Санкт-Петербургский государственный университет

Системное программирование

Группа 22.М05-мм

Системы рендеринга в Qt Framework и их производительность в различных комбинациях

Левков Данил Андреевич

Отчёт по учебной практике

Научный руководитель:
старший преподаватель каф. СП, М. Н. Смирнов

Санкт-Петербург
2023

Оглавление

Введение

В современном мире разработка нативных кроссплатформенных приложений до сих пор является сложной задачей. В первую очередь это касается высоконагруженных систем, использующих C++ в качестве основного языка. При этом самым значимым этапом является выбор инструментов и фреймворков, способных в полной мере обеспечить требуемый функционал.

В то время как в индустрии Web разработки существуют общепринятые и устоявшиеся подходы к реализации интерфейса, позволяющие разработчику не углубляться в детали реализации рендеринга и оптимизации, в создании нативных приложений, чтобы достичь аналогичного уровня качества интерфейса и при этом не испортить производительность, в подавляющем большинстве случаев необходимо проделать большое количество шагов таких как: выбор UI фреймворка (часто не одного), написание вручную кастомизированных элементов, отсутствующих в стандартном наборе, оптимизация и возможно полная переработка render движка. Последнее зачастую является самым сложным и при этом необходимым этапом в развитии проекта.

Самым популярным кроссплатформенным UI фреймворком на сегодняшний день является Qt. В данной работе будут рассмотрены плюсы и минусы его использования, а также тонкости различных встроенных типов отрисовки и сложности, возникающие при их сочетании.

1. Постановка задачи

Целью настоящей работы является определение наиболее оптимального способа написания графического интерфейса с использованием фреймворка Qt. Для достижения цели были поставлены следующие задачи:

1. Выполнить обзор предметной области - основных концепций компьютерной графики и их имплементации в рассматриваемом фреймворке.

2. Несколькими способами реализовать отображение всплывающих реакций в нативном приложении для видеоконференций ВК Звонки.

3. Измерить производительность для каждого из способов.

4. Выявить закономерности, позволяющие избежать потерь производительности.

2. Обзор

2.1. Способы разработки интерфейса на Qt

В Qt существует 2 основные парадигмы создания графического интерфейса: QtWidgets и QtQuick [1]. Первая предполагает написание C++ классов, в которых тип растеризации определяет QPaintDevice, вторая эмулирует веб-разработку с использованием QML (аналога CSS), где нет возможности управлять растеризацией. Мы не будем освещать QtQuick в данной работе, так как он использует совершенно другой подход к разработке интерфейса, который обособлен от всех других. Здесь же мы в первую очередь интересуемся эффектами, возникающими при сочетании различных типов отрисовки.

2.2. Софтверная растеризация

QPaintDevice является универсальной поверхностью для рисования векторной графики, текста и изображений, и может быть имплементирован в различных специальных классах, таких как QImage, QOpenGLPaintDevice, QWidget и QPrinter. Фактическое рисование происходит в QPaintEngine внутри QPaintDevice. Механизм растрового рисования — это программный растеризатор Qt, который используется при рисовании на QImage или QWidget. Его преимущество перед движком рисования OpenGL заключается в высоком качестве при включенном сглаживании и полном наборе функций. Именно он используется по умолчанию для отображения компонентов UI интерфейса. Однако, его главным минусом является высокая нагрузка на CPU при наличии хоть сколько-нибудь сложных элементов дизайна, таких как скругленные края, тени, градиенты и тд.

2.3. Растеризация на GPU

Одним из возможных вариантов выбора отрисовщика является QOpenGLPaintDevice, который использует для этого OpenGL 2.0. Доступ к

API библиотеки осуществляется через класс QOpenGLFunctions. Для отображения результатов рендеринга существует специальный виджет QOpenGLWidget. В этот виджете определены стандартные для OpenGL методы инициализации сцены initializeGL [3], изменения размеров окна resizeGL и метод вывода сцены paintGL. Работа с шейдерами производится с помощью класса QOpenGLShaderProgram. Перед тем, как осуществлять вызов функций OpenGL в Qt 5.8 необходимо провести их инициализацию. Данная процедура выполняется с помощью метода initializeGL класса QOpenGLFunctions. Обычно она вызывается в самом начале при инициализации сцены в методе initializeGL класса QOpenGLWidget. После чего производится установка различных параметров визуализации и компиляция шейдеров. В случае если компиляция пройдёт неудачно, то отобразить сцену согласно требованиям не получится.

Но чаще всего вместо функционала, описанного выше, используют QGraphicsScene/View, внутри которых скрыта инициализация OpenGL контекста. Для этого достаточно указать QOpenGLWidget в качестве viewport-а. При этом внутри QGraphicsWidget для отрисовки можно использовать стандартные OpenGL текстуры.

