CS/RBE 549 Computer Vision
Fall 2012
Final Project Report

Show Us What You Know, Robot:
Autonomous Identification and Indication of Plastic Automata Existence
Group 7

Member	**Signature**	**Contribution (%)**
Russell Toris		
David Kent		
Adrian Boteanu		

Grading:	Approach	______/25
	Justification	______/20
	Testing & Examples	______/25
	Documentation	______/15
	Presentation	______/10
	Difficulty	______/5
	Extra	______/10
	Total	______/100

Executive Summary

The goal of this project was to detect a toy computer using a variety of computer vision techniques. Our team decided to implement a full object recognition pipeline, while minimizing the use of predefined functions. The objectives of the project were to detect the target object, to use an approach generalizable to any object, to perform this recognition in real-time, and to present the detection and recognition data in such a way that it could be useful for robotic applications.

In keeping with the robotics-focused objective of the project, the group primarily used a Kinect RGB camera for object detection. The Kinect was mounted on a PR2 robot, which allowed for easy control of the camera position as well as transmission of the object detection results to the robot itself. The project was implemented within the Robot Operating System (ROS), and the OpenCV code base was used for lower level computer vision functionality.

The overarching approach of the project was to use feature-based classification to recognize objects. This first required segmentation of the original Kinect RGB image to detect potential objects. This image segmentation was implemented using noise removal, edge detection, morphological operations, and size-and-shape filtered closed contour detection. After individual objects were detected and segmented, the algorithm performs a custom feature extraction method on each image segment to create a feature vector. Each feature vector is used as input to a Naive Bayes Classifier. The classifier was trained on image segments of a set of seven distinct object classes, including the toy computer.

With object recognition implemented, a few adjustments had to be made to the pipeline to improve real-time performance. False positive identification of the target object made any physical robotic interaction hazardous, so the results of the classifier were filtered using a custom smoothing algorithm. The smoothing algorithm removed false positive measurements and only confirmed the existence of the target object after a set time interval.

Once the object was reliably detected in real-time with the implementation of the smoothing algorithm, the group focused on using this information for robotics. The PR2 robot is commonly used in grasping research, and we took our inspiration from this. Using the Kinect's depth data, the center point of the detected and recognized object could be mapped onto three-dimensional space. Using coordinate transformations implemented for the PR2 robot in ROS, this 3D point was translated into a position relative to the base coordinate frame of the robot. Information such as this allows robots to interact with physical objects, and the group demonstrated this functionality by programming the robot to point at the target object once it was detected and recognized. This functionality could theoretically be extended to grasping and object manipulation in future work.

Overall, the implemented pipeline was a success. The robot could consistently detect the target object in real-time at arbitrary orientations, and the information could be used by the robot to physically locate the object in 3D space. There is, however, further work on the system could improve its functionality. For example, the segmentation algorithm does not do as well in heavily cluttered environments, and parts of the feature vector are dependent on lighting conditions consistent with that of the training data. Improving these issues could generalize the system to work under more conditions, but as it stands now the system completed all of our original objectives.

Contents

1 Introduction

For our final Computer Vision project, our team was given the task of detecting a small toy computer as depicted in Figure 1. After discussing potential ways of tackling such a problem and noting the numerous computer vision techniques used to detect objects, we decided to base our approach around classification.

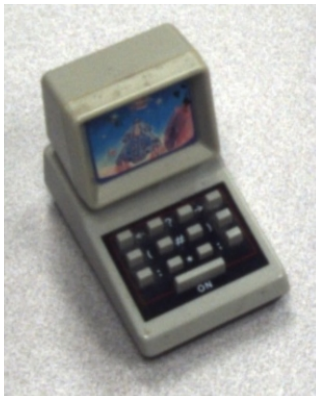


Figure 1: The toy computer.

Although there are many out-of-the-box solutions that could be used to aid us in a vision based classification system, we wanted to instead use lower level processing techniques to build a more robust and original pipeline. The pipeline, which is discussed in detail in the following section, consists of a custom image segmentation algorithm and feature extraction. By doing so, we would be able to customize exact parameters to help us more accurately detect target objects.

