Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

**Высшая школа кибербезопасности**

**КУРСОВАЯ РАБОТА**

**Оптимизация моделирования решения Кубика Рубика 3x3**

по дисциплине «Языки программирования»

Выполнили

студенты гр. 5151003/30002 И. А. Штарев

Д. А. Плотников

Руководитель

асс. преподавателя Е. М. Орел

«19» декабря 2024 г.

Санкт-Петербург

2024

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc185433742)

[1. ТЕОРЕТИЧЕСКИЕ ИССЛЕДОВАНИЯ 4](#_Toc185433743)

[1.1. ВОЗМОЖНОСТИ ДЛЯ МАШИННО-НЕЗАВИСИМОЙ ОПТИМИЗАЦИИ 4](#_Toc185433744)

[1.2. ВОЗМОЖНОСТИ ДЛЯ АЛГОРИТМИЧЕСКОЙ ОПТИМИЗАЦИИ 6](#_Toc185433745)

[2. МАШИННО-НЕЗАВИСИМАЯ ОПТИМИЗАЦИЯ 7](#_Toc185433746)

[2.1. ОПТИМИЗАЦИЯ БУФЕРОВ 7](#_Toc185433747)

[2.2. ОПТИМИЗАЦИЯ ГЛАВНОГО ОКНА 8](#_Toc185433748)

[2.3. ОБЩИЕ МАШИННО-НЕЗАВИСИМЫЕ ОПТИМИЗАЦИИ 9](#_Toc185433749)

[3. АЛГОРИТМИЧЕСКАЯ ОПТИМИЗАЦИЯ 10](#_Toc185433750)

[3.1. ОЦЕНКА АЛГОРИТМИЧЕСКОЙ СЛОЖНОСТИ 10](#_Toc185433751)

[3.2. ВНЕДРЕНИЕ АЛГОРИТМИЧЕСКОЙ ОПТИМИЗАЦИИ 11](#_Toc185433752)

[4. МАШИННО-ЗАВИСИМАЯ ОПТИМИЗАЦИЯ 13](#_Toc185433753)

[5. АНАЛИЗ ВНЕДРЕННЫХ ОПТИМИЗАЦИЙ 14](#_Toc185433754)

[ЗАКЛЮЧЕНИЕ 18](#_Toc185433755)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 20](#_Toc185433756)

[ПРИЛОЖЕНИЕ А. ЛИСТИНГИ ФУНКЦИЙ ОТРИСОВКИ ОКОН 22](#_Toc185433757)

[ПРИЛОЖЕНИЕ Б. ЛИСТИНГИ КЛЮЧЕВЫХ ФУНКЦИЙ АЛГОРИТМА РЕШЕНИЯ КУБИКА 31](#_Toc185433758)

# ВВЕДЕНИЕ

В современных вычислительных системах важно учитывать как эффективность алгоритмов, так и производительность программного обеспечения. Одной из задач, которая требует оптимизации, является решение кубика Рубика с использованием графических библиотек и алгоритмов для его моделирования. В предыдущей курсовой работе была реализована программа для решения кубика Рубика с использованием языка C и библиотеки OpenGL с GLFW. Однако, несмотря на свою функциональность, программа имеет возможности для улучшения как с точки зрения алгоритмической, так и машинно-независимой оптимизации, что обеспечит ускорение выполнения и улучшение пользовательского опыта.

Целью данной курсовой работы является оптимизация исходной программы, созданной в рамках предыдущей работы. В процессе оптимизации будет проведено профилирование программы с целью выявления узких мест и неэффективных участков кода. На основе полученных данных будут предложены и реализованы как алгоритмические, так и машинно-независимые оптимизации, что позволит снизить вычислительные затраты и улучшить общую производительность. Важным аспектом будет также оценка изменения алгоритмической сложности программы и сравнение результатов до и после оптимизаций.

В рамках выполнения работы будет проведен анализ исходной программы с применением методов профилирования и оценки сложности алгоритмов. Будет предложено и реализовано несколько вариантов оптимизаций, среди которых особое внимание будет уделено алгоритмическим улучшениям и кодовым оптимизациям, не зависящим от конкретной архитектуры. Программная система будет проанализирована с точки зрения полученных результатов, а также приведены графики, на которых будут отражены изменения в производительности программы после каждого этапа оптимизации.

Цель работы: получение навыков и умений в анализе производительности программного обеспечения, реализации алгоритмических и машинно-независимых оптимизаций, а также в оценке их эффективности на практике.

Задачи:

1. Провести машинно-независимые оптимизации программы, включая устранение избыточных операций, управление циклами и другие улучшения структуры кода, направленные на повышение эффективности ее выполнения.
2. Реализовать алгоритмическую оптимизацию программы, направленную на сокращение времени выполнения части кода, отвечающей за поиск решения кубика.
3. Провести анализ и тестирование всех внедренных оптимизаций и представить результаты в виде графиков и оценок изменений производительности.

# 1. ТЕОРЕТИЧЕСКИЕ ИССЛЕДОВАНИЯ

# 1.1. ВОЗМОЖНОСТИ ДЛЯ МАШИННО-НЕЗАВИСИМОЙ ОПТИМИЗАЦИИ

Одним из ключевых аспектов оптимизации производительности графических приложений является эффективное использование ресурсов графического процессора, таких как буферы и массивы вершин. В контексте OpenGL и других графических API процесс работы с буферами может существенно повлиять на производительность визуализаций, особенно в случае динамичных сцен, где необходимо часто обновлять геометрические данные. В традиционных подходах к рендерингу каждый кадр может включать полное пересоздание буферов и пересылку данных на видеокарту, что накладывает значительные затраты на производительность, особенно при высоком числе объектов или сложных сценах. Это связано с высокой стоимостью операций, связанных с выделением памяти и загрузкой данных в видеопамять, что увеличивает общие задержки и нагрузку на систему.

Для повышения производительности разработчики графических приложений ищут способы минимизации количества операций с видеопамятью. Одним из наиболее эффективных подходов является использование предварительной инициализации буферов с последующим их обновлением. Идея заключается в том, чтобы создать буферы и массивы вершин один раз в начале рендеринга и в дальнейшем только изменять их содержимое, что позволяет избежать лишней работы с выделением памяти и загрузкой данных в каждый кадр. В OpenGL это можно реализовать с помощью функции glBufferSubData, которая позволяет обновлять часть данных в уже существующих буферах без необходимости их повторного выделения или полной перезаписи. Это значительно снижает накладные расходы на операции с памятью и ускоряет работу приложения.