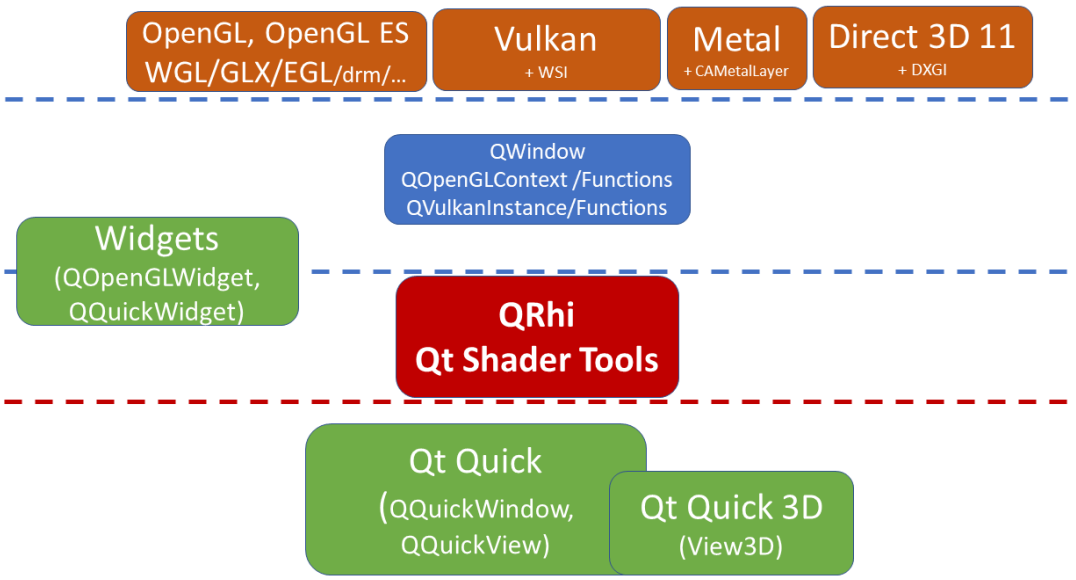

Рис. 1: Архитектура интерфейса работы с графикой в Qt6 [2]

3. Реализация

Рассмотрим применение вышеперечисленных способов на примере рабочей задачи: реализовать всплывающие реакции в приложении для видеоконференций. Везде далее будут освещаться лишь архитектурные особенности каждой из реализаций, так исходный код не может быть предоставлен ввиду требований работодателя. Далее будет описано поэтапное развитие решения и приведен его качественный анализ. Количественные оценки будут представлены в следующем разделе.

Сперва было принято решение реализовать поставленную задачу, используя виджеты. Так как дополнительно присутствовало требование по расположению реакций в определенном месте, зависящим от режима отображения участников, то необходимо было дополнительно продумать способ компоновки. Стандартным средством в Qt для управления взаиморасположением элементов друг над другом является QStackedLayout в режиме StackAll. Z-order в таком случае выставлялся автоматически в зависимости от положения в дереве родителей.

Однако при проверке данного способа выяснилось, что даже при отсутствии реакций общая производительность приложения снизилась. При более детальном профилировании и чтении исходного кода Qt, обнаружилось, что QStackedLayout кэширует результат рендеринга каждого слоя в QPixmap, а затем после синхронизации и наложения их друг на друга формируется конечная картина. Это приводило к тому, что каждый кадр, отрисованный на OpenGL, дополнительно обрабатывался на CPU, что очень негативно сказывалось на производительности. Для того, чтобы избежать негативное влияния QStackedLayout пришлось полностью отказаться от него в пользу самостоятельно реализованного layout менеджера без кэширования.

Тем не менее, сама по себе встроенная в QWidget отрисовка, хоть и удовлетворяла визуальным требованиям, но потребляла слишком много ресурсов центрального процессора. Поэтому было необходимо переписать имеющееся решение с использованием QGraphicsScene и QGraphicsView. Для этого была создана отдельная сцена, на которой с помощью

примитивов QOpenGLPaintDevice помещались реакции. При этом пересчет координат, анимаций и прозрачности производился явно. Таким образом получилось в два раза сократить потребление CPU, не пожертвовав при этом плавностью анимации.

Единственно проблемой последнего решения является то, что каждый новый объект QGraphicsView требует ещё одного полноценного OpenGL контекста. Большое количество контекстов на одно приложение в общем случае является плохой практикой. Дополнительной сложностью является задача сделать QOpenGLWidget прозрачным, которая не решается без изменения исходного кода Qt.

