**ОЛИМПИАДА ШКОЛЬНИКОВ «ШАГ В БУДУЩЕЕ»**

**НАУЧНО-ОБРАЗОВАТЕЛЬНОЕ СОРЕВНОВАНИЕ «ШАГ В БУДУЩЕЕ, МОСКВА»**

473

*регистрационный номер*

**Информатика и системы управления**
*название факультета*

**Программное обеспечение ЭВМ и информационные технологии**
*название кафедры*

# Визуализация интерактивной 3D модели кубика Рубика и алгоритмов по его сборке

*название работы*


**Автор:** ***Кутовой Егор Арсеньевич***
*фамилия, имя, отчество*
**ГБОУ школа 1501 г. Москвы, 11**
*наименование учебного заведения, класс*

**Научный руководитель:** **Филиппов Константин Сергеевич**
*фамилия, имя, отчество*
**ГБОУ школа 1501 г. Москвы**
*место работы*
**Учитель информатики**
*звание, должность*

_____
*подпись научного руководителя*


**Москва - 2019**

# *Визуализация интерактивной 3D модели кубика Рубика и алгоритмов по его сборке*

## Аннотация

В работе была поставлена цель создать программу, предоставляющую удобную интерактивную среду для изучения работы алгоритмов, решающих головоломку кубик Рубика.

Разработанная программа обладает следующими функциями:

1) визуализирует интерактивную 3D модель кубика рубика;
2) реализует оптимальные алгоритмы решения;
3) предоставляет несложный программный интерфейс для добавления новых алгоритмов;
4) позволяет вводить конфигурацию реального кубика рубика в программу с помощью веб-камеры.

В работе над проектом были использованы библиотеки GLFW, GLAD, FreeType, OpenCV, а также реализации оптимальных алгоритмов решения кубика рубика https://github.com/muodov/kociemba, http://kociemba.org/cube.htm. Для написания программного кода использовалась интегрированная среда разработки Microsoft Visual Studio 2017, а также система контроля версий git и сервис Github.

Репозиторий с кодом программы доступен по ссылке:
https://github.com/muji-4ok/Cube3D

# Оглавление

# Предпосылки проекта

Кубик Рубика – головоломка, ставшая одной из самых популярных в мире из-за ее математических свойств. Ее анализ способствует открытиям в некоторых областях математики (комбинаторика, теория множеств, теория графов, теория вычислимости), а также улучшает индивидуальное понимание этих областей.

Научиться решать кубик Рубика не так сложно, так как существует много инструкций по сборке. Но человек не может научиться решать его самым оптимальным способом, то есть используя минимальное количество ходов, без помощи компьютера. Поэтому поиск оптимального алгоритма является предметом исследований.

Сегодня существует много программ, предоставляющих интерактивную 3D модель головоломки, а также вычисляющих оптимальное решение для любой ее начальной конфигурации.

Наиболее популярные программы, визуализирующие кубик Рубика:

1. http://iamthecu.be/,
2. https://ruwix.com/online-puzzle-simulators/,
3. https://rubiks-cu.be/,
4. http://kociemba.org/cube.htm,
5. https://github.com/bluquar/cubr.

Первые три программы способны только визуализировать головоломку, четвертая находит оптимальные решения, но визуализирует только в 2D, а последняя – визуализирует в 3D и решает, но не самым оптимальным способом.

# Цели и задачи

## Цель

Создать программу, предоставляющую удобную интерактивную среду для изучения работы оптимальных алгоритмов.

## Задачи

1. Изучить библиотеки, применяемые в процессе работы;
2. Разработать программу.

## Требования к функционалу

Что должна делать программа:

1. Визуализировать интерактивную 3D модель кубика Рубика;
2. Реализовывать оптимальные алгоритмы решения;
3. Предоставлять понятный программный интерфейс для добавления новых алгоритмов решения;
4. Считывать и загружать конфигурацию реального кубика Рубика в модель с помощью веб-камеры.

