**­­­Augmented Reality Rubik Cube Solve Design Doc**

[1 - Introduction 2](#_Toc407205492)

[2 - Goals 2](#_Toc407205493)

[2.1 - Solve Rubik Cube from Arbitrary Starting Position 2](#_Toc407205494)

[2.2 - Target Devices (Smart Glasses) 2](#_Toc407205495)

[2.3 - Operation in a natural environment 2](#_Toc407205496)

[2.4 - Gesture Driven 2](#_Toc407205497)

[2.5 - Productization 2](#_Toc407205498)

[3 - Architecture 3](#_Toc407205499)

[3.1 - Controller 3](#_Toc407205500)

[3.2 - Model (or State) 3](#_Toc407205501)

[3.3 - View 3](#_Toc407205502)

[4 - Design 3](#_Toc407205503)

[4.1 - Image Processing 3](#_Toc407205504)

[4.2 - Face Recognition (i.e., 2D-Reconstruction) 4](#_Toc407205505)

[4.3 - Cube Recognition (i.e., 3D Reconstruction) 4](#_Toc407205506)

[4.4 - Gesture Recognition 4](#_Toc407205507)

[4.5 - Application State Machine 4](#_Toc407205508)

[4.6 - 4](#_Toc407205509)

[5 - Event Model 4](#_Toc407205510)

[5.1 - Thread Model 4](#_Toc407205511)

[5.2 - Tracking and Feedback 4](#_Toc407205512)

[6 - Class Diagram of Persistent Objects 5](#_Toc407205513)

[7 - Class Diagram of Per-Frame Objects 6](#_Toc407205514)

[8 - Parameters 6](#_Toc407205515)

[8.1 - Gaussian Blur Kernel Size 6](#_Toc407205516)

[8.2 - Canny Edge Upper Threshold 6](#_Toc407205517)

[8.3 - Canny Edge Lower Threshold 7](#_Toc407205518)

[8.4 - Dilation Kernel Size 7](#_Toc407205519)

[8.5 - Minimum Contour Area Size 7](#_Toc407205520)

[8.6 - Polygon Epsilon Threshold 7](#_Toc407205521)

[8.7 - Minimum Parallelogram Area Size 7](#_Toc407205522)

[8.8 - Maximum Parallelogram Area Size 7](#_Toc407205523)

[8.9 - Parallelogram Angle Outlier Threshold 7](#_Toc407205524)

[8.10 - Face Least Means Square Sigma Threshold 7](#_Toc407205525)

[8.11 - Manual Luminous Offset 7](#_Toc407205526)

[9 - Notes 7](#_Toc407205527)

[10 - Stable Face Event State Diagram 7](#_Toc407205528)

[11 - New Stable Face Event State Diagram 8](#_Toc407205529)

[12 - Application State Machine State Diagram 9](#_Toc407205530)

[12.1 - App State Definitions 10](#_Toc407205531)

[13 - 2D Model Reconstruction 11](#_Toc407205532)

[13.1 - Calculate Metrics 11](#_Toc407205533)

[13.2 - Initial Layout 11](#_Toc407205534)

[13.3 - Lease Means Square 11](#_Toc407205535)

[13.4 - Move Tile (if necessary) 11](#_Toc407205536)

[14 - 2D to 3D Reconstruction 12](#_Toc407205537)

[14.1 - Coordinate Systems 12](#_Toc407205538)

[14.2 - Rubik Cube definition in Real World Coordinates 12](#_Toc407205539)

# Introduction

This document describes the associated Android application that can visually recognize a Rubik Cube and guide the user through edge rotation steps to return a cube to its’ original configuration. The application is written nearly completely in Java and makes heavy use of OpenCV for image recognition. This document describes the architecture, design, implementation and performance achievements of this application.

# Goals

## Solve Rubik Cube from Arbitrary Starting Position

This is of course the main purpose of the application.

## Target Devices (Smart Glasses)

While developed on a standard commercial Android Smart Phone, the target platform is a set of Fully Immersive Smart Glasses such as CastAR from Technical Illusions, Moverio BT-200 from Epson, Meta from Space Glasses, and ATHEER AiR™ from Atheer One to name a few.

