Final Project Report

Realtime 3D Rubik's Cube State Tracker & Solver

1 Project Introduction

In our final project, an IMU-driven 3D Rubik's Cube viewer and solver demo was built on the RP2040, where face colors were entered by buttons and solution steps were found and shown on a VGA display.

RP2040 dual cores were used to separate IMU processing from VGA rendering, keeping the 3D cube display smooth and stable. IMU data from the MPU6050 and GY271 were sampled at 100Hz and used to update a rotation matrix for real-time cube visualization. A button-based 3×3 color grid for all six faces was implemented with a bottom preview to make color entry clear and reduce input errors. The inputs were converted into a consistent internal cube-state model shared by both the renderer and the solver. An RLFB solution was then found using iterative deepening search and was shown and applied one move at a time, so the solving steps could be followed easily. The whole system was shown as Figure 1.

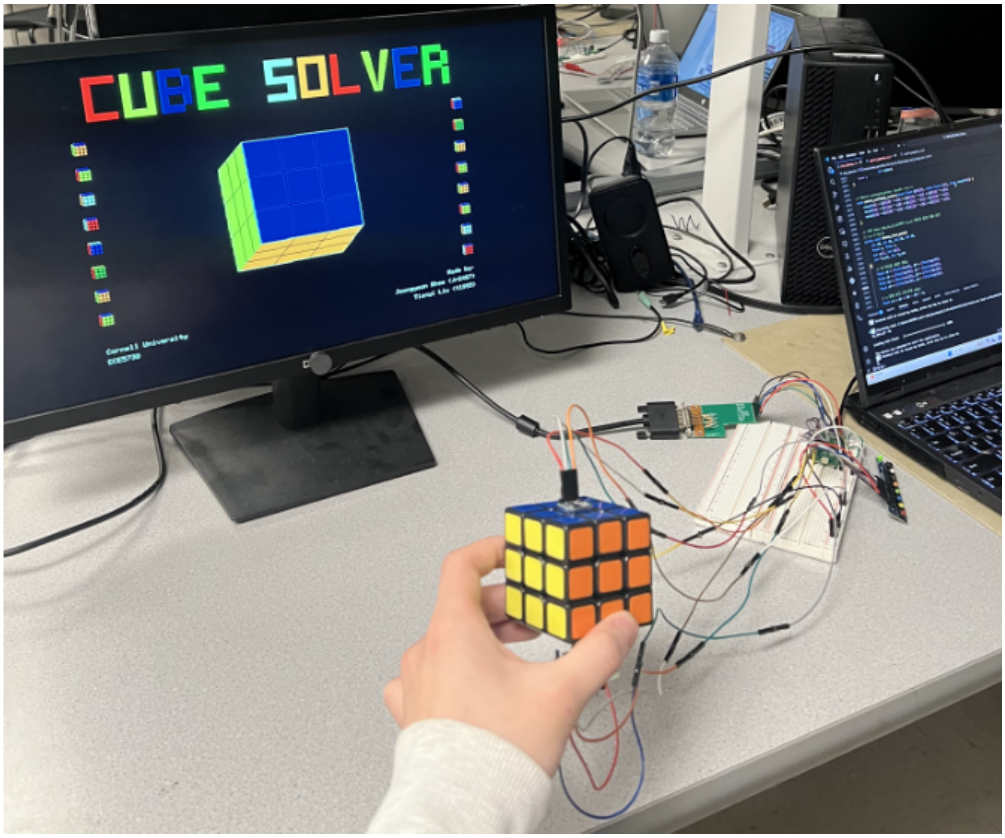


Figure 1: The whole system in our project.

2 High Level Design

2.1 Project Idea

Our inspiration came from Lab 2 and Lab 3. In those labs, the VGA was used to display a digital Galton board, measured angles and control inputs. However, the display content was limited to 2D graphics. This led us to explore whether 3D graphics could be shown on the VGA. When we thought about a 3D cube, it naturally pointed us to a Rubik's Cube. Therefore, we further decided to build a Rubik's Cube solver as our final project.

2.2 Background Math: TRIAD Attitude Determination

In our project, we estimate the cube's 3D orientation using the Tri-Axial Attitude Determination (TRIAD) method. The goal of attitude determination is to compute the rotation that maps a vector expressed in a reference (inertial) frame $\mathcal{N}$ into the same vector expressed in the body frame $\mathcal{B}$. TRIAD solves this problem by comparing two non-collinear reference directions measured in both frames.

Given two reference vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, TRIAD constructs an orthonormal triad by normalizing the first vector, forming an orthogonal second vector via a cross product, and obtaining the third by another cross product:

$$\hat{\mathbf{t}}_1 = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}, \qquad \hat{\mathbf{t}}_2 = \frac{\hat{\mathbf{t}}_1 \times \mathbf{v}_2}{\|\hat{\mathbf{t}}_1 \times \mathbf{v}_2\|}, \qquad \hat{\mathbf{t}}_3 = \hat{\mathbf{t}}_1 \times \hat{\mathbf{t}}_2. \tag{1}$$

Stacking these basis vectors into a matrix yields a direction cosine matrix (DCM):

$$\mathbf{T} = \left[\hat{\mathbf{t}}_1 \ \hat{\mathbf{t}}_2 \ \hat{\mathbf{t}}_3\right]. \tag{2}$$

We form such a triad in the reference frame, $\mathbf{T}_N$, and in the body frame, $\mathbf{T}_B$. Since each $\mathbf{T}$ is orthonormal, $\mathbf{T}^{-1} = \mathbf{T}^T$, and the rotation from $\mathcal{N}$ to $\mathcal{B}$ is computed as

$$\mathbf{Q} = \mathbf{T}_B \mathbf{T}_N^T. \tag{3}$$

In practice, TRIAD requires at least two reference vectors, and it becomes numerically unstable if the two vectors are nearly collinear (i.e., $\mathbf{v}_1 \times \mathbf{v}_2 \approx \mathbf{0}$).

