Herbert
Autonomous Robotic Rubik's Cube Solver

Jonathan Whitaker, Dylan Lytle, Matt Frandsen, Li Lao
Dept. of Electrical and Computer Engineering, University of Utah

***Abstract*—Project Herbert is an autonomous robotic Rubik's Cube solver that is composed of a complex network of mechanical and electrical devices. In this project our team interfaced with these complex mechanical and electrical devices so as to design and create an autonomous robot that is capable of solving a Rubik's cube within two minutes of time. This project is an integration of various technologies including: mechanical actuators, electrical stepper motors, single-board computers, field-programmable gate arrays, and video cameras.**

I. Introduction

The goal of this project was to create an autonomous robotic Rubik's Cube solver through the integration of several components. The main components integrated into our project include video cameras, electrical stepper motors, mechanical actuators, a single-board computer (SBC), and field-programmable gate arrays (FPGAs). A video camera connects to the computer through standard universal serial bus (USB) 2.0 protocol [7]. The video camera is responsible for capturing the initial configuration of the Rubik's Cube, and the single board computer is responsible for processing the video frame information and generating a matrix model for the initial state of the cube. After generating a matrix model for the initial cube, the computer applies Kociemba's algorithm [4] (an optimal algorithm used to solve a Rubik's Cube) to generate a solution sequence that can be processed sequentially. The solution sequence that Kociemba's algorithm returns is in the standard notation used in Rubik's Cube discussion and theory (see Appendix A). As each solution sequence is processed, the computer communicates through an RS232 serial connection to an FPGA control board which drives the mechanical actuations and stepper motor rotations needed to physically manipulate the cube. The FPGA acts as a system control board and is responsible for controlling the actions of two motor control boards and a relay board used to trigger the mechanical actuators. An overview of this process can be found below in Fig. 1.

II. Software Implementation

The single board computer in our system operates a software stack composed of Python, Cython, and OpenCV. We utilized a 3rd party Python library called *pyflycapture2* [3] to capture images from the camera that was provided to us by Point Grey Research. Once the images of the cube are captured, we apply color recognition and categorization using OpenCV-Python [6]. The characterization phase in OpenCV generates a matrix model of the Rubik's cube. We pass this matrix model on to an application called Kcube to generate a solution sequence needed to solve the cube permutation. We go into each these components in more detail in the next few sections.

A. Image Processing

The image processing aims to capture the initial state of the cube using a camera. The criteria for evaluating the success of the image processing implementation is robustness and speed of the algorithm.

1) Point Grey Camera and FlyCapture Software Development Kit: We interfaced with the cameras using Point Grey's API, which was provided to us through the FlyCapture software development kit (SDK) [2]. The FlyCapture SDK was implemented in C. Since we chose to use Python as our base language, we had to use Cython, an optimized static

