.

пример, изменение их видимости, положения, размеров и т. д. В случае, когда не используются предварительная компиляция QML скриптов, в GUI-потоке выполняется JavaScript-код, связанный с пользовательским интерфейсом.

Scene Graph [6] отвечает за формирование дерева графических элементов, а также за их рендеринг, расчет положения и размеров, применение эффектов и анимаций. Его поток реагирует на изменения, происходящие в GUI-потоке, и обновляет сцену в соответствии с ними, осуществляя необходимые перерисовки.

Взаимодействие между потоками отражено на рисунке 2. При изменении состояния QML компонента вызывается метод QQuickItem::update(), что может быть спровоцировано анимацией или пользовательским вводом. Затем событие регистрируется в потоке рендеринга, инициируя подготовку нового кадра. В это время в основном потоке происходит обработка вызова QQuickItem::updatePolish(), который выполняет финальное обновление элемента. Далее происходит синхронизация состояния QML компонентов с узлами графа, путем вызова функции QQuickItem::updatePaintNode() для всех элементов, которые изменились с момента предыдущего кадра. После этого начинается рендеринг сцены, который заканчивается выставлением полученного кадра на вершину

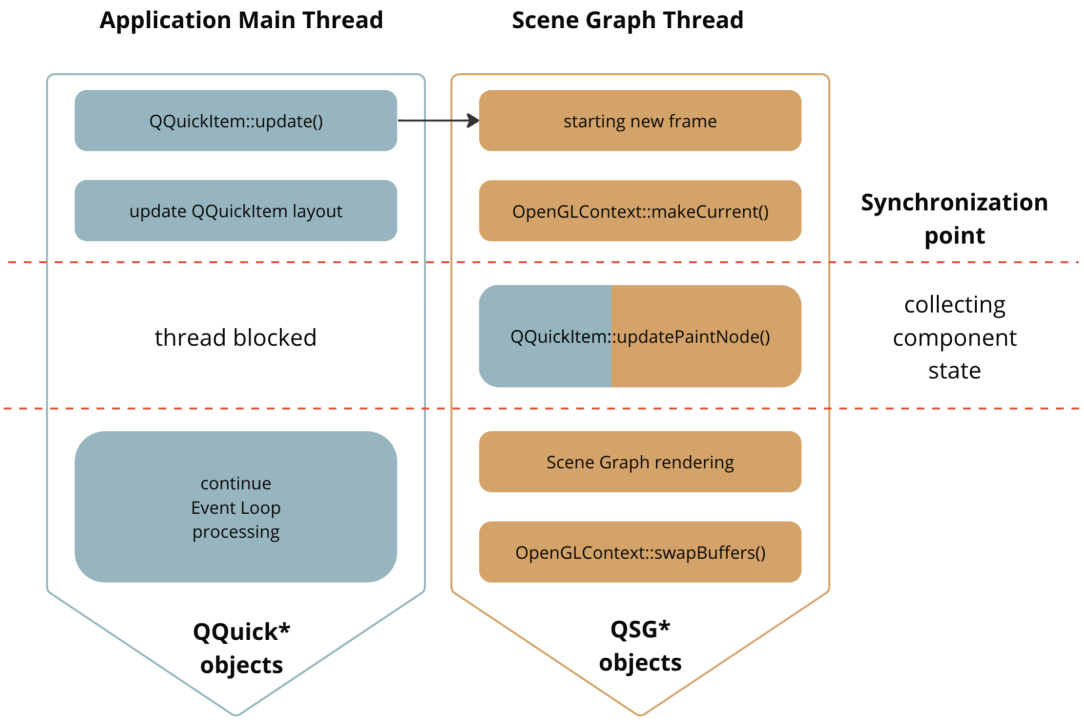

Рис. 2: Взаимодействие потоков в Qt Quick 2.

буфера кадров. Заметим, что при такой реализации, во время рендеринга GUI-поток свободен для выполнения анимации, обработки событий и других задач.

2.8. Описание проблемы

Рассмотренный фреймворк Qt относится к широко распространенному классу Retained-mode [7] графических фреймворков, что подразумевает сохранение состояния визуальных объектов (виджетов) в некой структуре данных (чаще всего в дереве) с целью предотвращения перерисовки неизменных областей. В подавляющем большинстве случаев данная оптимизация оправдана и позволяет сократить потребление вычислительных ресурсов. Но есть ситуации, когда на определенном уровне дерева возникает элемент с высокой частотой обновления, например, потоковое видео, изображение с камеры или 3D сцена. В таком случае, в зависимости от реализации композитора, нижнее поддерево или даже весь интерфейс нуждаются в постоянной перерисовке на каждом кадре. А это именно то, чего данная идеология старается избежать, ведь пол-

ный цикл перерисовки — это очень дорого. Плюс ко всему мы постоянно будем перестраивать дерево состояний.