In our implementation, we use the accelerometer to provide a gravity-related direction and the magnetometer to provide a magnetic-field-related direction. We initialize $\mathbf{T}_N$ once at startup using the first valid sensor measurements (defining our project's "reference" frame), and then recompute $\mathbf{T}_B$ at each update using the latest sensor readings. This produces a real-time rotation matrix $\mathbf{Q}$, which is then used to render the cube in 3D on the VGA display. Special thanks to Prof. Adams for introducing this method.

2.3 Logical Structure

For the logical structure, two cores were used in our project. Core0 was responsible for running the IMU thread, while Core1 was responsible for running the VGA thread. Core0

signals Core1 via a semaphore to refresh the display using the latest rotation matrix and UI state. A state machine was used to manage transitions between different cases. In Case 0, an IMU-driven 3D Rubik's Cube visualization was displayed. In Cases 1–6, the 3×3 grid colors for all six faces were collected one face at a time and previewed at the bottom of the screen. In Case 7, the supported move set, the real-time cube state and the computed solutions were presented.

2.4 Hardware/Software Tradeoffs

For our hardware/software tradeoffs, the solver's recovery capability was inherently bounded by both sensing constraints and compute limits. To estimate the cube's 3D pose accurately in real time, we used two sensors—an IMU (MPU6050) and a magnetometer (GY271). To rigidly mount both sensors on the physical cube while preserving stable axis alignment, we attached one sensor to the top face and the other to the bottom face. This mechanical arrangement introduced a key constraint: we could not perform U (up) or D (down) face rotations during operation, since those moves would disturb the fixed sensor alignment.

Our original plan was to solve the cube using *Kociemba's two-phase algorithm*; however, the restriction on U/D moves meant that a standard full-move solution pipeline was not applicable. As a result, we adopted a brute-force search approach using Iterative Deepening Depth-First Search (IDDFS) over the remaining four faces. Because the RP2040 has limited computational resources, we bounded the search to solutions within a finite number of moves to avoid excessively long runtime. We parameterized the maximum search depth in the code: increasing this limit can expand the set of solvable configurations (and, in principle, solve all cases), but it creates a direct tradeoff in computation time, which grows rapidly with depth.

2.5 Relevant Patents/Copyrights/Trademarks

For existing patents, copyrights and trademarks, this project did not incorporate any patented technology, third-party intellectual property or registered trademarks.

3 Program/Hardware Design

3.1 Software Design

In terms of program details in our project, the overall software architecture is summarized in Figure 2. The program is divided across two RP2040 cores: Core0 acquires IMU/magnetometer data, processes user inputs, and updates the rotation matrix, while Core1 is dedicated to VGA rendering and UI drawing. Core0 signals Core1 via a semaphore so the display can be refreshed using the latest orientation estimate and the current UI state. This separation keeps the 3D visualization responsive while maintaining a stable sensor update rate. The state machine then organizes the user workflow from live 3D viewing, through face-by-face color entry, to step-by-step solution playback.

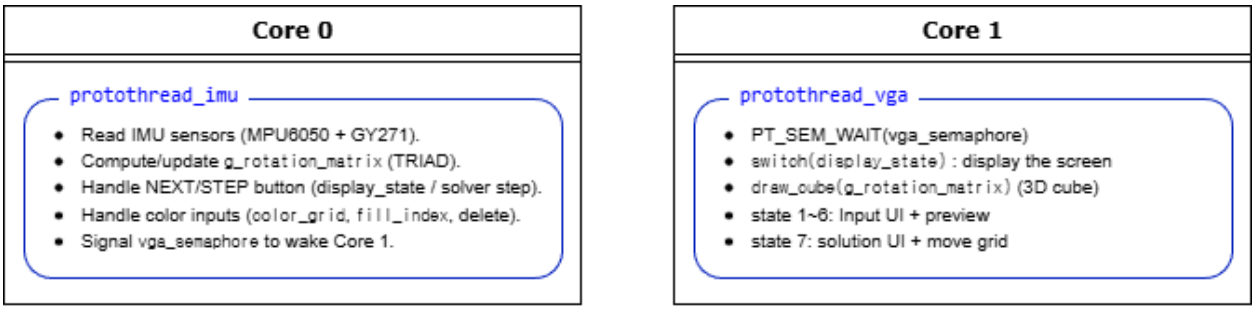


Figure 2: Software Diagram in our project.

3.1.1 Orientation Estimation (TRIAD)

In our project, we used the TRIAD method to estimate the cube orientation in real time and drive the live 3D Rubik's Cube visualization on VGA. The MPU6050 (accelerometer) and GY271 (magnetometer) axes were first aligned, and the sensors were rigidly mounted on the cube to preserve a consistent body frame. At each update, the 3-axis acceleration vector from the MPU6050 was used as $\mathbf{v}_1$, and the 3-axis magnetic-field vector from the GY271 was used as $\mathbf{v}_2$.

From these two non-collinear vectors, we constructed an orthonormal basis (triad) by $\mathbf{e}_1 = \hat{\mathbf{v}}_1$, $\mathbf{e}_2 = \widehat{\mathbf{e}_1 \times \mathbf{v}_2}$, and $\mathbf{e}_3 = \mathbf{e}_1 \times \mathbf{e}_2$, so that $\mathbf{T} = [\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3]$ formed a 3×3 attitude basis matrix. In our implementation, the triad computed at startup was stored as the reference frame $\mathbf{T}_N$, and each new sensor sample produced the current body-frame triad $\mathbf{T}_B$.

