Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

**Высшая школа кибербезопасности**

**КУРСОВАЯ РАБОТА**

**Моделирование решения Кубика Рубика 3x3**

по дисциплине «Структуры данных»

Выполнили

студенты гр. 5151003/30002 И. А. Штарев

Д. А. Плотников

Руководитель

асс. преподавателя И. Д. Панков

«07» июня 2024 г.

Санкт-Петербург

2024

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc168616463)

[1. ТЕОРЕТИЧЕСКИЕ ИССЛЕДОВАНИЯ 4](#_Toc168616464)

[1.1. ВЫБОР ПРОГРАММНОЙ ПЛАТФОРМЫ 4](#_Toc168616465)

[1.2. ВЫБОР АЛГОРИТМА СБОРКИ КУБИКА 6](#_Toc168616466)

[2. ГРАФИЧЕСКАЯ ЧАСТЬ 8](#_Toc168616467)

[2.1. ГЛАВНОЕ ОКНО 10](#_Toc168616468)

[2.1.1. ПРЕДСТАВЛЕНИЕ КУБИКА И ЕГО ОТРИСОВКА 13](#_Toc168616469)

[2.2. ОКНО ВВОДА КУБИКА 16](#_Toc168616470)

[2.3. СИНХРОНИЗАЦИЯ КУБИКОВ 19](#_Toc168616471)

[3. АЛГОРИТМ РЕШЕНИЯ КУБИКА 19](#_Toc168616472)

[3.1. ФУНКЦИИ ПРОВЕРОК СОСТОЯНИЯ КУБИКА 24](#_Toc168616473)

[3.2. ФУНКЦИИ РЕШЕНИЯ КУБИКА 26](#_Toc168616474)

[ЗАКЛЮЧЕНИЕ 28](#_Toc168616475)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 29](#_Toc168616476)

[ПРИЛОЖЕНИЕ А. БЛОК-СХЕМЫ ФУНКЦИЙ ОТРИСОВКИ ОКОН 31](#_Toc168616477)

[ПРИЛОЖЕНИЕ Б. ЛИСТИНГИ ФУНКЦИЙ ОТРИСОВКИ ОКОН 33](#_Toc168616478)

[ПРИЛОЖЕНИЕ В. БЛОК-СХЕМЫ КЛЮЧЕВЫХ ФУНКЦИЙ АЛГОРИТМА РЕШЕНИЯ КУБИКА 40](#_Toc168616479)

[ПРИЛОЖЕНИЕ Г. ЛИСТИНГИ КЛЮЧЕВЫХ ФУНКЦИЙ АЛГОРИТМА РЕШЕНИЯ КУБИКА 43](#_Toc168616480)

# ВВЕДЕНИЕ

Кубик Рубика – культовая головоломка, покорившая умы миллионов людей по всему миру. С виду простая конструкция, состоящая из цветных квадратов, таит в себе невероятное количество комбинаций и алгоритмов, позволяющих достичь заветного результата – собрать все цвета на своих гранях.

Данная курсовая работа посвящена разработке алгоритма, способного решить кубик Рубика, состояние которого задается пользователем. В работе рассматриваются различные подходы к решению головоломки, анализируются их преимущества и недостатки.

В рамках исследования был разработан и реализован алгоритм, основанный на трехслойном методе.

Практическая значимость данной работы заключается в создании универсального решения для сборки кубика Рубика, доступного для использования пользователями с различным уровнем подготовки. Программа может быть использована как в развлекательных целях, так и в качестве инструмента для изучения алгоритмов решения головоломки.

# 1. ТЕОРЕТИЧЕСКИЕ ИССЛЕДОВАНИЯ

# 1.1. ВЫБОР ПРОГРАММНОЙ ПЛАТФОРМЫ

OpenGL (сокращение от «Open Graphics Library») – это программный интерфейс для взаимодействия с графическим оборудованием. Интерфейс состоит из набора из нескольких сотен процедур и функций, которые позволяют программисту задавать объекты и операции, участвующие в создании высококачественных графических изображений, в частности, цветных изображений трехмерных объектов.

Большинство функций OpenGL требуют наличия кадрового буфера (далее – фреймбуфера) в графическом оборудовании.

Многие вызовы OpenGL относятся к рисованию объектов, таких как точки, линии и полигоны, но способ выполнения некоторых операций рисования (например, при включенном сглаживании) зависит от наличия фреймбуфера. Кроме того, некоторые функции OpenGL напрямую связаны с манипуляциями с фреймбуфером.

Для программиста OpenGL представляет собой набор команд, которые позволяют описывать геометрические объекты в двух или трех измерениях, а также команды, управляющие тем, как эти объекты будут отображаться (рендериться) во фреймбуфере.

Типичная программа, использующая OpenGL, начинается с вызовов для открытия окна во фреймбуфере, в которое программа будет рисовать. Затем следуют вызовы для выделения GL-контекста и связывания его с окном. После того, как GL-контекст выделен, программист может свободно выполнять команды OpenGL. Некоторые вызовы используются для рисования простых геометрических объектов (то есть точек, отрезков и полигонов), в то время как другие влияют на рендеринг этих примитивов, включая то, как они освещаются или окрашиваются, а также как они отображаются из двух- или трехмерного модельного пространства пользователя на двумерный экран. Существуют также вызовы для непосредственного управления фреймбуфером, такие как чтение и запись пикселей [9, 11].

GLFW (сокращение от «Graphics Library Framework») – это библиотека с открытым исходным кодом, предназначенная для упрощения разработки приложений с использованием OpenGL. Вместо того чтобы писать платформозависимый код для создания окон, контекстов OpenGL, обработки ввода и других низкоуровневых задач, GLFW предоставляет простой и кроссплатформенный API.

GLFW позволяет создавать окна с заданными размерами, заголовком, режимом отображения (оконный или полноэкранный) и другими параметрами. Также GLFW инициализирует контекст OpenGL, который необходим для использования функций OpenGL. Библиотека также позволяет управлять несколькими контекстами OpenGL, что полезно для многопоточных приложений. Библиотека предоставляет функции для отслеживания событий клавиатуры, мыши, геймпада и джойстика. Имеется возможность позволяет скрывать курсор, фиксировать его в пределах окна и получать его позицию. GLFW предоставляет информацию о подключенных мониторах, их разрешениях и частоте обновления, а также позволяет выбирать монитор для полноэкранного режима. Библиотекой предоставляются функции для измерения времени, установки таймеров и вертикальной синхронизации для синхронизации частоты кадров с частотой обновления монитора [6, 7, 13].