2.9. Парадигма Immediate Mode GUI

Альтернативным решением к описанному выше служит идеология ImGui, которая подразумевает под собой полную перерисовку всего интерфейса на каждом кадре. При этом главное отличие состоит в том, что подготовка каждого компонента является очень легковесной операцией, не требующей сложных вычислений, так как всё, что происходит в фреймворке – это подготовка вершин и операций над ними. В отличие от Retained-mode GUI, в нашем случае не существует состояния графических элементов, которое нужно поддерживать. Также нет дерева виджетов как и самих виджетов, что избавляет от большого количества ресурсоемких задач. Сам по себе подход формирования UI очень похож на функциональное программирование. Вместо виджета со сложной структурой обработки событий – одна функция prepare(), вместо дерева объектов – стек вызова функций, вместо таблиц стилей – scoped параметры. Восприятие UI фреймворка как функцию состояния на данный момент является актуальной темой в сфере мобильной разработки. Этот принцип заложен в основу Jetpack Compose, одного из самых передовых android инструментов, чей интерфейс и принцип формирования компонентов очень похож на ImGui. Разница лишь в том, что Jetpack Compose кэширует состояния элементов, а ImGui нет.

2.10. MVVM

Данный шаблон проектирования применяется для того, чтобы отделить код, отвечающий за бизнес логику приложения, от кода, формирующего визуальное представление интерфейса. Он состоит из следующих смысловых частей:

- View – отвечает за презентацию интерфейса пользователю. Она должна быть спроектирована максимально просто и не содержать

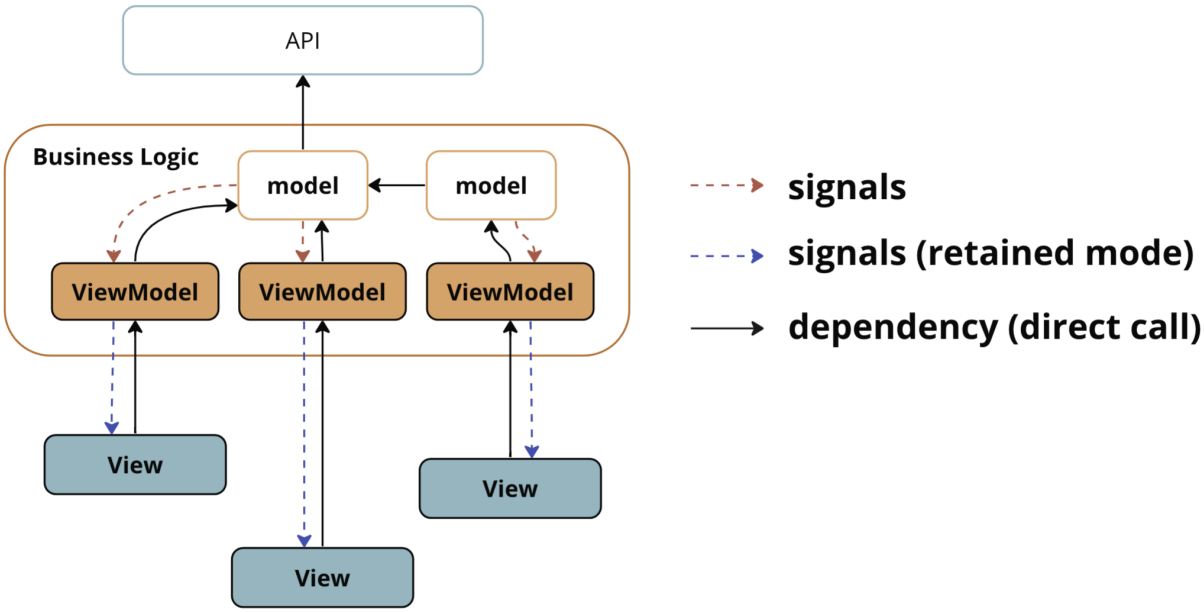

Рис. 3: Взаимодействие компонентов MVVM

в себе какую-либо логику. Её ответственность ограничивается визуальным представлением. Пример: Красная кнопка с текстом “Отмена”;

- ViewModel – представление данных, использующихся для правильного и однозначного определения View. Именно эта часть обновляет и хранит в себе состояние визуального элемента. Она отвечает за представление элемента со стороны предметной области, а также связывает его с моделью;

- Model – чистая бизнес-логика. Отвечает за процессы в приложении на верхнем уровне абстракции.