Подход, при котором данные обновляются в существующих буферах, а не перезаписываются целиком, является примером машинно-независимой оптимизации, так как его применение не зависит от особенностей конкретной аппаратной платформы или архитектуры. Вместо того чтобы на каждом кадре пересоздавать буферы и заново загружать данные в видеопамять, программа обновляет только измененные части данных, что сводит к минимуму количество операций, требующих взаимодействия с видеокартой. Это особенно эффективно при работе с динамическими объектами, такими как анимации, где геометрия меняется постепенно и частично. Такой подход позволяет существенно снизить время отклика и повысить общую производительность, что делает его важным инструментом в арсенале оптимизаций для графических приложений.

Также, в актуальной версии программы ресурсы главного окна продолжают обрабатываться даже при его неактивности, что приводит к избыточному потреблению памяти и снижению общей эффективности. В перспективе можно реализовать «заморозку» главного окна на время работы вспомогательных окон, при которой его ресурсы временно освобождаются, а обработка приостанавливается. Это позволит снизить пиковое потребление памяти и оптимизировать распределение ресурсов между окнами программы.

Наконец, среди потенциальных улучшений можно выделить оптимизацию цепочек условий, таких как switch/case, за счет поднятия наиболее вероятных ветвей вверх для сокращения времени выполнения. Также существуют возможности для устранения избыточных операций, например, через удаление общих подвыражений, свертку констант и переприсваивание переменных. Перспективным направлением является оптимизация циклов, включая управление переменными индукции и вынесение инвариантного кода, что позволит уменьшить издержки на повторяющиеся вычисления. Применение этих техник может способствовать снижению вычислительных затрат и улучшению общей производительности программы.

# 1.2. ВОЗМОЖНОСТИ ДЛЯ АЛГОРИТМИЧЕСКОЙ ОПТИМИЗАЦИИ

Для оптимизации алгоритма работы функции search, которая является ключевым элементом программы и определяет ее основную алгоритмическую сложность, предлагается изменить порядок обработки возможных вариантов вращений. На данный момент функция обрабатывает массив возможных вращений в фиксированном порядке, последовательно перебирая их в цикле. Если первое вращение из массива не приводит к решению, алгоритм смещается к следующему элементу и продолжает поиск, пока не найдет успешную комбинацию. Такой подход может быть неэффективным, так как в случае неудачно выбранного порядка вращений поиск требует значительного количества итераций.

Перспективным направлением для улучшения является динамическое перемешивание массива возможных вращений в процессе поиска. Это позволит разнообразить порядок их обработки и увеличит вероятность нахождения успешного решения быстрее. Для реализации этого подхода предлагается использовать функцию shuffle, которая будет выполнять частичное перемешивание массива с определенной периодичностью. Такой подход обеспечит регулярное изменение порядка вращений, минимизируя вероятность «застревания» на неэффективной последовательности.

Функция перемешивания будет выполнена с постоянной сложностью O(1), так как она затрагивает фиксированное количество элементов массива. Это позволяет сохранить общую сложность функции search, одновременно улучшив ее эффективность на практике. Дополнительно, для этапов, на которых перемешивание не обеспечивает должной оптимизации (например, для десятого этапа), может быть подобрана вручную оптимальная последовательность вращений, которая сокращает время выполнения. Этот подход нацелен на уменьшение числа итераций для нахождения решения и улучшение средней производительности алгоритма.

# 2. МАШИННО-НЕЗАВИСИМАЯ ОПТИМИЗАЦИЯ

# 2.1. ОПТИМИЗАЦИЯ БУФЕРОВ

В процессе оптимизации кода программы для решения кубика Рубика была внедрена предварительная инициализация буферов с последующим их обновлением. Этот подход позволил значительно улучшить производительность графического интерфейса, использующего OpenGL и GLFW. Основная идея заключалась в том, чтобы создать и инициализировать буферы один раз при запуске программы, а затем обновлять их содержимое по мере необходимости, вместо создания новых буферов каждый раз при отрисовке элементов.

Для реализации данной оптимизации были созданы функции инициализации буферов, которые выполняют все необходимые операции по созданию и настройке буферов и вершинных массивов. Эти функции вызываются один раз при инициализации окна, что позволяет избежать лишних затрат на создание и настройку буферов в процессе работы программы. В результате, каждый элемент интерфейса, будь то кнопка, квадрат или линия, использует уже существующие буферы, что значительно ускоряет процесс отрисовки.

После инициализации буферов была реализована функция обновления их содержимого. Эта функция использует команду glBufferSubData, которая позволяет обновлять данные в уже существующем буфере, не создавая новый. Таким образом, при изменении состояния элементов интерфейса, например, при нажатии кнопок или изменении цвета квадратов, обновляются только данные в буфере, что значительно снижает нагрузку на систему и улучшает общую производительность.

Оптимизация также включала в себя модификацию функций отрисовки элементов интерфейса, таких как drawButton, drawSquare и drawStrokeLine, для использования предварительно инициализированных буферов. Эти функции были изменены таким образом, чтобы они принимали в качестве параметров указатели на буферы и обновляли их содержимое по мере необходимости. Это позволило избежать повторной инициализации буферов и сократить количество вызовов OpenGL, что положительно сказалось на производительности.

Также было уменьшено количество вызовов gltInit и gltTerminate из текстовой библиотеки GL Text. Оптимизация заключается в том, чтобы инициализация и завершение библиотеки GLT (GL Text) выполнялись один раз за весь жизненный цикл программы, а не каждый кадр. Это позволяет избежать лишних затрат на повторную инициализацию и освобождение ресурсов, что может значительно улучшить производительность. В результате вызовы gltInit и gltTerminate переносятся из основного цикла отрисовки в начало и конец программы соответственно (для каждого окна по отдельности), что обеспечивает более эффективное использование ресурсов и уменьшает накладные расходы на управление памятью.

# 2.2. ОПТИМИЗАЦИЯ ГЛАВНОГО ОКНА

В ходе выполнения профилирования программы с использованием вспомогательных окон, таких как окно ввода кубика, было выявлено, что при открытии таких окон происходит рост потребления памяти. Это связано с тем, что даже при переключении контекста и временной неактивности главного окна его ресурсы продолжают обрабатываться и потреблять системные ресурсы. Такое поведение отрицательно сказывается на общей эффективности программы, поскольку параллельная обработка данных для обоих окон увеличивает нагрузку на оперативную память.