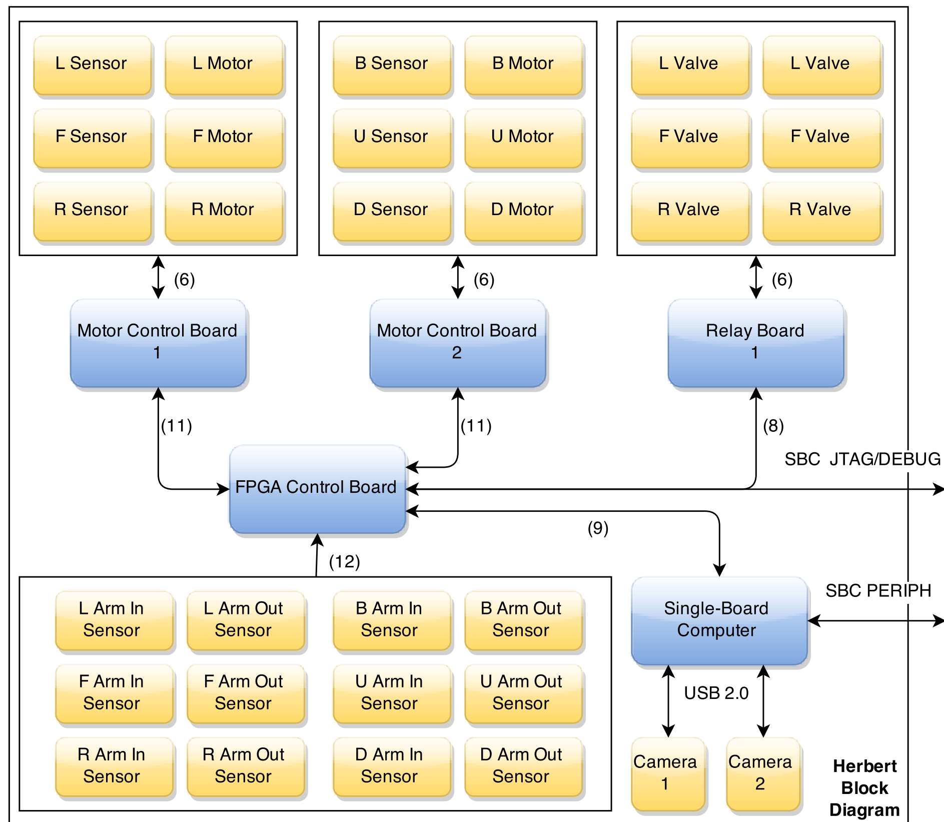


Fig. 1. Herbert Block Diagram

compiler for C extensions in Python. We use Cython to bind the C functions within the FlyCapture API to Python functions. The FlyCapture SDK provided us with a large amount of configurability, and it gave us a flexible interface for which the cameras settings could be adjusted to enhance capturing mode and synchronizing image buffer retrieval.

We initially had planned to use two cameras, each connected to the SBC through a standard USB 2.0 connection. Each one of these cameras was going to be responsible for capturing exactly three of the six faces of the cube with a view like that shown in Fig. 2. However, as construction of the main frame of our project came to a close, we soon realized that various parts of the mechanical system obscured the view of a large majority of the facelets on the three faces. On top of this, we also experienced challenges dealing with color recognition in ambient lighting. Because of these challenges we decided that our original camera configuration was not ideal.

We decided to compromise and use a single camera configuration. A single camera is used to capture the entire state of a Rubik's Cube. The single camera was best positioned like that shown in Fig. 4. A static mechanical rotation sequence is used to capture the cube in various configurations until all of the facelets on the cube are captured. The cube is then returned to its original state, and the collection of images are saved off for facelet characterization.

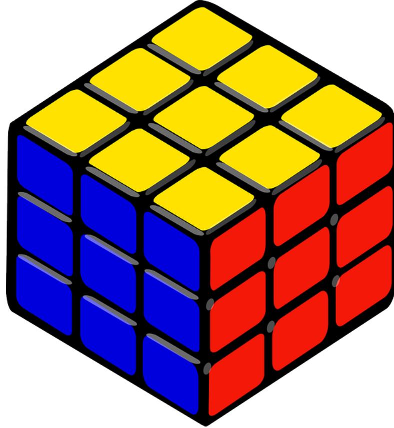


Fig. 2. Initial ideal cube image capture

2) Cube Orientation: The pixel region of each of the facelets must be identified for color recognition. We initially planned on using a combination of grayscale conversion and contour filtering to dynamically detect a bounding contour around each of the facelets. However, the dynamic approach to identifying a facelet pixel region proved to be difficult due to the lack of vision of the cube. The arm assemblies were too obtrusive to capture a clean image (see Fig. 4).

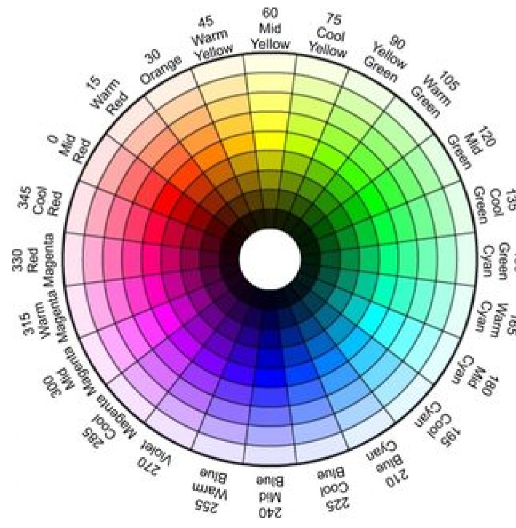


Fig. 3. HSV color wheel

Because the camera and the Rubik's Cube remains fixed inside the mechanical frame, we decided to implement a static solution. Through a GUI interface, a static polygon for each of the facelets was defined. Each of the static polygons creates a bounding rectangle around a facelet pixel region. The static solution that we adopted proved to be consistent, but it necessitated high maintenance. Each of the static polygons must be redefined each time the camera position changes or if a mechanical change affects the orientation of the cube. If we were to continue on with this project, this is one aspect that we could improve. It would be nice to implement a robust dynamic approach.