Идеология MVVM призвана отделить состояние интерфейса (ViewModel) от его отображения (View). View является по сути функцией, принимающей параметры из ViewModel, возвращающей визуальный компонент и не нуждающейся в механизме обновления. После изменения состояния внутри ViewModel, View сама обновится прочитав его на следующем кадре. Таким образом, мы избавляемся от связи, отмеченной синей стрелкой на рисунке 3. Что, в свою очередь, позволяет инкапсулировать внутри View Model бизнес логику любой сложности, вплоть до

реализации полноценных конечных автоматов, покрытых юнит-тестами. Последнее было бы гораздо сложнее реализовать, если бы состояние было распределено по классам, завязанным на UI фреймворке.

2.11. Dear ImGui

Базовые подходы и решения были взяты из открытой библиотеки Dear ImGui [2], используемой главным образом для создания пользовательского интерфейса в играх. Она вносит минимальный вклад в основное потребление ресурсов компьютера, встраиваясь в render loop приложения. Сравнивая её с более высокоуровневыми UI фреймворками, можно прийти к выводу, что библиотека является уникальной с точки зрения области применения. Её самобытность обусловлена совершенно иным подходом к оптимизации графического интерфейса. Именно поэтому реализация более функционального фреймворка на его основе представляет большой интерес. Далее будет вынесен на обсуждение вопрос о требованиях к этой реализации. Для этого рассмотрим характеристики Dear ImGui более подробно.

Достоинства:

- легковесный и не имеет зависимостей;
- исходный код составляет всего несколько .h/.cpp файлов;
- есть полный контроль над выбором механизмов рендеринга, в том числе нативных (DirectX, Vulkan, Metal);
- имеет максимальную гибкость.

Недостатки:

- отсутствует система адаптивной верстки (layouts);
- инвертированная модель обработки событий;
- отсутствие готового набора UI компонентов, отвечающих современным тенденциям в сфере дизайна приложений.

Реализованный в рамках данной работы фреймворк ImReady является более высокоуровневым по сравнению с Dear ImGui. Он сохраняет все указанные выше достоинства, при этом поддерживает дополнительные возможности, покрывающие недостатки. Ниже будут рассмотрены способы реализации этой функциональности.

3. Решение

3.1. Адаптивный layout

Под адаптивной системой верстки [8] подразумевается механизм, позволяющий элементам выстраиваться в упорядоченную последовательность по определенным правилам. Компоненты должны уметь выравниваться, располагаться с заданным отступом друг от друга, заполнять предоставленное пространство. Всё это можно реализовать в однопроходной системе построения интерфейса. Однако есть случаи, когда для определения положения элементов нужно заранее знать их размеры. Это и составляет главную сложность ввиду того, что появляется необходимость в разделении этапа формирования геометрии и её непосредственной отрисовки. Но данное решение проблемы не является наилучшим по нескольким причинам. Во-первых, практически для любого компонента геометрия и содержимое имеют высокую связанность между собой, поэтому вынос их расчёт в отдельные функции противоречит принципам хорошей архитектуры. Такой подход неизбежно приведет к дублирующемуся, запутанному коду, в который будет тяжело вносить изменения и тем более расширять функционал. Во-вторых, вычисленную геометрию нужно будет где-то сохранить для последующей передачи в функцию отрисовки, что противоречит декларативному подходу, к которому мы стремимся. Следовательно, необходимо разработать иной механизм адаптивного интерфейса. Сделаем важное уточнение: в данной работе мы не ставим перед собой задачу реализовать всё множество функций, предоставляемых стандартом CSS Flexible Box Layout [2]. Нас интересуют 2 случая, используемых в подавляющем большинстве современных интерфейсов. Первый из них – это фиксация некого параметра для всех элементов в группе. В качестве параметра может выступать ширина, длинна строки текста и тд. Реализовать необходимый функционал можно с помощью scope, который будет определять область, в которой каждый элемент будет следовать установленному правилу. Второй – это выравнивание элементов разных размеров, относительно

