Anna Manning, Tim Cushman, Ben Lehrburger and Jack Gnibus

[[1]](#footnote-1)

Balloon’s Tower Defense Robot

*Abstract*— Color detection and object sensing require fast computation and precise image processing. We present an autonomous robot that searches an environment for green balloons, targets the green balloons and pops them using an attached blade. The robot simultaneously avoids red balloons and any obstacles in the environment. We implemented this program on the Husarion ROSbot 2.0 and made use of the Orbbec Astra RGBD camera to sense green/red balloons and localize the robot so that it correctly targets green balloons. We also used the LiDAR scanner to avoid obstacles and walls. We evaluated our implementation by setting up an environment with various red and green balloons taped to the floor. The robot was able to successfully target and pop green balloons while avoiding red balloons and getting too close to the walls.

# INTRODUCTION

There is a large range of applications for mobile robot robotic color and object detection. For robots to move about their surrounding environment, whether it be self-driving cars or factory-inhabiting robotic limbs, they must be able to accurately sense objects and colors. Our task was to program the ROSbot 2.0 such that it can accurately detect objects based on color. In the present experiment, these objects are red and green balloons. We also added functionality allowing the robot to target and attack green colored objects while avoiding red colored objects. This has applications to any area of robotics concerning machine vision. Our specific motivation for this project was to emulate how self-driving cars perceive their environment and respond to cues such as traffic lights (which also use red and green signals). We discuss these implications further in the conclusion section.

To implement the balloon popping robot, we used a Finite State Machine (FSM) to control the motion of the robot and the OpenCV library to assist with processing image messages from the robot. In what follows, we discuss: (II) fieldwork related to the present experiment, (III) assumptions that we made and the environment that we created, (IV) the methodology that we followed to build our ROS node, (V) the results of our experiment, and (VI) what those results mean for future robotics initiatives.

# Related Works

Color sensing and object detection are widely studied areas of robotics. However, we were only able to find one paper discussing balloon tracking and popping [1]. In that report, they use a micro aerial vehicle (MAV) to pop balloons in open space. Like our intended goal, this paper also describes a method using a state machine and implements real time image processing. One issue we ran into is that tracking objects with color poses a challenge based on the high cost of image processing. The ROSbot 2.0’s camera is similar to the Kinect sensor (used in the aforementioned study), specifically that it has both depth and RGB sensors with the capacity for high real time color tracking [2]. One important factor in image processing is making sure the subscriber has a large buffer size (*buff\_size*) so that the queue of image messages drops the messages that the robot cannot process in time [3]. This was a crucial finding and made it so that we were able to process the images in real time and detect different colors. One issue we were finding was inconsistencies in lighting leading to bad results. One addition that would be useful for our application would be a light sensor, so that we could better determine the colors in different environments [4].

Figure 1, Model of Environment Setup

# Project and Environment Assumptions

We set up an environment for the robot in various locations, including Sudikoff and Sarner Underground. For our successful runs, we used tables as the walls for the environment. Our final environment in Sarner Underground was roughly 6x5 meters, and our walls were made with black tables. The black tables provided high walls with a consistent color that isolated the “arena” from the external environment. Additionally, the lighting in Sarner Underground was a bright florescent light, which made the color detection very accurate It was essential that the tables were a dark color, so that the camera did not sense them as red or green. This manner of controlling the environment was incredibly important, as keeping the robot in a sterile arena that lacked other colors ensured the robot would not detect other green and red objects. Interference from external factors, such as red or green chairs, could interfere with the robot’s image and LIDAR processing, making it more difficult or impossible to complete the desired task.

Chart, bubble chart

Description automatically generated

Within the arena, green and red balloons were

taped to the floor so that the robot would have enough force to pop the green balloons instead of simply pushing them forward. Assumptions we made include that the robot would pop the green balloons on the first try, which turned out to be correct, and that a red and green balloon would not be placed so close together that in the process of turning after popping a green balloon, the robot would pop the red balloon.

# Method

Our implementation can be divided into 3 subsections: hardware modification, image processing, and the FSM. The FSM controls the motion of the robot while the image processing and the LiDAR determine which state the FSM executes.