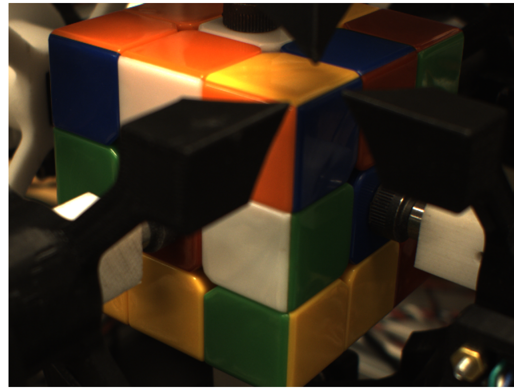


Fig. 4. Obscured cube image capture

3) Color Space: Lighting condition severally affects the robustness of the color recognition algorithm. Color values on a Rubik's Cube facelet drastically change if the facelet is reflective or if a shadow is cast upon it. This proved to be challenging for us. However, we were able to maintain a consistent lighting around the mechnical frame to mitigate a majority of these problems. We were very confident in the color recognition algorithm under normal lighting conditions. During demo day, we solved over 24 Rubik's Cubes without failure.

Choosing a color space that was robust to lighting conditions was critical to enabling a robust color recognition algorithm. Several color spaces such as RGB, CMYK, and HSV were examined. The HSV color space provided the best consistency under

various lighting conditions. The three components of the HSV color space are hue, saturation, and value. The hue component describes the similarity of the color to an unique hue: red, green, blue, and yellow as shown in Fig. 3. The saturation component measures the intensity or "colorfulness" of the specific hue. Finally, the value measures the brightness of the color value. HSV is ideal for developing a robust color recognition because shadows and high gloss merely change the value component of the color without affecting the hue and saturation component of the color. Additionally, the hue component can be used to distinctly distinguish each of the primary colors such as red, green, and blue that are found on a Rubik's Cube facelet.

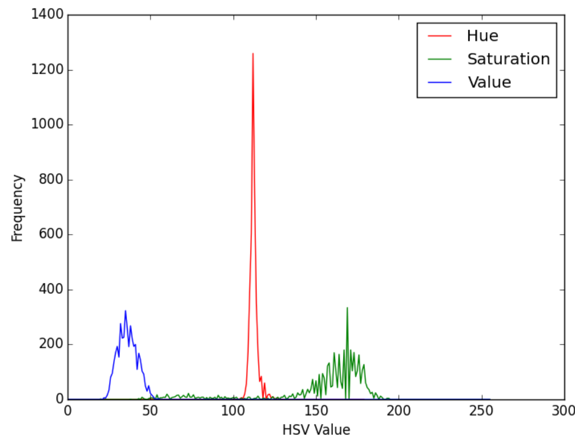


Fig. 5. Example HSV color histogram for blue

TABLE I
HSV COLOR RANGES

COLOR	HUE	SATURATION
WHITE	25 - 58	0 - 139
RED	1 - 12	0 - 255
BLUE	200 - 280	0 - 255
GREEN	80 - 120	0 - 255
ORANGE	15 - 20	0 - 255
YELLOW	25 - 58	140 - 255

4) Categorization Algorithm: After identifying a bounding contour around each of the facelets, a categorization algorithm was used to determine the color contained within the bounding rectangle. Each image was first converted to the HSV color space. The hue and saturation components of each pixel value is then used to categorize the color. Several color histograms such as the one shown in Fig. 5 were constructed for each facelet and were used to determine the ranges of each HSV component in order to categorize a single facelet color. For example, the histogram shown in Fig. 5 was constructed for a blue facelet. The hue values range between 100 to 130 and the saturation values ranges from 150 to 180. Our analysis led to the resulting HSV ranges shown in Table. I.