The TRIAD rotation from the reference frame $\mathcal{N}$ to the body frame $\mathcal{B}$ could be written in its expanded form as

$$ {}^{N}Q^{B} = {}^{N}\left[\hat{\mathbf{v}}_1 \ \ (\hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2) \ \ \left(\hat{\mathbf{v}}_1 \times (\hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2)\right)\right] \left({}^{B}\left[\hat{\mathbf{v}}_1 \ \ (\hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2) \ \ \left(\hat{\mathbf{v}}_1 \times (\hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2)\right)\right]\right)^{-1}. \quad (4) $$

Since $\mathbf{T}_N$ and $\mathbf{T}_B$ were orthonormal, $\mathbf{T}^{-1} = \mathbf{T}^T$, and the rotation could be computed as

$$ \mathbf{Q} = \mathbf{T}_B \mathbf{T}_N^T. \quad (5) $$

This was the form used in our real-time implementation. Figure 3 shows the `build_triad` function that constructs the triad matrix $\mathbf{T}$ from $\mathbf{v}_1$ and $\mathbf{v}_2$.

```
// Build rotation matrix Q from two basis vectors: TN = [v1, v1×v2, v1×(v1×v2)]
void build_triad(const float v1[3], const float v2[3], float Q[3][3]) {
    float e1[3] = {[0]=v1[0], [1]=v1[1], [2]=v1[2]};
    normalize_vector(v: e1);

    float e2[3];
    cross_product(a: e1, b: v2, result: e2);
    normalize_vector(v: e2);

    float e3[3];
    cross_product(a: e1, b: e2, result: e3);
    normalize_vector(v: e3);

    // Build Q: columns are body frame basis vectors in inertial frame
    Q[0][0] = e1[0]; Q[1][0] = e1[1]; Q[2][0] = e1[2];
    Q[0][1] = e2[0]; Q[1][1] = e2[1]; Q[2][1] = e2[2];
    Q[0][2] = e3[0]; Q[1][2] = e3[1]; Q[2][2] = e3[2];
}
```

Figure 3: `build_triad` function used to construct the TRIAD basis matrix $\mathbf{T} = [\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3]$.

3.1.2 Real-Time 3D Cube Rendering

Displaying the cube's real-time 3D orientation on VGA was one of the most important parts of our project. Based on the TRIAD method, we constructed a rotation matrix $\mathbf{Q}$. We then rotated the cube's eight predefined 3D vertices using $\mathbf{Q}^T$ and mapped the rotated coordinates onto the VGA display via a simple orthographic projection with a fixed scale factor. Figure 4 shows the corresponding code segment that computes $\mathbf{Q}^T$, applies the rotation to each vertex, and converts the results into screen coordinates.

```
// Q transpose : body -> world
float Q_T[3][3];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        Q_T[i][j] = Q[j][i];
    }
}

int   sx[8], sy[8];        // screen coords
float vx[8], vy[8], vz[8]; // "world" coords (for depth/face test)

// 1) Vertex rotation + screen coordinate transform
for (int i = 0; i < 8; i++) {
    float v_original[3] = {
        [0]=cube_vertices[i].x,
        [1]=cube_vertices[i].y,
        [2]=cube_vertices[i].z
    };
    float v_rotated[3];
    matrix_multiply_vector(Q: Q_T, v: v_original, result: v_rotated);
```

Figure 4: Vertex rotation and orthographic projection using $\mathbf{Q}^T$ in `draw_cube`.

To render solid faces, we estimated each face's depth using the average z value of its vertices, treated faces with positive depth as visible, and drew them from far to near. Figure 5 illustrates our visibility test using the face center depth along the camera's $+z$ direction. Each visible face was divided into a 3×3 grid, where every tile was filled using two triangles, and grid lines were overlaid to complete the 3D Rubik's Cube visualization.

```
//    to determine visibility toward the camera (+z direction)
int   face_visible[6] = {0};
float face_depth[6];  // use the center z-value as depth

for (int f = 0; f < 6; f++) {
    int i0 = cube_faces[f][0];
    int i1 = cube_faces[f][1];
    int i2 = cube_faces[f][2];
    int i3 = cube_faces[f][3];

    float cz = (vz[i0] + vz[i1] + vz[i2] + vz[i3]) * 0.25f;
    face_depth[f] = cz;

    // Assume camera direction is (0,0,+1);
    // if the face center is on the +z side, treat it as visible
    if (cz > 0.0f) {
        face_visible[f] = 1;
    }
}
```

Figure 5: Face visibility and depth estimation (camera direction $+z$).

It was also important to ensure consistent face mapping between the user's color inputs and the internal Rubik's Cube model. We adopted a fixed solver convention for the six faces and stored the cube state in a $6 \times 3 \times 3$ sticker array. During data entry, each 3×3 color grid entered on the VGA interface was converted into this canonical representation by matching each color to a predefined face-color table and applying a face-specific in-plane rotation (0°, 90°, 180°, or 270°) before writing into the sticker array.

During rendering, we applied a separate face-dependent display transform so that the 3×3 layout shown on screen always corresponded to the same logical indices in memory. Concretely, each face was assigned a fixed in-plane rotation `disp_rot` and optional horizontal/vertical mirroring flags (`disp_flip_h`, `disp_flip_v`), as summarized in Figure 6. This mapping decouples the solver's canonical sticker indexing from the camera-facing view used for visualization.

```
// ---------- Display mapping ----------
int disp_rot[6]    = { [0]=0, [1]=3, [2]=3, [3]=2, [4]=0, [5]=3 }; // 0/90/180/270
int disp_flip_h[6] = { [0]=1, [1]=0, [2]=0, [3]=1, [4]=0, [5]=0 }; // mirror LR
int disp_flip_v[6] = { [0]=1, [1]=0, [2]=1, [3]=0, [4]=1, [5]=1 }; // mirror UD
```

Figure 6: Face-dependent display mapping parameters used for rendering.