## Operation in a natural environment

The application must work in a normal cluttered environment without necessity of specific background or specific lighting. However, with that said, there will exist some backgrounds and some lighting conditions under which the application will not perform. It is the long term goal of this project to operate as robustly as possible in as many different environmental conditions as possible.

## Gesture Driven

In normal mode there are no menus or text displays, rather the application recognizes the current position and status of the cube and provides graphics instructions on how to solve the cube.

## Productization

It is undecided at the time of this writing whether this application will be published on Google Play Store for general use. It does not make sense to run this application on a Smart Phone or Tablet since two hands are required to rotate the cube and an additional hand to hold the device. Smart Glasses, especially Android centric devices, are in their infancy and little standardization has yet occurred. In all probability, Augmented Reality application like this will need to be hand ported to each and every Smart Glasses device for some time to come.

# Architecture

In reality, the core architecture is just the simple “Model-View-Controller” pattern, or for this application, more logically phrased as a “Controller-Model-View” pattern. Specifically:

## Controller

This lion-share of the application complexity in both code and CPU consumption is in this domain. Specifically, each incoming Camera Image Frame is fully analyzed in an attempt to recognize a Rubik Face. Then, additional processing is performed in class StateMachine to determine if the Model (or State) should be changed, and, when all six sides of the Rubik Cube have been recorded, the TwoPhase[] Rubik Cube Logic solution is called.

## Model (or State)

Rubik Cube state is captured in the simple class StateModel.java. Besides the cube six faces, a small amount of additional data (state) is held in this class. Otherwise, this represents pretty much the full state of the system.

## View

There are actually four separate “Views,” they are:

### Diagnostic Annotations

A wide range of diagnostic information is available through the Menu->Annotation user interface. This is rendered using OpenCV on the frame image itself. Information broader that what is in state is offered.

### Pilot Cube

At the time of this writing, the Pilot Cube is immature, but its long term purpose is to represent the state of the cube both in tiles and faces, and eventually to show requested Cube Rotations. This is rendered using OpenGL.

### Requested Text Directions

Text instructions to user to rotate entire cube or rotate an edge can be optionally displayed as a text message. This is rendered using OpenCV on the frame image itself.

### Requested Arrow Directions

Graphic instructions in the form of 3D arrows are rendered in a fashion overlaying the cube itself with proper position, orientation and size.

# Design

Clearly the “Controller” is the more interesting and challenging aspect of this application.

## Image Processing

Also correlates to what OpenCV offers.

### Grey Scale

### Gaussian Blur

### Edge Detection

### Contour Detection

### Polygon Detection

### Rhombus Detection

## Face Recognition (i.e., 2D-Reconstruction)

### Polygon Detection

### 3x3 Tile Grid Reconstruction

### Tile Color Detection

## Cube Recognition (i.e., 3D Reconstruction)

Purpose is to produce OpenGL required cube translation and rotation.

### Definition of Cube in GL space

### Homography and Perspective Transformation equations

### Over constrained Non-Linear equations with six unknowns

## Gesture Recognition

### Stable Face Detection

### Stable New Face Detection

## Application State Machine

### Guided Exploration Phase

### Solution Computation

### Guided Solution Phase

# Event Model

## Thread Model

Main Thread: - That which constructs the app?

UI Thread: - Menu events are handled here

OpenCV Frame Thread: - Camera Frame image is provided and processed.

OpenGL Thread: - Separate thread on which OpenGL renderers are executed.

## Tracking and Feedback

For simplicity, no frame to frame information path exists. That is, each frame is processed without any information from the previous frame.

# Class Diagram of Persistent Objects

**+**

Activity

Menu and Params

Image Recognizer

App State Machine

State Model

Face

Instructions View

Overlay View

Pilot Cube View

# Class Diagram of Per-Frame Objects

**+**

**+**

**+**

Rubik Face

Rhombus

Polygon

Contour

Constant Tile Color

Least Means Square

Profiler

Frame

Measured Tile Color

# Parameters

The image recognition process uses many “offline parameters” that can be adjusted to improve the process. Manual adjustment is, of course, undesired. However, achieving automatic (i.e., “online”) determination of these parameters shall be part of the maturing of this application.