# Изучение и анализ библиотек

1. Для визуализации выбрана библиотека OpenGL, гарантирующая высокую производительность приложения.
2. Для взаимодействия с пользователем и создания контекста OpenGL выбрана библиотека GLFW, обеспечивающая простоту использования и кроссплатформенность.
3. Для подключения функций OpenGL выбрана библиотека GLAD, потому что она предоставляет наиболее полный функционал и проста в использовании.
4. Библиотека FreeType выбрана для работы с текстом на экранах. Библиотека обладает всем необходимым набором функций и подробным руководством.
5. GLM выбрана для дополнительных математических функций и структур данных для работы с векторами и матрицами. Главные преимущества – высокая скорость работы и интеграция с GLSL (OpenGL Shading Language, язык шейдеров OpenGL).
6. OpenCV выбрана для работы с веб-камерой. Библиотека обладает всем необходимым набором функций и подробным руководством.
7. В процессе работы использовались два алгоритма по сборке кубика Рубика. Один – очень быстрый, но не оптимальный. Второй – наоборот, медленный, но оптимальный. В результате я остановил выбор на реализациях алгоритмов Two-Phase muodov/ckociemba и Optimal Solver от Kociemba. Преимущества этих реализаций в том, что для них есть исходные файлы на языке C. Поэтому я мог сделать из них библиотеки для C++ без чрезмерных усилий.

# Состав проделанной работы

1. Созданы классы для работы с объектами OpenGL.
2. Создан класс, реализующий основной цикл работы программы.
3. Создан класс, содержащий в себе состояние кубика Рубика, и реализована отрисовка кубика Рубика.
4. Реализовано вращение куба с помощью мыши и клавиатуры.
5. Алгоритм Two-Phase встроен в программу.
6. Для более эффективной дальнейшей работы над приложением выполнен переход к архитектуре MVC.
7. Созданы классы, описывающие виджеты пользовательского интерфейса.
8. Полностью реализован интерфейс пользователя.
9. Встроен алгоритм Optimal Solver.
10. Реализован вывод изображения с веб-камеры.
11. Реализована функция ввода состояния кубика Рубика с веб-камеры в приложение.
12. Добавлена инструкция для работы с API.

# Ход и специфика работы

## Классы для работы с OpenGL

Чтобы вывести что-либо с помощью OpenGL необходимо как минимум несколько объектов: Vertex Buffer Object, Vertex Array Object, Shader, Shader Program. Еще один важный объект – Texture, но он не является необходимым. При вызове создающих их функций эти объекты на самом деле создаются на GPU, а программа на CPU получает ID этих объектов, чтобы ими в дальнейшем манипулировать.

В Vertex Buffer Object (или VBO) содержатся все данные для каждой вершины треугольника. Vertex Array Object (или VAO) определяет, как необходимо интерпретировать данные вершин. Все описанные данные будут доступны в шейдерах во время отрисовки. Также, в шейдерах будут доступны дополнительные uniform переменные, загружаемые с CPU, и данные о текстурах.

Шейдер – это программа, которая параллельно выполняется на GPU. Существует 3 вида шейдеров: вершинные, геометрические и фрагментные. В разрабатываемой программе используются только вершинные и фрагментные. Перед использованием шейдеры должны быть скомпилированы, а их код хранится в так называемой шейдерной программе. Эта программа объединяет сразу все 3 вида шейдеров, потому что они очень тесно взаимодействуют друг с другом.

Когда объекты созданы, их нужно сделать доступными для считывания при непосредственном рисовании треугольников. Также, перед рисованием нужно установить некоторые параметры, такие как размер окна, поведение при пересечении нескольких треугольников или обработке полупрозрачных текстур.