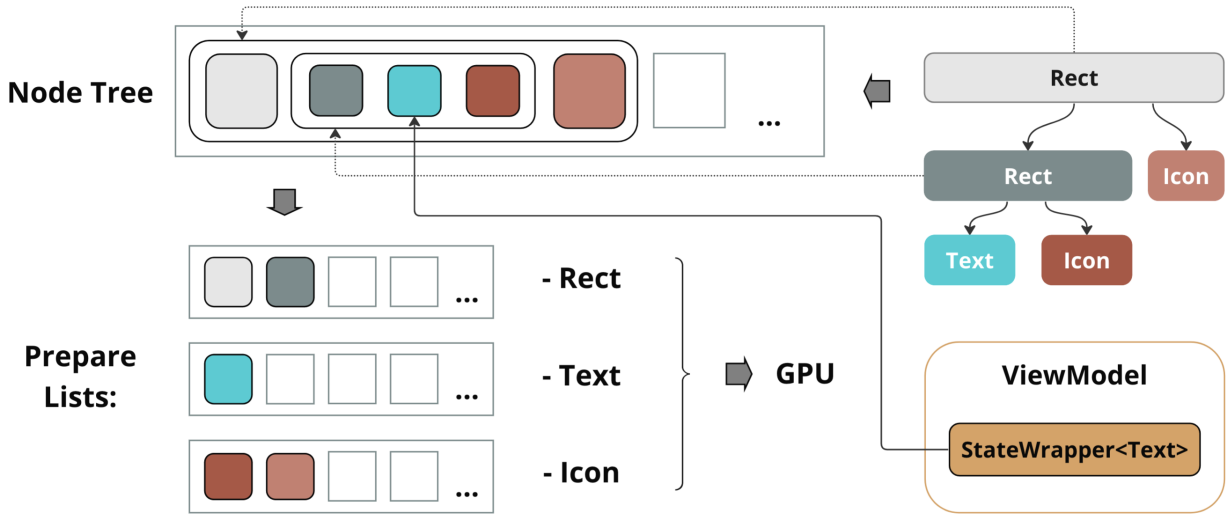

Рис. 4: Архитектура ImReady.

самого большого из них. Данный случай идеологически сложнее, ведь то что фактически нужно сделать, это отрисовать первый элемент, ещё не зная на какой конкретно позиции он находится, так как следующий за ним может оказаться больше и первый элемент должен быть выравнен, к примеру, относительно его центра. Но не будем забывать, что ImReady не рендерит непосредственно в фреймбуфер окна, он лишь предоставляет список операций, которые буду отданы на GPU. Поэтому в момент подготовки второго элемента, мы можем вернуться к примитивам первого и исправить их координаты так, чтобы получить правильное выравнивание. Таким образом, адаптивная верстка стала возможным в однопроходном режиме.

3.2. Обработка событий

Проблема с правильным порядком обработки кликов мыши так же является довольно существенной. Дело в том, что Retained-mode фреймворки при обработке событий, полученных от пользователя, проходят путь от дочерних элементов к родительским, то есть в порядке обратном отрисовке. В современном дизайне не так много случаев, когда компоненты, обрабатывающие одно и тот же событие, находятся друг над другом. Но они встречаются, поэтому необходимо поддержать данный