3.1.3 Solving Algorithm and Execution

Implementing our solving algorithm played a vital role in our project. After the user finished entering all six faces, we converted the six 3×3 color grids into the internal cube state. We then searched for a solution using iterative deepening depth-first search (IDDFS) with a limited move set.

At each depth, the DFS routine enumerates candidate moves and performs a standard backtracking step: it records the current move, temporarily applies it to the cube state, recursively searches the next depth, and undoes the move if the recursive call fails. Figure 7 shows the core backtracking loop, including the `apply_move`/`undo_move` pair that restores the cube state before trying the next option.

Once a valid sequence was found, we restored the cube to the original state, saved the resulting move list, and switched the interface into a solver mode. In this solver mode, each press of the NEXT button applied exactly one move from the saved sequence, allowing the user to watch the cube being solved gradually on the VGA display.

```
for (int i = 0; i < NUM_CAND; i++) {
    MoveType m = candidates[i];
    int mf = move_face_index(m);

    // 1) Skip if (current face == last face)
    if (mf == last_face) {
        continue;
    }
    // 2) Record the move
    g_moves[depth] = m;
    // 3) Apply the move
    apply_move(m);
    if (dfs_search(depth: depth + 1, depth_left: depth_left - 1, last_face: mf)) {
        return 1;
    }
    // 4) Undo the move
    undo_move(m);
}
```

Figure 7: Core DFS backtracking loop used in IDDFS.

3.1.4 User Interface State Machine

Figure 8 summarizes the user-interface state machine that organizes the entire workflow from sensing-driven visualization to solution playback. The interface is divided into three stages: a start stage (State 0), an input stage (States 1–6), and a solving stage (State 7). All transitions are triggered by the NEXT button, while color-button presses are only active during the input stage.

In State 0, the program continuously renders the IMU-tracked 3D cube and static UI elements. Pressing NEXT advances to the face-entry sequence. In States 1–6, the user enters one face at a time (UP, FRONT, BACK, LEFT, RIGHT, DOWN) by filling a 3×3 grid with color buttons; once a face reaches nine inputs, the system is ready to proceed to the next

face via NEXT. After the DOWN face is completed, the state machine transitions to State 7, where the solver output is displayed. In State 7, each press of NEXT applies exactly one move from the precomputed move list until the solution is finished; pressing NEXT again (after completion) returns the interface to State 0 for a new run.

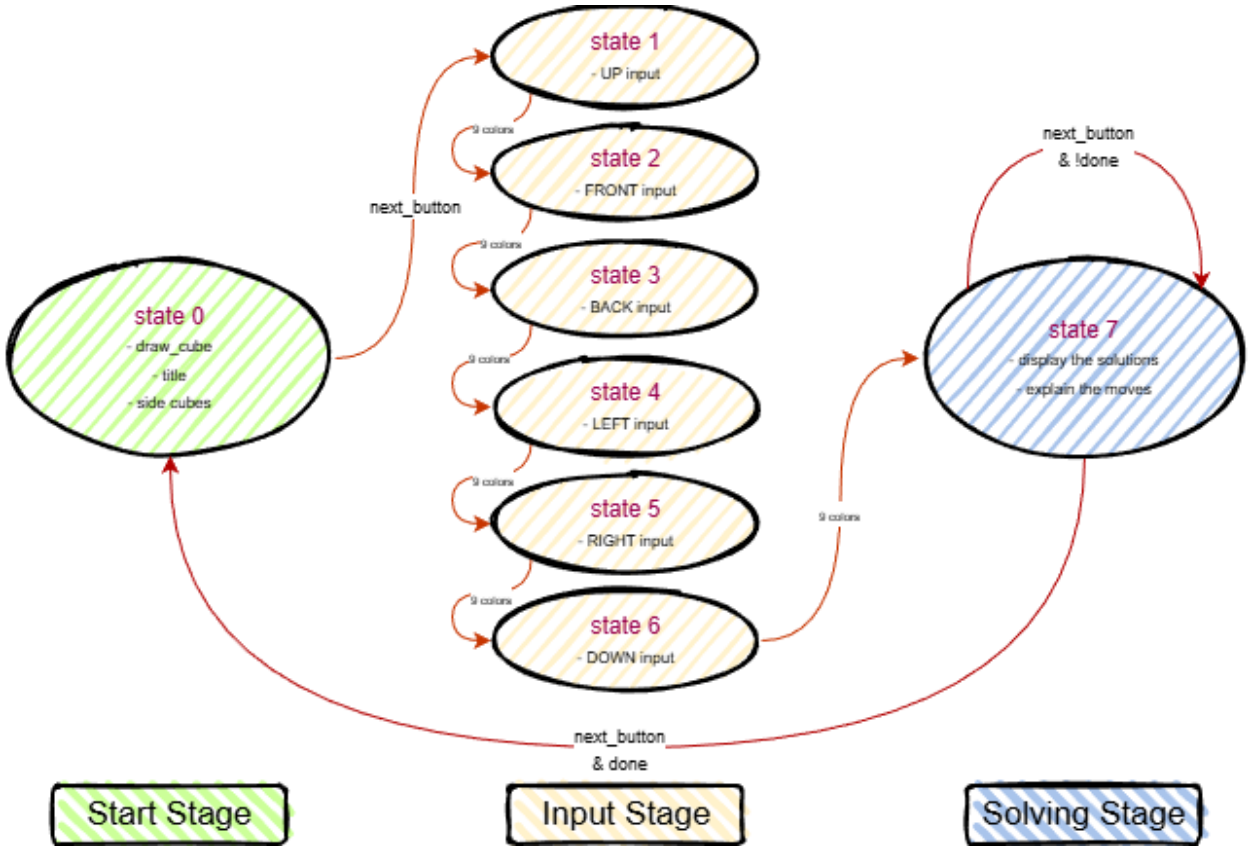


Figure 8: UI state machine for real-time.