# 1.2. ВЫБОР АЛГОРИТМА СБОРКИ КУБИКА

В данной работе для реализации решения кубика рассматривались несколько различных алгоритмов: двухфазный алгоритм Герберта Коцембы, метод Джессики Фридрих, а также послойный алгоритм сборки кубика.

Двухфазный алгоритм Герберта Коцембы позволяет искать оптимальные или субоптимальные решения кубика. Двухфазный алгоритм сводит задачу решения кубика к переходу от состояния собранности группы G0 (произвольно заданный кубик), до состояния группы G1, после чего без поворотов боковых граней кубик приводится в состояние группы G2 (решенный кубик). Особенностью состояния группы G1 является то, что любые конфигурации из ее группы будут иметь одну и ту же разметку если выполнить ее по следующему принципу: все элементы на верхних и нижних гранях пометить знаком "+", все смежные элементы на среднем ряду боковых гранях пометить знаком "-". Однако данный алгоритм довольно долго вычисляет решения, что связано с тем, что он пытается найти минимально возможное число ходов [3].

Метод Джессики Фридрих состоит из четырех этапов и используется в основном для сборки кубика на скорость. Особенностью алгоритма является то, что для решения кубика необходимо иcпользовать 119 различных наборов вращений в зависимости от текущей конфигурации куба. Наличие столь много количества возможных конфигураций и различных действий, которые необходимо реализовать для каждого, усложняло написание алгоритма приложения, поэтому от данного метода было решено отказаться [8].

Послойный алгоритм сборки кубика заключается в том, что сборка разделена на 7 этапов, где постепенно кубик приводится в более решенный вид. Простота и понятность послойного алгоритма стали ключевыми при выборе для реализации в программе по нескольким причинам. Послойный метод разбивает процесс сборки на понятные и логически последовательные этапы. Это облегчает как программирование, так и отладку алгоритма. Каждый этап четко определен, что позволяет легче выявлять и исправлять ошибки в коде. В отличие от двухфазного алгоритма Коцембы, который может требовать значительных вычислительных ресурсов для поиска оптимального решения, послойный алгоритм менее ресурсоемок. Реализация послойного алгоритма требует меньше кода и логических проверок по сравнению с методом Фридрих. Кроме того, алгоритм проще для понимания и изучения, что важно для разработчиков, не имеющих глубоких знаний в сборке кубика. Послойный метод широко используется новичками из-за своей простоты и доступности [1, 10].

Таким образом, выбор послойного алгоритма для реализации в программе обусловлен его простотой, понятностью, низкими требованиями к вычислительным ресурсам и легкостью внедрения. Эти факторы делают его оптимальным решением для создания приложения, способного эффективно и быстро решать кубик Рубика, обеспечивая при этом доступность и удобство использования для широкой аудитории.

# 2. ГРАФИЧЕСКАЯ ЧАСТЬ

Приложение «Rubik Solver» используется для решения кубика Рубика с состоянием, заданным пользователем из файла или вручную (через интерфейс ввода состояния кубика).

После ввода состояния кубика пользователю предлагается нажать на кнопку «решить» для сохранения в файл последовательности поворотов, необходимых для решения заданного им кубика. Решение можно загрузить и из заранее подготовленного файла с последовательностью поворотом.

После задания состояния кубика и получения необходимой последовательности действий пользователю предлагается просмотреть решение пошагово, вызывая каждый новый шаг очередным нажатием кнопки «следующий шаг». Повороты кубика на экране сопровождаются выведением информации о соответствующем повороте.

После просмотра всех шагов пользователю сообщается, что решение завершено.

Для отрисовки графической части разработанного решения была использована библиотека OpenGL в связке со вспомогательными библиотеками GLFW и GLAD. Самый очевидный ход при отрисовке окон в OpenGL – одинарная буферизация. При ней все изменения в кадре сразу же отображаются в экранном буфере, что может привести к мерцанию экрана. Для разрешения этих ситуаций в решении использована двойная буферизация. При ней вместо одного буфера кадра используются два:

* Задний буфер (back buffer): скрытый от пользователя буфер, где происходит рисование объектов.
* Передний буфер (front buffer): буфер, который в данный момент отображается на экране.

Все операции рисования происходят в заднем буфере, невидимом для пользователя. Как только кадр полностью отрисован в заднем буфере, происходит мгновенный обмен (swap) между передним и задним буферами. Задний буфер становится видимым, а передний становится доступным для рисования следующего кадра. При двойной буферизации достигается целостность кадров: пользователь видит только полностью отрисованные кадры [2, 6, 13].

Для обработки событий клавиатуры, таких как нажатие или отпускание клавиш, используется функция glfwSetKeyCallback [7]. Для обработки событий мыши, таких как нажатие кнопок мыши, используется функция glfwSetMouseButtonCallback [7]. В каждом из окон нажатие клавиши Escape закрывает данное окно, а в главном окне нажатие клавиши F1 открывает справку.

Функция framebufferSizeCallback предназначена для обработки изменения размера окна и поддержания квадратного соотношения сторон для его содержимого [7]. Функция принимает текущую ширину и высоту окна. Сначала определяется минимальный размер из двух, чтобы сохранить квадратное соотношение сторон. Если ширина меньше высоты, minSize будет равен ширине, иначе – высоте. Далее функция glfwSetWindowSize изменяет размер окна на квадрат, стороны которого равны минимальному значению, определенному на предыдущем шаге. Это действие делает окно квадратным. Далее функция glViewport задает параметры области просмотра. Она определяет часть окна, куда будут отрисовываться все последующие графические примитивы [9]. Параметры (0, 0, minSize, minSize) означают, что область просмотра начинается с координаты (0, 0) и имеет размеры minSize на minSize, что обеспечивает квадратное отображение.

Блок-схемы функций отрисовки представлены в приложении А. Листинги этих функций представлены в приложении Б.