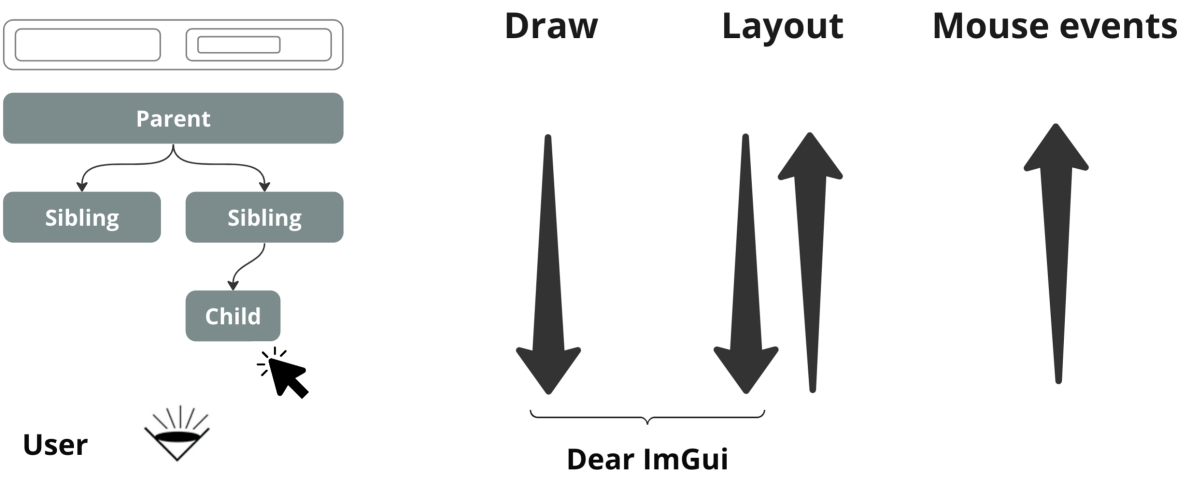

Рис. 5: Направление обхода дерева элементов.

функционал. Добиться требуемого поведения можно несколькими способами. Один из них предполагает добавление специальных флагов в структуру прямоугольника. Таким образом, при формировании сетки, для каждого отдельного фрагмента можно будет определить наличие дочерних фрагментов, способных обработать пришедшее событие. Однако это требует от разработчика явной классификации каждого фрагмента на активный и неактивный. При этом появление частично активного компонента сделает работу алгоритма некорректной. Существует альтернативное решение. Оно основано на той же идее формирования prepare list и последующей его обработке. После формирования списка команд снизу вверх мы можем пройти по нему в другую сторону и выставить состояния у активных элементов. Таким образом, разработчику необходимо указать лишь область, где может возникнуть конфликт обработки событий.

3.3. Композиция элементов

Главной сложностью в разработке GUI на языках, не имеющих специальных абстракция для него, является проработка архитектуры визуальных элементов [12]. Однако общепринятые подходы, широко используемые для решения обобщенных задач, оказываются неприменимы для написания графического интерфейса. Одной из таких ошибок яв-

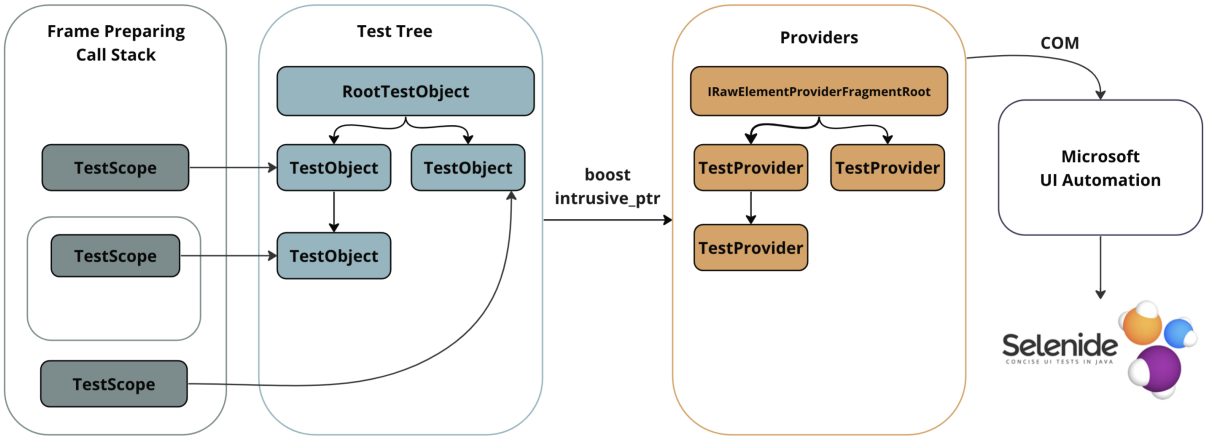

Рис. 6: Архитектура модуля для поддержки автотестов.

ляется полная инкапсуляция всех параметров, определяющих внешний вид, внутрь компонента. Правило состоит в том, что инкапсулировать необходимо сам алгоритм верстки, а не входные параметры. Дизайн макет может развиваться непредсказуемо. Поэтому, если на данный момент свойство используется только внутри элемента и не от чего не зависит, стоит предусмотреть его изменение снаружи. Второй распространенной ошибкой является связывание элементов через наследование, а не композицию. Разделение компонента на иерархичную структуру подразумевает, что каждая из частей будет независима от остальных. Но в случае с составным UI элементом гарантировать это невозможно.