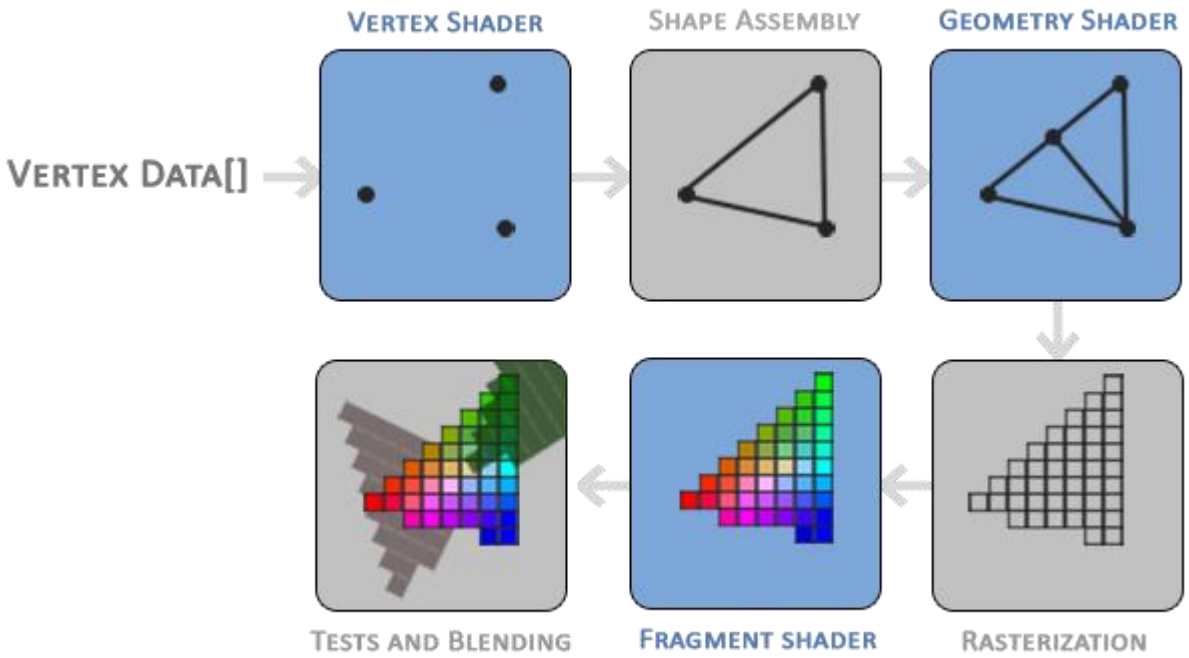



Рисунок 1 — Принцип работы OpenGL

## Основной цикл работы программы

Любая графическая программа содержит цикл, выполняющийся с момента старта программы до закрытия. Тело цикла:

1. Обработка пользовательских событий.
2. Обновление состояния объектов в соответствии с полученными событиями и предыдущим состоянием.
3. Вывод на экран.

Такой цикл присутствует в программе в явном или скрытом виде, в зависимости от используемых библиотек и фреймворков. В разрабатываемой программе цикл присутствует в явном виде.

Обработка пользовательских событий и вывод на экран всегда происходят по-разному в разных операционных системах (далее ОС). Поэтому, чтобы писать кроссплатформенный код, была использована библиотека GLFW, определяющая интерфейс, независимый от ОС.

GLFW предоставляет функции создания окон в ОС, создания контекста OpenGL, а также callback функции для обработки событий. Все эти функции были собраны в одном классе, чтобы облегчить к ним доступ.

Для рисования на экране, помимо контекста, нужно загрузить указатели на функции OpenGL. Это можно сделать вручную, загружая каждую используемую в программе функцию, а можно использовать библиотеку-загрузчик, подключающую все функции для используемой версии OpenGL. Второй вариант лучше, т. к. это освобождает от написания лишнего кода и, как следствие, ошибок.

Для загрузки функций была выбрана библиотека GLAD, из-за простоты конфигурации и использования.

## Состояние и отрисовка кубика

Для отрисовки куба и взаимодействия с ним создан специальный класс состояния.

Состояние включает: матрицы вращения и трансляции, цвета всех сторон, положение и поворот отдельных мини-кубов в 3D пространстве, неизменяемые данные вершин, необходимые для отрисовки куба в OpenGL и шейдеры.

```
16  struct CubeModel
17  {
18      CubeModel(const WindowModel* wm, glm::vec3 translation_vec = glm::vec3(0.0f, 0.0f, -3.0f));
19
20      void reset_view_rotation();
21      void reset_rotations();
22      void full_reset();
23      void copy_colors(const CubeModel& other);
24      bool is_valid() const;
25
26      RotationQueue rotationQueue;
27      HitModel hitModel;
28
29      std::array<std::array<std::array<Cubelet, 3>, 3>, 3> cubelets;
30
31      glm::mat4 view;
32      glm::mat4 rotation_view;
33      glm::mat4 translation_view;
34
35      const WindowModel* windowModel;
36  };
```

Рисунок 2 — Описание модели кубика рубика

## Вращение куба

В процессе я разработал два разных вида вращения: вращение отдельных элементов и вращение всей модели.