3.2 Hardware Design

Figure 9 shows the hardware configuration used in our project, consisting of an RP2040 microcontroller, an MPU6050 IMU, a GY271 magnetometer, a VGA display, and eight push buttons (six color inputs, NEXT, and DELETE). To achieve accurate real-time 3D orientation tracking, we carefully aligned the XYZ axes of the MPU6050 and GY271 and rigidly mounted the sensors to maintain a consistent body frame throughout operation.

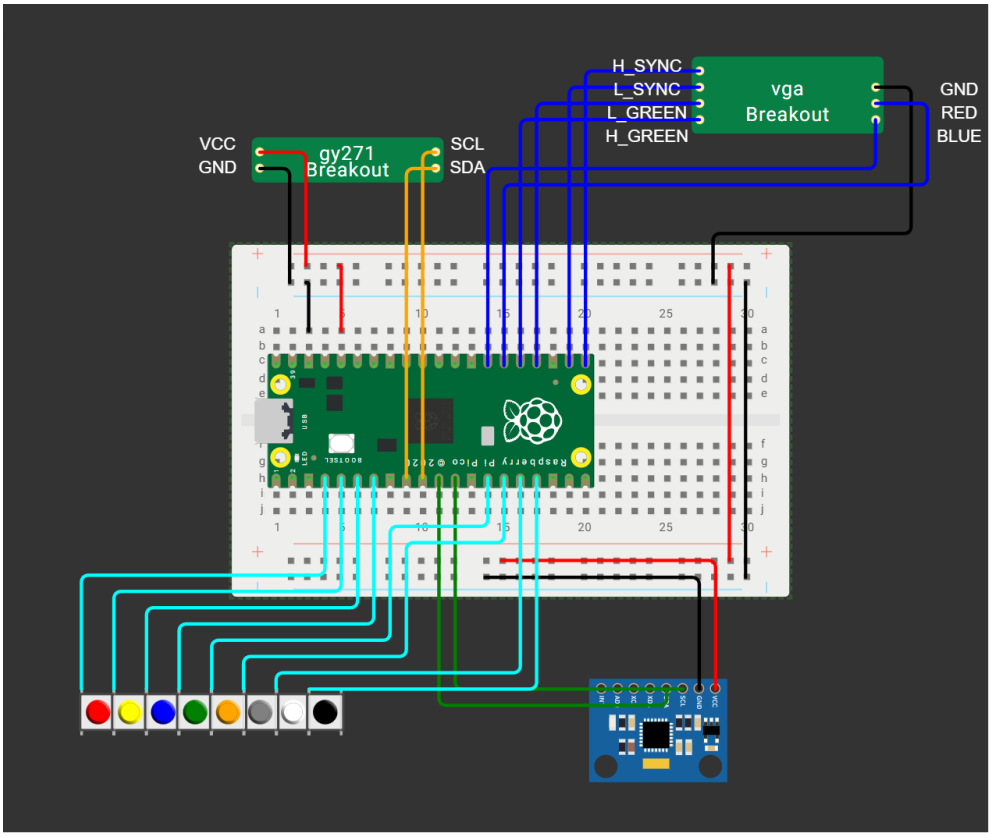


Figure 9: Hardware configuration of the RP2040-based Rubik's Cube solver system.

3.2.1 MPU6050 IMU (Accelerometer + Gyroscope)

MPU6050 was used in our project as shown in Figure 10 . It was placed on the cube's down face. Based on our TRIAD method, we only used the data obtained from the accelerometer to construct the gravity reference vector **v1**, which helped accurate real-time orientation visualization of the 3D cube. The gyroscope measured angular rate and would require integration and bias compensation to obtain attitude. Since it tended to introduce drift, we did not rely on it in our implementation. We also applied a low-pass filter to the accelerometer data to obtain a more stable estimate of the gravity direction. This filtering suppressed high frequency noise, improving the smoothness of the real-time 3D cube visualization. For MPU6050, we set the left direction as the x-axis, the direction pointing toward the user as the y-axis, and the upward direction as the z-axis in our project.

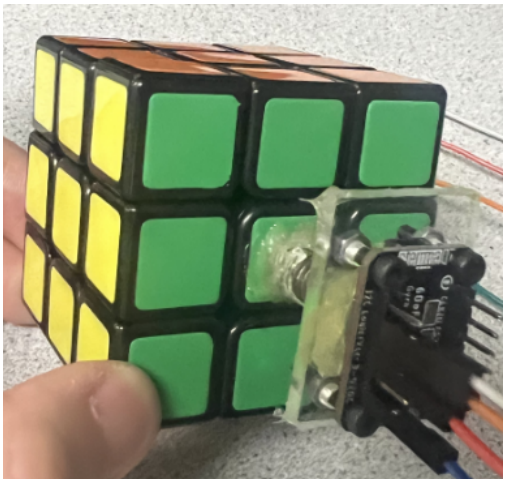


Figure 10: MPU6050 in our project.

3.2.2 GY271 (Magnetometer)

GY271 was also used in our project as shown in Figure 11. It was placed on the cube's up face. It provided a reference direction that was independent of gravity. Its measurements were used to construct the second TRIAD reference vector **v2**, completing the absolute heading information for full 3D orientation estimation. In order to avoid situations where the magnetic field indoors was too weak and easily disturbed by interference, we added a small magnet to provide a strong and repeatable magnetic field. For GY271, we set the left direction as the x-axis, the direction pointing toward the user as the y-axis, and the upward direction as the z-axis in our project. The XYZ axes of the MPU6050 and GY271 were aligned and these two sensors were placed on the bottom and top of the Rubik's Cube.