3.4. Поддержка автотестов

Возможность покрытия пользовательского интерфейса тестами является необходимой функциональностью для высокоуровневого UI фреймворка. Большинство решений, существующих на данный момент, предоставляют такую возможность. Поэтому было необходимо реализовать взаимодействие фреймворка с одной из технологий управления интерфейсами. В нашем случае был выбран Microsoft UI Automation ввиду его универсальности и поддержке всех необходимых функций. Главная сложность заключается в том, что UI Automation работает с деревом объектов, что в свою очередь противоречит идеологии Immediate Mode

Gui, где оно полностью отсутствует. Для решения данной проблемы, была разработана система, которая во время прохода по стеку вызова функций подготовки кадра конструирует тестовые объекты, отвечающие актуальному состоянию приложения. Далее с помощью COM [9] интерфейса доступ к объектам передается в UI Automation, после чего они становятся доступны в инструменте автоматизированного тестирования, в нашем случае это Selenide.

3.5. Применение

Разработанный фреймворк, поддерживающий описанный функционал, был использован для создания набора графических элементов VKUI, которые в дальнейшем были применены в интерфейсе ВК Звонков.

4. Апробация

4.1. Условия эксперимента

Все измерения производились на ноутбуке со следующими параметрами: MacOs Sonoma 14.4, 6-ти ядерный процессор Intel Core i7-8550U 1,8 GHz, встроенная видеокарта Intel UHD Graphics 630, оперативная память 16 Гб, экран Retina display 3072×1920. Исследуемые программы собирались при помощи компилятора Apple Clang 15.

В качестве средства для профилирования была выбрана утилита powermetrics, встроенная в операционную систему. Она позволяет получать точную информацию о потреблении ресурсов CPU и GPU в удобном для последующей обработки виде. Все данные, полученные с помощью данной утилиты были хорошо воспроизводимы. При запуске измерений в одних и тех же сценариях результаты совпадали в пределах погрешности, в том числе после нескольких циклов перезагрузки компьютера.

Приложением, на котором проводились все тесты, является платформа видеоконференций ВК Звонки, полностью реализованная в двух вариантах: Qt Widgets и ImReady, и частично реализованная на Qt Quick 2, который использован для отдельного окна чатов. Таким образом, можно сравнить ImReady с Qt Widgets на полном наборе тестовых сценариев. А также сравнить ImReady с Qt Quick 2 на примере окна чата с большим количеством сложно структурированного содержимого, среди которого может находится элемент, требующий постоянной перерисовки. Для более краткой записи далее в работе под Qt5 будем подразумевать модуль Qt Widgets, а под Qt6 - Qt Quick 2.

4.2. Исследовательские вопросы

ImReady является достаточно низкоуровневым фреймворком, предоствляющим разработчику контроль над полным циклом рендеринга. В нем отсутсвует дорогостоящая абстракция над моделью обработки состояний. Благодаря этому полная перерисовка всего кадра становится

достаточно дешевой операцией. Не смотря на то, что ImReady содержит инструменты для частичного обновления кадра, в данной работе мы будем проводить измерения в режиме максимального потребления ресурсов, зафиксировав частоту обновления на уровне 60 кадров в секунду. В этом режиме чаще всего работают рассматриваемые высоконагруженные системы, в нашем случае - платформа видеоконференций. Таким образом в первую очередь мы интересуемся следующими вопросами:

RQ1 : оправдано ли использование ImReady в случае статических интерфейсов?

RQ2 : каково различие в потреблении ресурсов между Qt и ImReady в различных сценариях использования приложения?

RQ3 : можно ли применять ImReady для написания современных пользовательских интерфейсов?

4.3. Метрики

В данной работе использовались две характеристики из powermetrics:

1. CPU ms/s - затраченное процессорное время, усредненное за 1 секунду;
2. GPU ms/s - затраченное время графического процессора, усредненное за 1 секунду.

Период одной выборки (sample period), равный 10 секундам, был выбран таким образом, чтобы полученные данные имели погрешность не более 10% и удовлетворяли нормальному распределению. Для проверки последнего условия применялся критерий Шапиро—Уилка (The Shapiro-Wilk test) из библиотеки scipy. Для каждой серии измерений были достигнуты значения p-value больше выбранного уровня значимости - 5%. Откуда можно заключить, что нет основания отвергнуть нулевую гипотезу о нормальном распределении полученных данных. Стоит отметить, что при вычислении ресурсов графического процессора,

были взяты значения, отвечающие процессу исследуемого приложения, просуммированные с показаниями, относящимися к процессу оконного менеджера WindowServer. Это необходимо, так как в зависимости от реализации графического бэкенда, часть вычислений может быть делегировано этому системному процессу. При этом, во время проведения эксперимента, все остальные окна были закрыты.