1.1 Objectives

Our project consisted of an overarching goal of accurately detecting the toy computer using the techniques mentioned above. To challenge ourselves to build a robust system, we decided to take this goal one step further. We began by noting that computer vision is an area of research that encompasses various areas of the sciences and engineering. Therefore, we wanted to build a system that would not only meet the main recognition objective, but, more importantly, we wanted to make practical use of this data. Thus, the decision was made to use the PR2 robot (seen in Figure 2).

Our end objective was to use the robot's Kinect RGB camera to accurately, confidently, and efficiently detect the image using only 2D image data. Once we had made a confident match, we would then use the Kinect's features to pinpoint a probable location of the object in 3D space. By converting this point into the robot's coordinate frame, we can then have the robot physically indicate the location of the object. This physical demo was the ultimate end goal of the project.

In hopes of releasing our code, we decided to make use of the widely used OpenCV code base. Furthermore, all of our code was implemented to be used within ROS (Robot Operating System). Details of the code, related documentation, and its release can be found later in this document. We begin by detailing our approach.

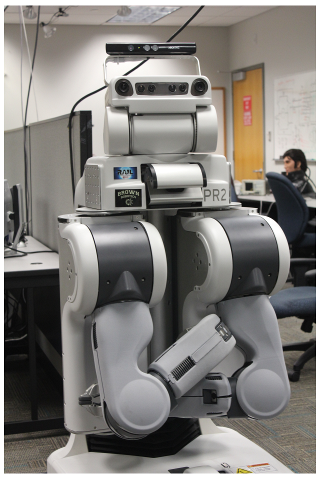


Figure 2: The PR2 robot.

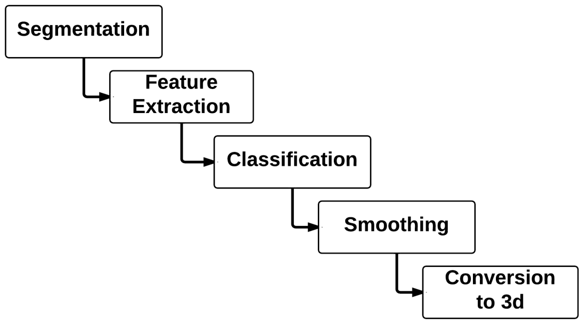


Figure 3: The object recognition pipeline.

2 Approach

As stated above, the main objective of the project was to implement an effective object recognition pipeline for use in real-time. An overview of our pipeline can be seen in Figure 3. Our approach began with image segmentation to divide the original image into smaller images of each potential object for separate processing. Features are then extracted from each image segment to create a feature vector. This vector is used as input for a classifier, which determines what class of object the image segment belongs to. To improve real-time performance, the output of the classifier is run through a smoothing algorithm to remove false positive classification of the target object. Finally, the detected target object is converted into a point in three-dimensional space, which can be used by the robot to physically locate the object. Each step of the pipeline is explained in detail below.

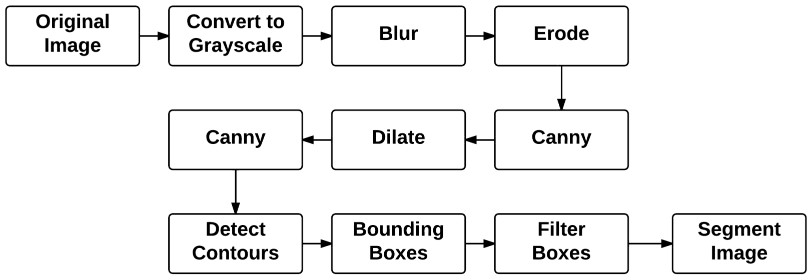


Figure 4: The segmentation pipeline.

2.1 Image Segmentation

Although a variety of image segmentation techniques exist, one of the goals of our project was to implement as much of the object detection pipeline as possible. As such, we implemented our own segmentation algorithm. The algorithm consists of three major steps: noise removal, edge detection, and closed contour detection. The detailed pipeline can be seen in Figure 4.

The segmentation pipeline begins with standard noise removal by converting the original Kinect RGB image to grayscale, applying a Gaussian blur, and eroding the image with an ellipse structuring element. With most of the noise removed, the algorithm then detects edge pixels using the Canny edge operator [1]. This creates a binarized image showing edge and non-edge pixels that divides the object into sections of closed contours around each object's detailed sections, as seen in Figure 5.

