Санкт-Петербургский государственный университет

Системное программирование

Группа 22.М05-мм

Разработка UI фреймворка, основанного на идеологии immediate mode

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. Однако, его использование в высоконагруженных системах, требующих большого количества графических вычислений, приводит к значительным потерям производительности. В данной работе будет рассмотрен процесс создания собственного UI фреймворка, потребляющего меньше вычислительных ресурсов и предоставляющего разработчикам сложных систем, возможность написания современного графического интерфейса.

1. Постановка задачи

Целью настоящей работы является определение наиболее оптимального способа построения графического интерфейса, основанного на парадигме Immediate Mode. Для достижения цели были поставлены следующие задачи:

1. Разработать собственный ImGui фреймворк, учитывающий ограничения данного подхода по сравнению с привычными способами формирования UI.
2. Разработать адаптивную layout-систему и модель обработки событий.
3. Добавить возможность тестирования графического интерфейса.
4. Сравнить производительность полученного фреймворка с Qt.

2. Обзор

2.1. Описание проблемы

Исследуемая идеология является противоположностью более привычного Retained-mode GUI [2], который используется в подавляющем большинстве существующих фреймворков и подразумевает сохранение состояния визуальных объектов (виджетов) в некой структуре данных (чаще всего в дереве) с целью предотвращения перерисовки неизменных областей. На первый взгляд оптимизация оправдана и в подавляющем большинстве случаев так и есть. 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 полный цикл перерисовки — это очень дорого. Плюс ко всему мы постоянно будем перестраивать дерево состояний.

2.2. 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 тем не менее кэширует состояния элементов, а ImGui нет.

2.3. MVVM

Данный шаблон проектирования призван 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 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 подписывается, 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.

В ImGui очень удобно использовать идеологию MVVM, отделяющую состояние интерфейса (ViewModel) от его отображения (View), так как у последней нет состояния. View является по сути функцией, принимающей параметры из ViewModel и возвращающей готовый визуальный компонент. Поэтому нет необходимости реализовывать механизм подписок и обновлений состояния в виде коллбеков или сигналов-

слотов Qt. После изменения состояния внутри ViewModel, View сама обновится прочитав его на следующем кадре.

2.4. Имеющиеся решения

Базовые подходы и решения были взяты из открытой библиотеки Dear ImGui [1], используемой главным образом для создания пользовательского интерфейса в играх. Она вносит минимальный вклад в основное потребление ресурсов компьютера, встраиваясь в render loop приложениея. Сравнивая её с более высокоуровневыми UI фреймворками, можно прийти к выводу, что библиотека является совершенно уникальным явлением с узкой, но всегда имеющей место, областью 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 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. Реализация

3.1. Адаптивный layout

Под адаптивной системой верстки [3] 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 добирающемуся, запутанному коду, в который будет очень 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 нашему изначальному желанию сделать однопроходный Immediate Mode GUI, не обремененный деревом состояний. 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 установлен-

ному правилу. Второй – это выравнивание элементов разных размеров, относительно 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 ImGui 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 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 draw list и последующей его обработке. После формирования списка команд снизу вверх мы можем пройти по нему в другую сторону и вы-

ставить 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. Однако общепринятые подходы, широко используемые для решения обобщенных задач, оказываются неприменимы для написания графического интерфейса. Одной из таких ошибок является полная инкапсуляция всех параметров, определяющих внешний вид, внутрь компонента. Правило состоит в том, что инкапсулировать необходимо лишь сам алгоритм верстки, а не входные параметры. Дизайн макет может развиваться непредсказуемо. Поэтому даже 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 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 ImGui фреймворк, поддерживающий описанный функционал, был использован для создания набора графических элементов VKUI, которые в дальнейшем были применены в интерфейсе ВК Звонков.

4. Эксперимент

4.1. Условия эксперимента

Все измерения производились на ноутбуке с процессором Intel Core i7-8550U 1,8 GHz, встроенной видеокартой Intel UHD Graphics 630, оперативной памятью 16 Гб и операционной системой Windows 10. Потребление CPU и GPU оценивалось на стандартном тестовом наборе шагов, используемых для мониторинга производительности приложения. При этом тесты запускались для двух разных сборок: старой, чей интерфейс реализован на Qt Framework, и новой, в которой использовалось решение, обсуждаемое в данной работе. Измерения производились в нескольких режимах:

- статический интерфейс, отображались только элементы управления в неизменном состоянии, видео и анимированные аватары отсутсвовали;
- динамический интерфейс, в котором производилось активное использование элементов управления, видео и анимированные аватары отсутствовали;
- отрисовка графической сцены, состоящей из 20-ти анимированных аватаров, обновляемых с частотой 60 кадров в секунду.

4.2. Метрики

Основным требованием к фреймворку ImGui было снижение потребления GPU в сценариях активного использования приложения. Также с точки зрения пользователя, при фиксированном качестве получаемой картинки, немаловажным является нагрузка на центральный процессор, так как это влияет на общее быстродействие системы и других приложений. Для количественной оценки результатов будем использовать метрики со следующими обозначениями:

Режим тестирования	CPU, %		GPU, %	
	ImGui	Qt	ImGui	Qt
Статический интерфейс	5 ± 3	3 ± 1	4 ± 2	< 1
Динамический интерфейс	6 ± 3	15 ± 5	10 ± 3	17 ± 3
Графическая сцена (отдельно)	10 ± 6		30 ± 3	
Графическая сцена (внутри приложения)	34 ± 9	60 ± 10	44 ± 8	72 ± 9

Таблица 1: Сравнение потребляемых ресурсов для различных фреймворков.

- CPU – усредненная по времени нагрузка на центральный процессор;
- GPU - усредненная по времени нагрузка на графический процессор.

4.3. Обсуждение результатов

Как видно из полученных данных, добавление графических элементов в интерфейс может очень сильно сказаться на общей производительности интерфейса. Это особенно заметно в случае с Qt, когда потребление ресурсов сложного интерфейса, не равно сумме его составных частей: элементов управления и графической сцены по отдельности. В этом случае на первый план выходит влияние эффектов композиции кадра. В случае Qt это композиция софтварно растерезованного пользовательского интерфейса и GPU текстур. ImGui не подвержен данному эффекту, так как в его случае растерезация интерфейса встраивается в основной цикл рендеринга. Ввиду чего ImGui практически не вносит дополнительных расходов в процесс подготовки кадра. Заметим также, что в случае статического интерфейса Qt показывает себя намного лучше как по GPU, так и по CPU.

Заключение

В ходе работы были получены следующие результаты.

1. Был разработан собственный ImGui фреймворк, поддерживающий адаптивную layout-систему и модель обработки событий.

2. Разработано средство тестирования графического интерфейса.

3. 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, сопровождающих пользовательский интерфейс.

Список литературы

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

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

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