Для решения этой проблемы была реализована оптимизация, заключающаяся в «замораживании» главного окна на время работы вспомогательного. В рамках данной оптимизации главному окну временно присваивается состояние, при котором его ресурсы не обрабатываются (isAuxWindowActive == 1), что позволяет освободить занимаемую память. Такая мера не только устраняет избыточные затраты ресурсов, но и снижает общее потребление памяти в моменты, когда активным является только одно окно. Замораживание главного окна на время работы вспомогательных окон не повлияло на функциональность приложения, сохранив возможность корректной обработки пользовательского ввода.

# 2.3. ОБЩИЕ МАШИННО-НЕЗАВИСИМЫЕ ОПТИМИЗАЦИИ

В рамках работы также были реализованы различные общие оптимизации, направленные на повышение эффективности программы за счет упрощения и реструктуризации кода. Эти оптимизации включали в себя работу с цепочками условий, оптимизацию циклов, устранение избыточных операций и работу с повторяющимися выражениями. Например, наиболее вероятные условия были подняты вверх в цепочках switch/case, что позволяет быстрее достичь верного ветвления и снизить количество ненужных проверок. Это особенно важно в часто вызываемых функциях, где даже незначительное ускорение каждой итерации приводит к заметной экономии времени.

Дополнительно применялись техники, такие как свертка констант, устранение избыточных операций загрузки и сохранения, а также удаление общих подвыражений. Эти меры позволили снизить количество операций, выполняемых процессором, за счет более эффективного использования ресурсов и минимизации избыточных вычислений. Вынесение инвариантного кода из циклов и управление переменными индукции также способствовали оптимизации выполнения повторяющихся операций. Несмотря на то, что эти изменения не оказали значительного влияния на общую производительность, их совокупный вклад улучшил отклик программы и снизил ее накладные расходы. Применение перечисленных оптимизаций позволяет сделать код программы более компактным и предсказуемым для компилятора, что повышает эффективность выполнения.

Листинги ключевых функций представлены в приложении А.

# 3. АЛГОРИТМИЧЕСКАЯ ОПТИМИЗАЦИЯ

# 3.1. ОЦЕНКА АЛГОРИТМИЧЕСКОЙ СЛОЖНОСТИ

Алгоритмическая сложность данного кода, решающего кубик Рубика, можно оценить, анализируя основные функции и их взаимодействие. Основные аспекты, влияющие на сложность, включают:

Функции вращения граней кубика:

* rotateSideBy90: вращает одну грань кубика. Сложность O(1), так как выполняется фиксированное количество операций.
* rotateOtherSides: вращает линии на других гранях кубика. Сложность O(1), так как выполняется фиксированное количество операций.

Функции вращения кубика:

* cubeFrontRotate, cubeBackRotate, cubeUpRotate, cubeDownRotate, cubeLeftRotate, cubeRightRotate: каждая из этих функций вызывает rotateOtherSides и rotateSideBy90, что делает их сложность O(1).
* Функция cubeDoOp: вызывает одну из функций вращения кубика. Сложность O(1).
* Функция randCube: выполняет n случайных вращений кубика. Сложность O(n).

Функции проверки состояния кубика:

* checkIfCrossFront, checkIfCrossSides, checkIfCornersFront, checkIfMiddleRow, checkIfBackCross, checkIfBackSide, checkIfBackCorners, checkIfBackRow: все эти функции имеют сложность O(1), так как проверяют фиксированное количество элементов.
* Функция doCheck: вызывает одну из функций проверки состояния кубика. Сложность O(1).
* Функция checkAllCube: вызывает doCheck до 8 раз. Сложность O(1).
* Функция search: основная функция поиска решения. Выполняет поиск в ширину (BFS) с ограничением по памяти. Функция в цикле перебирает различные варианты вращений кубика, пока либо не закончится память, либо не найдется комбинация ходов, сложность O(n).
* Функция cubeSolve: вызывает функцию search до 19 раз (SMALLSTEPSCOUNT).
* Сложность будет O(19 ∙ n) = O(n), так как константа 19 не влияет на асимптотическую сложность.

Таким образом, основная алгоритмическая сложность данного кода определяется функцией search и составляет O(n), где n – количество итераций поиска решения.

# 3.2. ВНЕДРЕНИЕ АЛГОРИТМИЧЕСКОЙ ОПТИМИЗАЦИИ

После оценки алгоритмической сложности было принято решение оптимизировать работу функции search, так как именно она определяет основную алгоритмическую сложность всего кода. При анализе возможности различных оптимизаций было замечено, что порядок возможных вариантов вращения в массиве возможных вращений на каждом шаге влияет на скорость поиска решения каждого из этапов сборки. Это связано с тем, что алгоритм в цикле последовательно перебирает вращения, смещаясь в случае неудачи на 1 и начиная поиск решения с нового вращения. В результате алгоритм находит комбинацию, которая собирает нужный этап, однако на это, в случае неудачно выбранного порядка вращений, может понадобиться значительное время. Для создания более выгодных комбинаций была использована перестановка элементов в массиве вращений в случайном порядке на каждой пятнадцатой итерации цикла поиска решения. Для этого была написана функция shuffle, которая в цикле переставляет 5 или 3 чисел местами, в зависимости от количества вариантов вращений. Сложность O(1), поэтому на общую сложность функции search существенного влияния не оказывает. Число 15 неслучайно, так как это наименьшее число кратное и 5 и 3. Пять – это количество возможных вращений первых 16 этапов сборки, а три – последних трех. В результате перемешивание происходит по числу возможных вариантов вращения, которое используется во вложенном цикле перебора вращений, что обеспечивает перемешивание массива с вариантами вращения каждый раз кратный 5 или 3. Делать перемешивание 30, 45, 60 и так далее раз кратным 15 увеличивает время поиска решения, так как чем меньше перемешиваний, тем меньше они оказывают влияние на работу поиска. Среднее время сборки всех этапов после внедрения системы перемешиваний уменьшилось, так как программа начала делать меньше итераций для поиска решения, однако десятый этап все равно собирался в несколько раз дольше на фоне всех остальных, поэтому для него вручную была подобрана оптимальная последовательность вращений, которая начала быстрее приводить к поиску решения этапа.

Листинги ключевых функций представлены в приложении Б.