While clearly defined edges result in a connected set of edge pixels, some edges still have some gaps. These gaps are filled in with a dilate morphology operator, using the same structuring element as the erode morphology operator executed earlier in the pipeline. This dilation has the added effect of restoring the objects to their original size, since the objects' outer edges were diminished in the erosion. With the edges complete and the objects restored to their correct size, the algorithm runs the Canny edge operator a second time. This combines all of the smaller closed-contour sections of each object into a single outline contour for the object, which can be seen in Figure 5.

The final section of the segmentation pipeline involves detecting closed contours and fitting bounding boxes to them. The algorithm next detects closed contours using an OpenCV function. OpenCV also includes a function which fits a minimum area rotated bounding box to a contour. Using these two functions, all closed contours in the image are detected and bounded with rotated rectangles. Depending on how cluttered the image is, this can result in many closed contours and bounding boxes, so we filtered the bounding boxes by size to remove any particularly large or small contours, and by aspect ratio to remove any long, skinny contours. The results of the contour detection and bounding box filtering can be seen in Figure 6, which contains only bounding boxes of a size and shape consistent with the toy computer. These bounding boxes are then overlayed onto the original Kinect RGB image, and a separate image segment is extracted for each bounding box, completing the image segmentation process.

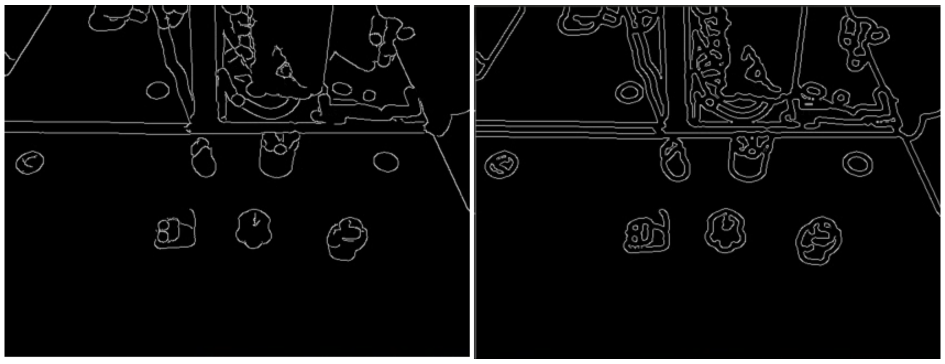


Figure 5: **Left**: Result of the first execution of the Canny edge operator. **Right**: Result of the second execution of the Canny edge operator.

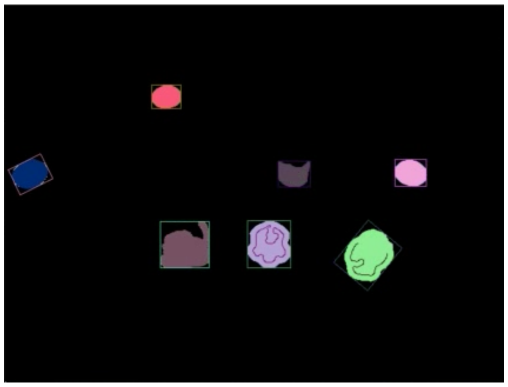


Figure 6: The filtered output of bounded contour detection.

2.2 Feature Vector Extraction

We processed each image segment separately to extract a feature vector for later use in classification. We took into consideration features that use both color and geometric information, all normalized on a real scale from 0 to 1. For color, we used the RBG channels. For geometry, we considered three categories: results from a Hough transform, a count of SIFT features, and how similar the closed contours in the image segment were when compared to a reference rectangle.

The first three features we used were the average red, green, and blue colors of the entire image segment. For each color, we took the arithmetic mean of each pixel intensity in the image segment and then normalized the result by 255, the maximum value per channel. These three features helped us to distinguish amongst objects of different color; however, this was not enough information to rely on alone. We also note that these features are invariant to scale, rotation, translation, and skew.