All determined, more or less, empirically.

These parameters can be adjusted using the menu option “parameters.” All parameters are modeled as double precision floating point, and the purpose they serve in the application is as follows:

## Gaussian Blur Kernel Size

The square kernel size of the Gaussian Filter. The default is set to 3. The filter sigma will be computed from the kernel size using the formula: sigma = 0.3\*((ksize-1)\*0.5 - 1) + 0.8.

## Canny Edge Upper Threshold

Gradient threshold at which an edge is declared to exist.

## Canny Edge Lower Threshold

Gradient threshold at which an edge (once detected) is declared to no longer exist.

## Dilation Kernel Size

## Minimum Contour Area Size

Minimum contour area size to be considered a tile candidate.

## Polygon Epsilon Threshold

## Minimum Parallelogram Area Size

Minimum parallelogram area size to be considered a tile candidate.

## Maximum Parallelogram Area Size

Maximum parallelogram area size to be considered a tile candidate.

## Parallelogram Angle Outlier Threshold

The angles (in degrees, there are two angles) that the parallelogram sides make with respect to the absolute coordinate system is calculated. The mean of these angles among the entire candidate set is calculated. They should roughly have all the same value. If any parallelogram exhibits a sufficiently different shape (i.e., angle of sizes) it is rejected.

## Face Least Means Square Sigma Threshold

Threshold for residual Least Means Square error of fitting Parallelogram centers into a 3x3 grid as defined by their candidate grid location and their actual measured centers. If the residual is less than this value, the candidate tile grid locations are considered valid.

## Manual Luminous Offset

# Notes

* Prefix Rubik on all allows for easy identification and use of common names like Util and Controller.
* Significant use of public key word on member variables.
* Public vs Private: Member data is often made public in this implementation. Compliance with the convention of keeping member data private as much as possible results in too much code being in a very few files: particularly cube state and face state information. The convention of getters and setters is possible, but really accomplished nothing in terms of member data containment. I believe this issue is practically an outcome of the use of the MVC design pattern.
* Inter UI vs Frame thread contention handled by use of simple Boolean variables: i.e., atomic by nature.

# Stable Face Event State Diagram

The purpose of this state machine is to determine when a stable Rubik face is present. Stable is defined by the observance of four consecutive faces that have the identical tiles.

Other State Member Data:

Last :== Last Valid Face Object

Count :== Integer for counting

On Face Event

[ Face == null ]

Count = 0

[ Face != null ]

Last = Face

Count ++

[ Face == Last && Count < 4 ]

Count ++

[ Face == Last && Count >= 4 ]

[ Face == Last ]

[ Face != Last ]

[ Face != Last]

Count = 0

[ Face == Last ]

[ Face != Last && Count < 4 ]

Count ++

[ Face != Last && Count >= 4 ]

On Stable Face Event

Off Stable Face Event

# New Stable Face Event State Diagram

The purpose of this state machine is to determine when a new and different stable face is observed.

Other State Member Data:

Last :== Last Valid Face Object

Count :== Integer for counting

On Stable Face Event

[ Face == null ]

Count = 0

[ Face != null ]

Last = Face

Count ++

[ Face == Last && Count < 4 ]

Count ++

[ Face == Last && Count >= 4 ]

[ Face == Last ]

[ Face != Last ]

[ Face != Last]

Count = 0

[ Face == Last ]

[ Face != Last && Count < 4 ]

Count ++

[ Face != Last && Count >= 4 ]

On Stable Face Event

Off Stable Face Event

Off Stable Face Event

# Application State Machine State Diagram

Other State Member Data:

Status :== True if all six sides of Rubik Cube have been observed

Count :== Integer for counting

On Face Event

Off Stable Face Event

On Stable Face Event

Count= 0

[ Count >= 4 ]

[ Count < 4 ]

Count++

[ Status == False ]

Count = 0

[ Status == True && colors OK ]

[ Status == True && colors ! OK ]

Off New Stable Face Event

On New Stable Face Event

## App State Definitions

START: Ready

GOT\_IT: A Cube Face has been recognized and captured.

ROTATE: Request user to rotate Cube.

SEARCHING: Attempting to lock onto new Cube Face.