# 4. МАШИННО-ЗАВИСИМАЯ ОПТИМИЗАЦИЯ

В ходе выполнения работы была применена машинно-зависимая оптимизация компилятора GCC с использованием флага -Ofast. Этот флаг представляет собой агрессивный уровень оптимизации, включающий все стандартные оптимизации уровня -O3, а также дополнительные улучшения. Ключевое отличие заключается в том, что -Ofast игнорирует строгие ограничения стандарта языка C, такие как соблюдение спецификации IEEE для вычислений с плавающей точкой, что позволяет добиться максимальной производительности за счет упрощения некоторых вычислительных операций. Оптимизация выполняется за счет использования инструкций процессора для обработки данных, уменьшения числа вызовов функций, предсказания ветвлений, а также устранения избыточного кода.

Применение -Ofast может оказать значительное влияние на производительность программы, особенно в задачах с высокой вычислительной нагрузкой, таких как работа функции поиска решения кубика Рубика. В частности, использование этого флага может ускорить работу алгоритма за счет более эффективного использования ресурсов процессора и памяти. Однако важно учитывать, что оптимизация -Ofast может быть неприменима для некоторых задач, требующих строгой точности вычислений или корректности в соответствии с установленными стандартами. В данном случае ее использование было оправдано, так как оно способствовало значительному улучшению времени выполнения без нарушения логики работы программы.

# 5. АНАЛИЗ ВНЕДРЕННЫХ ОПТИМИЗАЦИЙ

В ходе выполнения работы были реализованы три машинно-независимые оптимизации, направленные на улучшение производительности программы, использующей OpenGL для рендеринга кубика Рубика, а также алгоритмическая оптимизация.

Первая оптимизация заключалась в инициализации буферов и массивов вершин только один раз в начале рендеринга с последующим обновлением их содержимого через функцию glBufferSubData, а также уменьшении количества вызовов gltInit и gltTerminate из текстовой библиотеки GL Text. Ранее, до применения этой оптимизации, буферы пересоздавались каждый кадр, что приводило к значительным накладным расходам на память и замедляло работу приложения.

Для оценки эффективности оптимизации буферов было проведено сравнение времени отрисовки одного кадра как для главного окна программы, так и для окна ввода кубика. Измерения проводились на выборке из 100 кадров, что обусловлено минимальными колебаниями результатов, при которых разница во времени рендеринга остается незначительной. Это позволило получить достаточно точные данные для оценки влияния изменений на производительность.

На основе полученных данных были построены два графика. Первый график (рисунок 1) отображает время отрисовки 100 различных кадров главного окна до и после оптимизации. На графике видно, как улучшения привели к снижению времени рендеринга в среднем на 35% с оптимизацией -Ofast и на 20% без ее использования. Второй график (рисунок 2) демонстрирует аналогичное сравнение для окна ввода кубика, где также отмечено снижение времени отрисовки кадров после применения оптимизаций в среднем на 70% с оптимизацией -Ofast и на 31% без ее использования. Эти графики наглядно показывают, как оптимизации снизили время, необходимое для рендеринга, что в свою очередь повысило отзывчивость программы и улучшило пользовательский опыт.



Рисунок 1 – Производительность главного окна

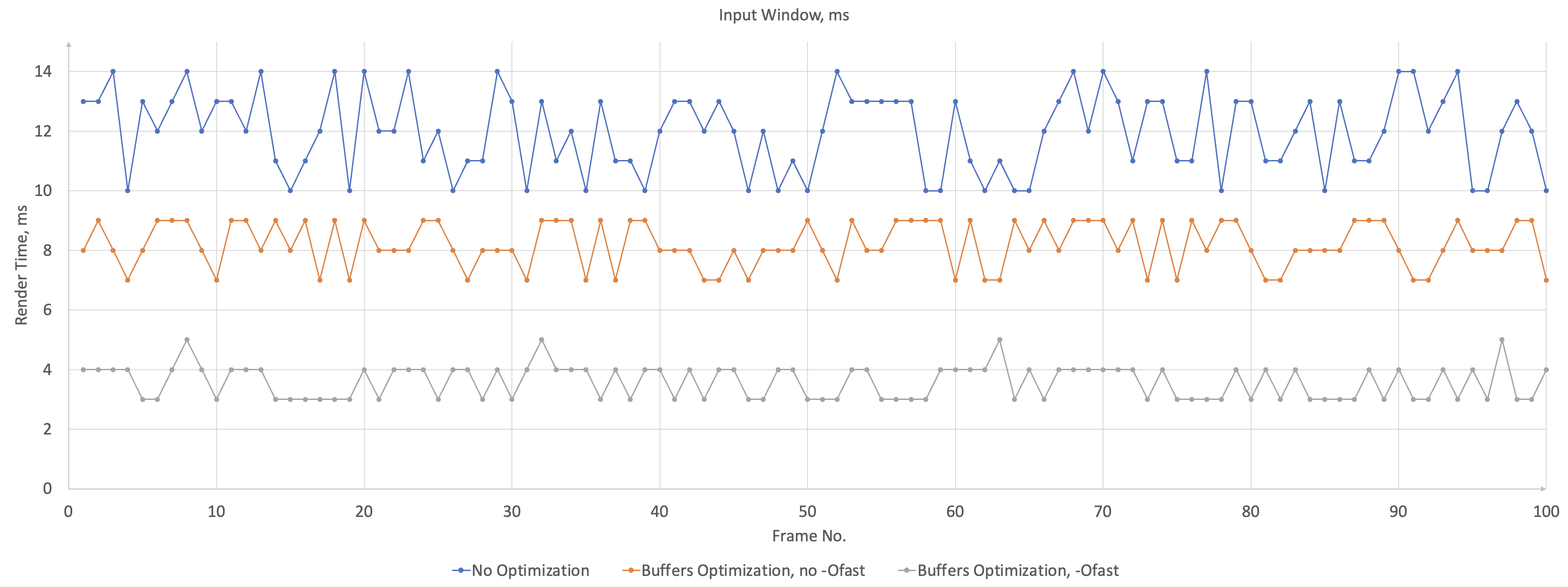


Рисунок 2 – Производительность окна ввода кубика

Для анализа оптимизации главного окна было выполнено профилирование программы с помощью встроенного профилировщика IDE Visual Studio. В изначальной версии программы при открытии вспомогательных окон ресурсы главного окна продолжали обрабатываться, несмотря на его неактивное состояние. Это приводило к увеличению общего потребления памяти, так как одновременно обрабатывались оба окна. Данный эффект был зафиксирован в ходе профилирования, результаты которого представлены на рисунке 3. На графике заметен рост потребления памяти в момент вызова функции fillCubeFromUserInput, что связано с обработкой главного окна наравне с окном ввода кубика.