Next, we looked at geometric features using line segments. Before applying the Hough transform to identify these line segments, we applied a blur mask, followed by the same Canny edge operator mask used throughout the project. We created two features from the resulting lines: the total number of line segments in the current image segment and the average line segment length. Both feature values were limited by a predefined maximum (150 pixels and 25 pixels respectively) and

[R | G | B | # Lines | Avg. Line Len. | # SIFT | Rectangle %]

Figure 7: The Feature Vector used for Classification.

then normalized by the same respective values. By using the Hough transform to find line segments, we created a feature that can help to describe aspects of the objects geometry. For example, a circular object like the turtle produced a large number of small line segments on the outline while the toy computer had a small number of longer segments corresponding to the rectangle edges. These features are invariant to rotation and translation.

We next used the number of SIFT features as a measure of the object's complexity. We did not use any matching here to avoid going through the computationally intensive process of matching the features to a reference image. Furthermore, matching features would have required analyzing multiple images of the rotated object and choosing the best image. In the case of our main target, the toy computer, there were numerous angles from which reference images would have to be compared to, since the object was not symmetric. Nevertheless, the number of features returned by the SIFT algorithm on each segmented image gave us a measure of the objects' complexity that is invariant to rotation, scale, and translation. To use in the feature vector, we limited and normalized the number of SIFT key points found by a predefined maximum (20).

Furthermore, since the shape of the toy computer contains, from any perspective, a large number of rectangles relative to the total number of contours in the image (the shape of the screen, the keyboard keys, parts of the sides, the bottom, etc.), we computed a feature which measured how many closed contours in the image are rectangular. First, we passed the image segment through our custom image segmentation algorithm described in the previous section to gather contours. Using a function included in OpenCV, we compared each of the resulting contours with a reference rectangle with the aspect ratio of 5:7. This function uses seven Hu moments[4] for comparing shape. Hu moments are invariant to translation, scale, and rotation. For each contour that matched the reference rectangle, we incremented a counter. After all contours were considered, we divided the result by the number of contours to normalize the value.

The resulting feature vector, shown in Figure 7, had rotationally, translationally, and scale invariant elements. Also, the geometric features were effectively scale invariant within a reasonable interval. This vector could then be used for classification.

2.3 Classification

With ability to segment an image into probable objects and obtain seven dimensional feature vectors describing each image segment, we were now able to attempt to classify each of the segmented objects. To do so, we needed to decide on two important factors: the classification method and the set of training data.

After discussing the pros and cons of various techniques such as Gaussian Mixture Models (GMM) or Support Vector Machines (SVM), we decided to use a fairly simplistic, yet powerful, classifier: the Naive Bayes Classifier[2, 5] which was provided as part of OpenCV. In short, this model makes use of the widely known Bayes Theorem ($P(A|B) = \frac{P(B|A)P(A)}{P(B)}$) to find the most likely classification for a given feature vector. That is, for a set of classifications C and our seven numerical features $F_1, \dots, F_7$, we look for the classification C_i that maximizes the conditional probability $P(C_i|F_1, \dots, F_7)$.

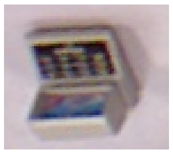
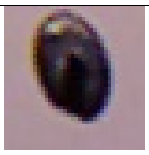
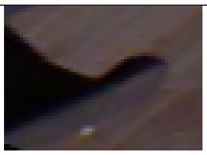
Toy Computer		Pens	
Squishy Robot		Squishy Turtle	
Toy *Cylon*		Desk Wire-Hole Cover	
Other			

Table 1: The 7 classification classes and an example associated segmented image.

What makes this model efficient to use is its assumption of independence amongst the features in our feature vector. This is the key point in the Naive Bayes model. We note that although this is not necessarily true with the features of our image (as they rarely are in any feature vector), in practice this is a valid assumption. This is further justified by the efficiency of the overall pipeline and the accuracy performance of the model discussed in later sections.

2.3.1 Training

As with any classifier, we started by providing a set of training data. To make the approach of our pipeline more robust and generic, we decided to not make our classifier binary; that is, we did not simply want to train on a set of positive images (i.e., the toy computer) and a set of negative images (i.e., anything else in the scene). We choose to instead train the classifier to recognize seven object classes. The seven classes as well as an example image produced from our segmentation algorithm are given in Table 1.