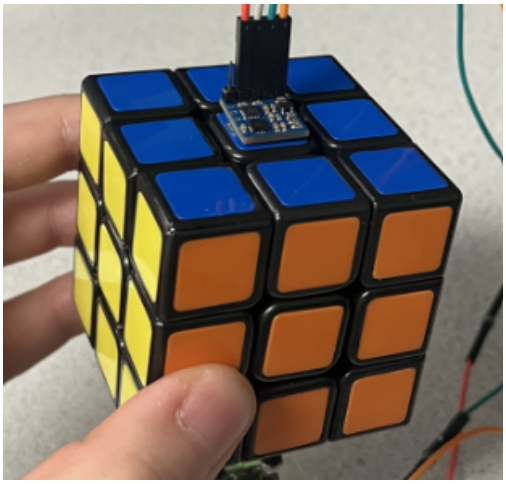


Figure 11: GY271 in our project.

3.2.3 8-Button Color Input Module

In our project, six buttons were required to enter the color information for all six faces of the Rubik's Cube. Two more buttons were needed to switch to the next state and provide recoverability when the user made input mistakes. Therefore, our system required eight buttons to achieve the full functionality. We chose to use an 8-button color input module shown in Figure 12, rather than eight individual push buttons, so that users could identify the colors more clearly. The brown button was used to switch to the next state and the black delete button was used for error recovery.

Figure 12: 8-Button Color Input Module in our project.

3.2.4 Magnetic Reference Fixture

In this project, we used a small permanent magnet in our project to provide a strong and repeatable magnetic field reference for the GY271 magnetometer as shown in Figure 13. In an indoor environment, the Earth's magnetic field was relatively weak and could be easily distorted, which might lead to unstable or inconsistent heading measurements. By introducing a magnet with a fixed placement and orientation, we created a controllable local magnetic field that improved measurement robustness and made it easier to validate the sensor's axis alignment and sign convention during debugging and calibration. As a result, the system could obtain a more reliable magnetic reference for displaying the cube's real-time orientation.

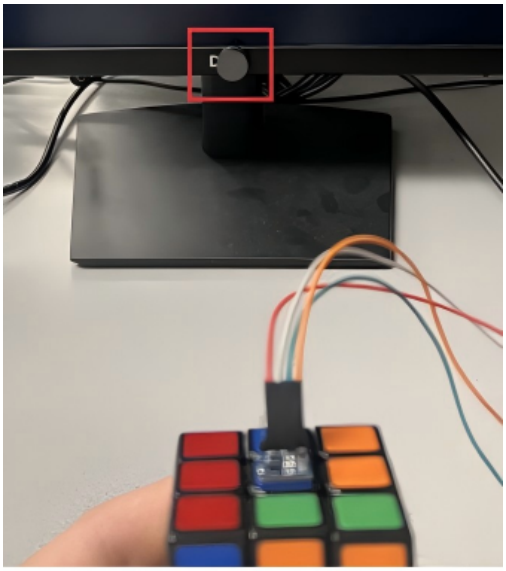


Figure 13: Magnet in our project.

3.3 Unsuccessful Attempts

3.3.1 Sixth-Face Color Inference

In this project, we initially explored inferring the colors of the sixth face from the color information of the other five faces to reduce the required user inputs. While this idea is feasible in principle because a valid cube has strict color-count and permutation constraints, we were not able to build a robust inference procedure within our implementation timeline. In practice, small input mistakes, ambiguous corner/edge consistency, and the need for additional validity checks made the approach unreliable, so we ultimately required explicit entry for all six faces.

3.3.2 Euler-Angle Orientation Estimation

We also attempted to estimate the cube's orientation using an Euler-angle approach. Roll and pitch were computed from the gravity vector measured by the accelerometer, and yaw was obtained from the horizontal projection of the magnetometer reading. However, the results were unstable. Yaw estimation was highly sensitive to accurate axis alignment between sensors and required proper tilt compensation. When the cube tilted, the magnetometer no longer represented only the north direction in the horizontal plane because a vertical component was introduced, easily causing yaw to deviate significantly or jump abruptly. We also observed a severe wrap-around artifact at the $-180°/+180°$ boundary: when the heading crossed this overlap region, the computed yaw could flip by nearly 360°, making the 3D cube appear to "spin" suddenly even under smooth motion. As a result, the Euler-angle method led to drift, flicker, and sudden heading changes in the visualization. Therefore, we switched to the TRIAD method to construct a rotation matrix directly, which provided a more stable real-time orientation estimate.

3.3.3 Kociemba Solver

Moreover, we could not adopt a Kociemba(-Lite) solver under our project constraints. Beyond the implementation complexity and the memory/compute overhead of two-phase methods (e.g., precomputed pruning tables and cube-coordinate encodings), our move set was fundamentally restricted by hardware. Because the MPU6050 and GY271 had to remain rigidly mounted with consistent axis alignment, our physical setup prevented performing U and D face rotations during operation. Since Kociemba's standard pipeline assumes access to the full move set (including U/D) to guarantee completeness and optimality, it was ultimately not applicable in our system. In contrast, an IDDFS-based solver over the remaining four faces required minimal memory and was straightforward to implement and verify, so we adopted IDDFS for our project.

3.4 AI Use

Regarding the use of AI, we used it to assist with debugging and to generate reference implementations of certain functions. We reviewed, understood and modified these functions to meet our specific requirements and to implement the desired features in our project.