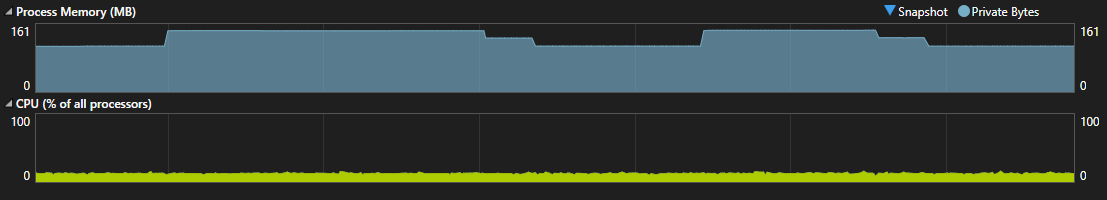


Рисунок 3 – Потребление памяти программой до оптимизации главного окна

После внедрения оптимизации профилирование показало снижение пикового потребления памяти во время вызова окна ввода кубика. На рисунке 4 видно, что при активации окна ввода общее потребление памяти уменьшилось примерно на 7%.

A black and blue screen with a green line

Description automatically generated with medium confidence

Рисунок 4 – Потребление памяти программой после оптимизации главного окна

В ходе анализа общих машинно-независимых оптимизаций было выявлено, что они не оказали значительного влияния на общую производительность программы. Тем не менее, их вклад проявился в виде снижения накладных расходов на уровне отдельных операций. Это позволило улучшить читаемость кода, а также повысить эффективность его обработки компилятором. В совокупности эти изменения создали более оптимизированную и устойчивую базу для дальнейших улучшений программы.

Завершающей машинно-независимой оптимизацией стала алгоритмическая. В результате внедрения системы рандомизации возможных вариантов вращений для всех этапов сборки удалось сократить время поиска решения на 92% с оптимизацией -Ofast и на 55% без ее использования. Тестирование проводилось путем создания 1000 случайных кубиков Рубика и замером времени, которое необходимо алгоритму на поиск решения. Результаты замеров представлены на графике (рисунок 5). Алгоритмическая сложность алгоритма не изменилась, но благодаря перемешиванию вариантов вращений в среднем делается гораздо меньше итераций перебора при поиске, что приводит к ускорению работы алгоритма.

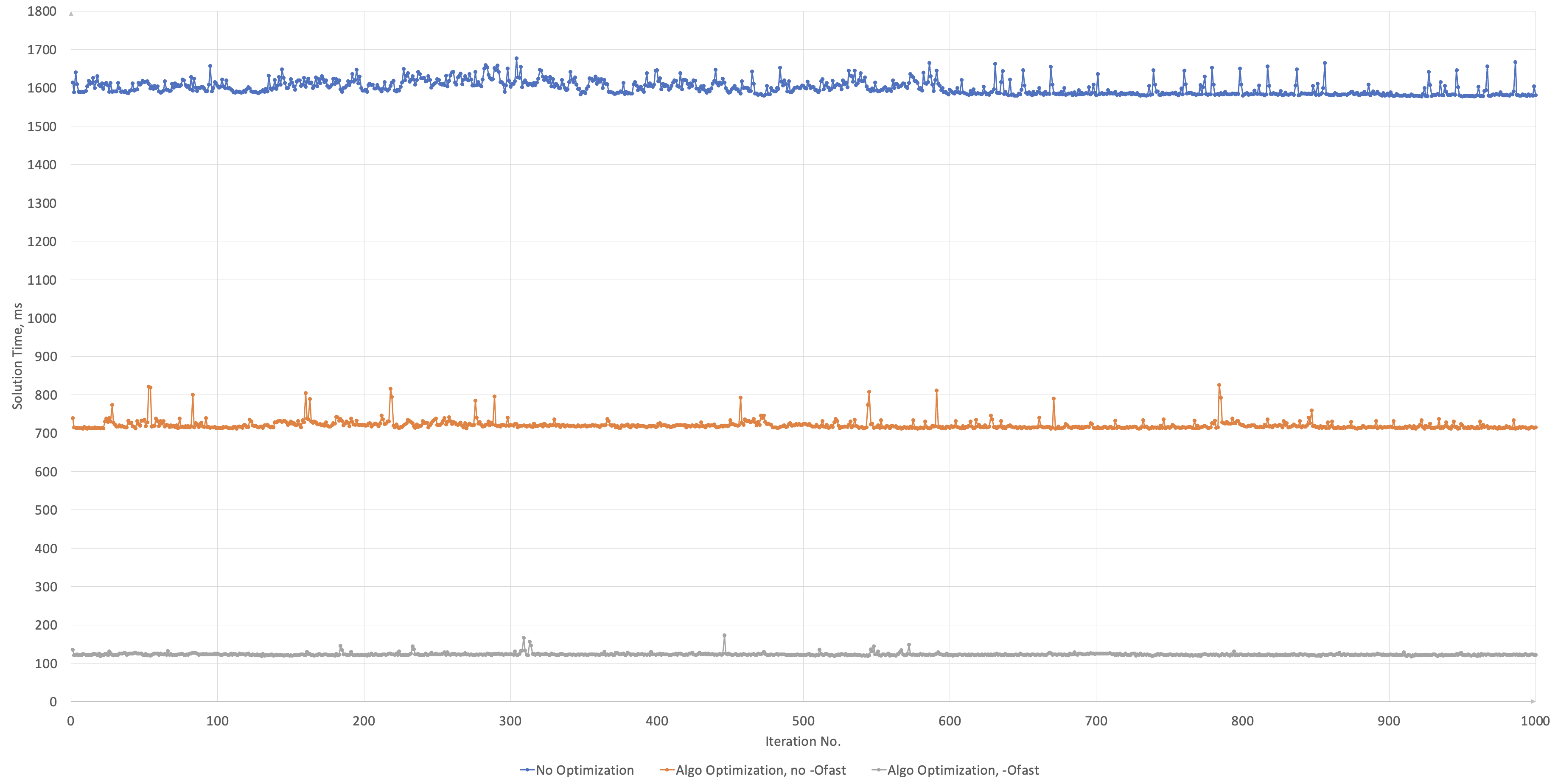


Рисунок 5 – Производительность алгоритма

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были реализованы и проанализированы три ключевые направления оптимизации программы. Первой и наиболее значимой стала оптимизация работы с буферами графического рендеринга. За счет переноса операций инициализации буферов и массивов вершин в начало рендеринга с последующим обновлением их содержимого удалось существенно сократить накладные расходы на пересоздание данных. Эта оптимизация оказала значительное влияние на производительность за счет снижения времени отрисовки каждого кадра.