# 2.1. ГЛАВНОЕ ОКНО

Первое, что видит пользователь при запуске приложения – главное окно. Главное окно представлено на рисунке 1.

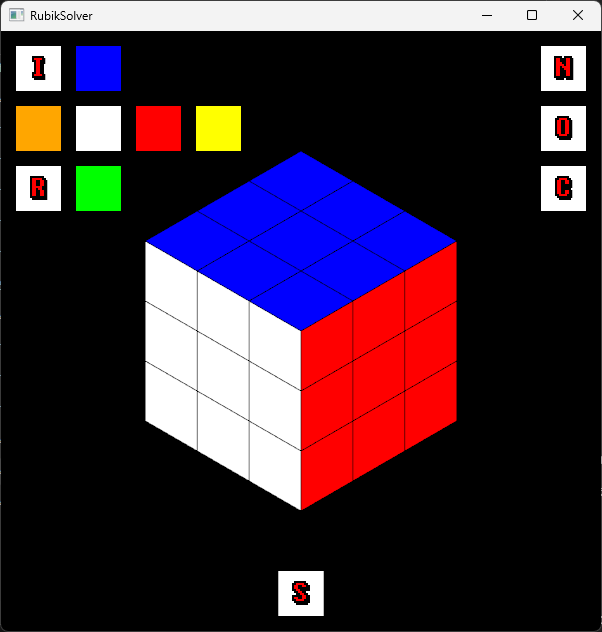


Рисунок 1 – Главное окно разработанного приложения

Как можно пронаблюдать по рисунку 1, главное окно состоит из кубика и подписанных кнопок управления. Преставление кубика в памяти и процесс его отрисовки описаны в последующих главах.

Кнопка I отвечает за приведение кубика к изначальному (собранному) виду.

Кнопка R запутывает куб, то есть совершает над ним последовательность случайных поворотов.

Цветные кнопки без надписей рядом с кнопкой R отвечают за повороты соответствующих сторон по и против часовой стрелки. Если нажать на одну из таких кнопок левой клавишей мыши, соответствующая сторона повернется по часовой стрелке, если правой – против часовой.

Кнопка O отвечает за открытие файла с шагами решения. Если у пользователя уже есть готовое решение для кубика, он может нажать на эту кнопку, далее откроется системный диалог выбора файла. После выбора файла он загружается в программу, становясь готовым для исполнения. Для реализации файлового диалога в Windows используются системные функции GetOpenFilename и GetSaveFileName, в Linux используется встроенная в стандартный пакет Ubuntu утилита zenity [4, 5, 14]. В файле с шагами каждое действие записывается в новой строке. Далее записывается название стороны в соответствии с общепринятой нотацией (F – Front, B – Back, U – Up, D – Down, L – Left, R – Right). Допускаются как строчные, так и прописные буквы, однако при решении куба в файле с шагами используются прописные. Далее опционально записывается требуемое число поворотов. Далее опционально записывается символ ‘'’ (одинарная кавычка), являющийся показателем того, что соответствующий поворот происходит против часовой стрелки [10, 12]. Пример файла с шагами представлен на рисунке 2.

A screenshot of a computer

Description automatically generated

Рисунок 2 – Пример файла с шагами

Кнопка N исполняет следующий шаг из открытого файла с шагами, выводя инструкцию к шагу наверху окна. В случае если файл не открыт, либо в случае если все шаги из открытого файла исполнены, выводится соответствующее сообщение.

Кнопка С открывает окно ввода кубика, который необходимо решить. Вызов этого окна переключает так называемый контекст, делая главное окно неактивным. Окно ввода кубика описано в последующих главах.

Кнопка S решает кубик и сразу предлагает пользователю сохранить решение в файл с шагами, а далее загружает этот файл в программу. После приведения алгоритма решения в действие пользователю следует дождаться появления надписи «STEPS LOADED» либо надписи «UNSOLVEABLE». После решения кубика пользователь может последовательно исполнить шаги решения, нажимая на кнопку N. Алгоритм решения кубика описан в последующих главах.

В главном окне при нажатии клавиши F1 открывается окно, содержащее информацию о приложении и краткую справку по использованию интерфейса.

# 2.1.1. ПРЕДСТАВЛЕНИЕ КУБИКА И ЕГО ОТРИСОВКА

В программе используется два представления кубика: в виде одномерного массива и в виде двумерного. Такое решение было принято по причине того, что для отрисовки кубика с использованием OpenGL более оптимальным является линейное представление, в то время как решение с помощью метода обхода в ширину становится более оптимальным с точки зрения программного кода именно при использовании представления в виде матрицы. В обоих случаях кубик хранится в памяти в виде структуры, поля которой отражают цвета каждого квадрата на каждой грани. Порядок записи квадратов для каждой грани линейного представления изображен на рисунке 2. Индексы двумерных массивов, указывающие на цвета соответственных квадратов в матричном кубе, представлены на рисунке 3.

A colorful squares with numbers

Description automatically generated

Рисунок 2 – Индексы квадратов линейного куба

A colorful squares with numbers

Description automatically generated

Рисунок 3 – Индексы квадратов матричного куба

При запуске программы экземпляры этой структуры создаются в виде глобальных переменных. Хотя использование глобальных переменных и является небезопасной практикой, в данной случае решение об их использовании является оправданным, так как это обеспечивает удобный доступ к редактированию кубика из любого места программы.

Важно отметить, что в программе установлено четкое соответствие между общепринятыми обозначениями сторон и их цветами:

U (Up) – Синий

L (Left) – Оранжевый

F (Front) – Белый

R (Right) – Красный

B (Back) – Желтый

D (Down) – Зеленый

В связи с этим все повороты, предлагаемые программой, следует совершать на кубике, ориентированном именно таким образом.

Пользователю не дается возможность выбрать цвета центральных квадратов при вводе начального состояния кубика с помощью мыши как раз по причине того, что соответствие сторон и их цветов строго зафиксировано.

Оба кубика хранятся в памяти программы, их состояния синхронизируются при каждом изменении состояния кубика. Синхронизация кубиков описана в последующих главах.