Таким образом мы пришли к финальному варианту реализации, а именно к отрисовке реакций в основном OpenGL контексте. Это потребовало дополнительного контроля со стороны layout менеджера, определяющего не только положение каждой из реакций, но также их z-order. Немаловажным преимуществом данного подхода является возможность использовать общие статические текстуры. Всё это позволило достичь максимальной производительности.

4. Эксперимент

4.1. Условия эксперимента

Все измерения производились на ноутбуке с 6-ядерным процессором Intel Core i7 2,6 GHz, встроенной видеокартой Intel UHD Graphics 630 1536 МБ, экраном Retina (3072 × 1920) и операционной системой MacOS 13.1. Потребление CPU и GPU определялось с помощью набора утилит XCode Instruments.

Тестирование производилось на непрерывной последовательности реакций, где каждая новая добавлялась через 200 миллисекунд. Частота обновления главного окна была зафиксирована на уровне 60 FPS. Само приложение было запущено в стандартном сценарии использования: 2 участника с включенными камерами, звук выключен, приложение развернуто на полный экран.

4.2. Метрики

С точки зрения пользователя, при фиксированном качестве получаемой картинки, самый главный параметр, влияющий на опыт использования, это нагрузка на процессор. Поэтому для количественной оценки результатов будем использовать метрики со следующими обозначениями:

- CPU – усредненная по времени нагрузка на центральный процессор, выраженная в процентах в расчёте на одно ядро, из которой вычиталась нагрузка, измеренная без отображения реакций;
- GPU - усредненная по времени нагрузка на графический процессор, выраженная в процентах в расчёте на все ядра, из которой вычиталась нагрузка, измеренная без отображения реакций;
- Memory – величина оперативной памяти в мегабайтах, дополнительно выделенной на отрисовку реакций.е.

Реализация	CPU, %	GPU, %	Memory, MB
QWidget + QStackedLayout (без реакций)	85.0	3.7	1.5
QWidget + QStackedLayout	110.6	3.9	2.2
QWidget + custom layout	47.1	4.3	1.9
OpenGL + multiple context	24.0	4.1	2.1
OpenGL + one context	8.2	2.8	1.2

Таблица 1: Сравнение потребляемых ресурсов при различных способах компоновки и рендеринга.

Во всех случаях из расчётов были убраны затраты ресурсов на дополнительную обработку информации о реакциях, не относящуюся напрямую к отрисовке.

4.3. Обсуждение результатов

Как видно из полученных данных, итоговое решение более чем в 10 раз превосходит изначальное по ключевому параметру. При этом потребление GPU и памяти во всех случаях оставалось примерно одинаковым. Так же заместим, что использование OpenGL не только не увеличило нагрузку на графический процессор, но в случае с единым контекстом, даже уменьшило её. То же самое можно сказать и про память. Объяснить данное поведение можно тем, что отрисовка примитивов графического интерфейса является для GPU крайне легкой задачей. Намного больше времени уходит на ожидание фрейма, прошедшего растеризацию на CPU. В течении этого времени графический процессор находится в бездействии и при этом удерживается стандартным композитором. Поэтому софтверная растеризация влияет на нагрузку не только центрального, но и графического процессора.

Отдельно хочется отметить высокое потребление CPU при использовании QStackedLayout в первом случае, когда в приложении не отображалась дополнительная графика, по сравнению с исходным вариантом. Ведь 85% нагрузки одного ядра или 14% нагрузки всего CPU тратилось исключительно на накладные расходы, что значительно сказываются на общей производительности приложения.

Заключение

В ходе работы были получены следующие результаты.

1. Изучены механизмы растеризации, использующиеся в Qt Framework.
2. Реализованы и проанализированы три различных подхода к организации и отрисовке графических элементов.
3. Измерена производительность каждого подхода в сочетании с различными методами композиции.
4. Выработаны принципы, позволяющие избежать потерь производительности.

Резюмируем описанные принципы, с помощью которых в случае рассмотренного примера удалось повысить производительность в 10 раз. Таким образом, при использовании нескольких движков для рендеринга в рамках одного приложения внутри одного окна стоит:

- исключить наложение друг на друга элементов с различной отрисовкой;
- не использовать дополнительные средства композиции наподобие QStackedLayout;
- использовать OpenGL в максимальном количестве случаев.

Список литературы

[1] Gois Joao Paulo, Batagelo Harlen Costa. Interactive Graphics Applications with OpenGL Shading Language and Qt. // SIBGRAPI Tutorials. — IEEE Computer Society, 2012.

[2] Ltd The Qt Company. Qt6 Documentation. Graphics. — URL: https://doc.qt.io/qt-6/topics-graphics.html.

[3] Shirley Peter. Fundamentals of computer graphics. — Wellesley, Mass : Peters, 2005. — ISBN: 1-56881-269-8.