## A. Hardware Modification

#### To accomplish our goal of balloon popping, we added various blades to the front of the ROSbot so that it could successfully pop the balloon. We purchased a craft knife set from the dollar store and the blades were surprisingly sharp. Originally, we planned to have a state for the robot if it was unsuccessful in popping the green balloon on the first try, but our blades were so sharp that we never ran into this problem. Also, we believed initially that the robot would only have enough force to pop if the balloons were taped to the walls, but we found that the blades were share enough that the balloons could be taped to ground, which cause the robot to run into walls less. One modification from our original design was that we added multiple blades to the front of the robot. Instead of just one blade in the center, we added 4 across the front of the ROSbot, which can be seen in Figure 2.

## A toy car with balloons Description automatically generated with low confidence

Figure 2, Hardware Modifications

## B. Image Processing

To handle the image processing, we decided to leverage OpenCV for color detection. The callback function “image\_callback(msg)” sets up this process by converting the ROS image into an image accessible to OpenCV. This is achieved by representing the image data in a 1D array that can then be converted into an OpenCV image using “cv2.imdecode()”. Because OpenCV reads images in a BGR format, the image is then converted from RGB to BGR, and subsequently passed to “determineBalloonColor()” to handle the actual color detection.

“determineBalloonColor()” is heavily based on a color detection algorithm using OpenCV [5]. First, it is important to convert the image to an HSV color space to ensure more accurate results under different lighting conditions in comparison to a BGR format, as hue values change less drastically than BGR values. Thus, we can apply a tighter threshold on the color values for red and green balloons so the robot will only notice the values we intend it to notice, which we can control through consistently colored balloons. We then create a color mask for said red and green balloons by defining a range of HSV values for each respective color, which is then stored in a variable using “cv2.inRange()”. The function effectively creates a binary image, with each pixel not within the color range having a value of 0 and each pixel within the range having a nonzero value. Thus, makes it easier to perform contour finding, as the program now needs to find contours composed of nonzero values instead of attempting to find contours in the image itself.

With red and green colors isolated from the rest of the images, we run this check for contours using “cv2.findContours()”. Each contour identified is a part of the original image that had HSV values that fell within the color mask. Applying a check for contour area size allows us to remove any small extraneous flicks of color that somehow slipped through the mask. We set the minimum required area to 4000 (obtained via experimental testing), as we found that the size corresponded well to the size of balloons. For each valid contour, we take appropriate actions depending on the color of the balloon. For red balloons, the “self.red” flag is set, we rotate 90 degrees to avoid the balloon. For green balloons, the “self.green” flag is set to true, indicating that we have a green balloon in the robot’s field of view. After this, OpenCV draws a bounding box around the contour, which we found very helpful for adjusting the robot’s heading towards the center of the balloon. Information regarding the coordinates of the vertices of the bounding box are then passed to “self.isCentered”, so the robot can take action to successfully navigate towards the green balloon to pop it.

## C. Finite State Machine

The main motion of the robot was controlled by a Finite state machine, see in Figure 3. The ROSbot begins in the TURN state, so that it can do a scan of the room before it begins the random walk.

* *Random Walk State:*

Our implementation is a modified version of the random walk that we implemented in programming assignment 0. Like in PA0, the robot begins by moving forward until the self.\_close\_obstacle flag is set, or until it senses a green or red balloon. When the robot does not sense a green balloon, and the self.\_close\_obstacle is set, the robot will stop, calculate a random angle between 60° and 180°, and rotate that angle, so that the robot is facing away from the wall. When the robot senses a green balloon in the random walk state, it will enter the Green Balloon state. When the robot senses a red balloon, it will enter the turn state. Otherwise, it will continue to random walk.

* *Green Balloon State:*

A robot senses a green balloon in its field of vision once it enters the green balloon state. The robot first calculates if the balloon is in the center of its field of view. This is accomplished by calculating where the center of the object is in the image frame, then calculating the angle that the robot needs to turn. To calculate this angle, we take the difference between the center of the green object and the center of the screen, divide this difference by the width of the screen, and then multiply by π/3 (which is the field of view of the robot). After the robot rotates to this angle and verifies that the balloon is in the center of the screen, the robot moves forward. While the robot is moving forward it will continually check if the green object is in the center of the screen and adjust accordingly. This is colloquially known as a proportional gain controller.

When the robot pops the balloon, it will no longer sense green in the image frame, and will transition back into the random walk state. However if the robot is near a wall, it will enter into the turn state.

* *Turn state:*

The turn state is quite simple. It tells the robot to rotate 60 degrees and continue with the random walk. This is used when the robot sees a red balloon, or has just popped a green balloon and is near a wall.