Ниже представлена команда для получения выборки из 20-ти измерений по 10 секунд.

```
sudo powermetrics —samplers tasks —show-process-gpu —n 20 —i 10000
```

Для сравнительного анализа были выбраны следующие сценарии, некоторые из которых для лучшей воспроизводимости были автоматизированы с помошью инструмента Selenium.

1. Элемент 60 FPS - статический интерфейс, в котором присутсвует один легковесный компонент, обновляющийся с частотой 60 кадров в секунду.

2. Скроллинг списка - пролистывание длинного списка с элементами сложной структуры.

3. Vmoji - отрисовка анимированного виртуального персонажа, являющегося сложным графическим элементом, обновляющимся 60 кадров в секунду.

Дополнительно было проведено профилирование отдельных этапов формирования кадра. В Qt Quick 2 для этого были использованы специальные сигналы объекта QQuickWindow: beforeRendering, afterRendering, beforeSynchronizing, afterSynchronizing и frameSwapped. С их помощью можно определить на каком этапе находится поток Scene Graph, см. рисунок 2, и сколько времени заняло каждое из действий. В ImReady точки профилирования были выставлены явно в коде. Данное измерение проводилось для трех случаев: Qt6 (static), когда состояние дерева элементов не изменилось по сравнению с прошлым кадром, Qt6 (dynamic), когда в статическом интерфейсе чата находится один элемент с постоянным обновлением 60 кадров в секунду.

Статистическая обработка данных производилась с помощью языка Python. Для каждой выборки было рассчитано среднее значение и стандартное отклонение. Далее было произведено удаление выбросов по правигу 3σ. Погрешность рассчитывалась как произведение стандартного отклонения и коэффициента Стьюдента с выбранным уровнем значимости 5%.

4.4. Обсуждение результатов

Система	**Qt6 (static)**	**Qt6 (dynamic)**	**ImReady**
Выставление контектса	160±10	48±5	110±10
Синхронизация состояния	74±9	320±20	-
Подготовка кадра	-	-	700±10
Рендеринг	850±20	16400±100	390±20
Переключение кадра	30±10	30±10	50±10

Таблица 2: Процессорное время в микросекундах, требуемое для подготовки одного кадра

Рассмотрим результаты, описанные в таблице 2. Мы можем увидеть, что на утилитарные задачи такие, как выставление текущего OpenGL контекста в самом начале и обмен кадрами в frame bufer в конце, расходуется примерно одинаковое количество ресурсов во всех сценариях. Также заметим, что суммарное время цикла рендеринга Qt6 в случае статичесского интерфейса по порядку величины совпадает с временем подготовки кадра в ImReady. Qy6 выигрывает в данном случае за счет того, что он обновляет кадры значительно реже, чем ImGui в режиме фиксированного FPS. Таким образом, отвечая на вопрос RQ1, можно заключить, что в случае обычного интерфейса, который большую часть времени находится в неизменном состоянии, использование Qt6 более оправдано, так как этот фреймворк более развит и имеет ряд преимуществ. Тем не менее, как видно из случая Qt6 (dynamic), даже небольшого динамического элемента в общей структуре UI достаточно для того, чтобы значительно увеличить время рендеринга вплоть до

десятков миллисекунд. И такое потребление ресурсов может привести к снижению плавности анимации, потере кадров или подлагиваниям.

CPU	**Qt5**	**Qt6**	**ImReady**
Элемент 60 FPS	113 ± 2	228 ± 4	125 ± 2
Скроллинг списка	750 ± 30	610 ± 80	282 ± 40
Vmoji	240 ± 20	-	220 ± 20

Таблица 3: Потребление ресурсов CPU в различных сценариях, в мс/с

GPU	**Qt5**	**Qt6**	**ImReady**
Элемент 60 FPS	210 ± 20	250 ± 10	260 ± 10
Скроллинг списка	600 ± 20	200 ± 10	270 ± 20
Vmoji	420 ± 5	-	260 ± 20

Таблица 4: Потребление ресурсов GPU в различных сценариях, в мс/с

В таблицах 3 и 4 представлены результаты сравнения всех трех моделей и фреймворков, рассматриваемых в настоящей работе. Замметим, что ImReady показал самый стабильный результат по потреблению как GPU, так и CPU. Это закономерно вытекает из его узкоспециализированности на системах с перманентными трудоемкими графическими вычислениями. Не смотря на то, что в достаточно простом первом сценарии все системы показали себя одинаково, в более сложных сценариях ImReady потреблял в 3 раза меньше процессорного времени по сравнению с Qt6 и в 2 раза меньше времени графического процессора по сравнению с Qt5. Стоит также указать на то, что в сценарии пролистывания длинного списка Qt6 ожидаемо показал результат лучше, чем Qt5. Потребление CPU с учетом погрешности практически не изменилось, в то время как потребление GPU уменьшилось с 600 мс до 200 мс. В этом проявляется преимущество нового архитектурного решения (Scene Graph) по отношению к старому (QPainter).