Based on these color ranges, the categorization algorithm can be implemented based on the average of all the pixel values in the facelet region or the frequency of each pixel value. A categorization algorithm based on the average of HSV values proved to be quick, but it was also error-prone due to outliers. A categorization algorithm based on frequency of each HSV component proved to be more robust, but less optimized. Our final implementation computes a histogram of the facelet pixel region using OpenCV. Based on the histogram, the frequency of each of the facelet colors is determined. The highest frequency of a facelet color determines the color that facelet is binned into.

Our initial trivial algorithm that we implemented took approximately a minute to categorize each facelet, because of the unoptimized looping in Python and hardware limitations. After several iterative optimizations using the OpenCV library, the overall color categorization time was reduced to approximately three seconds. In the future, the categorization can still be optimized for both robustness and speed. For example, the image resolution or the overall pixel region can be reduced to improve performance. Additionally, a filtering algorithm can be used to reduce the effects on abnormal lighting.

B. Kcube and the Solution Sequence

Kcube is a C++ application developed by Greg Schmidt that utilizes Kociemba's two-phase algorithm which uses two stages of an iterative depth first search algorithm [5]. We utilized the Kcube ap-

```
              |************|
              |*U1**U2**U3*|
              |************|
              |*U4**U5**U6*|
              |************|
              |*U7**U8**U9*|
              |************|
|************|************|************|************|
|*L1**L2**L3*|*F1**F2**F3*|*R1**R2**F3*|*B1**B2**B3*|
|************|************|************|************|
|*L4**L5**L6*|*F4**F5**F6*|*R4**R5**R6*|*B4**B5**B6*|
|************|************|************|************|
|*L7**L8**L9*|*F7**F8**F9*|*R7**R8**R9*|*B7**B8**B9*|
|************|************|************|************|
              |************|
              |*D1**D2**D3*|
              |************|
              |*D4**D5**D6*|
              |************|
              |*D7**D8**D9*|
              |************|
```

Fig. 6. Rubik's Cube matrix representation

TABLE II
CUBELET COLOR TO ASCII CHARACTER MAPPING

COLOR	CHARACTER
WHITE	'W'
RED	'R'
BLUE	'B'
GREEN	'G'
ORANGE	'O'
YELLOW	'Y'

plication to generate the solution sequence needed to solve the Rubik's Cube that was captured during the image processing phase. The matrix model generated from the image processing phase allows us to provide Kcube's command-line interface with the cube representation needed to generate a solution sequence. Kcube's command-line interface takes six parameters, one for each face of the cube. The values for these parameters are the color characters at each of the cubelet locations for that face (as seen in Fig. 6). For example, to solve the scrambled cube shown in Fig. 7 you invoke Kcube with the following command:

```
c:>kcube L:GGWWOWBRB
         F:GWGBGYWBO
         U:YOOOWYROY
         D:ORGWYYYRB
         R:OGBBRYWRR
         B:YBROBGWGR
```

Kcube processes the input parameters and generates a sequence of twenty-three or less moves (see Appendix B) that, when applied to the cube, will solve the cube. Each move is mapped to a unique character or set of characters (see Table III), and these characters are transmitted over an RS232 serial connection to the FPGA control board, at which point the control board takes responsibility for controlling the electro-mechanical stepper motors and mechanical actuators needed to physically manipulate the cube.

III. HARDWARE IMPLEMENTATION

A. Mechanical Actuators

Herbert employs a six arm design to physically manipulate the cube. One arm for each face of the

TABLE III
CUBE MOVES TO CHARACTER SET MAPPING

MOVE	CHAR	MOVE	CHAR	MOVE	CHAR
F	F	R	R	D	D
F'	Fb	R'	Rb	D'	Db
F2	F2	R2	R2	D2	D2
L	L	U	U	B	B
L'	Lb	U'	Ub	B'	Bb
L2	L2	U2	U2	B2	B2

cube. In order to achieve a six arm design, each arm actuates in and out so as to avoid conflict with the other arms. This actuation process is a time critical component of the design. We initially planned on implementing the arm actuation with motors. However, preliminary testing showed that using motors to convert rotary motion into linear motion is too slow, and using a linear motor actuator is too costly. We decided that pneumatic actuation is the solution to this problem. Each of the arms is attached to a double action pneumatic air cylinder as show in Fig. 8.