Второй важной оптимизацией стало внедрение «замораживания» главного окна программы на время работы вспомогательных окон. Это решение позволило временно освободить ресурсы главного окна, исключив избыточное потребление памяти в моменты, когда оно неактивно. Оптимизация продемонстрировала положительный эффект в виде снижения пикового потребления памяти, что особенно важно для повышения отзывчивости и эффективности программы на системах с ограниченными ресурсами.

Третьим направлением стали общие машинно-независимые оптимизации, включавшие улучшения структуры кода и устранение избыточных операций. Несмотря на менее выраженное влияние на производительность по сравнению с другими оптимизациями, эти изменения способствовали созданию более компактного и легко поддерживаемого кода. Они также повысили эффективность компиляции и создали основу для дальнейших улучшений программы.

Наконец, была внедрена алгоритмическая оптимизация. Использование системы рандомизации возможных вариантов вращений для всех этапов сборки показало высокую эффективность. Проведенное тестирование на выборке из 1000 случайных кубиков Рубика продемонстрировало сокращение времени поиска решения на 92% при использовании оптимизации компилятора -Ofast и на 55% без ее применения. Несмотря на сохранение алгоритмической сложности, динамическое перемешивание порядка вращений существенно уменьшило количество итераций перебора, что позволило значительно ускорить выполнение алгоритма.

Применение машинно-зависимой оптимизации с использованием флага -Ofast позволило существенно ускорить выполнение программы, особенно в задачах с высокой вычислительной нагрузкой. Например, это улучшение обеспечило до 92% сокращения времени работы алгоритма поиска решения, подтверждая эффективность использования агрессивных оптимизаций компилятора для повышения производительности.

Все подходы к оптимизации в совокупности позволили значительно улучшить производительность и стабильность программы, сделав ее более эффективной как с точки зрения времени выполнения, так и с точки зрения использования системных ресурсов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Buffer Object // OpenGL Wiki. - URL: https://www.khronos.org/opengl/wiki/Buffer\_Object (дата обращения: DD.MM.YYYY).
2. Cube Exproler // Herbert Kociemba. - URL: https://kociemba.org/cube.htm (дата обращения: DD.MM.YYYY).
3. GetOpenFileNameA function (commdlg.h) // Windows App Development. - URL: https://learn.microsoft.com/en-us/windows/win32/api/commdlg/nf-commdlg-getopenfilenamea (дата обращения: DD.MM.YYYY).
4. GetSaveFileNameA function (commdlg.h) // Windows App Development. - URL: https://learn.microsoft.com/en-us/windows/win32/api/commdlg/nf-commdlg-getsavefilenamea (дата обращения: DD.MM.YYYY).
5. Hello Window // LearnOpenGL. - URL: https://learnopengl.com/Getting-started/Hello-Window (дата обращения: DD.MM.YYYY).
6. Input guide // GLFW. - URL: https://www.glfw.org/docs/3.3/input\_guide.html (дата обращения: DD.MM.YYYY).
7. My system for solving Rubik's cube // Jessica Fridrich. - URL: http://www.ws.binghamton.edu/fridrich/system.html (дата обращения: DD.MM.YYYY).
8. OpenGL 4 Reference Pages // Khronos Registry. - URL: https://registry.khronos.org/OpenGL-Refpages/gl4/html/ (дата обращения: DD.MM.YYYY).
9. The Mathematics of the Rubik’s Cube // MIT. - URL: https://web.mit.edu/sp.268/www/rubik.pdf (дата обращения: DD.MM.YYYY).
10. The OpenGL Graphics System: A Specification // The Khronos Group Inc. - URL: https://registry.khronos.org/OpenGL/specs/gl/glspec40.core.pdf (дата обращения: DD.MM.YYYY).
11. WCA Regulations // World Cube Association. - URL: https://www.worldcubeassociation.org/regulations/ (дата обращения: DD.MM.YYYY).
12. Window guide // GLFW. - URL: https://www.glfw.org/docs/3.3/window\_guide.html (дата обращения: DD.MM.YYYY).
13. Zenity Manual // The GNOME Project. - URL: https://help.gnome.org/users/zenity/stable/ (дата обращения: DD.MM.YYYY).

# ПРИЛОЖЕНИЕ А. ЛИСТИНГИ ФУНКЦИЙ ОТРИСОВКИ ОКОН

Листинг функции «main»

int main(int argc, char \*\*argv) {

#ifdef DEBUG\_MODE

printTime = 0;

logFile = fopen("log.txt", "w");

#endif

needToReInitGLT = 0;

isAuxWindowActive = 0;

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

GLFWwindow \*window;

if (!glfwInit()) {

fprintf(stderr, "failed to initialize GLFW!\n");

return -1;

}

glfwWindowHint(GLFW\_SAMPLES, 8); // anti aliasing

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR,

3); // openGL major version to be 3

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR,

3); // minor set to 3, which makes the version 3.3

#ifdef \_\_APPLE\_\_

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE); // for MAC OS only

#endif

glfwWindowHint(GLFW\_OPENGL\_COMPAT\_PROFILE,

GLFW\_OPENGL\_CORE\_PROFILE); // avoid using old OpenGL

window = glfwCreateWindow(600, 600, "RubikSolver", NULL, NULL);

if (!window) {

fprintf(stderr, "failed to create GLFW window!\n");

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebufferSizeCallback);

// glfwSetWindowAspectRatio(window, 1, 1);

glfwSetKeyCallback(window, keyCallback);

glfwSetMouseButtonCallback(window, mouseButtonCallback);

#ifndef \_\_APPLE\_\_

gladLoadGL();

#endif

initBuffers(&mainWndData.VAO\_button, &mainWndData.VBO\_button);

initBuffers(&mainWndData.VAO\_square, &mainWndData.VBO\_square);

initBuffers(&mainWndData.VAO\_stroke, &mainWndData.VBO\_stroke);

GLuint shaderProgram = createShaderProgram(vertexShaderInstanceSource,

fragmentShaderInstanceSource);

// printf("%s\n", glGetString(GL\_VERSION));

isCubeFilled = 0;

currentFlatCubeIndex = -1;

initButtons(mainButtons, flatButtons);

initLinearCube(&Cube2D);

initCube(&Cube3D);

GLTtext \*stepText = gltCreateText();

GLTtext \*revertText = gltCreateText();

GLTtext \*openText = gltCreateText();

GLTtext \*nextText = gltCreateText();

GLTtext \*cubeText = gltCreateText();