Отрисовка кубика происходит в изометрии: пользователю видны три смежных стороны, чего вполне достаточно для полноценного отображения любого поворота. За отрисовку кубика отвечают функции drawSide и drawSquare. Первая принимает на вход информацию о том, какую сторону следует отрисовать (левую, правую или верхнюю) и массив цветов, соответствующих этой стороне. Вторая вызывается первой девять раз, по каждому на каждую клетку. Эта функция принимает на вход координаты угла клетки и три компоненты цвета этой клетки.

Состояние кубика обновляется каждый кадр, а хранение его состояния в глобальной переменной позволяет делать повороты сторон из любой точки программы, моментально получая отрисованный результат.

# 2.2. ОКНО ВВОДА КУБИКА

На рисунке 4 представлен внешний вид окна ввода кубика.

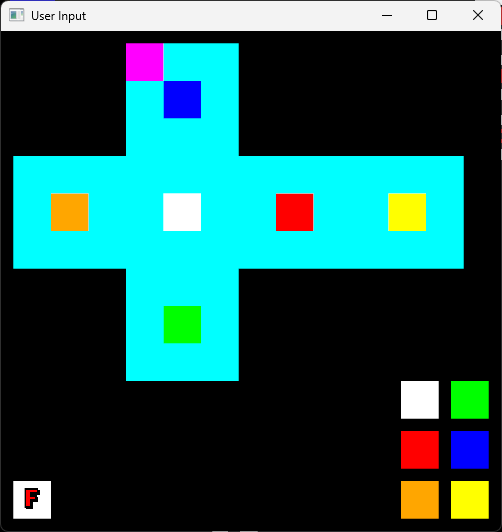


Рисунок 4 – Окно ввода кубика

Бирюзовым цветом отмечены квадраты, цвет которых еще не определен. Фиолетовым цветов отмечен квадрат, ввод цвета которого в данный момент ожидается от пользователя. Цвет выбирается с помощью одной из цветных кнопок, расположенных в правом нижнем углу окна ввода кубика. После указания цветов всех квадратов окно ввода кубика автоматически закрывается, а введенный пользователем кубик переносится на главное окно.

Пользователь может выбрать кубик из файла, нажав кнопку в F в левом нижнем углу окна ввода кубика. Нажатие кнопки создает системный диалог выбора файла аналогично диалогу открытия файла с шагами. Файл с состоянием кубика состоит из шести строк, порядок ввода цветов квадратов изображен на развертке куба, представленной на рисунке 5 (индексация начинается с нуля). Сначала вводится синяя сторона, потом оранжевая, потом белая, потом красная, потом желтая, и, наконец, зеленая. Допускаются как строчные, так и прописные буквы. Пример того, как должно записываться состояние кубика в этом файла, представлен на рисунке 6.

A colorful squares with numbers

Description automatically generated

Рисунок 5 – Порядок ввода сторон, ожидаемый от пользователя

A screenshot of a computer

Description automatically generated

Рисунок 6 – Пример файла с состоянием кубика

В случае ввода из файла все квадраты, введенные пользователем вручную, игнорируются, а на главном окне появляется кубик, состояние которого отражено в файле.

На этом этапе выполняется первичная проверка кубика на корректность по двум факторам:

Во-первых, у корректного кубика всегда совпадает количество квадратов всех цветов, логично, что оно должно быть равно 9.

Во, вторых, если следовать ранее введенной системе соответствий цветов и сторон, цвет центральных квадратов каждой стороны у корректного куба всегда совпадают с цветом, назначенным данной грани. Эта проверка неактуальна для ввода мышью (так как в этом режиме центральные квадраты не поддаются модификации), но в случае с вводом из файла позволяет не допустить попытки решения заведомо некорректного куба.

Важно отметить, что эти две проверки не отсекают все возможные некорректные кубы. Если куб некорректен, но этот факт не является очевидным, вывод о том, что решение отсутствует, будет сделан уже на этапе исполнения алгоритма.

# 2.3. СИНХРОНИЗАЦИЯ КУБИКОВ

Для синхронизации двух типов кубиков (линейного и матричного) были реализованы две функции, принимающие на вход кубик-назначение и кубик-источник.

Первая функция работает следующим образом: она построчно перебирает квадраты матричного куба, записывая в них цвета из соответствующих индексов линейного куба. Соответствие для каждой грани установлено с помощью массивов, заданных с упором на данные, изображенные на рисунке 5, сами же массивы представлены на рисунке 7. Эта функция не затрагивает служебные поля матричного куба, так как в линейном имеется лишь информация, необходимая для отрисовки, в то время как служебные поля необходимы только для алгоритма решения куба.

Вторая функция работает обратным образом: она построчно перебирает квадраты матричного куба, записывая их в соответствующие индексы линейного куба, руководствуясь тем же соответствием.

A screen shot of a computer code

Description automatically generated

Рисунок 7 – Соответствие индексов квадратов на развертке куба

# 3. АЛГОРИТМ РЕШЕНИЯ КУБИКА

В данной работе рассматривается трехслойный метод решения кубика, который является одним из наиболее популярных благодаря своей простоте и логичности. Метод предполагает сборку кубика слой за слоем и состоит из нескольких ключевых этапов:

Сборка креста передней грани: на первом этапе необходимо собрать крест на передней (обычно белой) грани кубика. Для этого нужно расположить четыре ребра так, чтобы их цвета совпадали с центральным элементом передней грани и с центральными элементами боковых граней. Используются простые повороты любых граней, кроме задней, для достижения этого результата [1, 10]. Пример кубика на данном этапе представлен на рисунке 8.



Рисунок 8 – Вид кубика после сборки белого креста

Сборка углов передней грани (всего верхнего слоя): следующий шаг включает установку угловых элементов передней грани. Углы должны быть размещены так, чтобы их цвета совпадали с цветами соседних центральных элементов, не нарушая собранный крест. Пример кубика на данном этапе представлен на рисунке 9.



Рисунок 9 – Вид кубика после сборки верхнего слоя

Сборка среднего слоя: на данном этапе необходимо расположить ребра среднего слоя кубика. Применяются алгоритмы, которые перемещают ребра в нужное положение, не разрушая при этом уже собранный верхний слой [1, 10]. Пример кубика на данном этапе представлен на рисунке 10.



Рисунок 10 – Вид кубика после сборки среднего слоя

Этот процесс аналогичен сборке креста на передней грани, однако используются другие алгоритмы для правильного расположения ребер. Например, не используется поворот передней грани. Пример кубика на данном этапе представлен на рисунке 11.