To train our classifier, we began by taking various images of the objects at various positions and angles using the robot's Kinect camera. We then fed the images into our segmentation algorithm to ultimately produce 231 segmented training images (36 computer samples, 39 turtle samples, 18 robot samples, 19 toy *Cylon* samples, 35 pens samples, 54 desk wire-hole cover samples, and 30 other samples). We labeled these outputs by hand such that we could then feed them into our feature extraction method, which was finally put into the classifier as a pair between the classification and the corresponding feature vector. Furthermore, since training data was given of our target object at various orientations, we could correctly classify the object regardless of orientation. This is shown in a video referred to later in this report.

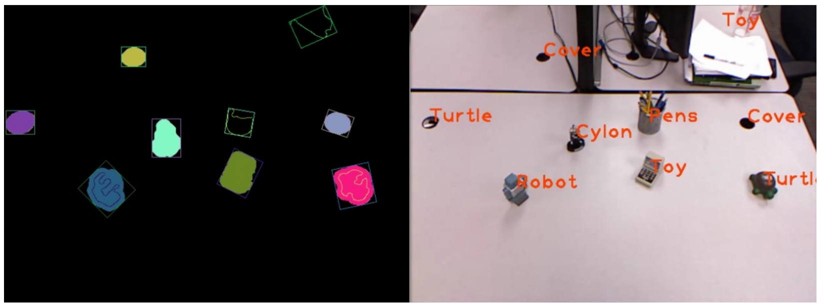


Figure 8: **Left**: The resulting segmented objects. **Right**: The results of the classifier for each of the segmented objects found in the image.

2.3.2 Multi-Object Recognition

As a result of above approach, we could now not only attempt to detect the target object, but we can also label various other objects as well. At this point in our pipeline, we could now take an image from the robot's Kinect, segment the image into a set of possible objects, run each segmented object through the feature extraction method to get its corresponding seven-dimensional feature vector, run each feature vector through the classifier, and label the results on the original image. The output of this process is depicted in Figure 8. For a better example of this procedure, please refer to the demo video located at `https://www.youtube.com/watch?v=ORFOlFgczsk`.

To give a rough estimate of the classifier's performance, we looked at the percentage of correctly classified training images and false positive classifications on the entire training set when run through the final classifier. We note that a more accurate statistic would be on a set of unseen training data; however, the decision was made to include all sample images as training data to improve the accuracy of the final, live system. The results showed an accuracy of 83.33% for correct classifications, and a false positive rate of 1.54%. While it is impractical to expect a perfect classifier, we make additional steps to make a more confident classification further down the pipeline by smoothing the classification results.

2.4 Smoothing & False Positive Reduction

As previously mentioned, classification will not always provide the correct solution. This problem can be viewed as two separate issues: failure to detect the toy computer when it is in the scene, and false positive classification. The latter is largely a problem when attempting to classify objects that the classifier has never seen before. To overcome these two problems we introduced a smoothing function to the pipeline.

The algorithm, presented in Algorithm 1, is run after each iteration of the recognition pipeline. That is, it assumes it will be running in real-time. Before beginning, it initializes a set P which is indexed by $\{x, y\}$ coordinates (representing the center location of a segmented image) and a counter c. During each iteration of the recognition pipeline, it takes the set of all segmented images that were classified as the target object. For each one of these segmented images, it checks to see if their location is already contained in the set P (within some distance threshold ϵ). If it is, the algorithm increments that location's counter. This value is maxed out by Ω (set to 20 in our setup) which prevents the smoothing algorithm from keeping a point in the set for longer than approximately one second. After incrementing these counts, the algorithm iterates through each point in the set

Algorithm 1 Smoothing classifications over a set time period.

```
Require: $P$ = empty set of $\{x, y\} \to c$ pairings
 1: for all $s_i \in$ Target Classified Segmented Image Center Pixel Locations do
 2:     if $s_i \in P$ within some $\epsilon$ then
 3:         $P(s_i).c \to \min(p_i.c + 1, \Omega)$
 4:     else
 5:         $P(s_i).c \to 1$
 6:     end if
 7: end for
 8: for all $p_i \in P$ not visited do
 9:     $p_i.c \to \max(p_i.c - 2, 0)$
10: end for
11: $f \to p_i \in P$ with max count $c$
12: if $f.c \geq \Phi$ then
13:     return $f$
14: else
15:     return $null$
16: end if
```

P that was not recently visited. For each of these points has their count decremented by 2 (note that we prevent this count from becoming negative). Finally, the updated set P is searched for the location with the highest count. If this count is above some threshold Φ (set to 5 in our setup), the algorithm returns this location as the most confident location of the object. If no point meets this criteria, it simply returns no object location.