4 Results of the Design

With the help of the VGA screen, we visualized the real-time 3D state of the cube, the color inputs for all six faces and the final solving process. The first state of our design was shown in Figure 14. Besides the central 3D rotating cube, we added some static cubes that matched our Rubik's Cube color layout and relevant text information such as the title and authors in the first state to make our system more informative.

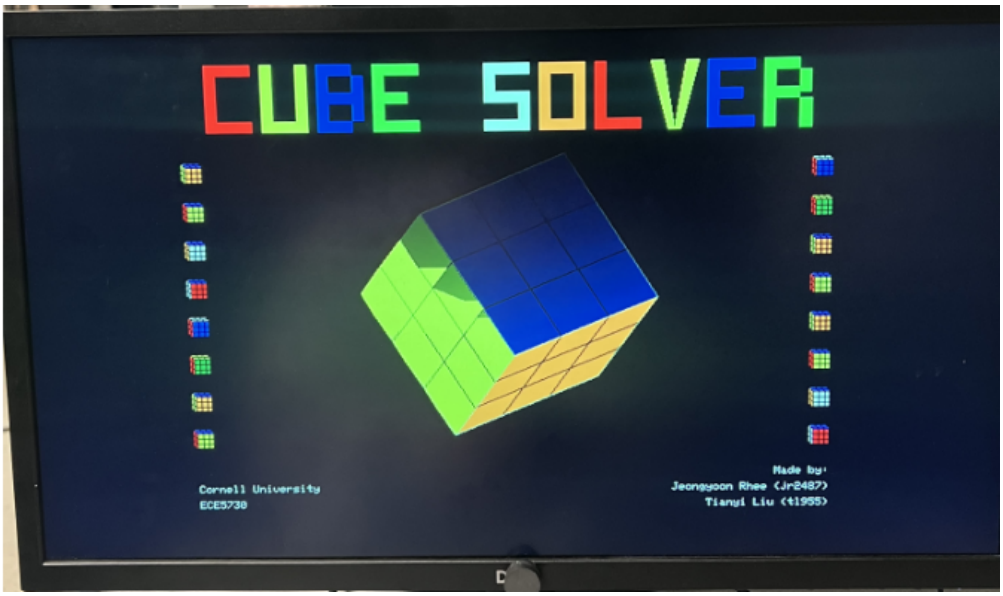
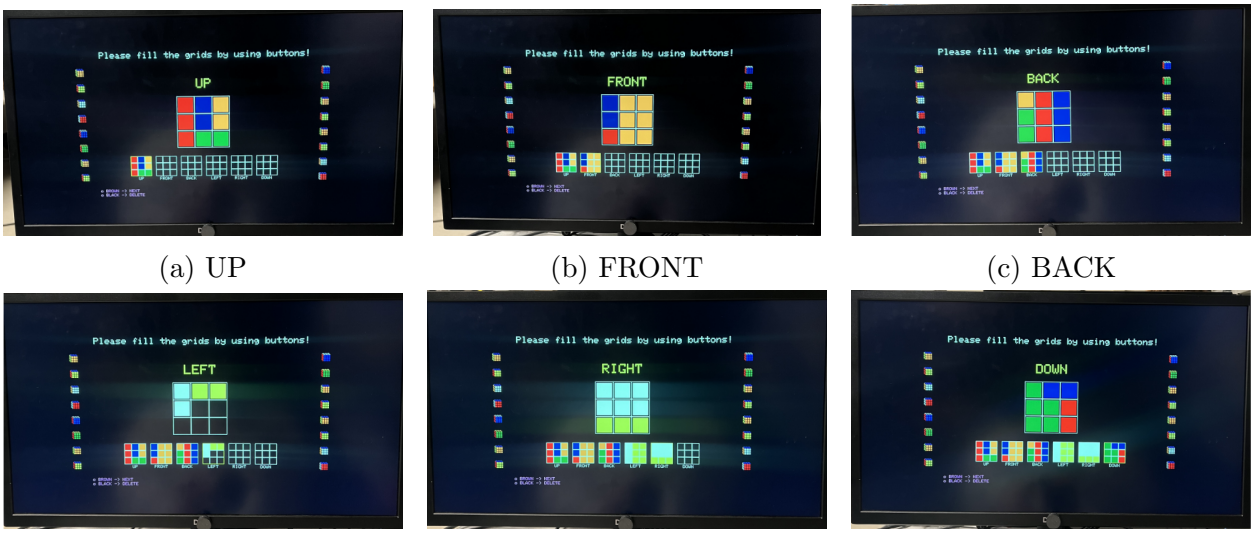


Figure 14: First state of the project.