Рисунок 11 – Вид кубика после сборки креста на задней грани

Сборка углов на задней грани: на этом этапе угловые элементы задней грани должны быть установлены в правильные позиции. Применяются алгоритмы, позволяющие перемещать углы задней грани, сохраняя при этом собранный крест.



Рисунок 12 – Вид кубика после перемещения углов на задней грани

Ориентирование углов на задней грани: заключительный этап включает правильное ориентирование углов задней грани, чтобы все стороны кубика были одного цвета. Для этого используются всего лишь повороты трех граней. Пример кубика на данном этапе представлен на рисунке 12.



Рисунок 13 – Вид полностью собранного кубика

Полностью собранный кубик представлен на рисунке 13.

Трехслойный метод обладает преимуществами, такими как простота запоминания алгоритмов и относительная легкость выполнения. Он является отличной основой для начинающих и служит хорошей отправной точкой для изучения более сложных методов сборки кубика [10].

В приложении для реализации данного алгоритма были созданы функции проверки состояния каждого из этапов, а также функции поворота каждой из шести граней. Алгоритм основан на переборе различных поворотов граней и возвращении наиболее приближенных к нужному этапу состояний.

# 3.1. ФУНКЦИИ ПРОВЕРОК СОСТОЯНИЯ КУБИКА

Для реализации алгоритма перебора с возвратом были созданы функции проверки соответствия куба каждому из этапов сборки. Количество функций проверок больше, чем этапов сборки кубика, что связано с тем, что некоторые шаги алгоритма сборки были разбиты на подшаги для ускорения поиска. Каждая из восьми функций проверки возвращает числовое значение от 0 до 4, которое обозначает уровень соответствия куба нужному этапу.

Функция checkIfCrossFront проверяет собранность переднего креста только относительно передней грани, без учета соответствия цветов боковым граням. Данная проверка выполняется во второй функции – checkIfCrossSides.

Функция checkIfCornersFront проверяет собранность углов передней грани, а также их ориентированность, чтобы на смежных с передней гранях также было соответствие цветов. В данной функции значение собранности куба после каждой проверки увеличивается на 1 только при условии соответствия всех цветов каждой грани данного угла.

Функция checkIfMiddleRow проверяет собранность среднего ряда, а именно боковых краев на каждой из четырех смежных гранях. Функция проверки аналогична предыдущей, значение собранности куба после каждой проверки увеличивается на 1 только при условии соответствия всех цветов каждой грани данного края.

Функция checkIfBackCross проверяет собранность креста на задней части кубика. Особенностью функции является то, что она может вернуть только 0, 2 или 4 в результате проверки, так как она проверяет собранность креста как двух пересекающихся линий одного цвета. Это необходимо для реализации разбиения этапа сборки заднего креста на несколько шагов, то есть имеется возможность сначала собрать первую половину креста, и только потом вторую.

Функция checkIfBackSide проверяет собранность всей задней грани, функция аналогична checkIfCrossFront, только вместо частей передней грани проверяются части задней.

Функция checkIfBackCorners проверяет собранность углов задней грани. Функция по своему строению аналогична checkIfCornersFront.

Функция checkIfBackRow проверяет собранность всего последнего слоя, а именно только соответствие цветов на смежных с задней стороной гранях, так как цвета задней грани уже были проверены в функции checkIfBackSide, а углы в функции checkIfBackCorners.

Функция checkAllCube выполняет проверку собранности кубика. Она работает в двух режимах проверки, которые зависят от флага, установленного при вызове функции. Если флаг равен нулю, то функция последовательно запускает все функции проверок состояния кубика до тех пор, пока хотя бы одна из них вернет значение неравное четырем или пока не будут выполнены все проверки. Это необходимо для того, чтобы убедиться в том, что текущее состояние кубика не нарушило собранность всех предыдущих этапов. Если флаг больше нуля, то в таком случае функция будет возвращать значение собранности кубика вне зависимости от того нарушены предыдущие этапы или нет. Функция возвращает полученное значение контрольной суммы собранности кубика, которое в дальнейшем будет использоваться в алгоритме сборки.

# 3.2. ФУНКЦИИ РЕШЕНИЯ КУБИКА

Для решения кубика используется перебор с возвратом различных вращений. Для оптимизации перебора на различных этапах решения доступны только некоторые возможные вращения кубика. Для сборки переднего слоя используются все повороты, кроме вращения задней грани. Для сборки среднего слоя и креста на задней грани используются все повороты, кроме поворотов передней грани. Для последних этапов решения, а именно ориентировки углов и сборки всего последнего слоя, используются только три вращения. Такой подход позволяет отсекать повороты, которые бы не помогли привести куб в более собранное относительно конкретного этапа состояние [8].

Для реализации алгоритма с перебором создается специальная таблица поиска. SolutionTable – структура данных, которая хранит в себе счетчик открытых решений куба, счетчик закрытых решений куба, размер таблицы, а также массив состояний кубов.

В цикле по количеству этапов сборки, запускается функция search с различными настройками, которые обеспечивают поиск комбинации поворотов для нужного состояния. Функция search принимает указатель на таблицу поиска, куб, из состояния которого продолжается поиск, числовое значение собранности куба, которое должно быть у куба по окончании функции, массив операций, которые могут быть применены к кубу на данном этапе сборки, числовое значение максимального погружения поиска, а также минимальное числовое значение собранности куба, которое служит для отрезания заведомо неверных путей, в случае если ветвь поиска вернула слишком маленькое значение собранности. В зависимости от этапа сборки данные параметры изменяются, повышается значение собранности куба, к которому необходимо прийти, меняются массивы операций, а также постепенно увеличивается минимальное значение для отрезания неверных путей. Функция search возвращает указатель на собранный на данном этапе куб и, в случае если этот кубик не равен кубику, с которого был запущен поиск, то в файл с операциями записывается последовательность вращений, при помощи которых было получено текущее состояние кубика. В противном случае, если вернувшийся кубик равен тому, с которого был запущен поиск, в файл ничего не записывается. Если поиск не увенчался успехом (функция вернула NULL), в файл записывается символ ‘\_’, который будет означать, что куб невозможно собрать. В таком случае приложение выведет на экран «UNSOLVEABLE».