The result of this algorithm becomes immediately apparent in a real-time system. Without the smoothing in place, we frequently observe faults such as losing the object when it is not detected for a few frames even though we have observed its location for multiple prior frames, or displaying false positives when the misclassified object has only been incorrectly classified for a few frames at a time, but not over a longer time period. The output of this process is depicted in Figure 9. For a better example of this procedure and proof of its effectiveness in real-time systems, please refer to the demo video located at `https://www.youtube.com/watch?v=0Gw8YbK4WMw`. Furthermore, this video exemplifies the robustness of the pipeline thus-far as one of our group members arbitrarily rearranges the scene and as the camera changes angles.

2.5 3D Space Transformation and Robot Pointing

With this new smoothed output from our pipeline, we were now able to correctly locate the object confidently while ignoring the vast majority of false positives. As previously mentioned, this is more clearly shown in the example videos referenced in this video.

At this point, we were confident enough with our recognition of the object to begin passing information about the object back to the robot. Since our end goal was to have the robot point its end effector at the target object, we first would need to locate the object in 3D space. Capabilities of the robot's Kinect helped with this process. By matching up the resulting 3D point cloud to the Kinect's RGB data, we could take the 2D pixel location of the center of the object found in our object recognition pipeline up to this point and find its depth away from the Kinect. Furthermore,

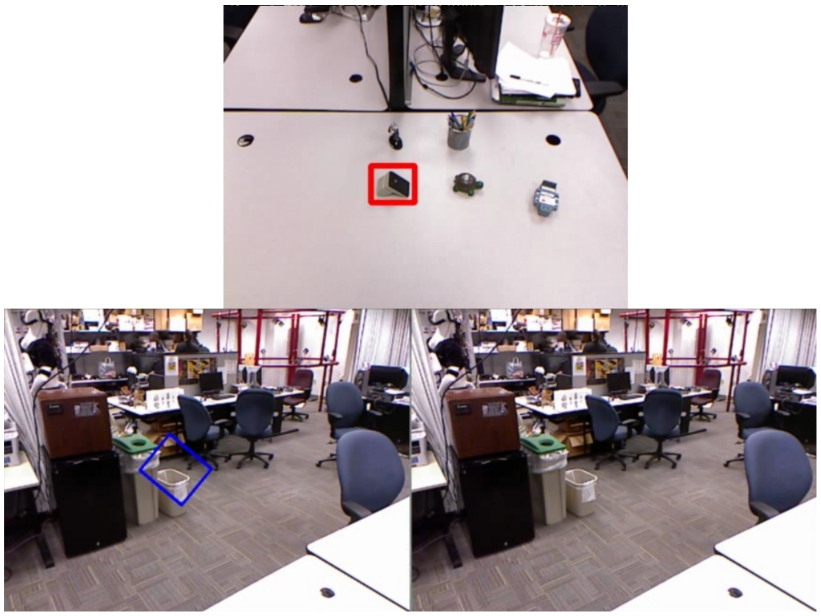


Figure 9: **Top**: The output of the smoothing function after confidently finding a match (outlined in red). **Bottom-Left**: A misclassified object (outlined in blue) found by the classifier. **Bottom-Right**: The smoothed output that successfully removes the brief false-positive.

since the PR2 was pre-calibrated to know the location of the Kinect relative to its own coordinate frame, an easy calculation could be made to convert the objects location into the robot's coordinate frame. We therefore could now track the object in 3D space. This is depicted in Figure 10.

Finally, we could query the robot's inverse kinematics services to move the robot's end effector. The goal position was set to one centimeter away from the object to prevent collission. To stop repetition of the point action, we had the robot re-point to the object if the object had moved at least a few centimeters. An example of the final pointing action is depicted in Figure 11. For a better example of this procedure, please refer to the demo video located at `https://www.youtube.com/watch?v=35rbeE1c-D4`.

3 Implementation, Code Documentation, & Release

In addition to providing the raw source code, we also include a brief, high-level overview of the code-base, as well as information on where documentation, source code, and distribution packages can be found.