Принцип работы алгоритма для вращения всей модели мышью:

1. При перемещении курсора (с зажатой правой кнопкой мыши) вычисляется вектор перемещения курсора относительно его положения на предыдущем кадре.
2. По вектору перемещения вычисляется ось вращения, а по длине вектора – угол поворота.

3. Вычисляется матрица поворота вокруг вычисленной оси на вычисленный угол.
4. Существующая матрица вращения доумножается на полученную.

Принцип работы алгоритма для вращения отдельных элементов куба мышью:

- При нажатии левой кнопки мыши:
  a. Вычисляется пересечение луча, идущего из центра окна и проходящего через точку положения курсора со всеми треугольниками, составляющими модель куба.
  b. Из всех пересеченных треугольников выбирается ближайший к экрану.
  c. По полученному треугольнику определяется мини-куб и его сторона, в которую попал луч.
- При перемещении мыши с зажатой левой кнопкой:
  a. По вектору перемещения определяется – какая из осей, перпендикулярных граням куба, лучше всего ему подходит.
  b. По длине вектора перемещения и направлению определяется угол поворота.
  c. Вычисляется матрица поворота.
  d. Матрицы вращения для каждого мини-куба доумножаются на полученную.
- При отпускании левой кнопки мыши:
  a. Запускается анимация, доворачивающая все мини-кубы, повернутые на какой-либо угол, до ближайшего угла, кратного 90°.
- При нажатии клавиш “U”, “D”, “L”, “R”, “F”, “B”:
  a. Запускается анимация, поворачивающая грани.

## Алгоритм Two-Phase

Кубик Рубика имеет примерно 43 квинтиллиона различных конфигураций (43 * $10^{18}$). Удивительно то, что все возможные конфигурации могут быть решены за 26 поворотов боковых граней.

Задача оптимального алгоритма заключается в поиске такой комбинации поворотов минимальной длины, которая решает текущую конфигурацию. В основе всех современных алгоритмов лежит “умный” перебор ходов.

Алгоритм Two-Phase не гарантирует оптимальность найденных решений, хотя перебирает большое количество комбинаций ходов.

Из-за того, что алгоритм Two-Phase ищет не самое оптимальное решение, он работает очень быстро и во всех достаточно сложных конфигурациях кубика его решения будут быстрее любых решений, которые способен найти человек. Именно поэтому в программах для спидкубинга или для механического решения роботом реализован данный алгоритм, так как в реальных условиях оптимальный алгоритм всегда работает слишком долго, и выигрыш в количестве ходов не компенсирует сложность нахождения оптимальной комбинации.

Оригинальная программа для командной строки, была переделана в библиотеку для моей программы. Реализован алгоритм перевода данных, обозначающих состояние кубика Рубика, из разрабатываемой программы в формат CKociemba.

## Переход к MVC

Архитектура MVC (Model-View-Controller) - одна из самых популярных объектно ориентированных архитектур для графических приложений с пользовательским интерфейсом. Основная идея MVC, в том, чтобы разделять данные приложения,

пользовательский интерфейс и логику работы программы на 3 компонента: модель (model), представление (view) и контроллер (controller).

Модель содержит данные и реагирует на команды контроллера, изменяя свое состояние. Представление отвечает за отображение данных модели на экране и получение пользовательских действий. Контроллер обрабатывает пользовательские действия и дает модели команды об изменении ее состояния.

Самое главное преимущество MVC – масштабируемость, необходимая для добавления новых функций в программу. Поэтому, я провел рефакторинг кода программы так, чтобы он соответствовал архитектуре MVC.

## Классы, описывающие виджеты

Добавлены модели, контроллеры и представления для кнопок, списков, текстовых полей и панелей с изображением.

## Интерфейс

Разработан полностью масштабируемый интерфейс для двух экранов программы:

1. Интерактивный экран, где можно вращать куб и решать.
2. Экран ввода с веб-камеры.

## Алгоритм Optimal Solver

В отличие от описанного выше алгоритма Two-Phase, этот алгоритм находит оптимальное решение Кубика Рубика.

Для добавления этого алгоритма мне пришлось переделать оригинальную программу и реализовать перевод данных состояния кубика рубика в формат, принимаемый алгоритмом Optimal Solver.