В функции search идет цикл, количество итераций которого равно числовому значению максимального погружения поиска, в котором к кубику применяется один из поворотов массива операций. После этого при помощи функции checkAllCube возвращается числовое значение полученного состояния кубика. Если по итогу поворота кубик пришел в необходимое состояние (или даже более собранное), то функция завершает работу и возвращает кубик. Если же в функции кончается память в таблице поиска или ей не удается найти решение для данного кубика, она возвращает NULL.

Блок-схемы ключевых функций, необходимых для сборки кубика, представлены в приложении В. Листинги этих функций представлены в приложении Г.

# ЗАКЛЮЧЕНИЕ

Данная курсовая работа посвящена разработке программного решения для моделирования и решения кубика Рубика 3x3 с использованием метода обхода в ширину. Приложение позволяет пользователю вводить состояние кубика как вручную, так и из файла, а затем просматривать пошаговое решение, которое генерируется алгоритмом. Работа включает в себя реализацию графического интерфейса с использованием библиотеки OpenGL, позволяющего визуализировать кубик и его повороты в изометрии. В программе применяются два представления кубика: одномерный массив для отрисовки и двумерная матрица для алгоритма решения. Состояние кубика синхронизируется между двумя представлениями для обеспечения корректности отображения.

Алгоритм решения кубика, основанный на трехслойном методе, реализован с помощью функции перебора с возвратом. Программа использует функции проверки состояния куба на каждом этапе сборки для оптимизации поиска решения.

Программа позволяет решать кубик Рубика с любым начальным состоянием, вводимым пользователем. Пользователь может наблюдать процесс решения куба в реальном времени, что позволяет изучать алгоритм и повышать понимание принципов решения головоломки. Программа может использоваться как инструмент для изучения алгоритмов решения кубика Рубика.

Разработан и реализован графический интерфейс приложения, позволяющий пользователю задавать начальное состояние кубика, управлять его поворотами и просматривать пошаговое решение. Реализован алгоритм решения кубика Рубика, основанный на трехслойном методе с использованием функции перебора с возвратом. Разработаны функции проверки состояния куба на каждом этапе сборки, позволяющие оптимизировать поиск решения. Создана система синхронизации состояний кубика в двух представлениях (одномерном массиве и двумерной матрице).

Несмотря на успешную реализацию алгоритма решения, в работе можно отметить некоторые ограничения:

Сложность алгоритма: алгоритм решения, основанный на переборе с возвратом, может быть ресурсозатратным для сложных начальных состояний куба.

Ограниченность алгоритма: программа не поддерживает другие методы решения кубика Рубика, такие как метод CFOP.

Данная курсовая работа демонстрирует успешную реализацию программного решения для моделирования и решения кубика Рубика 3x3, которое может использоваться как в развлекательных целях, так и в качестве инструмента для изучения алгоритмов решения головоломки.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Мячин, Т. С. Математика кубика Рубика / Т. С. Мячин, Д. С. Михеенко. — Текст : непосредственный // Юный ученый. — 2022. — № 5 (57). — С. 74-79. — URL: https://moluch.ru/young/archive/57/3018/ (дата обращения: 06.06.2024).
2. Buffer Object // OpenGL Wiki. - URL: https://www.khronos.org/opengl/wiki/Buffer\_Object (дата обращения: 06.06.2024).
3. Cube Exproler // Herbert Kociemba. - URL: https://kociemba.org/cube.htm (дата обращения: 06.06.2024).
4. GetOpenFileNameA function (commdlg.h) // Windows App Development. - URL: https://learn.microsoft.com/en-us/windows/win32/api/commdlg/nf-commdlg-getopenfilenamea (дата обращения: 06.06.2024).
5. GetSaveFileNameA function (commdlg.h) // Windows App Development. - URL: https://learn.microsoft.com/en-us/windows/win32/api/commdlg/nf-commdlg-getsavefilenamea (дата обращения: 06.06.2024).
6. Hello Window // LearnOpenGL. - URL: https://learnopengl.com/Getting-started/Hello-Window (дата обращения: 06.06.2024).
7. Input guide // GLFW. - URL: https://www.glfw.org/docs/3.3/input\_guide.html (дата обращения: 06.06.2024).
8. My system for solving Rubik's cube // Jessica Fridrich. - URL: http://www.ws.binghamton.edu/fridrich/system.html (дата обращения: 06.06.2024).
9. OpenGL 4 Reference Pages // Khronos Registry. - URL: https://registry.khronos.org/OpenGL-Refpages/gl4/html/ (дата обращения: 06.06.2024).
10. The Mathematics of the Rubik’s Cube // MIT. - URL: https://web.mit.edu/sp.268/www/rubik.pdf (дата обращения: 06.06.2024).
11. The OpenGL Graphics System: A Specification // The Khronos Group Inc. - URL: https://registry.khronos.org/OpenGL/specs/gl/glspec40.core.pdf (дата обращения: 06.06.2024).
12. WCA Regulations // World Cube Association. - URL: https://www.worldcubeassociation.org/regulations/ (дата обращения: 06.06.2024).
13. Window guide // GLFW. - URL: https://www.glfw.org/docs/3.3/window\_guide.html (дата обращения: 06.06.2024).
14. Zenity Manual // The GNOME Project. - URL: https://help.gnome.org/users/zenity/stable/ (дата обращения: 06.06.2024).