The FPGA control board is responsible for controlling a relay control board (see Fig. 1) which controls coaxial pairs of air cylinders. The actuation distance for each control arm is fixed. The coaxial pairs of arms are kept in an extended or retracted position, depending on the current move being executed. Potential optimizations and more stable cube manipulations are obtained by simultaneously extending and retracting coaxial pairs of arms. The linear actuation motion allows a coaxial pair of arms to extend, thus encasing two sides of the cube in the sockets of the arms. After a pair of arms extend, a stepper motor spins the arm corresponding to the appropriate move from the solution sequence (see section III-B). Each air cylinder is provided approximately 40-60 psi supplied from an air compressor. To protect against any arm collisions, only one pair of arms is in the extended position at any given time.

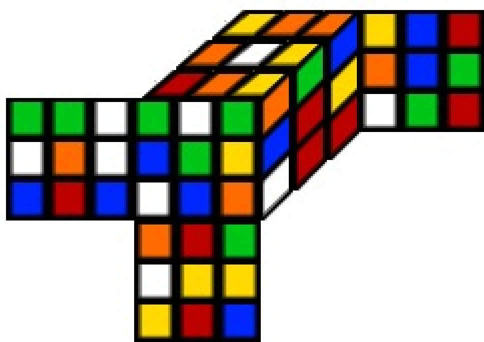


Fig. 7. A scrambled cube

Initially our design incorporated mechanical switches that would be triggered by physical contact of the arm assembly. These switches were mounted on the assembly in such a way that contact would occur in either the extended or retracted arm positions. The goal of the switch based design was to remove an element of manual timing approximation. By using the switches we would know the position of each arm at all times. Taking advantage of this, there would be a much smaller hard coded delay, or potentially no delay. However, due to time constraints as well as accuracy issues because of inconsistent and difficult mounting, we chose not to implement the switches in the final design.

If we were to continue the project there are many improvements and optimizations that could be made. The first optimization would be to implement the switch design we initially planned on. Improving the robustness of the switches would help reduce the time the mechanical motion takes. This would reduce the overall time to completion, because the mechanical motion is the most time intensive component of the solution. Another mechanical optimization that would help reduce the time to completion is to modify the arm. The image processing was a very difficult component of this project, because the arm assembly was obtrusive to the view of the cube for the cameras. Our thought is to remove two of the four prongs of each arm. This modification would give the cameras a more clear view of the cube, as well as potentially giving us the ability to actuate in and out more quickly, because potential arm collisions would be reduced.

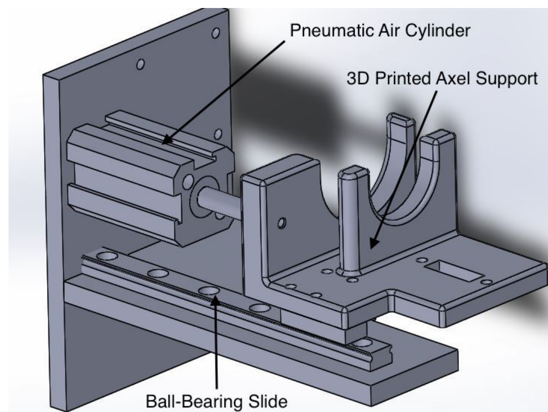


Fig. 8. Pneumatic Air Cylinder and 3D Printed Axel Support

B. Electro-mechanical Stepper Motors

Each actuating arm has a stepper motor which is responsible for rotating a single face. A stepper motor rotates a face either 90 degrees or 180 degrees clockwise or counter-clockwise based on the solution move that is being processed (as specified in Appendix A). A 3D printed axle (as seen Fig. 9) is fastened to the stepper motors. This axle twists a 3D printed arm piece in order to spin a face of the Rubik's Cube.

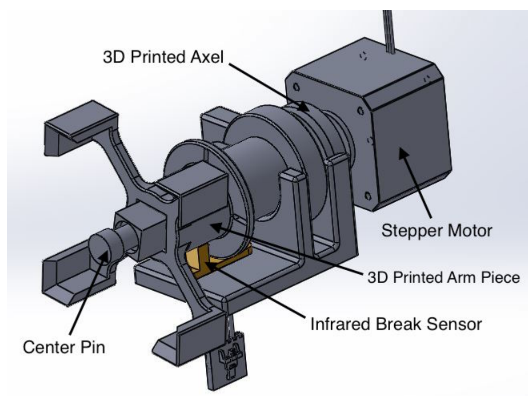


Fig. 9. Stepper Motor Arm Assembly

Each stepper motor is driven by a motor control board which is controlled by the FPGA control board (see Fig. 1). The motor control boards contain a motor driver chip for each stepper motor. The FPGA control board is responsible for controlling the angular and temporal timing of each stepper motor rotation.