COMPLETE: All six faces have been captured, and we seem to have valid color.

BAD\_COLORS: All six faces have been captured, but we do not have properly nine tiles of each color.

VERIFIED: Two Phase solution has verified that the cube tile/colors/positions are a valid cube.

WAITING: Waiting for TwoPhase Prune Tree generation to complete.

INCORRECT: Two Phase solution has analyzed the cube and found it to be invalid.

SOLVED: Two Phase solution has analyzed the cube and found a solution.

DO\_MOVE: Inform user to perform a face rotation

WAITING\_FOR\_MOVE\_COMPLETE: Wait for face rotation to complete

DONE Cube should be completely physically solved.

# 2D Model Reconstruction

**X**

**Y**

**M**

**N**

Screen

Space

Coordinates

Face

Indices

Coordinates

**0**

**0**

**1**

**1**

**2**

**2**

In the first phase of model recognition, the top Rubik Face is treated as a two dimensional surface which should contain nine parallelograms (called Rhombi in the software). Class RubikFace contains several two dimensional arrays with indices M and N that are depicted in the diagram above. The earlier image recognition steps may or may not recognize each parallelogram (Rhombus), but as long as at least one tile is recognized in each row and column (i.e., M and N), then the algorithms discussed below can successfully reconstruct the face in 2D coordinates.

## Calculate Metrics

First, from the available recognized Rhombi, the average angles with respect to the X axis are calculated. Also, the center of each Rhombi is calculated.

## Initial Layout

Next, the Rhombus center X and Y screen coordinates are, using the above angle information, normalized to coordinates along the alpha and beta axis. The coordinated transformed Rhombus centers are then sorted along each axis and optimum partitions are found that subdivide the set into three groups with the conditions that 1) there is at least one Rhombi in each group, and 2) the axis coordinate RMS of each group is calculated and minimized. In this manner, depending upon the group that the Rhombus was assigned to, each Rhombus can then be assigned an N and M indices. Of course an error occurs if two Rhombus and assigned the same N and M indices, and the attempt to recognize the face is rejected. All of this step in performed in a separate class named TileLayoutAlgorithm.

## Lease Means Square

Next, given the tentative Rhombus grid assignment from above, and the angle calculations in step one, the available Rhombi centers are tested for an overall fit in a three by three grid. If the residual position error is less that a certain threshold (parameter faceLmsThresholdParam), the Rhombi grid assignment is accepted.

## Move Tile (if necessary)

If the residual position error calculated above is not less than the threshold, an attempt to change the grid assignment of the Rhombus with the greatest error is made. A direction can be obtained from the LMS calculations. The Rhombus grid assignment is changed (assuming the new position is not already assigned) and the overall LMS is recalculated. As long as the LMS is decreasing with each Rhombus move, the process is completed. However, typically, the initial layout assignment is good, and at most only one Rhombus move is necessary.

# 2D to 3D Reconstruction

Since the shape of the Rubik Cube is well defined, from simply one face (always the up face) tile location in 2D space, the position and orientation of the cube in 3D space can be calculated.

## Cube Face Definition

Internally, the application will define names for the faces as it initially observes the cube as show in the diagram below.

**Front**

**Right**

**Up**

The full set of face names is: Up, Down, Front, Back, Right and Left. This naming convention is leveraged from the TwoPhase algorithm and can be found in file Facelet.java. This naming convention is relevant in several locations in the code base:

* Expected face designation during initial six-turn observation phase: see StateModel.adoptFace().
* Representing the cube state in a string format: see StateModel.getStringRepresentationOfCube().
* Interpreting the TwoPhase algorithm results and applying them to OpenGL coordinate space: see UserInstructionsGLRenderer.renderCubeEdgeRotationArrow().
* Representation of faces in OpenGL coordinates space: see Xxxxxxx.xxxxx().

## Rubik Cube definition in Real World Coordinates

By convention, the top face is the face that is observed by the application. OpenCV and OpenGL have different coordinated space definitions. The conversion between the two systems occurs primarily in function CubeReconstructor.poseEstimation(). In all other code areas, coordinates are in OpenGL convention.

**X**

**Y**

**Z**

**X**

**Y**

**Z**

**OpenGL**

**OpenCV**