Diagram

Description automatically generated

Figure 3, Finite State Machine

# Results

At first, our results were sub-standard. Our image processing subscriber was too data-potent and could not process RGB-D data fast enough to keep up with our publishers. So, the robot would see a green balloon, move towards it, and subsequently pop it, but then the green balloon didn’t leave its field of view for 30 seconds to 4 minutes beyond the pop. As a result, the robot would continue trying to pop a green balloon that was no longer there.

In response, we set the queue size to 1 and fixed the buffer size in our image callback to constrain the amount of data being processed in each iteration of data intake. This allowed our image processing subscriber and action publishers to operate on the same time scale. Once we implemented these changes, the robot was able to appropriately transition between states and move from one green balloon to the next, popping each in succession.

When testing the final model of our code, the robot successfully popped the initial 3 green balloons in the arena without popping any red balloons, and successfully popped another green balloon when we added it to the arena. The robot avoided running into any walls or making erroneous movements.

# Conclusion

Using the methodologies described above, we programmed an ROS node that (a) detects green and red balloons in the ROSbot’s environment and (b) directs the ROSbot towards green balloons to pop them while avoiding red balloons.

In the process, we learned valuable lessons about machine vision. The ROSbot does not have the computational capacity to process every feature of the image it’s receiving from the RGB-D camera. Much like the human visual system does not process every feature of the color input it receives – concentrating receptors in the center of the visual field and limiting perception in the periphery – we had to put a cap on the amount of data we processed in each image. By setting our queue size to one and fixing a buffer size in our image callback, we were able to process images in nearly real-time. This allowed us to align our publishers with the data we received from our RGB-D camera subscriber. We concluded that when machine vision is limited by computing power, data received in image processing needs to be narrow and focused.

When machine vision is employed in higher-stakes scenarios, specifically with self-driving cars, the question of what information to prioritize and what data to omit will be of utmost concern. If a Tesla can’t process every bit of information in its visual field, how will it decide where to focus? If we drop the Tesla in the middle of Times Square, will it be able to process each pedestrian? If not, will the pedestrians in its periphery be taken into account? Will it only process the pedestrians passing in front of the vehicle? These are all questions that our project raises and that machine vision engineers will have to consider.

Another nuance of machine vision that we had to work through was making color detection consistent across environments with different lighting. Our color detection program for identifying red and green objects was determined by the HSV (hue, saturation, value) ranges for each color. However, these parameters were hard coded and color detection was only consistent in well-lit, heavily constrained environments. When we tried to run our program in dim spaces or on colored floors, the ROSbot detected red and green colors erroneously. In the case of a self driving car, The vehicle’s color detection program will have to account for these confounding variables.

Lastly, and perhaps most importantly, we learned just how sensitive ROS systems can be. In doing something as simple as programming a robot to move towards green balloons and avoid red ones, we spent about two weeks trying to control for every nuance in the ROSbot’s environment. If the overhead lights didn’t hit the balloons properly, the wall colors were too red, the ROSbot was too close to the wall to turn, or the floor color was too tinted, we had to edit our program. As robots become more commonplace in spaces that humans inhabit, it will be very important for programmers to control for every situation the robot may find itself in.

Acknowledgment

We would like to thank Professor Quatrinni Li and Mingi Jeong for all their help and support throughout the term.

References

1. M. Beul et al., "Visually Guided Balloon Popping with an Autonomous MAV at MBZIRC 2020," 2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR), 2020, pp. 34-41, doi: 10.1109/SSRR50563.2020.9292612.
2. M. Clark, D. Feldpausch and G. S. Tewolde, "Microsoft kinect sensor for real-time color tracking robot," IEEE International Conference on Electro/Information Technology, 2014, pp. 416-421, doi: 10.1109/EIT.2014.6871800.
3. “rospy subscriber queue\_size acting strange when buff\_size is too small” <https://github.com/ros/ros_comm/issues/536>
4. J. Darma, Frederick, L. Setyawan and E. Kuantama, "Color recognition robot," 2011 IEEE 3rd International Conference on Communication Software and Networks, 2011, pp. 85-89, doi: 10.1109/ICCSN.2011.6014855.
5. *Multiple color detection in real-time using python-opencv*. GeeksforGeeks. (2020, May 10). Retrieved November 20, 2021, from https://www.geeksforgeeks.org/multiple-color-detection-in-real-time-using-python-opencv/.

1. [↑](#footnote-ref-1)