C. Infrared Break Sensors

Rotations must end at a 90 degree angle so as to not interrupt the rotation of another arm. These exact angles are hard to accomplish by counting steps, because it is difficult to detect whether steps are skipped. To compensate for this we used infrared break sensors like that shown in Fig. 10. Each arm assembly has an infrared sensor (see Fig. 9). The infrared sensor is broken at all angles that are not a multiple of 90 degrees. These break sensors are monitored and controlled through the motor control boards.

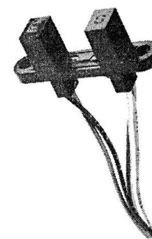


Fig. 10. Infrared Break Sensor

D. Arm Progression

The 3D printed arm piece at the end of each arm assembly (see Fig. 9) underwent a great progression. Fig. 11 highlights the progression of the arm pieces from left to right. The first iteration was a simple arm that had a square socket and connected directly to the stepper motor without any intermediate axel piece. The inner part of the socket had chamfered edges to correct for any error in alignment. The second iteration included tabs for break sensor detection, and it was elongated to accommodate the length of the center pin. We also modified the socket into four prongs. This allowed better visibility of the cube for the image processing. In the last arm revision we changed from tabbed break sensor detection to slits. This

improvement allowed more precise angle detection. We also shortened the arm, added more chamfered edges for minor alignment error corrections, and included slight height changes to fit the slider piece and circular cutouts to avoid arm collisions.

Fig. 11. Arm Progresssion

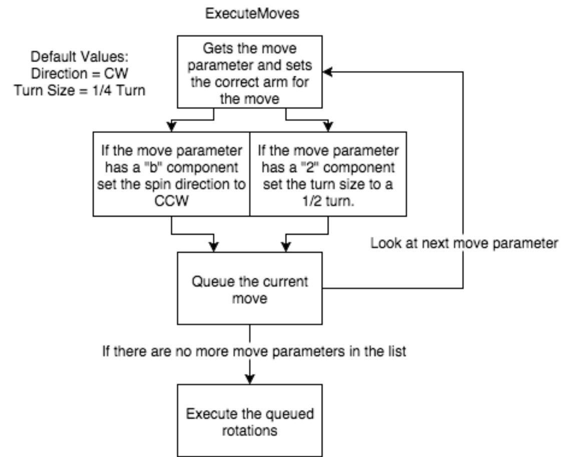


Fig. 12. "ExecuteMoves" State Diagram

IV. Firmware

The firmware stemmed from code received from BioFire. Much of the code provided was only used as a skeleton, and was modified to implement our design. Table IV outlines the commands we used for this project. For brevity only the main command "ExecuteMoves" will be described, for details on the other commands see table IV. "ExecuteMoves" is the main command that performs the solution sequence generated by Kcube. Each move is encoded with a specified face(U,F,R,D,B,L) and an optional component either a 'b' indicating a counter-clockwise rotation, or a '2' indicating the move to execute a half turn instead of a quarter turn. A detailed state diagram outlining this process is shown in Fig. 12

V. Required Resources

Table V is the bill of materials (BOM) needed to realize this project. This BOM defines the main materials needed to realize the system as outlined in Fig. 1. Many of the components that we used were donated to us by industry sponsors. The components that were donated to us include: the stepper motors, FPGA system control board, the motor control boards, and the Chameleon USB2.0 camera. The stepper motors, motor control boards, and the FPGA system control board are "in-house" proprietary boards developed by BioFire Defense Systems. The FPGA board embeds a Xilinx Spartan3 with a softcore Microblaze processor. The Chameleon USB2.0 cameras were donated to us by Point Grey Research. They a 1.3 megapixel camera with a Sony ICX445 CCD, 1/3", 3.75 micron sensor.