## Вывод изображения с веб-камеры

Виджет панели с изображением пришлось доработать так, чтобы он мог принимать ввод с веб-камеры на каждом цикле работы программы. На данном этапе была добавлена библиотека OpenCV.

## Обработка ввода с веб-камеры

Ввод цветов с камеры происходит для каждой из 6 граней куба. А каждая грань состоит из 9 областей.

Алгоритм определения цвета считываемой области:

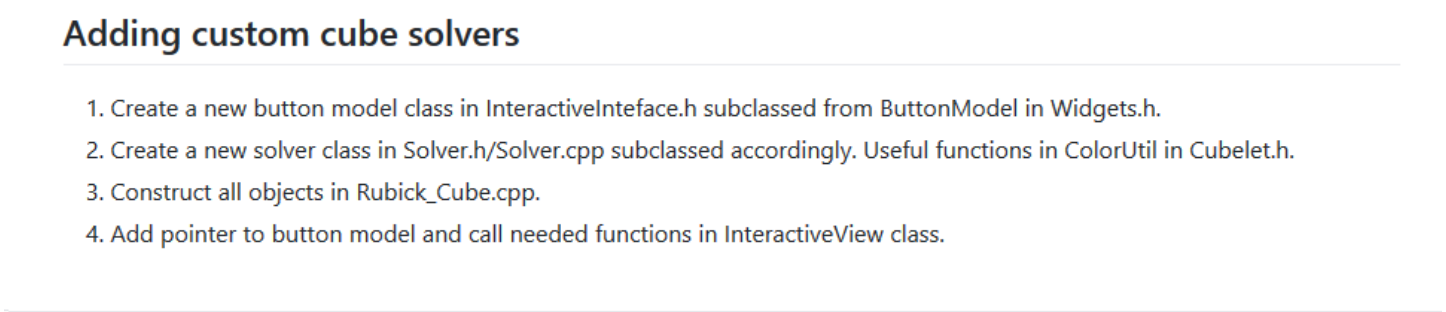
1. Вычисляется среднее значение цвета в области.
2. Вычисляется его тон.
3. Тон области сравнивается с 6 тонами стандратных цветов граней кубика рубика, и выбирается наиболее похожий.

```
118  SideColor ColorUtil::guessColor(const glm::vec3 & colorVec)
119  {
120      static std::map<SideColor, float> colorMap{
121          { Yellow, ColorUtil::getHue(ColorUtil::toGlmVec(Yellow)) },
122          { White, ColorUtil::getHue(ColorUtil::toGlmVec(White)) },
123          { Red, ColorUtil::getHue(ColorUtil::toGlmVec(Red)) },
124          { Orange, ColorUtil::getHue(ColorUtil::toGlmVec(Orange)) },
125          { Blue, ColorUtil::getHue(ColorUtil::toGlmVec(Blue)) },
126          { Green, ColorUtil::getHue(ColorUtil::toGlmVec(Green)) },
127      };
128
129      auto colorHue = ColorUtil::getHue(ColorUtil::normalizedColor(colorVec));
130      SideColor bestGuess = White;
131      float bestDiff = 1e6;
132
133      for (auto it = colorMap.begin(); it != colorMap.end(); ++it)
134      {
135          auto diff = std::abs(it->second - colorHue);
136
137          if (it->first == Red)
138              diff = std::min(diff, std::abs(360.0f - colorHue));
139
140          if (diff < bestDiff)
141          {
142              bestDiff = diff;
143              bestGuess = it->first;
144          }
145      }
146
147      return bestGuess;
148  }
```

Рисунок 3 — Функция для определения цвета области

## Программный интерфейс для добавления новых алгоритмов

Добавлено описание объектов программы, отвечающих за добавление новых алгоритмов.



### Adding custom cube solvers

1. Create a new button model class in InteractiveInteface.h subclassed from ButtonModel in Widgets.h.
2. Create a new solver class in Solver.h/Solver.cpp subclassed accordingly. Useful functions in ColorUtil in Cubelet.h.
3. Construct all objects in Rubick_Cube.cpp.
4. Add pointer to button model and call needed functions in InteractiveView class.