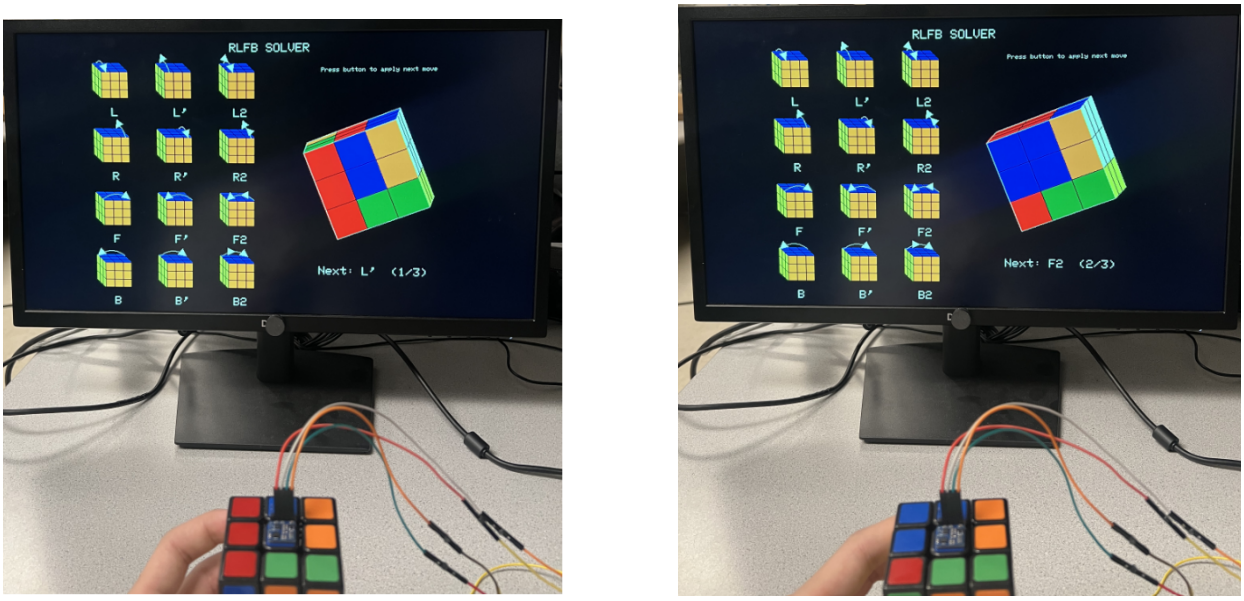
As for the color input state of our design, it was shown in Figure 15. We added some static cubes for decoration and some texts for instruction. We also displayed the color states of all six faces at the bottom of the screen to provide real-time feedback and reduce the possibility of input errors. In terms of the solving process, all results were shown in Figure 16. We showed all 12 moves on the left side of the screen to guide the user and avoid mistakes during operation.



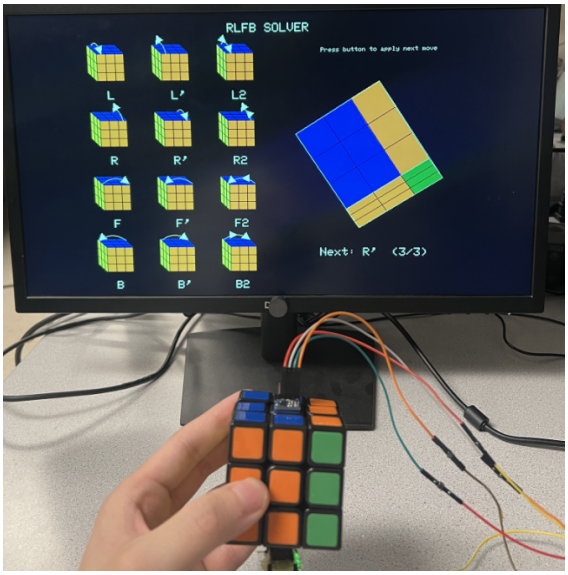
(a) UP (b) FRONT (c) BACK

(d) LEFT (e) RIGHT (f) DOWN

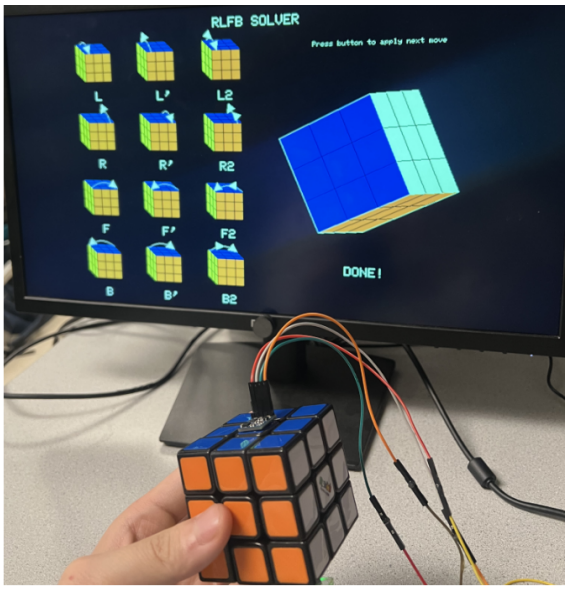
Figure 15: Color input state of the project.



(a) Step 1 (b) Step 2



(c) Step 3



(d) Step 4

Figure 16: Solving process of the project (continued).

In our project, the speed of execution was fast and the system was highly interactive. With the help of two cores, we also achieved strong concurrency. However, because of the asynchrony, the display could sometimes flicker. Moreover, our project had an accuracy of 100% and achieved high safety. In addition, our design enabled both us and other users to use the system conveniently and smoothly.

5 Conclusions

All the results met our expectations in our project. We might infer the sixth face's colors from the color information of the other five faces and eliminate the flicker next time. Our design conformed to the applicable standards by following the established interface and timing conventions of each subsystem in the RP2040 platform. The VGA output was produced using standard video signal timing so it could be displayed reliably on a regular monitor. The MPU6050 and GY271 sensors were accessed through the I2C protocol on two separate buses, using the correct initialization and read/write transaction sequences specified by the devices. All buttons and inputs were configured with pull-up resistors to prevent floating signals and improve operational safety and usability.

For intellectual property considerations, we did not reuse code or someone else's design. We did not use anyone else's IP. We did not use code in the public domain. We were not reverse-engineering a design. We did not meet patent/trademark issues. We did not have to sign non-disclosure to get a sample part. There were not patent opportunites for our project.

6 Appendix A

The group approves this report for inclusion on the course website. The group approves the video for inclusion on the course youtube channel.

7 Additional Appendices

In this project, Jeongyoon Rhee was responsible for real time 3D cube visualization, consistent face mapping and the solving algorithm. Tianyi Liu was responsible for application of GY271, the state machine and the color input. We both improved the user interface of our system and wrote the report. Since the code was too long to be included in our report, we added it to our website.