3.1 Source Code

In an effort to give back to the ROS community during this project, source code has been released under the Robot Autonomy & Interactive Learning (RAIL) lab at WPI (a lab to which all three group members belong). Source code can be found at the following location: `https://github.com/WPI-RAIL/rail_cv_project`.

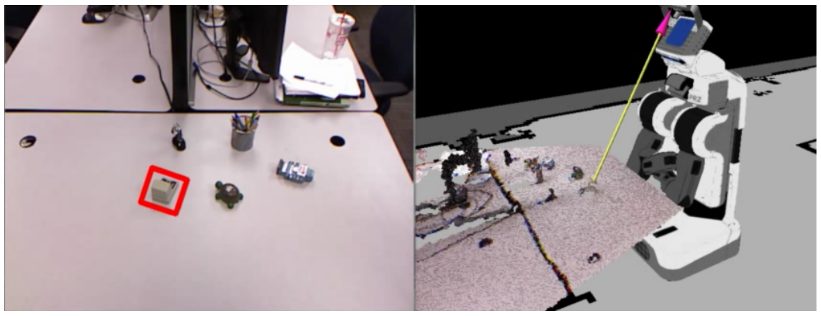


Figure 10: **Left**: The smoothed output from our pipeline. **Right**: A 3D visualization of the robot (using the RVIZ program included with ROS) with a vector drawn to display the location of the object in 3D space.

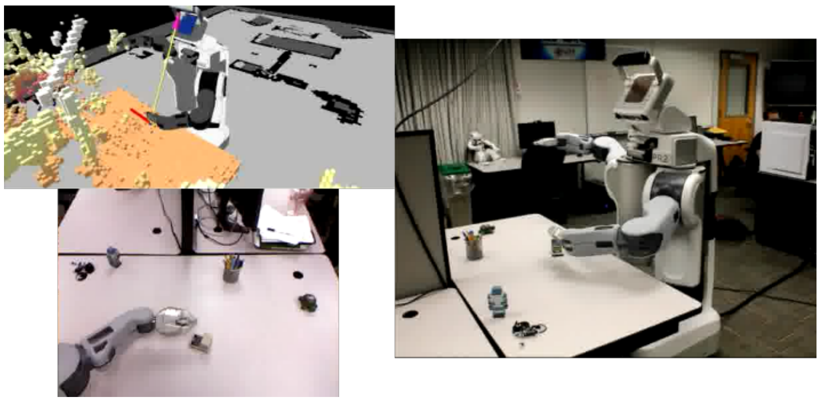


Figure 11: **Top-Left**: The point action as seen by the 3D visualizer. **Bottom-Left**: The point action as seen by the robot's Kinect. **Right**: The point action as seen by the robot's side.

In addition to the various ROS build files, the implementation for this project consists of four main files: `processor.cpp`, `robot.cpp`, and their associated header files.

3.1.1 Image Processing

The main object recognition pipeline is fully implemented in `processor.cpp`. It's header file not only contains the function and class declarations used in `processor.cpp`, but also the definitions of all constants and parameters (e.g., Canny kernel size, blur factors, etc.) used throughout the project.

Once compiled, the `processor` can be used in three different modes: gathering training data, running the accuracy test, or running the full pipeline. A prompt is displayed in the terminal at runtime to determine which mode to run.

When gathering training data, the process grabs a frame from the Kinect's RBG camera, runs the image through the segmentation algorithm, and saves all of the resulting segmented images. These images can then be labeled and used as training data for the classifier. Once one iteration has completed, a prompt is displayed asking if the user would like to continue or if they are finished. This allows users to rearrange the scene in order to easily gather more unique training data.

In the second mode, the program can be used to run an accuracy test of the classifier. Here, the program loads all training data, trains the classifier, and runs the training data back through the classifier. The percentage of correctly classified computer images and the number of false positives are outputted.