# ПРИЛОЖЕНИЕ А. БЛОК-СХЕМЫ ФУНКЦИЙ ОТРИСОВКИ ОКОН

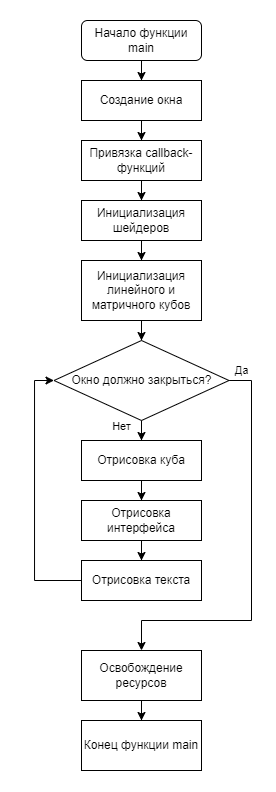


Рисунок 17 – Блок-схема функции main

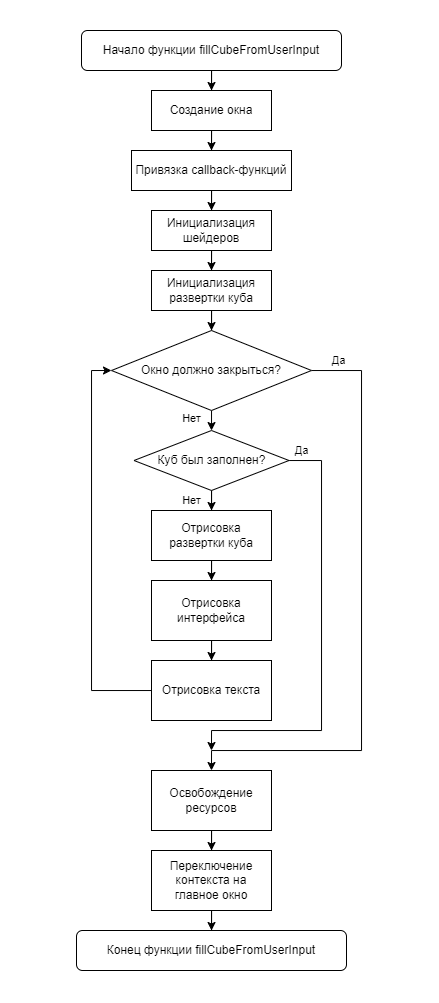


Рисунок 18 – Блок-схема функции fillCubeFromUserInput

# ПРИЛОЖЕНИЕ Б. ЛИСТИНГИ ФУНКЦИЙ ОТРИСОВКИ ОКОН

Листинг функции «main»

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

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

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

GLuint shaderProgram = createShaderProgram(vertexShaderInstanceSource, fragmentShaderInstanceSource);

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

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");

int width, height;

currentWindow = MAINWND;

// Main loop

while (!glfwWindowShouldClose(window)) {

glClear(GL\_COLOR\_BUFFER\_BIT);

glUseProgram(shaderProgram);

drawSide(RIGHT, Cube2D.redSide);

drawSide(LEFT, Cube2D.whiteSide);

drawSide(TOP, Cube2D.blueSide);

drawUI();

gltSetText(stepText, currentStepText);

gltInit();

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();

gltTerminate();

glfwSwapBuffers(window);

glfwPollEvents();

}

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);

currentWindow = FLATWND;

// gltInit();

GLTtext \*fileText = gltCreateText();

gltSetText(fileText, "F");

int width, height;

while (!glfwWindowShouldClose(flatCubeWindow)) {

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();

gltInit();

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();

gltTerminate();

glfwSwapBuffers(flatCubeWindow);

glfwPollEvents();

}

currentWindow = MAINWND;

gltDeleteText(fileText);

currentFlatCubeIndex = -1;

glfwDestroyWindow(flatCubeWindow);

glfwMakeContextCurrent(window);