VI. Summary

This project is evidence of our team's ability to design a complex system containing software, electrical hardware, and mechanical hardware components. Project Herbert is a project that integrates various technologies and domains of engineering into one complete package. Working on this project has exposed us to a real-world application of system integration and, most importantly, teamwork. Project Herbert brought many challenges to our team, and we were able to overcome them with clever compromises. The image processing proved to be very difficult due to the lack of vision and poor lighting effects. We overcame this by creating a clever execution of moves that allowed us to capture all of the facelets using a single camera, as well as a unique color characterization that uses histogram calculations. We were unable to get robust and reliable switches for the mechanical actuations, and so we slowed down the actuations to ensure that

TABLE IV
FIRMWARE COMMANDS

Command	Parameters	Description
ActuateArm	1	Toggles the relay board to control the pneumatic actuations of coaxial pairs of arms.
DisableMotors	0	This command disables the motors.
TimingTest	0	Perform an actuation and spin motion for time testing.
Abort	0	Halts all motion and clears buffers holding the solution sequence.
MoveRelative	2	Manually spin the arm by moving the specified stepper motor the given number of steps.
MR	2	Equivalent to MoveRelative just a shortened alias.
GetRawSwitches	0	Gets all of the current values for all of the switches.
GetRawSensors	0	Gets all of the current values for all of the break sensors.
GetSwitch	1	Gets the current value for the specified switch.
GetSensor	1	Gets the current value for the specified break sensor.
ExecuteMoves	Variable	Performs the correct actuation and spin sequence for each of the moves provided.
IsIdle	0	Queries the motion state machine to determine if motion is still in progress.
InitArms	0	Spins each arm until its corresponding break sensor is not blocked.

TABLE V
MAIN COMPONENT BOM

Part Description	Quantity	Vendor	Vendor PN	Price/Unit (dollars)
Stepper Motor	6	BioFire Defense	NA	DONATED
Pneumatic 12mmx25mm Double Action Thin Air Cylinder	6	Amico	A12030500UX0057	7.86
24V 2 Position 5 Way Pneumatic Solenoid Valve	6	Uxcell	A11102700UX0130	10.31
FPGA System Control Board	1	BioFire Defense	NA	DONATED
Motor Control Board	2	BioFire Defense	NA	DONATED
8 Channel 5V Relay Board	1	SainSmart	20-018-102	11.99
Chameleon USB2.0 Camera	2	Point Grey Research	CMLN-13S2C-CS	DONATED

any two adjacent arms would not collide with one another. Likewise, team scheduling proved to be another challenge for us. Half of our team works part-time. This made it difficult to find times to work on the project together. However, despite these challenges, we were able to implement the Rubik's cube solver as we had planned. We did not have enough time to make further optimizations to go for a Guinness World Record, but we are very satisfied to have a completely automated solver despite all of the compromises that we had to make.

VII. ACKNOWLEDGEMENTS

Our team would like to thank BioFire Defense LLC, Point Grey Research Inc., and Futura Industries for all the support and resources they have provided us. Your contributions are greatly appreciated.

BioFire Defense donated the various control boards and mechanical components needed for this project. They also gave us access to various prototyping tools including high precision 3D printers and laser cutters. A special thanks goes out to BioFire engineers Logan Taylor (Mechanical), Pat Riley (Electrical/Systems), Matt Murdock (Electri-

cal), and David Nielsen (VP of Product Development). These individuals provided invaluable time and knowledge to our team.

We'd also like to thank Vladimir Tucakov of Point Grey Research. He provided our team with their Chameleon CMLN-13S2M-CS camera which we used for image acquisition.

We couldn't have put all these components together without a nice chassis to house them all. For this, we would like to thank Futura Industries. They helped us in the design and construction of the aluminum frame we used to house Herbert. Another special thanks goes out to Futura's Kenton Frandsen (Mechanical/Manufacturing Engineer) who assisted in the mechanical design of the mechanical arms and frame of our project.

REFERENCES

[1] Joseph Converse. *Basic Notation*. URL: http://astro.berkeley.edu/~converse/rubiks.php?id1=basics&id2=notation.

[2] *FlyCapture SDK*. Point Grey Research. URL: https://www.ptgrey.com/flycapture-sdk.

[3] Robert Jordens. *pyflycapture2*. URL: https://github.com/jordens/pyflycapture2.git.

[4] Herbert Kociemba. *The Two-Phase Algorithm*. URL: http://kociemba.org/cube.htm.

[5] Herbert Kociemba. *The Two-Phase Algorithm*. URL: http://kociemba.org/twophase.htm.