In the third mode, the full object recognition pipeline is run in real-time. It begins by setting up communication to ROS and subscribing to the robot's Kinect. Once this is made, the classifier is trained. Upon the arrival of each new image from the Kinect, the image is segmented; each segmented image is run through the classifier. For each of the segments classified as the target object, we run them through the smoothing function. We take the output (if any) from that function as the location of our object. As this is running, the following images are constantly streamed to the user's screen:

- The output after a blur, erosion, and first Canny edge operation
- The output after the above steps, a dilation, and the second Canny edge operation
- The output after the closed-contour detection, bounding box fitting, and bounding box filtering
- The original image with classification labels on all segmented parts of the image
- The original image with blue boxes drawn around any segmented parts that were classified as the target object
- The original image with a red box around the segmented part of the image that is the computer after smoothing

Examples of these views are seen throughout the project. In addition to this pipeline, this program is also responsible for broadcasting the position of the object in 3D space. To do so, once the object is detected and passes our smoothing test, we query the Kinect to find the location of the corresponding pixel in 3D space. From here, we use ROS's transform (TF) library to publish this position relative to the robot's Kinect. This will allow us to later locate the object in a separate ROS node when we want to move the robot's end effector.

3.1.2 Robot Control

The second main program in this project is the robot node found in `robot.cpp`. This ROS node is what is responsible for all control of the robot in response to our object recognition node. At a high level, the node acts on a single loop. To begin, the node listens for the 3D position of the object relative to the robot via ROS's TF library that is being published from the node discussed previously. If the position has moved at least a few centimeters in 3D space, then we calculate a target point 1cm away from the object. Finally, we make a request to the PR2's inverse kinematics services to move the robot's end effector to our target position. Once there, we wait for a few seconds before returning the arm to our pre-defined home position. Once again, we wait for a change in the objects location before repeating the pointing action.

3.2 Code Documentation

In an effort to make the code reusable as both an example and a reference, we have published two sets of documentation for the ROS community. For example, if a researcher wanted to modify the package to recognize a new set of objects, these sets of documents would be extremely helpful.

The first set is a collection of helpful notes on how to download, compile (or simply install), and run the code on an existing PR2 setup. This documentation can be found on the ROS wiki site: `http://www.ros.org/wiki/rail_cv_project`.

Furthermore, the source code of the project contains detailed Doxygen comments. These comments have been generated and posted on the ROS documentation website to provide insight on how the code works for future developers. This documentation can be found at `http://ros.org/doc/fuerte/api/rail_cv_project/html/`.

3.3 Release & Packaging

In addition to the source code and adequate documentation, we have also run our code through the official ROS release process. This process allows servers at Willow Garage to automatically download, compile, and package our code as an Ubuntu package that can be distributed with the default ROS repositories. By doing so, we allow researchers who are interested in trying out our code to easily install on an Ubuntu machine with ROS installed via a single command: `sudo apt-get install ros-fuerte-rail-cv-project`.

4 Conclusions & Future Work

Over the course of the project we designed and implemented an extendable system for object recognition from 2D images. Given the modular architecture of our system and availability of the source code, any stage of the classification pipeline could be changed by other researchers, e.g. new features could be introduced or the classifier could be changed. Also, the target object of the classification can be changed by providing additional training data. Furthermore, by integrating our system with ROS, the image coordinates of the identified object's position can be used in more complex applications. We demonstrated such an application via the physical pointing functionality. We consider our project successful with our demonstrated ability of having the PR2 reliably accomplish the task.

We leave open to future work the possibility of improving aspects of the recognition system itself. For example, we note the reliance on adequate lighting in our pipeline. Fluctuations in lighting conditions negatively impact the classification accuracy. Current methods for correcting such an issue are complex, such as computing the eigenvalues for the color representation in multiple passes [3]. Also, the segmentation pipeline does not perform as well in high-clutter environments, which could be improved by adjusting the segmentation process. Furthermore, we note that the use case of the 3D location was merely an example of the possibilities this information could be used for. Notable extensions include using the pipeline to help with grasp planning.

References Cited

[1] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, June 1986.

[2] Rich Caruana and Alexandru Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning (ICML 06*, pages 161–168, 2006.

[3] Zhenyong Lin, Junxian Wang, and Kai-Kuang Ma. Using eigencolor normalization for illumination-invariant color object recognition. *Pattern Recognition*, 35(11):2629 – 2642, 2002.

[4] Hu Ming-Kuei. Visual patern recognition by moment invariants. In *IRE Transactions on Information Theory*, volume IT-8, pages 179–187, 1962.

[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.