GLTtext \*solveText = gltCreateText();

GLTtext \*shuffleText = gltCreateText();

gltSetText(revertText, "I");

gltSetText(openText, "O");

gltSetText(nextText, "N");

gltSetText(cubeText, "C");

gltSetText(solveText, "S");

gltSetText(shuffleText, "R");

gltInit();

int width, height;

currentWindow = MAINWND;

// Main loop

while (!glfwWindowShouldClose(window)) {

if (!isAuxWindowActive) {

#ifdef DEBUG\_MODE

clock\_t start = clock(); // start measuring time

#endif

if (needToReInitGLT) {

gltInit();

needToReInitGLT = 0;

}

glClear(GL\_COLOR\_BUFFER\_BIT);

glUseProgram(shaderProgram);

drawSide(RIGHT, Cube2D.redSide);

drawSide(LEFT, Cube2D.whiteSide);

drawSide(TOP, Cube2D.blueSide);

drawUI();

gltSetText(stepText, currentStepText);

glfwGetWindowSize(window, &width, &height);

// printf("%f %f\n", xpos, ypos);

gltBeginDraw();

gltColor(1.0f, 1.0f, 1.0f, 1.0f);

// gltDrawText2D(text, xpos, ypos, 1.0f);

gltDrawText2DAligned(stepText, NDCToPixels(0.0f, width, 'x'),

NDCToPixels(0.875f, height, 'y'), 2.0f, GLT\_CENTER,

GLT\_CENTER);

gltColor(1.0f, 0.0f, 0.0f, 1.0f);

gltDrawText2DAligned(

revertText, NDCToPixels(mainButtons[0].xPos + 0.075f, width, 'x'),

NDCToPixels(mainButtons[0].yPos - 0.075f, height, 'y'), 2.0f,

GLT\_CENTER, GLT\_CENTER);

gltDrawText2DAligned(

openText, NDCToPixels(mainButtons[9].xPos + 0.075f, width, 'x'),

NDCToPixels(mainButtons[9].yPos - 0.075f, height, 'y'), 2.0f,

GLT\_CENTER, GLT\_CENTER);

gltDrawText2DAligned(

nextText, NDCToPixels(mainButtons[7].xPos + 0.075f, width, 'x'),

NDCToPixels(mainButtons[7].yPos - 0.075f, height, 'y'), 2.0f,

GLT\_CENTER, GLT\_CENTER);

gltDrawText2DAligned(

cubeText, NDCToPixels(mainButtons[8].xPos + 0.075f, width, 'x'),

NDCToPixels(mainButtons[8].yPos - 0.075f, height, 'y'), 2.0f,

GLT\_CENTER, GLT\_CENTER);

gltDrawText2DAligned(

solveText, NDCToPixels(mainButtons[10].xPos + 0.075f, width, 'x'),

NDCToPixels(mainButtons[10].yPos - 0.075f, height, 'y'), 2.0f,

GLT\_CENTER, GLT\_CENTER);

gltDrawText2DAligned(

shuffleText, NDCToPixels(mainButtons[11].xPos + 0.075f, width, 'x'),

NDCToPixels(mainButtons[11].yPos - 0.075f, height, 'y'), 2.0f,

GLT\_CENTER, GLT\_CENTER);

gltEndDraw();

glfwSwapBuffers(window);

glfwPollEvents();

#ifdef DEBUG\_MODE

clock\_t end = clock(); // end measuring time

if (printTime) {

printf("main window took %0.f ms to render\n",

((double)(end - start)) / (CLOCKS\_PER\_SEC / 1000));

fprintf(logFile, "main window took %0.f ms to render\n",

((double)(end - start)) / (CLOCKS\_PER\_SEC / 1000));

printTime = 0;

}

#endif

}

}

glDeleteVertexArrays(1, &mainWndData.VAO\_button);

glDeleteBuffers(1, &mainWndData.VBO\_button);

glDeleteVertexArrays(1, &mainWndData.VAO\_square);

glDeleteBuffers(1, &mainWndData.VBO\_square);

glDeleteVertexArrays(1, &mainWndData.VAO\_stroke);

glDeleteBuffers(1, &mainWndData.VBO\_stroke);

gltDeleteText(stepText);

gltDeleteText(revertText);

gltDeleteText(openText);

gltDeleteText(nextText);

gltDeleteText(cubeText);

gltDeleteText(solveText);

gltDeleteText(shuffleText);

gltTerminate();

glDeleteProgram(shaderProgram);

glfwDestroyWindow(window);

glfwTerminate();

return 0;

}

Листинг функции «fillCubeFromUserInput»

void fillCubeFromUserInput(GLFWwindow \*window) {

GLFWwindow \*flatCubeWindow =

glfwCreateWindow(500, 500, "User Input", NULL, NULL);

if (!flatCubeWindow) {

fprintf(stderr, "failed to create flat cube window!\n");

return;

}

glfwMakeContextCurrent(flatCubeWindow);

glfwSetFramebufferSizeCallback(flatCubeWindow, framebufferSizeCallback);

glfwSetMouseButtonCallback(flatCubeWindow, mouseButtonCallback);

glfwSetKeyCallback(flatCubeWindow, keyCallbackEscapeOnly);

GLuint shaderProgram = createShaderProgram(vertexShaderInstanceSource, fragmentShaderInstanceSource);

initFlatCube(&flatCube);

updateFlatCube(WHITE, 1);

isAuxWindowActive = 1;

currentWindow = FLATWND;

initBuffers(&flatWndData.VAO\_button, &flatWndData.VBO\_button);

initBuffers(&flatWndData.VAO\_square, &flatWndData.VBO\_square);

gltTerminate();

gltInit();

needToReInitGLT = 1;

GLTtext \*fileText = gltCreateText();

gltSetText(fileText, "F");

int width, height;

while (!glfwWindowShouldClose(flatCubeWindow)) {

#ifdef DEBUG\_MODE

clock\_t start = clock(); // start measuring time

#endif

glClear(GL\_COLOR\_BUFFER\_BIT);

glUseProgram(shaderProgram);

if (isCubeFilled) {

isCubeFilled = 0;

break;