[6] *OpenCV-Python*. OpenCV Developers Team. URL: http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#opencv-python.

[7] *USB 2.0 Specification*. URL: http://www.usb.org/developers/docs/usb20_docs/.

APPENDIX

In order to solve a cube, it is standard to define the terminology and orientation layout used in Rubik's Cube theory and analysis. This section describes the basic notation that is used throughout this document.

A. Faces

A Rubik's Cube is composed of six faces: right (**R**), left (**L**), up (**U**), down (**D**), front (**F**), and back (**B**) (see Fig. 13). The exact color of each face is relative to the orientation in which you are holding the cube. For example, if you align the blue face towards you then the blue face is defined as the front face. Each face can be rotated in two different directions: *clockwise* or *counter-clockwise*. These rotations are defined as the direction of rotation when looking directly at that face.

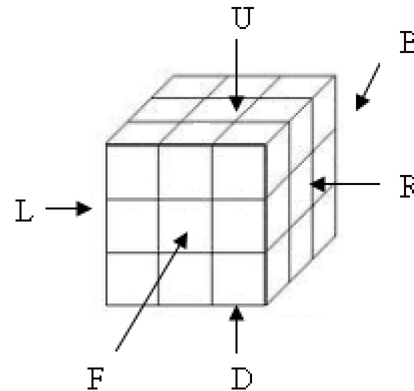


Fig. 13. Cube orientation

B. Fundamental Moves

The most fundamental moves are 90-degree clock-wise rotations for each of the faces outlined above. These moves are outlined below [1]:

- **R** - Indicates a 90-degree clockwise rotation of the right face such that the side on top rotates towards the back.
- **L** - Indicates a 90-degree clockwise rotation of the left face such that the side on top rotates towards the front.
- **U** - Indicates a 90-degree clockwise rotation of the upper face such that the side in front moves to the left.
- **D** - Indicates a 90-degree clockwise rotation of the downward face such that the side in front moves to the right.
- **F** - Indicates a 90-degree clockwise rotation of the front face such that the side on top moves to the right.

- **B** - Indicates a 90-degree clockwise rotation of the back face such that the side on top moves to the left.

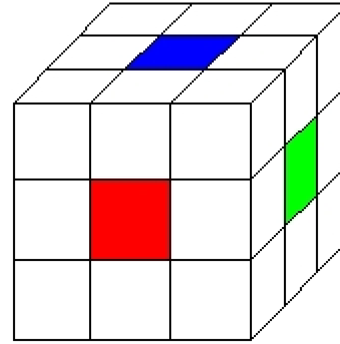
C. Modifiers

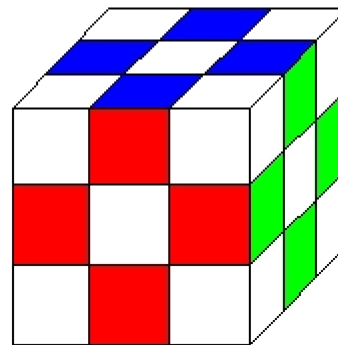
For each of the fundamental moves above, there are modifiers that can be appended to the move to change the rotation of the face. My example below uses **L** as the base move, but these modifiers can be applied to any of the fundamental moves.

- **L'** - Indicates a 90-degree counter-clockwise rotation of the left face such that the side on top rotates towards the back (opposite direction as that defined above).
- **L2** - Indicates a 180-degree rotation of the left face (two rotations).

D. Cubelets

A cubelet refers to a particular piece on the cube. Cubelets are categorized based on their position. There are three types of cubelets: center cubelets, edge cubelets, and corner cubelets (see Fig. 14). A center cubelet is unique. All other cubelets revolve around the center cubelets, they never move (go ahead, try and move the center piece). Edge cubelets connect two face pieces together at an edge. A corner cubelet connects three pieces together at the corner of the cube.





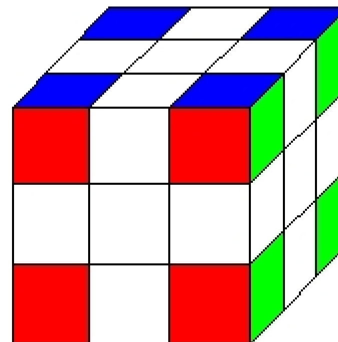


Fig. 14. Cubelet categories