}

# ПРИЛОЖЕНИЕ В. БЛОК-СХЕМЫ КЛЮЧЕВЫХ ФУНКЦИЙ АЛГОРИТМА РЕШЕНИЯ КУБИКА

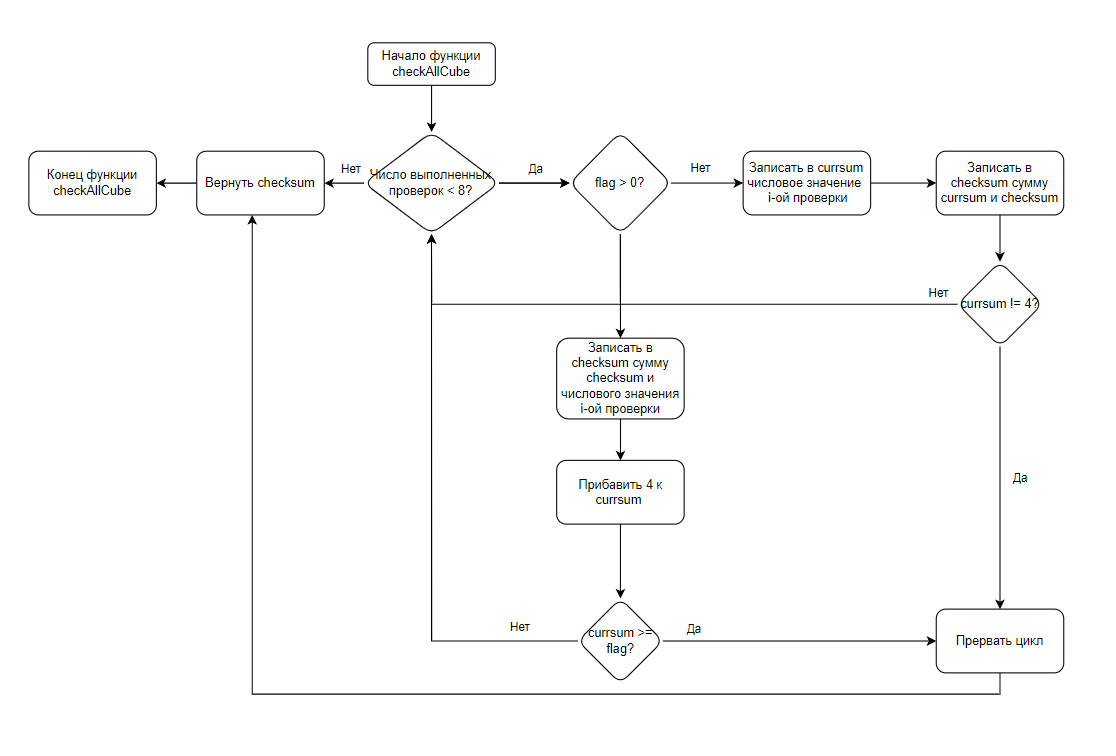


Рисунок 14 – Блок-схема функции checkAllCube

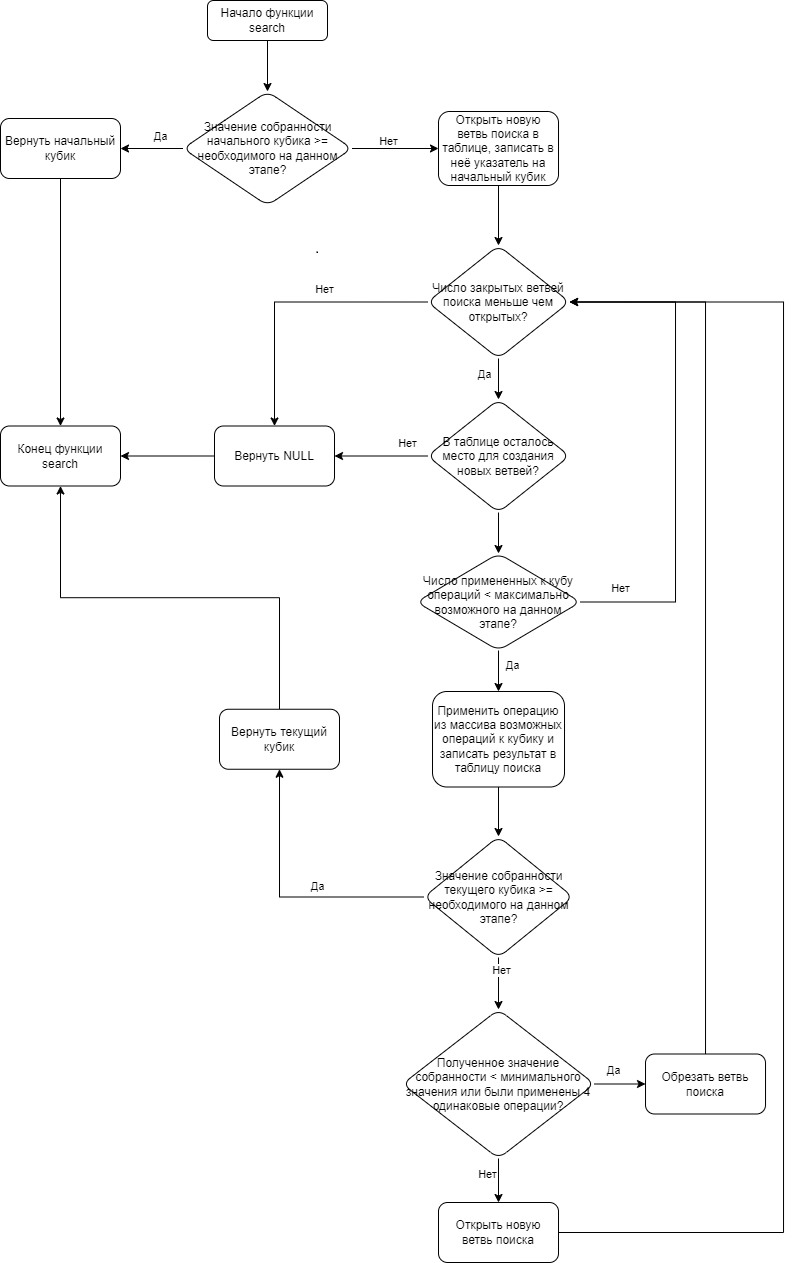


Рисунок 15 – Блок-схема функции search

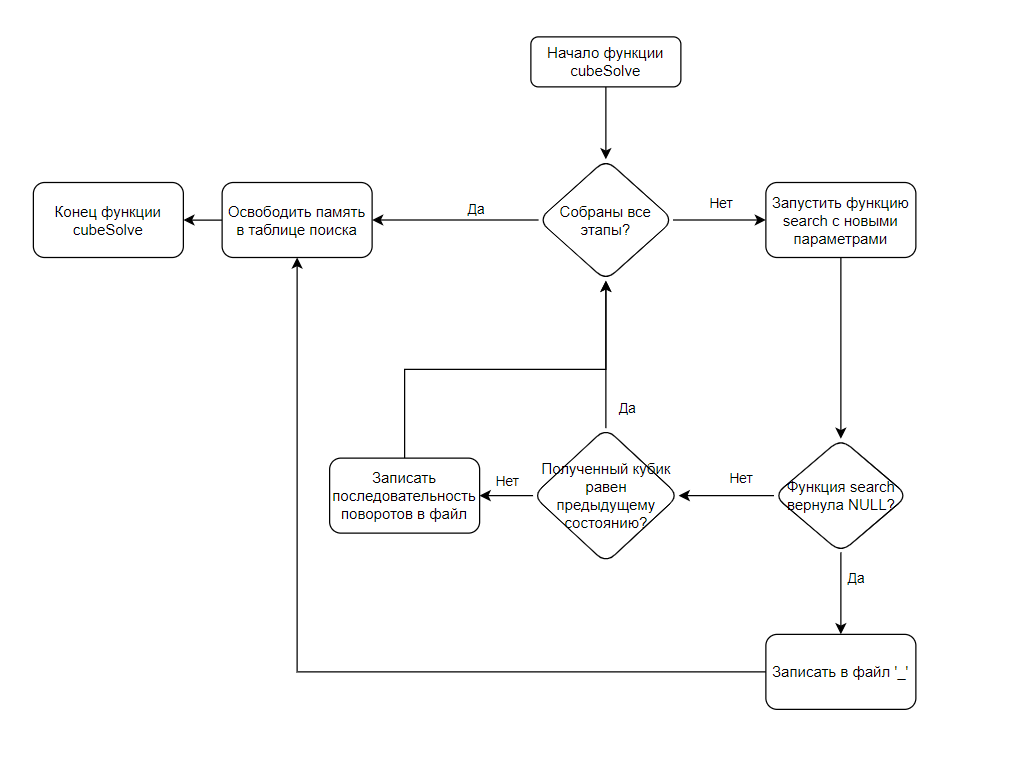


Рисунок 16 – Блок-схема функции cubeSolve

# ПРИЛОЖЕНИЕ Г. 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++;

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

{

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

{

//all memory was used

break;

}

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

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

{

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

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

cubeDoOp(newcube, operlist[i]);

newcube->operation = operlist[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

{ 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\* foutput = fopen("ops.txt", "w");

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

fputc('\_', foutput);

//Couldn't find

// fclose(foutput);

break;

}

}

deleteTable(&table);

}