}

drawFlatSide(-0.5f, 0.95f, BLUE, flatCube.blueSide);

drawFlatSide(-0.95f, 0.5f, ORANGE, flatCube.orangeSide);

drawFlatSide(-0.5f, 0.5f, WHITE, flatCube.whiteSide);

drawFlatSide(-0.05f, 0.5f, RED, flatCube.redSide);

drawFlatSide(0.4f, 0.5f, YELLOW, flatCube.yellowSide);

drawFlatSide(-0.5f, 0.05f, GREEN, flatCube.greenSide);

drawFlatWindowUI();

glfwGetWindowSize(flatCubeWindow, &width, &height);

gltBeginDraw();

gltColor(1.0f, 0.0f, 0.0f, 1.0f);

gltDrawText2DAligned(

fileText, NDCToPixels(flatButtons[6].xPos + 0.075f, width, 'x'),

NDCToPixels(flatButtons[6].yPos - 0.075f, height, 'y'), 1.75f,

GLT\_CENTER, GLT\_CENTER);

gltEndDraw();

glfwSwapBuffers(flatCubeWindow);

glfwPollEvents();

#ifdef DEBUG\_MODE

clock\_t end = clock(); // end measuring time

if (printTime) {

printf("input window took %0.f ms to render\n",

((double)(end - start)) / (CLOCKS\_PER\_SEC / 1000));

fprintf(logFile, "input window took %0.f ms to render\n",

((double)(end - start)) / (CLOCKS\_PER\_SEC / 1000));

printTime = 0;

}

#endif

}

gltTerminate();

glDeleteVertexArrays(1, &flatWndData.VAO\_button);

glDeleteBuffers(1, &flatWndData.VBO\_button);

glDeleteVertexArrays(1, &flatWndData.VAO\_square);

glDeleteBuffers(1, &flatWndData.VBO\_square);

currentWindow = MAINWND;

gltDeleteText(fileText);

currentFlatCubeIndex = -1;

glfwDestroyWindow(flatCubeWindow);

glfwMakeContextCurrent(window);

isAuxWindowActive = 0;

}

# ПРИЛОЖЕНИЕ Б. ЛИСТИНГИ КЛЮЧЕВЫХ ФУНКЦИЙ АЛГОРИТМА РЕШЕНИЯ КУБИКА

Листинг функции «checkAllCube»

int checkAllCube(Cube \*thisCube, int flag) {

int checksum = 0;

int cursum = 0;

for (int i = 0; i < 8; i++) {

if (flag) {

checksum += doCheck(thisCube, i);

cursum += 4;

if (cursum >= flag) {

break;

}

} else {

cursum = doCheck(thisCube, i);

checksum += cursum;

if (cursum != 4) {

break;

}

}

}

return checksum;

}

Листинг функции «search»

Cube\* search(SolutionTable\* table, Cube\* startCube, int state, char\* operlist,

int opernum, int cutvalue) {

Cube\* newcube = NULL;

Cube\* currcube = NULL;

int newstate = 0;

int newvalue = 0;

startCube->previousState = NULL;

startCube->operation = LASTOP;

if (checkAllCube(startCube, 0) >= state) {

return startCube;

}

table->openCubes = 0;

table->closedCubes = 0;

table->cubes[table->openCubes] = \*startCube;

table->openCubes++;

int arrI[] = { 0,1,2,3,4,5 };

int countshuffle = 0;

if (state == 15)

{

arrI[0] = 0;

arrI[1] = 1;

arrI[2] = 2;

arrI[3] = 3;

arrI[4] = 4;

}

while (table->closedCubes < table->openCubes) {

if (table->openCubes + 6 >= table->size - 1) {

// all memory was used

break;

}

currcube = &(table->cubes[table->closedCubes++]);

if ((state != 32 && state != 15) && countshuffle % 15 == 0) // %3 == 3s all; %4 == 7s all; %

{

shuffle(arrI, opernum);

}

countshuffle++;

for (int i = 0; i < opernum; i++) {

newcube = &(table->cubes[table->openCubes]);

memcpy(newcube, currcube, sizeof(Cube));

cubeDoOp(newcube, operlist[arrI[i]]);

newcube->operation = operlist[arrI[i]];

newcube->previousState = currcube;

newstate = checkAllCube(newcube, 0);

newvalue = checkAllCube(newcube, state);

if (newstate >= state) {

return newcube;

}

if (cutTheWay(newvalue, cutvalue) || checkIfSameOps(newcube)) {

continue;

}

table->openCubes++;

}

}

return NULL;

}

Листинг функции «cubeSolve»

void cubeSolve(Cube \*thisCube, FILE \*foutput, char filename[]) {

SolutionTable table;

createTable(&table, TABLESIZE);

char operationsfirstseps[6] = { FRONTROTATE, UPROTATE, DOWNROTATE,

LEFTROTATE, RIGHTROTATE };

char operatoinssecondsteps[6] = { UPROTATE, DOWNROTATE, LEFTROTATE,

RIGHTROTATE, BACKROTATE };

char operationslastspets[4] = { BACKROTATE, RIGHTROTATE, UPROTATE };

char\* opeartionsarray[3] = { operationsfirstseps, operatoinssecondsteps,

operationslastspets };

int parametrs[][4] = { {2, 0, 5, 0}, // начало белого креста1

{4, 0, 5, 2}, // конец белого креста1

{8, 0, 5, 2}, // начало и конец белого креста2

{9, 0, 5, 3}, // начало сборки верхнего слоя

{10, 0, 5, 4}, //

{11, 0, 5, 5}, //

{12, 0, 5, 5}, // конец сборки верхнего слоя

{13, 1, 5, 6}, // начало сборки среднего ряда

{14, 1, 5, 6}, //

{15, 1, 5, 7}, //

{16, 1, 5, 8}, // конец сборки среднего ряда

{18, 1, 5, 10}, // начало сборки желтого креста

{20, 1, 5, 11}, // конец сборки желтого креста

{21, 1, 5, 11}, // начало сборки желтого креста

{24, 1, 5, 11}, // конец сборки желтого креста

{26, 2, 3, 12}, // начало сборки всей желтой стороны

{28, 2, 3, 12}, // конец сборки всей желтой стороны //Было 12

{32, 2, 2, 12}, // начало и конец сборки нижнего слоя

{0, 0, 0, 0} };

Cube\* startCube = thisCube;

Cube\* find = NULL;

for (int i = 0; i < SMALLSTEPSCOUNT; i++) {

find = search(&table, startCube, parametrs[i][0],

opeartionsarray[parametrs[i][1]], parametrs[i][2],

parametrs[i][3]);

if (find) {

if (find != startCube) {

startCube = find;

\*thisCube = \*find;

printOpToFile(find, foutput);

}

}

else {

FILE\* r = freopen(filename, "w", foutput);

fputc('\_', foutput);

break;

}

}

deleteTable(&table);

}