Таким образом, вопрос RQ2 может быть разрешен следующим утверждением. Фреймворки Qt5 и Qt6 должны быть использованы в приложениях, где обновление дерева элементов интерфейса и ресурсоемкие

графические вычисления происходят достаточно редко и не занимают значительную часть от общего времени использования. При этом предпочтение стоит отдать более новой версии Qt, а именно модулю Qt Quick 2, ввиду использования более совершенной модели взаимодействия с GPU. В то же применение фреймворка ImReady оправдано в случае высоконагруженных систем.

Заключение

В ходе работы были получены следующие результаты.

1. Были выявлены особенности систем Qt Widgets и Qt Quick 2, подробно описаны применяемые архитектурные решения и их различия.

2. Был разработан UI фреймворк ImReady, поддерживающий адаптивную layout-систему и модель обработки событий.

3. Разработано средство тестирования графического интерфейса.

4. Было проведено сравнение производительности полученного фреймворка с Qt в различных сценариях использования приложения.

Таким образом, был получен фреймворк, с помощью которого можно создавать современный пользовательский интерфейс, состоящий из множества сложных элементов. При этом благодаря встраиванию в основной цикл рендеринга он позволяет минимизировать потребление ресурсов на отрисовку элементов управления. Однако его использование оправдано лишь в случае трудоемких вычислений на графическом процессоре, которые составляют основную часть потребляемых приложением ресурсов.

Список литературы

[1] Ahonen Santtu. Development hosts and targets in Qt 6.0. — 2020. — URL: https://www.qt.io/blog/qt6-development-hosts-and-targets.

[2] Cornut Omar. Dear ImGui: Bloat-free Immediate Mode Graphical User interface. — 2018. — URL: https://github.com/ocornut/imgui.

[3] Criswell Paul. The Evolution of the LGPL License Agreement. — Qt Virtual Tech Con. — 2020.

[4] Johan Thelin Jürgen Bocklage-Ryannel Cyril Lorquet. Qt6 QML Book. — 2021. — URL: https://www.qt.io/product/qt6/qml-book.

[5] Ltd. The Qt Company. Graphics View Framework. — 2024. — URL: https://doc.qt.io/qt-6/graphicsview.html.

[6] Ltd. The Qt Company. Qt Quick Scene Graph. — 2024. — URL: https://doc.qt.io/qt-6/qtquick-visualcanvas-scenegraph.html.

[7] Radich Quinn. Retained Mode Versus Immediate Mode. — 2019. — URL: https://learn.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode.

[8] Recommendation W3C Candidate. CSS Flexible Box Layout Module. — 2018. — URL: https://www.w3.org/TR/css-flexbox-1/flex-containers.

[9] Rob Costello Richard Maunsell Mitch Tulloch. Introduction to Microsoft Automation Solutions. — Redmond, Washington : Microsoft Press, 2014. — ISBN: 978-0-7356-9581-8.

[10] Serpa Yvens, Rodrigues Maria Andreia. A draw call-oriented approach for visibility of static and dynamic scenes with large number of triangles // The Visual Computer. — 2019. — 04. — Vol. 35.

[11] Smith Brett. A Quick Guide to GPLv3. — Free Software Found. — 2007. — http://www.gnu.org/licenses/quick-guide-gplv3.html.

[12] Роберт М. Чистая архитектура. Искусство разработки программного обеспечения. — "Издательский дом ""Питер""", 2018. — ISBN: 9785446107728. — URL: https://books.google.com/books?id=d6JSDwAAQBAJ.

[13] Швец Александр. Погружение в паттерны проектирования. — 2021.

[14] Шлее Марк. Qt 5.10. Профессиональное программирование на C++. — Санкт-Петербург : БХВ-Петербург, 2018. — P. 1072. — ISBN: 978-5-9775-3678-3.