Рисунок 4 — Описание процесса добавления новых алгоритмов

# Итоги

Разработана программа, решающая поставленные задачи:

- визуализировать оптимальные или любые другие алгоритмы сборки кубика Рубика в 3D;
- использовать веб-камеру для загрузки реальной конфигурации кубика для поиска ходов ее решения.

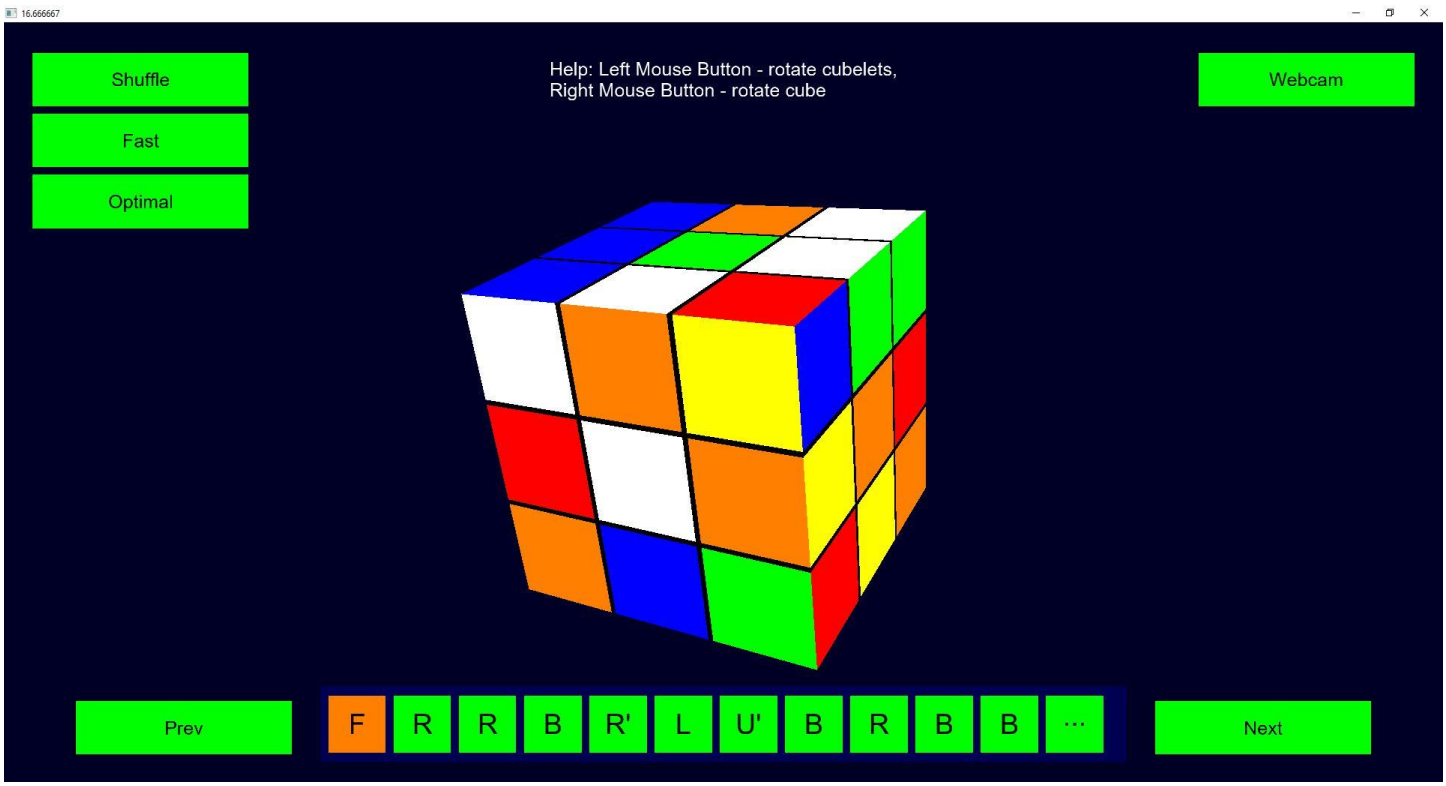



Рисунок 5 — Интерактивный экран

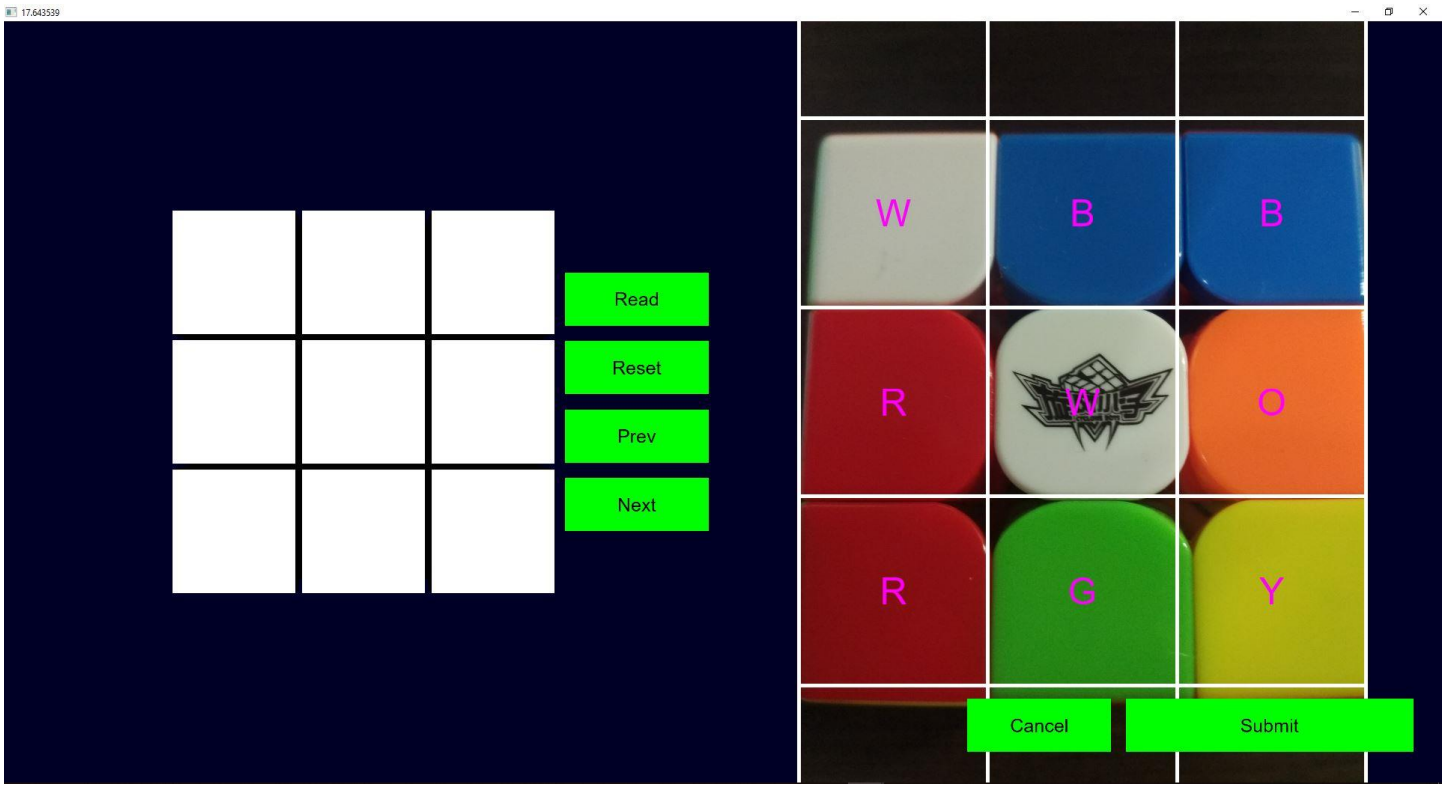



Рисунок 6 — Экран ввода с веб-камеры

# Использованные источники

1. https://learnopengl.com/
2. http://antongerdelan.net/opengl/raycasting.html
3. https://en.cppreference.com/w/
4. https://www.glfw.org/docs/latest/
5. http://glm.g-truc.net/0.9.9/api/index.html
6. https://www.khronos.org/registry/OpenGL-Refpages/gl4/
7. https://docs.opencv.org/3.4.2/d9/df8/tutorial_root.html
8. https://tproger.ru/translations/design-patterns-simple-words-1/
9. https://stackify.com/solid-design-principles/